Department of Computer Science

On Pragmatic System Design through Learning and Implementation-oriented Reachability Analysis

Georgios Giantamidis

Aalto University

DOCTORAL THESES

Aalto University publication series
DOCTORAL THESES 126/2023

On Pragmatic System Design through Learning and Implementation-oriented Reachability Analysis

Georgios Giantamidis

A doctoral thesis completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Science, at a public examination held online on 30 August 2023 at 15:00.
Remote connection link: https://aalto.zoom.us/j/7445914070

Aalto University
School of Science
Department of Computer Science

Supervising professor
Associate Professor Chris Brzuska, Aalto University, Finland

Thesis advisors
Associate Professor Stavros Tripakis, Northeastern University, USA
Doctor Stylianos Basagiannis, Collins Aerospace, Ireland

Preliminary examiners
Professor Marco Di Natale, Scuola Superiore Sant'Anna, Italy
Associate Professor Panagiotis Katsaros, Aristotle University of Thessaloniki, Greece

Opponent
Associate Professor Panagiotis Katsaros, Aristotle University of Thessaloniki, Greece

Aalto University publication series
DOCTORAL THESES 126/2023

ISBN 978-952-64-1386-0 (printed)
ISBN 978-952-64-1387-7 (pdf)
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)
http://urn.fi/URN:ISBN:978-952-64-1387-7

Unigrafia Oy
Helsinki 2023

Finland

NORDIC SWAN ECOLABEL
Printed matter
4041-0619

Aalto University

Abstract

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Author
Georgios Giantamidis

Name of the doctoral thesis
On Pragmatic System Design through Learning and Implementation-oriented Reachability Analysis

Publisher School of Science

Unit Department of Computer Science

Series Aalto University publication series DOCTORAL THESES 126/2023

Field of research Computer Science

Manuscript submitted 17 August 2022 **Date of the defence** 30 August 2023

Permission for public defence granted (date) 18 October 2022 **Language** English

☐ **Monograph** ☒ **Article thesis** ☐ **Essay thesis**

Abstract

The need for formalization and verification in the design of complex systems is now more evident than ever. However, formal methods practices can sometimes be challenging to adopt in industrial environments. In particular, two broad categories of challenges can be identified: (a) The algorithmic challenge, which is about the ability of related tools and algorithms to scale to industrial size problems, and (b) the modeling challenge, which is about obtaining a formal system model as well as a formal specification of its behavior. To the end of easing integration of formal methods in industrial model based system engineering workflows, a solution is developed in this thesis aiming to help address the modeling challenge through contributions to four key areas of the process: (1) requirements formalization, (2) monitor generation, (3) model extraction from example behavior traces, and (4) reachability analysis for dynamical system implementations (C/C++ code).

Keywords Formal Methods, Learning, Requirements Formalization, Monitor Generation, Reachability Analysis

ISBN (printed) 978-952-64-1386-0 **ISBN (pdf)** 978-952-64-1387-7

ISSN (printed) 1799-4934 **ISSN (pdf)** 1799-4942

Location of publisher Helsinki **Location of printing** Helsinki **Year** 2023

Pages 180 **urn** http://urn.fi/URN:ISBN:978-952-64-1387-7

Preface

I would like to thank Prof. Stavros Tripakis for introducing me to the world of formal methods, giving me the opportunity to study under his guidance, the insightful discussions leading to new ideas, as well as the honest feedback required to methodically transmute a half-baked algorithm into a peer-reviewed publication.

I would also like to thank Prof. Chris Brzuska for the invaluable support towards the end of my studies and finalization of the thesis; in particular, the very helpful discussions and feedback throughout preparation of the final manuscript, as well as the availability and guidance towards swiftly addressing any arising issues.

Special thanks go to Dr. Stelios Basagiannis for giving me the opportunity to apply my ongoing research in an industrial environment, as well as his guidance along the way, including the countless brainstorming sessions in front of the whiteboard.

I would also like to give very many thanks to Dr. Vassilios Tsachouridis and Dr. Kostas Kouramas for their immense help, including the many interesting discussions out of which novel ideas emerged (and some of which were turned into publications included in this thesis), as well as my family and friends who supported me in (and some of whom partly shared with me) this journey.

Special thanks also go to my pre-examiners, as well as my opponent, Panagiotis Katsaros, for taking the time to carefully go through the thesis and their valuable feedback.

Finally, I would like to, once again, thank all the above for their superhuman patience and making sure to always keep me motivated and focused towards finalization of my doctoral studies – to paraphrase a well known quote, I guess it really takes a village to raise a PhD.

August 4, 2023,

Georgios Giantamidis

Contents

List of Publications

This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.

I Georgios Giantamidis, Georgios Papanikolaou, Marcelo Miranda, Gonzalo Salinas-Hernando, Juan Valverde-Alcalá, Suresh Veluru, Stylianos Basagiannis. ReForm: A Tool for Rapid Requirements Formalization. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, Vol 79, 2020.

II Georgios Giantamidis, Stylianos Basagiannis, Stavros Tripakis. Efficient Translation of Safety LTL to DFA Using Symbolic Automata Learning and Inductive Inference. In *Computer Safety, Reliability, and Security*, 2020.

III Georgios Giantamidis, Stavros Tripakis, Stylianos Basagiannis. Learning Moore machines from input–output traces. *International Journal on Software Tools for Technology Transfer*, Vol 23, 1-29, 2021.

IV Georgios Giantamidis, Stavros Tripakis. Learning Moore Machines from Input-Output Traces. In *FM 2016: Formal Methods*, 2016.

V Vassilios A. Tsachouridis, Georgios Giantamidis, Stylianos Basagiannis, Kostas Kouramas. Formal analysis of the Schulz matrix inversion algorithm: A paradigm towards computer aided verification of general matrix flow solvers. *Numerical Algebra, Control & Optimization*, Vol 10(2), 177-206, 2020.

VI Vassilios A. Tsachouridis, Georgios Giantamidis. Computer-aided verification of matrix Riccati algorithms. In *58th Conference on Decision and Control*, 2019.

Author's contributions

Publication I: "ReForm: A Tool for Rapid Requirements Formalization"

The author wrote 100% of the article and is the sole core contributor and current maintainer of the presented workflow and tool. In particular, he came up with the idea and implemented the initial workflow including extraction of natural language requirements from documents, requirement preprocessing, requirement clustering, pattern identification and formalization, monitor generation, consistency checking, as well as the graphical user interface. Georgios Papanikolaou reimplemented the clustering algorithm in a different language for better performance, and made various usability improvements on the user interface. Marcelo Miranda implemented a monitor generation algorithm for an additional specification language added later in the tool. Suresh Veluru wrote some of the NLP analysis heuristics used in the requirement preprocessing phase. Gonzalo Salinas-Hernando and Juan Valverde-Alcalá built the Simulink models used in the industrial case studies and subsequently verified them by using the monitors generated by the tool and MATLAB's Simulink Design Verifier Toolbox. Stylianos Basagiannis tested the tool extensively and provided useful feedback during development.

Publication II: "Efficient Translation of Safety LTL to DFA Using Symbolic Automata Learning and Inductive Inference"

The author came up with the idea, designed the algorithm, derived the theoretical results, wrote nearly 100% of the article, implemented the proposed approach and conducted the experimental evaluation. The co-authors contributed with useful feedback on structuring the text and the experimental evaluation.

Publication III: "Learning Moore machines from input–output traces"

The author wrote around 90% of the article, designed the proposed algorithm, derived the theoretical results, and implemented the accompanying code and experimental evaluation. Stavros Tripakis came up with the idea, wrote part of the paper (mainly introduction section), and together with Stylianos Basagiannis provided useful feedback on structuring the text and the experimental evaluation.

Publication IV: "Learning Moore Machines from Input-Output Traces"

The author wrote around 90% of the article, designed the proposed algorithm, derived the theoretical results, and implemented the accompanying code and experimental evaluation. Stavros Tripakis came up with the idea, wrote part of the paper (mainly introduction section) and provided useful feedback on structuring the text and the experimental evaluation.

Publication V: "Formal analysis of the Schulz matrix inversion algorithm: A paradigm towards computer aided verification of general matrix flow solvers"

The author wrote around 50% of the article (the section related to verification and part of the introduction), implemented the reachability analysis framework and conducted the experimental evaluation. Vassilios Tsachouridis came up with the idea and derived the theoretical bound formulas that were used in the experimental evaluation. Stylianos Basagiannis and Kostas Kouramas contributed parts of the introduction as well as overall feedback on the approach.

Publication VI: "Computer-aided verification of matrix Riccati algorithms"

The author wrote around 50% of the article (the section related to verification and part of the introduction), implemented the reachability analysis framework and conducted the experimental evaluation. Vassilios Tsachouridis came up with the idea and derived the theoretical bound formulas that were used in the experimental evaluation.

List of Figures

Abbreviations

CEGIS CounterExample-Guided Inductive Synthesis

CS Characteristic Sample

CTL Computation Tree Logic

DDL Differential Dynamic Logic

DFA Deterministic Finite Automaton

DMD Data-driven and Model-based Design

DSL Domain Specific Language

FPGA Field Programmable Gate Array

FSM Finite State Machine

LTL Linear Temporal Logic

MBD Model-Based Design

MBSE Model-Based Systems Engineering

ML Machine Learning

NLP Natural Language Processing

ODE Ordinary Differential Equation

PSL Property Specification Language

PTA Prefix Tree Acceptor

SMT Satisfiability Modulo Theories

STL Signal Temporal Logic

1. Introduction

Given my interest in both mathematics and computer science from a very young age, encounter with formal methods was inevitable, as they can be found at the intersection of the two. To me, formal methods were the ultimate form of magic: Synthesizing a system in a correct-by-construction way that guarantees specific behavior expressed in a set of requirements looked akin to crafting a spell carefully tailored to carry out a specific task. And this was more than enough motivation to get me involved in the field and the pursuit of improving the state of the art. While doing so, I realized that, even though the usefulness of formal methods is well understood, there are hindrances that prevent widespread adoption in certain parts of the industry. These can broadly be split into two categories: (a) algorithmic challenges and (b) modeling challenges. The former are about how well the underlying procedures scale on systems of realistic size, while the latter are about the effort required for modeling a system as well as its expected behavior in terms of requirements. I decided to focus on the latter set of challenges for my PhD thesis, in order to help others who want to become wizards too, to do so in an easier way.

