The background of the cover is black. It features a grid of small white dots. A diagonal line of white squares runs from the top-left towards the bottom-right. The squares are more prominent in the lower right area, where they form a staircase-like pattern. The title is centered in the upper half in a large, white, sans-serif font.

Efficient Learning and Analysis of System Behavior

Jeroen Meijer

Efficient Learning and Analysis of System Behavior

Jeroen Meijer

EFFICIENT LEARNING AND ANALYSIS OF SYSTEM BEHAVIOR

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
prof.dr. T.T.M. Palstra,
on account of the decision of the Doctorate Board,
to be publicly defended
on Friday the 20th of September 2019 at 12:45 hours

by

Jeroen Johan Gerardus Meijer

born on the 6th of August 1989
in Oldenzaal, the Netherlands

This dissertation has been approved by:

prof.dr. J.C. van de Pol (supervisor)
prof.dr. M.I.A. Stoelinga (supervisor)

**DIGITAL SOCIETY
INSTITUTE**

DSI Ph.D. Thesis Series No. 19-015

Digital Society Institute
P.O. Box 217, 7500 AE
Enschede, the Netherlands

IPA Dissertation Series No. 2019-10

The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Stichting voor de