1.1 Background

1.1.1 System Design

The variety of system design methodologies in practice today can be categorized based on several dimensions. One such dimension is the high-level structured (or not) workflow they may follow. Some examples here are the traditional waterfall approach [116], the widely used V-model approach (Figure 1.1) [114], and the more recent agile approach [115], which tends to be popular among startups. Another important dimension is whether we move directly from the mind of the designer to a system prototype (implementation) or whether this transition is gradual and involves build-

ing (abstract) system models in the process, in which case we talk about Model-Based Design (MBD) [15, 105, 66, 78, 44, 93, 94, 99, 55, 104]. In the case where models are used, we can further classify based on whether these models are built manually or automatically (e.g. from specifications and / or example behaviours). In addition to that, there is also the question of which kinds of models are used. These can, for example, be (finite) state machines, differential equations, hybrid automata [14, 54], neural networks etc. Some of these models can actually also become part of the final system implementation; for example, a neural network model could be used as (part of) the image recognition software module of a self-driving car. Alternatively, the models can be further refined into more efficient implementations; for example, a neural network model could be implemented on an FPGA.

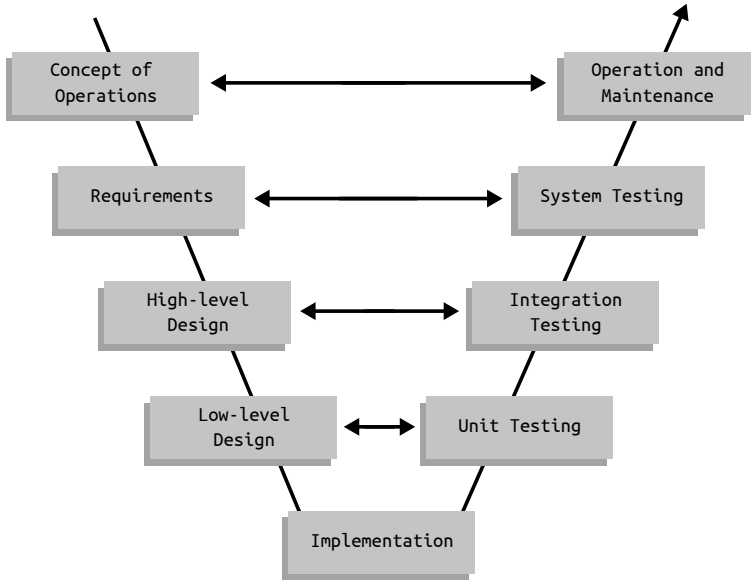


Figure 1.1. V-Model system design methodology

Regardless of the specifics of a particular system design methodology, it is well understood today that model-based design offers several concrete advantages over prototype-based design (where no models are involved) [107]. In particular, models are safer than prototypes, cheaper and faster to build, modify and maintain, as well as cheaper and faster to simulate (e.g. for testing purposes). In addition to that, one can perform more rigorous types of analysis on models (such as static analysis and formal verification) that cannot be performed on prototypes.

One disadvantage of the current MBD state of practice is that, more often than not, these models are typically built by hand, which can be

quite expensive and error prone. In particular, it requires manual effort by domain experts, who may need several attempts to build a model conforming to the given set of requirements. An emerging paradigm w.r.t. this aspect of system design is the so called Data-driven and Model-based Design (DMD) [107]. In this context, models are synthesized automatically from specifications and / or example behaviour [16, 17, 18, 30, 39, 19], the end goal being to reduce human effort, as well as to obtain correct-by-construction models, guaranteed to conform to the requirements.

The focus of this thesis w.r.t. the system design aspect is in providing processes and algorithms to help migrate from the typical MBD setting into a more DMD-enabled one.

1.1.2 Formal Methods - What and Why

In 1970 Edsger Dijkstra famously stated that "Testing can only show the presence, not the absence of bugs"; in order to achieve the latter, a different approach is necessary. Formal methods constitutes such an approach, consisting of mathematically rigorous ways for specification and verification of hardware and software.

In the context of formal methods we can distinguish three specific activities: (1) modeling, (2) specification and (3) verification. Modeling is about describing the system, or rather an abstraction of the system, to be verified using an appropriate formalism, such as state machines (for finite state systems) or hybrid automata [54] (for cyber-physical systems). Specification focuses on describing the property to be verified and is typically done in some form of logic, such as propositional, first order, higher order, or modal logics, such as Linear Linear Temporal Logic (LTL) [86], Computation Tree Logic (CTL) [35] and Differential Dynamic Logic (DDL) [85].

Verification is about taking a model and a specification and applying a procedure in order to determine whether the model conforms to the specification. We distinguish two main categories here: Model checking [36] and deductive verification [51]. The former is an automatic approach of systematically performing exhaustive exploration of the given model. The latter is typically carried out with the help of proof assistants and requires manual effort, but can in principle handle more types of properties as well as larger models than model checking can.

1.1.3 A Brief History of Formal Methods

One can trace the beginning of formal methods [38] back to 1954, when Martin Davis developed the first computer generated proof for the theorem stating that the product of two even numbers is even [80]. Important milestones since then include the development of the Stanford Pascal

Verifier (1960s) [70], ACL (1970s) [1], Isabelle [4], Coq [2] and PVS [5] proof assistants (1980 - 2000) on the deductive verification side, as well as temporal logics (LTL [86], CTL [35] – 1970s), the first model checking algorithms (1980s) [34, 87], symbolic model checking (1993) [72], as well as bounded [33] and probabilistic [63] model checking (1999-2005) on the model checking side.

1.1.4 Formal Methods in Industry Today

Presently, formal methods are in use by leading hardware vendors [52, 45, 89] (their use was initially facilitated by the advent of symbolic model checking, which drastically increased the number of system states that can be explored automatically). Adoption on the software side is also growing by the day, so that leading software companies now have dedicated verification groups [77, 31, 21, 90, 29].

We can identify two broad categories of challenges that need to be addressed in order to increase adoption of formal methods in the industry: the algorithmic challenge and the modeling challenge. The former is related to the (in)ability of tools and algorithms used for verification to scale to industrial size problems – the so called state explosion issue of model checking is a representative example. Potential solutions here include abstraction and compositional verification approaches [36, 106].

The modeling challenge is about system model definition and requirement formalization. The algorithms used to conduct verification require a formal model of the system as well as a formal specification of the expected behavior. Generating each of these artefacts typically requires expert manual effort, the volume of which can sometimes be prohibitive in cases of legacy systems. Potential solutions here include automatic model extraction approaches, verification algorithms able to work on actual system implementations , as well as approaches for automatic requirement formalization. In this thesis, the focus is on providing solutions for the modeling challenge, primarily from an industrial point of view.

1.1.5 Learning

One can encounter several different forms of learning in the current state of practice; to name a few, consider system identification [69] and machine learning [75]. The goal of the former is to extract information about structure and / or parameters of an unknown system, while the latter is typically linked with artificial intelligence [91] and focuses on solving a variety of related problems, such as (image) classification, optical character recognition, natural language processing / understanding, etc.

Within each of these two categories, one can identify more refined partitions, based on the amount of training data needed, the learned model type,

as well as how easy the learned model is to analyze. For example, in the system identification category, the learned model could be a (finite) state machine, a dynamical system or a hybrid system, each of which would typically need more training data and be more difficult to analyze than the previous model type. Correspondingly, in the machine learning category, the learned model could be a decision tree, a random forest or a neural network, with similar characteristics w.r.t. amount of training data needed and amenability to analysis.

In this thesis we focus mainly on the system identification type of learning, and in particular on white-box (finite state machine) model learning.

1.2 Research Questions and Contributions

Arguably, the earlier formal methods are introduced in the design lifecycle of a system, the easier this is done. The real challenge lies in legacy systems that are implemented without best engineering practices in mind and end up in monolithic implementations that are practically black boxes (i.e. difficult to reason about or change).

More often than not, the problems in such cases begin with how requirements are handled. Typically, requirements are expressed in unstructured, natural language format, which is prone to ambiguities and prevents early potential inconsistency detection, as well as analysis and tool support opportunities in general. In addition, test cases and requirement monitors, if existent, are typically constructed manually, which is time consuming and error prone. While formalization of requirements could address these issues, it is often not performed as simply the vast volume of legacy requirements makes this prohibitively time consuming.

To facilitate the shift towards proper model based system engineering practices, including integration of formal methods, in such cases, we would need ways for rapid requirements formalization as well as model extraction from black-box systems. Practical verification approaches that can be applied on implementations (e.g. code) – and not just models – can also be useful here. In this context, we formulate the following four research questions which we address in the thesis.

Research Question 1: *Approaches for (semi-)automated requirements formalization typically have two flavours: (a) either go directly from natural language to a specification language or (b) go from controlled / constrained natural language to a specification language. In the former case, translation accuracy is typically not sufficiently high to be of practical use, while the latter case is typically limited to a particular domain and does not address the potentially big volume of natural language legacy requirements that have to be rewritten. Is it possible to employ learning techniques in order to get the best of both worlds?*

The answer is affirmative and the related contributions can be found in Chapter 2 and publication I. In particular, the developed requirements formalization workflow leverages NLP and ML techniques to automatically identify patterns in natural language requirements and, by doing so, significantly reduce the required formalization effort for both new and legacy requirements.

Research Question 2: *Existing approaches for safety LTL to DFA translation exhibit issues such as unbounded size of intermediate translation results and inability to take into account a-priori knowledge about the target automaton in order to speed up the trnanslation process. Is it possible to use learning in order to address these shortcomings?*

The answer is affirmative and the related contributions can be found in Chapter 3 and publication II. In particular, the developed monitor generation algorithm, by leveraging active automata learning techniques, provides theoretical guarantees about the size of the intermediate translation results, is able to leverage a-priori knowledge about the target automaton in order to accelerate the translation process, and manages to significantly outperform state of the art approaches w.r.t. execution time and memory consumption in some cases.

Research Question 3: *Is it possible to extend the RPNI passive automata learning algorithm to learn Moore machines, preserving efficiency (i.e. polynomial complexity) and other properties (e.g. identification in the limit)?*

The answer is affirmative and the related contributions can be found in Chapter 4 and publications III and IV. In particular, the developed finite state machine extraction algorithm is accompanied by theoretical results on convergence as well as an efficient implementation, outperforming the state of the art w.r.t. execution time and memory consumption.

Research Question 4: *Is it possible, in the context of dynamical systems and, in particular, matrix iterative algorithms, to perform automated reachability analysis directly on system implementations (e.g. C++ code) without the need to manually generate corresponding abstract models? And if so, what are the benefits of doing so over alternative approaches (e.g. translation of the code to model checker / theorem prover input)?*

The answer is affirmative and the related contributions can be found in Chapter 5 and publications V and VI. The developed workflow enables instrumentation of C/C++ code describing the behavior of a dynamical system towards performing automated reachability analysis without the need of deriving a separate model of the system. The developed approach is demonstrated through application of the workflow on iterative matrix algorithms, viewed as dynamical systems, where it enables a-priori computation of convergence bounds for given initial matrix ranges, for which existing theoretical (i.e. closed form) approaches are not able to provide an answer.

1.3 Thesis Organization

In this thesis, we present a solution towards aiding re-engineering of legacy systems using model based design best practices. This is done through contributions in four key areas: requirements formalization (Chapter 2), automated monitor generation (Chapter 3), model learning from examples (Chapter 4), and practical reachability analysis for system implementations (Chapter 5). Finally (Chapter 6), we conclude and discuss possible future extensions of the developed workflows and algorithms.

2. Requirements Formalization

Managing requirements in industrial environments is typically done in unstructured, natural language format, which prevents the adoption of automated analysis that can improve both quality and speed of development by e.g. detecting inconsistencies early in the design phase. In addition, requirement monitors and test cases are typically created manually, which, apart from being time consuming, is error prone. Formalization of requirements can provide a solution here, however the sheer volume of legacy requirements often makes this prohibitively time consuming. In order to address these issues, we developed an end-to-end workflow and tool for rapid requirements formalization, starting from natural language requirements and going all the way down to automatically generated monitors. Specifically, by using NLP and ML techniques for requirement pattern extraction, we accelerate formalization for both legacy and new requirements. Formalized requirements can then be used for consistency checking (in order to prevent early design error propagation), as well as for automatic test-case and monitor generation.

Approaches for automatic requirement formalization (natural language to formal language) have been explored before and generally fall into two broad categories. In particular, there are approaches that (a) translate from natural language to a specification language, e.g. [79, 53] and approaches that (b) translate from controlled natural language (typically domain specific) to a specification language, e.g. [23, 43, 3]. In the former case, the reported translation accuracy is generally not sufficiently high to be of practical use, particularly when applied on data that differ non-trivially from those used for training. In the latter case, while the approach is adequate for introducing new requirements, it does not enable efficient handling of the potentially big volume of legacy requirements that have to be rewritten.

The novelty of our approach lies in the fact that it combines useful parts from both worlds by essentially learning a controlled natural language (the extracted requirement patterns) from legacy requirements. And while the formalization part is manual, the overall workload is reduced drastically,

since the engineer only needs to formalize the (typically small) set of extracted patterns. To the best of our knowledge, the work closest to ours here is [23], the main differences being as follows: (i) They focus on *continuous* time properties by making use of Signal Temporal Logic (STL), while we focus on *discrete* time properties. (ii) They focus on requirements specified in a template-based constrained natural language, while we focus on automatically discovering such templates / patterns by analyzing unconstrained natural language requirements.

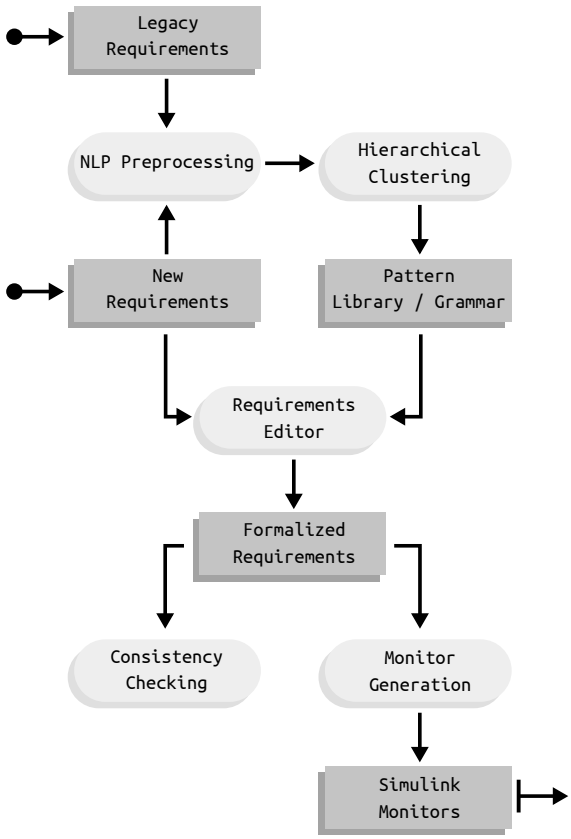


Figure 2.1. Requirements formalization workflow

The developed workflow is outlined in Figure 2.1. Legacy natural language requirements are preprocessed using off the shelf NLP tools [57, 71], as well as our own heuristics, in order to identify and abstract away domain entities and details (such as actual signal names and mathematical expressions) not relevant to pattern discovery. Abstract requirements are then clustered into groups using a hierarchical clustering algorithm. Several approaches have been explored here to define similarity between two requirements (necessary for the clustering algorithm to work), based on purely syntactic information, on purely semantic information, as well

as on combinations of the two, along with additional heuristics. Once the abstract requirements are placed into clusters, individual representatives of each cluster are essentially the patterns we are looking for. These patterns are formalized manually, however we reduce the required effort by employing several high level specification languages, namely PSL [7], SpeAR DSL [43] and SALT DSL [24]. Legacy requirements are formalized in batch during this process; once a pattern is formalized, all requirements following that particular pattern are automatically formalized as well.

New requirements can then be formalized using existing patterns through an editor supporting pattern and signal name autocompletion, as well as syntax checking using a context free grammar automatically derived by the set of identified patterns. In case no existing pattern is suitable, going through the same process as with legacy requirements to derive new patterns is always possible. Once a formalized set of requirements is obtained, consistency checking and monitor generation can be performed automatically. Consistency checking works across the supported specification languages by translating them into LTL or past LTL and then employing an existing algorithm [47] adapted to support linear arithmetic expressions as atomic propositions by leveraging the Z3 SMT solver [40]. Monitor generation currently targets Simulink models [41]. However, additional targets are not difficult to add, since we first generate a target-agnostic intermediate representation.

The developed approach has been applied on two industrial case studies: (a) Low-level requirements for the FPGA specification of Airbus A350 ETRAC (Electrical Thrust Reverser Actuation Controller), and (b) High-level requirements for the brake control unit of Mitsubishi Regional Jet. In the first case study, the entire workflow was used, from natural language requirements all the way down to formal verification of the Space Vector Modulation (SVM) subsystem of the design. We were able to fit 40% of the 750 requirements into 25 clusters, and formalized the 100 requirements related to the SVM subsystem using only 6 patterns. In the second case study, only the parsing and clustering parts of the workflow were applied, in order to demonstrate that our approach provides benefits (e.g. facilitating mapping of requirements to more structured representations) even for high-level requirements that cannot be easily mapped to Simulink monitors. In particular, we were able to fit 50% of the 700 requirements into 15 clusters.

3. Monitor Generation

Safety properties are ubiquitus in model based design. Capturing the notion that 'nothing bad should ever happen', they are typically expressed in Safety LTL and can be used for formal verification, runtime monitoring, test-case generation, as well as consistency checking. The first step in the aforementioned processes is translating the property at hand into an automaton. One drawback of existing approaches for this is that the size of intermediate tranlsation results can be significantly larger than the final automaton. In addition, to the best of our knowledge, existing implementations are unable to make use of a priori information about the translation target that may be available. In this work, we develop a novel approach for Safety LTL to symbolic DFA translation that addresses these limitations. In particular, our algorithm returns a minimal automaton (w.r.t. number of states) and provides theoretical guarantees that all intermediate results contain strictly fewer states than the learned automaton. In addition, the algorithm is able to incorporate a priori knowledge about the target automaton for a significant performance gain.

The problems of translating LTL to automata and specifically Safety LTL to DFA have received a lot of attention over the years [65, 12, 61, 46, 20]. To the best of our knowledge, the state of the art in the former case is Spot [12] and Rabinizer [61], while in the latter case we have scheck [65]. The problems of automata learning and grammatical inference, in general, have also been studied extensively [39]. While we do not claim to advance the state of the art in symbolic automata learning, note that in our extension of an existing learning algorithm we make specific assumptions about the nature of the automaton to be learned, which allows us to provide a more efficient approach than we could have done otherwise.

In particular, we focus on translating from the Syntactic Safety subset of LTL [100] into symbolic DFA [37] by adapting Angluin's L* algorithm for active automata learning [19]. In this setting, a learner tries to identify an automaton by submitting queries to a teacher. These can be membership queries, where the learner submits a word and gets back an 'accept' or 'reject' answer, or equivalence queries, where a hypothesis automaton is

submitted and either the process ends with success or a counterexample is generated that drives more subsequent queries.

An overview of our algorithm is shown in Figure 3.1. A data structure called the observation table is used throughout the algorithm to collect information made from membership queries. Once enough information is available, a hypothesis automaton is generated and submitted through an equivalence query to the teacher. In our case, membership queries are implemented by recursive traversal on the LTL formula to be translated, while for equivalence queries we employ the NuSMV symbolic model checker [32].

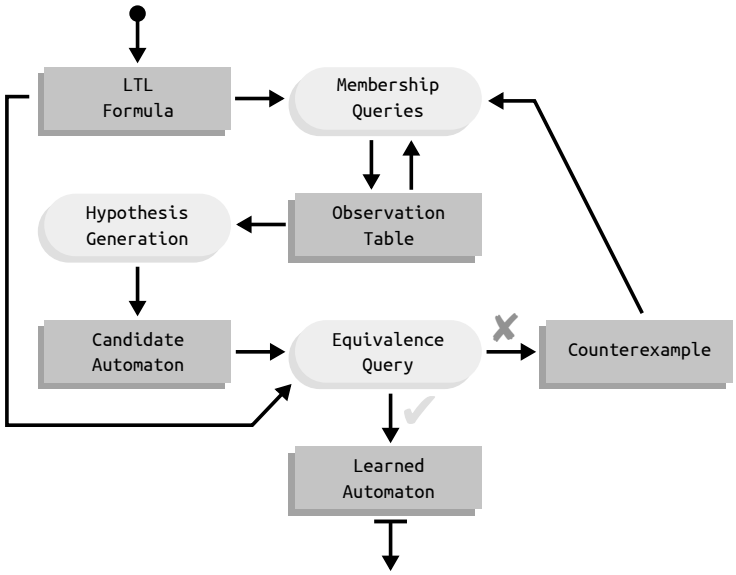


Figure 3.1. Monitor generation algorithm

Regarding the properties of the extended algorithm, minimality of the learned automaton, as well as theoretical guarantee that the intermediate hypothesis automata are strictly smaller than the learned automaton, directly follow from the properties of the L* algorithm. Regarding computational complexity, the L* algorithm is guaranteed to terminate after at most N equivalence queries and a number of membership queries bounded by a polynomial quadratic on N and linear on M, where N is the number of states of the learned automaton and M the maximum length of any counterexample returned by the teacher. In addition, the complexity of a membership query is polynomial on the trace length and exponential on the formula length, while the worst-case complexity of an equivalence query is at least doubly exponential on the length of the formula to be translated.

The query complexity results for equivalence queries motivated the

Table 3.1. Counter property families

N	Counter family A	Counter family B
1	$G(\neg p \vee X(\neg p \vee \neg q \vee r \vee Xr))$	$G(\neg p \vee X(\neg q \vee r))$
2	$G(\neg p \vee X(\neg p \vee X(\neg p \vee \neg q \vee r \vee Xr)))$	$G(\neg p \vee X(\neg q \vee (r \wedge Xr)))$
3	$G(\neg p \vee X(\neg p \vee X(\neg p \vee X(\neg p \vee \neg q \vee r \vee Xr))))$	$G(\neg p \vee X(\neg q \vee (r \wedge X(r \wedge Xr))))$

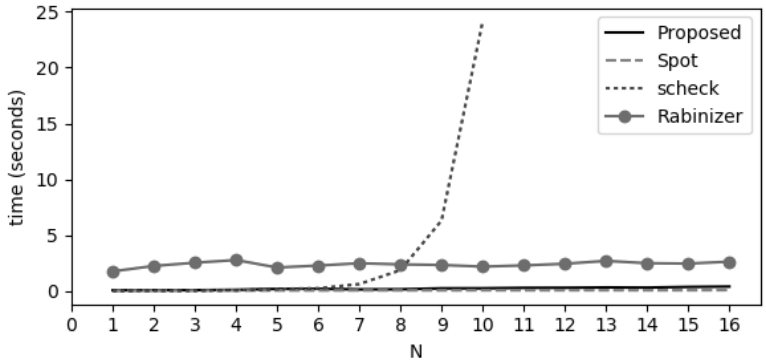
search for a modified approach that eliminates this type of queries altogether. It turns out that this is possible to do if we have some sort of a priori knowledge about the target automaton, which is relatively straightforward to obtain in cases where we deal with property families with members of increasing length such as these [1] shown in Table 3.1.

We implemented the proposed algorithm and compared against scheck v1.2 [65], Spot v2.6.1 [12] and Rabinizer v4 [61] on (i) 500 randomly generated syntactically safe LTL formulas, (ii) 54 formulas from the Spot benchmarks [6], as well as (iii) the 2 counter formula families from Table 3.1 and their conjunction. The results are summarized in Table 3.2 and Figures 3.2 and 3.3 (memory consumption generally closely follows running time in all cases). It can be seen that the proposed approach is comparable with existing ones for formulas of small size. Moreover, by guaranteeing that intermediate results do not explode in size, it outperforms existing approaches in long instances of the property families in Table 3.1, by orders of magnitude. In addition, unlike existing approaches, it can take into account a priori information about the target automaton, which leads to even better performance.

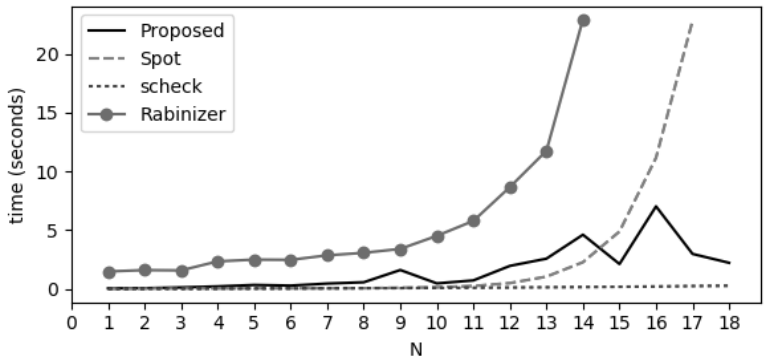
Table 3.2. Execution times (in seconds) for 500 random and 54 Spot formulas

Algorithm	500 random formulas		54 Spot formulas	
	Average	Median	Average	Median
Proposed	0.0693	0.0457	0.1262	0.0545
Spot	0.0397	0.0373	0.0406	0.0401
scheck	0.0082	0.0065	0.0161	0.0072
Rabinizer	1.4821	1.3668	1.8128	1.6885

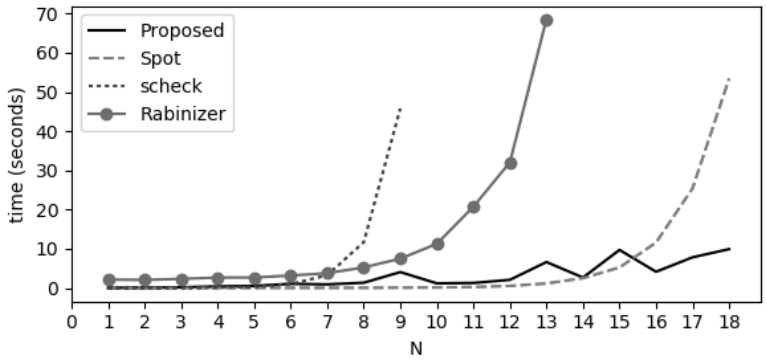
[1] These formulas, in particular, come from industrial requirements for aerospace domain digital hardware verification, a domain where formulas of this kind with many (typically > 50) nested next operators, expressing timing requirements for FPGAs, appear quite frequently.



(a) Counter family A

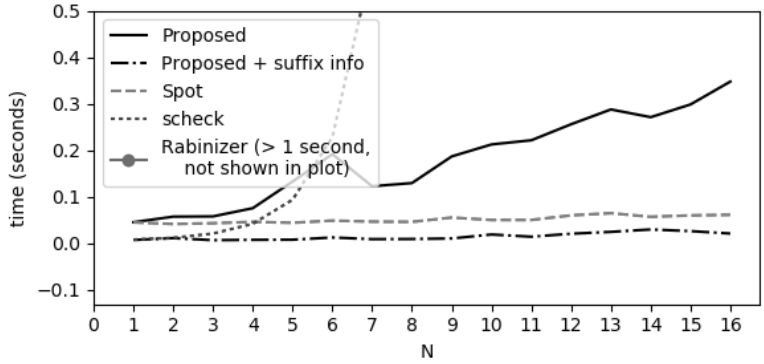


(b) Counter family B

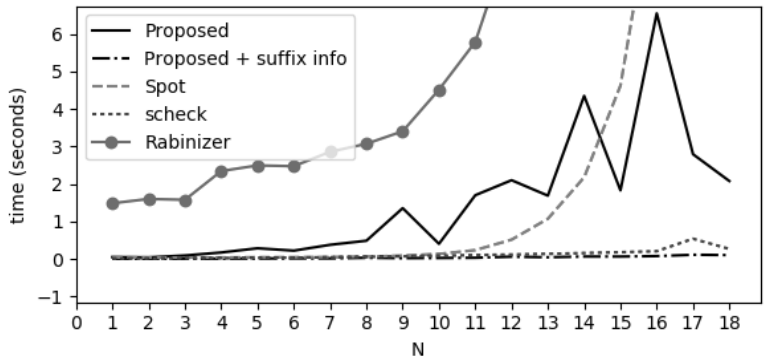


(c) Counter family conjunction

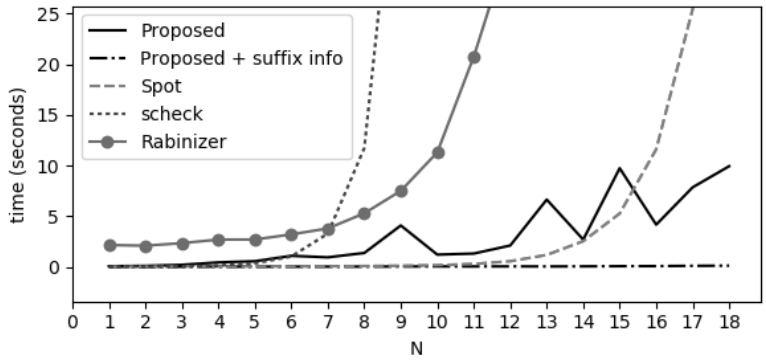
Figure 3.2. Results on counter formulas



(a) Counter family A



(b) Counter family B



(c) Counter family conjunction

Figure 3.3. Effect of suffix information on counter formulas

4. Model Learning

In the area of system design, and in particular within a DMD context, an important problem is automatically obtaining models from data [109, 107]. Depending on the type of models to be learned, as well as the provided input data and other assumptions or constraints, several variants of this problem exist. For example, there is the classic field of system identification [68], but also more recent works on generating programs, controllers, or other artifacts from examples [101, 50, 96, 16, 18, 107]. The motivation and objectives for this type of work include, but are not limited to, reduction of human effort in model creation, which in turn can reduce design errors and accelerate iteration times, as well as, at the same time, harness the abundance of available data being constantly generated by (potentially safety-critical) systems in an efficient way, in order to enable kinds of analyses not possible otherwise [76]. Another potential application for model generation from data is system reimplementation, particularly in cases where we have undocumented, essentially black-box, legacy systems not built with best MBSE practices in mind and, as a result, are difficult to modify and extend. In such a context, a first step could be employing model learning approaches able to also take into account requirements the learned model should satisfy, and use them to generate abstract models that (a) faithfully capture the interface between the various system components, as well as between the system and its environment, and (b) satisfy the desired requirements by construction.

Model learning from examples has been studied for several types of state machines, including DFA, Mealy machines, probabilistic automata, register automata, extended Mealy machines and subsequential transducers. Related work in this area can be classified into active learning, i.e., learning from (examples and) queries [19, 97, 60, 30, 10, 58, 9] and passive learning, i.e., learning only from examples. In the latter category we can also distinguish between exact approaches, which learn the smallest machine, w.r.t. number of states [56, 108] and heuristic approaches, which do not necessarily learn the smallest machine [49, 81, 42, 64, 82, 26, 111, 113, 102, 11, 28, 74, 110].

In this thesis, we focus on the problem of learning deterministic and complete Moore machines, from input-output traces. Despite this being a basic problem, it appears to not have received a lot of attention in the literature so far, however it is nevertheless worth studying as such state machines have many applications; for example, they can be used to represent digital circuits and controllers. In addition, the algorithms we propose can be used as building blocks for learning more complex types of models, such as hybrid automata [73]. The authors of [73], in particular, employ an active Mealy machine learning algorithm but adapt it to operate on a passive learning setting (i.e. only by examining the provided traces) and also postprocess the learned machine in order to ensure that no state has multiple incoming edges that produce different outputs. These modifications together imply that a passive learning approach that learns Moore machines, such as the one we provide here, would be a much better fit for this purpose.

Specifically, in our work, which is situated in the heuristic approach subcategory of the passive learning area, we formally define the problem of learning Moore machines from input-output traces, develop three algorithms, MooreMI, PRPNI and PTAP, that solve the problem, study their theoretical properties and compare them through experimental evaluation. In addition, we adapt MooreMI, our best algorithm, to learn Mealy machines and conduct a performance comparison against LearnLib [88] and flexfringe [112].

The input to all three algorithms we propose is a set of input-output traces, each trace being a pair of an input word and an output word, and each word being a finite sequence of symbols. The output of all three algorithms is a deterministic and complete Moore machine. An overview of our MooreMI algorithm is shown in Figure 4.1. The algorithm consists of two main phases, much like the RPNI [81] algorithm for learning DFA, of which it is a natural extension. Initially, the provided set of traces is converted into a more compact, tree based representation, called the Prefix Tree Acceptor (PTA). Subsequently, an iterative merging phase follows where nodes / states of the PTA compatible with each other are merged together in order to reduce the number of states in the learned state machine. Our PRPNI algorithm directly uses the RPNI algorithm as a building block, by decomposing the given input-output traces into $N = \lceil log_2|O| \rceil$ (where O is the set of distinct output symbols appearing in the traces) sets of positive and negative examples (which can be used as input for RPNI), invoking RPNI N times, and then computing and completing the product of the N learned DFA in order to obtain the learned Moore machine. Finally, our PTAP algorithm, being the simplest of the three approaches, simply computes the prefix tree acceptor, completes it and returns it as the learned Moore machine.

All three algorithms exhibit polynomial complexity w.r.t. to the total

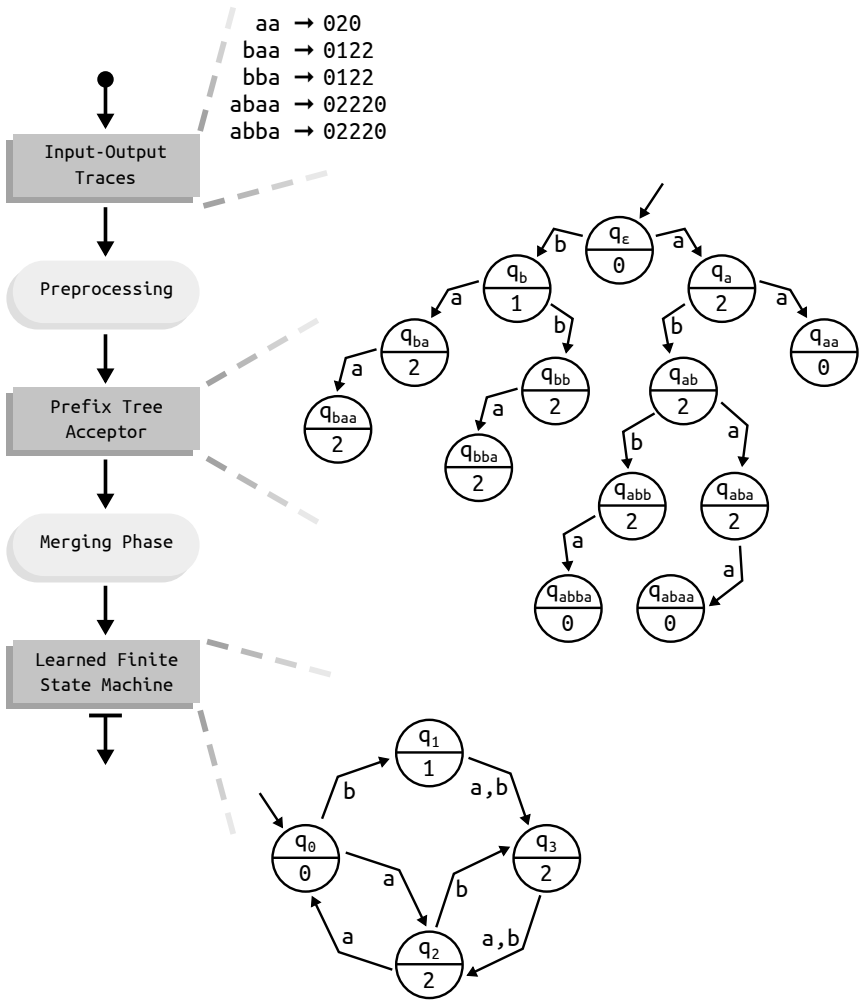


Figure 4.1. FSM learning algorithm

symbol length of the training set (input-output traces), and are guaranteed to return machines consistent with the training set, meaning that when fed with an input word from any of the training traces, they will return the corresponding output word. Our MooreMI algorithm also has the identification in the limit property [48]. This ensures that the algorithm will eventually learn the correct machine when provided with a sufficiently large set of examples. In our case, we also formally define 'sufficiently large' by extending the notion of characteristic sample, which is known for DFA [39], in the context of Moore machines. Experimental evaluation shows that MooreMI is superior to PTAP and PRPNI not only in theory, but also in practice, as shown in Tables 4.2, 4.3, 4.4, 4.5 (a dash indicates timeout). In particular, one can observe that MooreMI outperforms the

other two algorithms in terms of execution time, number of states in the learned machine, as well as three notions of accuracy we introduce in this thesis. Finally, our MealyMI algorithm (adaptation of MooreMI to learn Mealy machines) outperforms LearnLib [88] and flexfringe [112] in both execution time and memory consumption, as shown in Table 4.1.

Table 4.1. Performance comparison results with existing tools that learn Mealy machines.

Tool	Time (s)			Peak Memory Usage (GB)
	Parsing	Learning	Total	
LearnLib	3.851	7.143	11.994	1.8
flexfringe	13.806	181.032	194.838	2.8
MealyMI	3.062	2.891	5.953	1.1

Table 4.2. avg training set size: 140.9 (50 states), 109.0 (150 states), avg input word len: 8.0513 (50 states), 10.0227 (150 states)

Algorithm		50 states					150 states				
		Time (s)	States	Accuracy (%)			Time (s)	States	Accuracy (%)		
				Strong	Medium	Weak			Strong	Medium	Weak
PTAP	avg	0.0059	1000	0.031	25.614	28.785	0.0067	1000	0.04	20.18	23.339
	mdn	0.0058	1000	0.03	25.545	28.765	0.0062	1000	0.04	20.265	23.43
	sdv	0.0008	0	0.003	0.2731	0.3421	0.001	0	0	0.2297	0.276
PRPNI	avg	—	—	—	—	—	—	—	—	—	—
	mdn	—	—	—	—	—	—	—	—	—	—
	sdv	—	—	—	—	—	—	—	—	—	—
MooreMI	avg	0.0218	65.9	0.534	31.938	35.374	0.0277	93.3	0.04	21.158	24.408
	mdn	0.0199	65.5	0.515	31.885	35.42	0.0273	92	0.04	21.24	24.475
	sdv	0.0035	2.8089	0.0684	0.4904	0.408	0.0024	5.1391	0	0.2906	0.3032

Table 4.3. avg training set size: 1594.4 (50 states), 1184.7 (150 states), avg input word len: 8.0028 (50 states), 10.0325 (150 states)

Algorithm		50 states					150 states				
		Time (s)	States	Accuracy (%)			Time (s)	States	Accuracy (%)		
				Strong	Medium	Weak			Strong	Medium	Weak
PTAP	avg	0.0752	10000	0.371	34.737	37.492	0.0688	10000	0.399	27.547	30.413
	mdn	0.0701	10000	0.37	34.705	37.49	0.0678	10000	0.4	27.585	30.41
	sdv	0.0146	0	0.003	0.0986	0.1179	0.0031	0	0.003	0.1116	0.1341
PRPNI	avg	—	—	—	—	—	—	—	—	—	—
	mdn	—	—	—	—	—	—	—	—	—	—
	sdv	—	—	—	—	—	—	—	—	—	—
MooreMI	avg	0.1911	125.5	51.989	79.065	80.207	1.1478	354.2	0.489	31.123	34.16
	mdn	0.1825	126	52.95	79.635	80.71	1.1425	352	0.49	31.145	34.16
	sdv	0.0443	13.025	9.1848	4.5481	4.2777	0.051	5.2498	0.0094	0.304	0.311

Table 4.4. avg training set size: 18104.9 (50 states), 13019.5 (150 states), avg input word len: 8.0061 (50 states), 10.0076 (150 states)

Algorithm		50 states					150 states				
		Time (s)	States	Accuracy (%)			Time (s)	States	Accuracy (%)		
				Strong	Medium	Weak			Strong	Medium	Weak
PTAP	avg	0.8065	100000	4.131	45.378	47.605	0.7858	100000	4.366	36.522	39.03
	mdn	0.755	100000	4.13	45.385	47.64	0.7801	100000	4.36	36.555	39.01
	sdv	0.1354	0	0.0104	0.0935	0.1763	0.0342	0	0.0162	0.1211	0.1621
PRPNI	avg	3.5585	24651.7	98.637	99.562	99.683	—	—	—	—	—
	mdn	2.2394	3073	98.88	99.66	99.745	—	—	—	—	—
	sdv	3.9425	68215.5	1.4605	0.4823	0.3457	—	—	—	—	—
MooreMI	avg	0.3631	50	100	100	100	1.1815	220.4	95.923	98.439	98.508
	mdn	0.3622	50	100	100	100	1.0768	223.5	95.84	98.4	98.47
	sdv	0.0144	0	0	0	0	0.3627	34.1532	2.0841	0.7941	0.76

Table 4.5. avg training set size: 210700.0 (50 states), 144881.0 (150 states), avg input word len: 8.0059 (50 states), 9.9993 (150 states)

Algorithm		50 states					150 states				
		Time (s)	States	Accuracy (%)			Time (s)	States	Accuracy (%)		
				Strong	Medium	Weak			Strong	Medium	Weak
PTAP	avg	10.2782	1000000	47.558	74.448	75.448	10.9528	1000000	48.463	69.195	70.392
	mdn	9.9208	1000000	47.55	74.445	75.44	10.7495	1000000	48.46	69.195	70.4
	sdv	1.8331	0	0.0352	0.0655	0.0953	2.4395	0	0.0215	0.0385	0.0673
PRPNI	avg	27.8298	50	100	100	100	30.8077	11420	99.941	99.98	99.987
	mdn	27.5391	50	100	100	100	29.7683	150	100	100	100
	sdv	3.3386	0	0	0	0	3.819	13846	0.0779	0.0261	0.0168
MooreMI	avg	3.5939	50	100	100	100	4.2064	150	100	100	100
	mdn	3.5039	50	100	100	100	4.1011	150	100	100	100
	sdv	0.2197	0	0	0	0	0.2373	0	0	0	0

5. Reachability Analysis

Iterative matrix algorithms are fundamental components in many real-time control systems and, as such, have been studied extensively by control and applied mathematicians [83, 84], as well as embedded systems engineers [62]. Such components can be part of safety-critical systems (e.g. in avionics), which explains the interest in development and application of relevant V&V approaches [92, 25]. In this thesis, we present such an approach and demonstrate its application on the Schulz generalized matrix inversion algorithm as well as the discrete time matrix algebraic Riccati equation, both of which are fundamental building blocks in several approaches for optimization and control [27, 67, 98]. In particular, we are interested in performing reachability analysis for these algorithms in order to determine number of steps required for convergence given an initial matrix range. We do so by treating the algorithms as (discrete time) dynamical systems (or equivalently, hybrid systems with trivial dynamics where the actual computation takes place on mode transitions) and employing a reachability analysis framework we develop to handle such systems implemented in C++ code.

While there is no shortage of state set representations and corresponding propagation algorithms for identification of reachable states [13], one major characteristic of such approaches that hinders adoption in industry is that they require formal models (e.g. hybrid automata [14, 54]) of the system at hand to operate on. In particular, translation of the system model to an appropriate representation introduces an additional step in the verification process and concerns about preservation of semantics, which makes it more difficult to convince certification authorities to accept the results of the approach as evidence for system safety. For example, the translation step might involve some sort of abstraction that may not be wanted; in the case of dynamical systems, in particular, it may abstract away the specific method used for solving of the involved ODEs (e.g. Runge-Kutta method), which widens the gap between the model being verified and the actual implementation. The alternative approach of C++ code instrumentation we propose here addresses these concerns, since it is able to operate on the

same level of abstraction as the final system implementation.

Approaches involving code instrumentation for checking behavior correctness have been explored before, but the focus there is typically in test case generation [95, 22]. To the best of our knowledge, the work closest to ours is [117]. They develop an Affine arithmetic [103] based framework for instrumentation of SystemC code towards reachability analysis, but the focus is on extending Affine arithmetic to be able to handle hybrid behavior (i.e. including mode switching), while we focus on dynamical systems (in particular, matrix iterative algorithms) and use Affine arithmetic as a building block (without extending it). Specifically, we develop a framework for C++ code instrumentation towards reachability analysis of dynamical systems and, in particular, matrix iterative algorithms, by providing matrix data types and associated operations (e.g. multiplication, inversion, determinant and norm computation etc.), convergence criteria for the two algorithms we study (Schulz matrix inversion and discrete time algebraic Riccati equation), as well as an adaptive domain subdivision procedure together with two domain splitting heuristics.

In implementing an instrumentation framework for reachability analysis, we distinguish two key components, in general: First, a state set representation and associated propagation algorithm implemented in the language of choice (C++ in our case). Second, domain-specific utility data structures and procedures that facilitate minimally intrusive instrumentation of the system implementation and corresponding simulation / integration scheme (e.g. Runge-Kutta method) to enable reachability analysis. In principle, any state set representation and corresponding propagation algorithm can be used but, to keep things simple in our initial implementation, we opted for an Affine arithmetic [103] solution, since C++ libraries for it already exist [8] and, by virtue of making it easy to maintain a reachable set for each state variable, also simplifies the instrumentation step.

The bulk of the work in our implementation was building the instrumentation infrastructure. Since the initial application of the framework was iterative matrix algorithms viewed as dynamical systems, appropriate matrix data types had to be defined, supporting all relevant operations in safe (i.e. conservative w.r.t. reachable state set computation) ways. In addition to that, we had to provide safe implementations for a few domain specific bound computations (that served as stopping / convergence criteria for the iterative matrix algorithms under analysis), as well as an adaptive domain subdivision scheme (Figure 5.1), along with two associated domain splitting heuristics (Figure 5.2), in order to partially counteract the conservativeness of Affine arithmetic and provide tighter (but still safe) analysis results. In particular, in the context of the two iterative matrix algorithms we studied, there were cases where, given the same initial matrix range, the algorithm would diverge without domain subdivision, but converge when subdivision was performed (Figure 5.1). In addition,

no splitting heuristic of the two we tried was strictly better than the other – there were problem instances where the first performed better (smaller total number of subdivisions, as well as shorter execution time) and other problem instances where the second performed better (Figure 5.2).

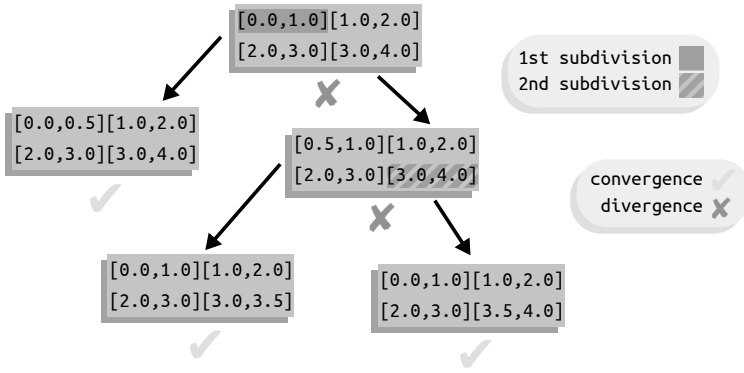


Figure 5.1. Adaptive domain subdivision scheme – when the iterative algorithm diverges, subdivide the domain and rerun on the resulting matrices

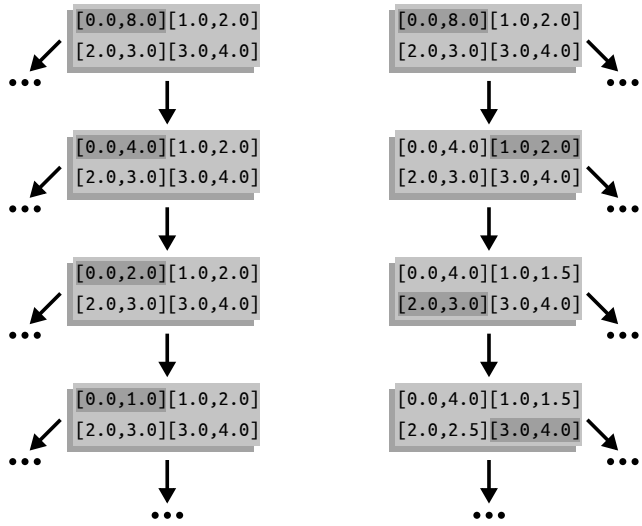


Figure 5.2. Domain splitting heuristics – always subdivide wider matrix element (left) vs subdivide all matrix elements in order (right)

6. Conclusion and Perspectives

In this thesis, we present a solution towards aiding re-engineering of legacy systems using model based design best practices through contributions in the areas of requirements formalization, automated monitor generation, model learning from examples, and practical reachability analysis for system implementations.

6.1 Requirements Formalization

The developed workflow for requirements formalization leverages NLP and ML methods for pattern identification from legacy requirements, which in turn accelerates formalization of both legacy and new requirements. A variety of formal languages is supported and, once requirements are formalized, consistency checking and automatic monitor generation can be performed as well. The approach has been tested on industrial case studies with several hundreds of requirements in each case and the results have been very promising.

One limitation here is that the approach currently only focuses on functional requirements (i.e. system behavior). Therefore, a direction worth exploring in the future is handling non-functional requirements as well (e.g. timing and architectural constraints). Another interesting direction for future development would be extending the tool with more specification languages and monitor generation targets in order to enable further interoperability with other tools and ease adoption from industrial users.

6.2 Monitor Generation

The developed approach for monitor generation of safety LTL properties is comparable performance-wise with existing ones for formulas of small size. Moreover, by providing theoretical guarantees (through leveraging of an active automata learning technique) that intermediate results do

not explode in size, it outperforms the state of the art in translation times for certain property families, by orders of magnitude. In addition, unlike implementations of existing approaches, it can take into account a-priori information about the target automaton, which leads to even better performance.

Interesting directions for future work here include using more optimized versions of the underlying learning algorithm, employing incremental model checking approaches for equivalence queries, as well as extending the work to translation of general (not just safety) LTL properties to Büchi automata.

6.3 Model Learning

The developed algorithm for finite state machine learning from examples has desirable theoretical properties (it converges to the 'correct' machine if given 'enough' example traces) as well as competitive performance compared to existing approaches.

Apart from further experimentation w.r.t. learning various types of black-box systems, an interesting direction to explore in the future would be extending the algorithm to also take into account requirements the learned machine should satisfy, by employing e.g. a CEGIS [59] outer loop.

6.4 Reachability Analysis

The developed framework for reachability analysis of dynamical system implementations has been successfully applied on analysis of iterative matrix algorithms, enabling derivation of convergence related results that were not possible through analytical (i.e. closed form) means.

In the context of iterative matrix algorithms, exploring different domain splitting heuristics would be an interesting direction for future work. In a more general view, we believe it would be worth exploring integration with different reachability analysis algorithms, as well as extension of the work to be able to handle hybrid systems too.

References

[1] A Computational Logic for Applicative Common Lisp (ACL2). https://www.cs.utexas.edu/users/moore/acl2/. Accessed: 2021-06-01.

[2] Coq formal proof management system. https://coq.inria.fr/. Accessed: 2021-06-01.

[3] FRET: Formal Requirements Elicitation Tool. https://github.com/NASA-SW-VnV/fret/. Accessed: 2022-08-08.

[4] Isabelle generic proof assistant. https://isabelle.in.tum.de/. Accessed: 2021-06-01.

[5] Prototype Verification System (PVS). https://pvs.csl.sri.com/. Accessed: 2021-06-01.

[6] Spot 1.0 benchmarks. https://www.lrde.epita.fr/~adl/ijccbs/.

[7] IEEE Standard for Property Specification Language (PSL). *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pages 1–182, 2010.

[8] aaflib - An Affine Arithmetic C++ Library. http://aaflib.sourceforge.net, 2019. [Online; accessed 9-August-2022].

[9] Fides Aarts, Paul Fiterau-Brostean, Harco Kuppens, and Frits W. Vaandrager. Learning Register Automata with Fresh Value Generation. In *Theoretical Aspects of Computing - ICTAC*, volume 9399 of *LNCS*, pages 165–183, 2015.

[10] Fides Aarts and Frits Vaandrager. Learning I/O Automata. In *CONCUR*, pages 71–85. Springer, 2010.

[11] A. V. Aleksandrov, S. V. Kazakov, A. A. Sergushichev, F. N. Tsarev, and A. A. Shalyto. The use of evolutionary programming based on training examples for the generation of finite state machines for controlling objects with complex behavior. *J. Comput. Sys. Sc. Int.*, 52(3):410–425, 2013.

[12] Alexandre Duret-Lutz and Alexandre Lewkowicz and Amaury Fauchille and Thibaud Michaud and Etienne Renault and Laurent Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129. Springer, oct 2016.

[13] Matthias Althoff, Goran Frehse, and Antoine Girard. Set Propagation Techniques for Reachability Analysis. *Annual Review of Control, Robotics, and Autonomous Systems*, 4, 05 2021.

[14] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995. Hybrid Systems.

[15] Rajeev Alur. *Principles of Cyber-Physical Systems*. The MIT Press, 2015.

[16] Rajeev Alur, Milo M. K. Martin, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. Synthesizing Finite-State Protocols from Scenarios and Requirements. In Eran Yahav, editor, *Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings*, volume 8855 of *Lecture Notes in Computer Science*, pages 75–91. Springer, 2014.

[17] Rajeev Alur, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. Automatic Completion of Distributed Protocols with Symmetry. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 395–412. Springer, 2015.

[18] Rajeev Alur and Stavros Tripakis. Automatic Synthesis of Distributed Protocols. *SIGACT News*, 48(1):55–90, mar 2017.

[19] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[20] Tomás Babiak, Mojmír Kretínský, Vojtech Rehák, and Jan Strejcek. LTL to Büchi Automata Translation: Fast and More Deterministic. *CoRR*, abs/1201.0682, 2012.

[21] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Integrated Formal Methods*, pages 1–20, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[22] Thomas Ball and Jakub Daniel. Deconstructing Dynamic Symbolic Execution. In *Dependable Software Systems Engineering*, 2015.

[23] Alessio Balsini, Marco Di Natale, Marco Celia, and Vassilios Tsachouridis. Generation of Simulink monitors for control applications from formal requirements. In *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–9. IEEE, 2017.

[24] Andreas Bauer, Martin Leucker, and Jonathan Streit. SALT—Structured Assertion Language for Temporal Logic. volume 4260, pages 757–775, 11 2006.

[25] Christine M. Belcastro. *Validation and Verification (V&V) of Safety-Critical Systems Operating under Off-Nominal Conditions*, pages 399–419. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[26] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. Inferring Models of Concurrent Systems from Logs of Their Behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 468–479, New York, NY, USA, 2014. ACM.

[27] D. Boley and B. N. Datta. Numerical Methods for Linear Control Systems. In Christopher I. Byrnes, Biswa N. Datta, Clyde F. Martin, and David S. Gilliam, editors, *Systems and Control in the Twenty-First Century*, pages 51–74, Boston, MA, 1997. Birkhäuser Boston.

[28] I. P. Buzhinsky, V. I. Ulyantsev, D. S. Chivilikhin, and A. A. Shalyto. Inducing finite state machines from training samples using ant colony optimization. *J. Comput. Sys. Sc. Int.*, 53(2):256–266, 2014.

[29] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving Fast with Software Verification. In Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 3–11, Cham, 2015. Springer International Publishing.

[30] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Learning Extended Finite State Machines. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Software Engineering and Formal Methods*, pages 250–264, Cham, 2014. Springer International Publishing.

[31] Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. Code-Level Model Checking in the Software Development Workflow. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '20, page 11–20, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, pages 359–364, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[33] E. Clarke, Armin Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19:7–34, 2001.

[34] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite State Concurrent System Using Temporal Logic Specifications: A Practical Approach. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, page 117–126, New York, NY, USA, 1983. Association for Computing Machinery.

[35] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[36] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of Model Checking*. Springer Publishing Company, Incorporated, 1st edition, 2018.

[37] Loris D'Antoni and Margus Veanes. The Power of Symbolic Automata and Transducers. In *CAV*, 2017.

[38] Ashish Darbari. A Brief History of Formal Verification. `https://www.eeweb.com/a-brief-history-of-formal-verification/`, 2019. Accessed: 2021-06-01.

[39] Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, New York, NY, USA, 2010.

[40] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, 4963:337–340, 04 2008.

[41] Simulink Documentation. Simulation and Model-Based Design, 2020.

[42] Pierre Dupont. Incremental regular inference. In *ICGI-96*, pages 222–237, 1996.

[43] Aaron Fifarek, Lucas Wagner, Jonathan Hoffman, Benjamin Rodes, M. Aiello, and Jennifer Davis. SpeAR v2.0: Formalized Past LTL Specification and Analysis of Requirements. In *NASA Formal Methods*, pages 420–426, 04 2017.

[44] Amit Fisher, Clas A. Jacobson, Edward A. Lee, Richard M. Murray, Alberto Sangiovanni-Vincentelli, and Eelco Scholte. Industrial Cyber-Physical Systems – iCyPhy. In Marc Aiguier, Frédéric Boulanger, Daniel Krob, and Clotilde Marchal, editors, *Complex Systems Design & Management*, pages 21–37, Cham, 2014. Springer International Publishing.

[45] Arthur Flatau, Matt Kaufmann, David Reed, David Russinoff, Eric Smith, and Rob Sumners. Formal Verification of Microprocessors at AMD. 01 2002.

[46] Paul Gastin and Denis Oddoux. Fast LTL to Büchi Automata Translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, pages 53–65, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[47] Nicola Gigante, Angelo Montanari, and Mark Reynolds. A One-Pass Tree-Shaped Tableau for LTL+Past. In Thomas Eiter and David Sands, editors, *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 46 of *EPiC Series in Computing*, pages 456–473. EasyChair, 2017.

[48] E. Mark Gold. Language Identification in the Limit. *Information and Control*, 10(5):447–474, 1967.

[49] E. Mark Gold. Complexity of Automaton Identification from Given Data. *Information and Control*, 37(3):302–320, 1978.

[50] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *38th POPL*, pages 317–330, 2011.

[51] Reiner Hähnle and Marieke Huisman. *Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools*, pages 345–373. Springer International Publishing, Cham, 2019.

[52] John Harrison. Formal Verification at Intel. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, LICS '03, page 45, USA, 2003. IEEE Computer Society.

[53] He, Jie and Bartocci, Ezio and Ničković, Dejan and Isakovic, Haris and Grosu, Radu. DeepSTL - From English Requirements to Signal Temporal Logic. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 610–622, 2022.

[54] Thomas A. Henzinger. *The Theory of Hybrid Automata*, pages 265–292. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[55] Thomas A. Henzinger and Joseph Sifakis. The Discipline of Embedded Systems Design. *Computer*, 40(10):32–40, 2007.

[56] Marijn J. Heule and Sicco Verwer. Software Model Synthesis Using Satisfiability Solvers. *Empirical Softw. Engg.*, 18(4):825–856, August 2013.

[57] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. spaCy: Industrial-strength Natural Language Processing in Python. https://doi.org/10.5281/zenodo.1212303, 2020.

[58] Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring Canonical Register Automata. In *VMCAI 2012, Proceedings*, pages 251–266, 2012.

[59] Susmit Jha and Sanjit A. Seshia. A Theory of Formal Synthesis via Inductive Learning. *Acta Inf.*, 54(7):693–726, November 2017.

[60] Bengt Jonsson. Learning of Automata Models Extended with Data. In *SFM 2011, Advanced Lectures*, pages 327–349, 2011.

[61] Jan Křetínský, Tobias Meggendorfer, Salomon Sickert, and Christopher Ziegler. Rabinizer 4: From LTL to Your Favourite Deterministic Automaton. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 567–577, Cham, 2018. Springer International Publishing.

[62] Gurrala Ajay Kumar, Thadigotla Venkata Subbareddy, Bommepalli Madhava Reddy, N Raju, and V Elamaran. An approach to design a matrix inversion hardware module using FPGA. In *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, pages 87–90, 2014.

[63] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 585–591, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[64] Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm. In *ICGI-98*, pages 1–12, 1998.

[65] Timo Latvala. Efficient Model Checking of Safety Properties. In Thomas Ball and Sriram K. Rajamani, editors, *Model Checking Software*, pages 74–88, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[66] Insup Lee, Joseph Y-T. Leung, and Sang H. Son. *Handbook of Real-Time and Embedded Systems*. Chapman amp; Hall/CRC, 1st edition, 2007.

[67] W.S. Levine. *The Control Handbook*. CRC Press, 2011.

[68] Lennart Ljung. *System Identification: Theory for the User*. Prentice Hall, 2 edition, 1999.

[69] Lennart Ljung. *System Identification: An Overview*, pages 1–20. 01 2014.

[70] David C. Luckham, Steven M. German, Friedrich W. von Henke, Richard A. Karp, P. W. Milne, Derek C. Oppen, Wolfgang Polak, and William L. Scherlis. Stanford Pascal Verifier User Manual. Technical report, Stanford, CA, USA, 1979.

[71] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. The Stanford CoreNLP Natural Language Processing Toolkit. 01 2014.

[72] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, USA, 1993.

[73] Ramy Medhat, S. Ramesh, Borzoo Bonakdarpour, and Sebastian Fischmeister. A framework for mining hybrid automata from input/output traces. In *2015 International Conference on Embedded Software (EMSOFT)*, pages 177–186, 10 2015.

[74] Karl Meinke. CGE: A Sequential Learning Algorithm for Mealy Automata. In *Grammatical Inference: Theoretical Results and Applications, 10th International Colloquium, ICGI 2010, Valencia, Spain, September 13-16, 2010. Proceedings*, pages 148–162, 2010.

[75] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., USA, 1 edition, 1997.

[76] Elisa Negri, Luca Fumagalli, and Marco Macchi. A Review of the Roles of Digital Twin in CPS-based Production Systems. *Procedia Manufacturing*, 11:939–948, 12 2017.

[77] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Commun. ACM*, 58(4):66–73, mar 2015.

[78] Gabriela Nicolescu and Pieter Mosterman. *Model-Based Design for Embedded Systems*. 09 2018.

[79] Allen Nikora and Galen Balcom. Automated Identification of LTL Patterns in Natural Language Requirements. In *2009 20th International Symposium on Software Reliability Engineering*, 11 2009.

[80] Eugenio G. Omodeo and Alberto Policriti, editors. *Martin Davis on Computability, Computational Logic, and Mathematical Foundations*, volume 10 of *Outstanding Contributions to Logic*. Springer, 2016.

[81] Jose Oncina and Pedro Garcia. Identifying Regular Languages In Polynomial Time. In *Advances in Structural and Syntactic Pattern Recognition*, pages 99–108, 1992.

[82] José Oncina, Pedro García, and Enrique Vidal. Learning Subsequential Transducers for Pattern Recognition Interpretation Tasks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(5):448–458, 1993.

[83] P. Hr. Petkov, N. D. Christov, and M. M. Konstantinov. *Computational Methods for Linear Control Systems*. Prentice Hall International (UK) Ltd., GBR, 1991.

[84] Marko D. Petković and Predrag S. Stanimirović. Generalized Matrix Inversion is Not Harder than Matrix Multiplication. *J. Comput. Appl. Math.*, 230(1):270–282, aug 2009.

[85] André Platzer. Differential Dynamic Logic for Hybrid Systems. *J. Autom. Reason.*, 41(2):143–189, August 2008.

[86] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, 1977.

[87] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming*, pages 337–351, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[88] Harald Raffelt and Bernhard Steffen. LearnLib: A Library for Automata Learning and Experimentation. *FMICS'05 - Proceedings of the Tenth International Workshop on Formal Methods for Industrial Critical Systems*, 3922:377–380, 03 2006.

[89] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-End Verification of Processors with ISA-Formal. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 42–58, Cham, 2016. Springer International Publishing.

[90] Alastair Reid, Luke Church, Shaked Flur, Sarah de Haas, Maritza Johnson, and Ben Laurie. Towards making formal methods normal: meeting developers where they are, October 2020. Accepted at HATRA 2020.

[91] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009.

[92] T. Samad and A.M. Annaswamy. The Impact of Control Technology, 2nd Edition. http://ieeecss.org/impact-control-technology-2nd-edition, 2014. [Online; accessed 9-August-2022].

[93] Alberto Sangiovanni-Vincentelli and Marco Di Natale. Embedded system design for automotive applications. *Computer*, 40(10):42–51, 2007.

[94] Alberto Sangiovanni-Vincentelli, Haibo Zeng, Marco Di Natale, and Peter Marwedel. *Embedded systems development*. Springer, 2015.

[95] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. *SIGSOFT Software Engineering Notes*, 30:263–272, 09 2005.

[96] S. A. Seshia. Sciduction: Combining induction, deduction, and structure for verification and synthesis. In *DAC*, pages 356–365, June 2012.

[97] Muzammil Shahbaz and Roland Groz. Inferring Mealy Machines. In *FM 2009*, pages 207–222, 2009.

[98] V. Sima. *Algorithms for Linear-Quadratic Optimization*. Chapman and Hall/CRC, 1996.

[99] A Sindico, M Di Natale, and A Sangiovanni-Vincentelli. An industrial application of a system engineering process integrating model-driven architecture and model based design. In *ACM/IEEE 15th MODELS Conference, Innsbruck, Austria*, volume 10, pages 978–3, 2012.

[100] A. Prasad Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–511, Sep 1994.

[101] Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

[102] M. Spichakova. An approach to inference of finite state machines based on gravitationally-inspired search algorithm. *Proc. of Estonian Acad. of Sci.*, 62(1):39–46, 2013.

[103] Jorge Stolfi, L. FIGUEIREDO, and Estrada Dona. An Introduction to Affine Arithmetic. *TEMA. Tendências em Matemática Aplicada e Computacional*, 4, 12 2003.

[104] J. Sztipanovits and G. Karsai. Model-integrated computing. *Computer*, 30(4):110–111, 1997.

[105] Janos Sztipanovits, Xenofon Koutsoukos, Gabor Karsai, Nicholas Kottenstette, Panos Antsaklis, Vijay Gupta, Bill Goodwine, John Baras, and Shige Wang. Toward a Science of Cyber–Physical System Integration. *Proceedings of the IEEE*, 100(1):29–44, 2012.

[106] Stavros Tripakis. Compositionality in the Science of System Design. *Proceedings of the IEEE*, 104(5):960–972, 2016.

[107] Stavros Tripakis. Data-driven and model-based design. *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, pages 103–108, 2018.

[108] Vladimir Ulyantsev, Ilya Zakirzyanov, and Anatoly Shalyto. BFS-Based Symmetry Breaking Predicates for DFA Identification. In *Language and Automata Theory and Applications (LATA)*, volume 8977 of *LNCS*, pages 611–622. Springer, 2015.

[109] Frits Vaandrager. Model Learning. *Commun. ACM*, 60(2):86–95, January 2017.

[110] L. P. J. Veelenturf. Inference of Sequential Machines from Sample Computations. *IEEE Trans. Computers*, 27(2):167–170, 1978.

[111] Sicco Verwer, Mathijs de Weerdt, and Cees Witteveen. A Likelihood-Ratio Test for Identifying Probabilistic Deterministic Real-Time Automata from Positive Data. In José M. Sempere and Pedro García, editors, *Grammatical Inference: Theoretical Results and Applications*, pages 203–216, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[112] Sicco Verwer and Christian Hammerschmidt. flexfringe: A Passive Automaton Learning Package. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 638–642, 09 2017.

[113] Neil Walkinshaw, Ramsay Taylor, and John Derrick. Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, 03 2015.

[114] Wikipedia contributors. V-Model (software development) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=V-Model_(software_development)&oldid=1050067631, 2021. [Online; accessed 29-March-2022].

[115] Wikipedia contributors. Agile software development — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Agile_software_development&oldid=1078799901, 2022. [Online; accessed 29-March-2022].

[116] Wikipedia contributors. Waterfall model — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Waterfall_model&oldid=1075719146, 2022. [Online; accessed 29-March-2022].

[117] Carna Zivkovic and Christoph Grimm. Symbolic Simulation of SystemC AMS Without Yet Another Compiler. In *2018 Forum on Specification Design Languages (FDL)*, pages 5–16, 09 2018.

Aalto-DT 126/2023

ISBN 978-952-64-1386-0 (printed)
ISBN 978-952-64-1387-7 (pdf)
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)

Aalto University
School of Science
Department of Computer Science
www.aalto.fi

BUSINESS + ECONOMY

ART + DESIGN + ARCHITECTURE

SCIENCE + TECHNOLOGY

CROSSOVER

DOCTORAL THESES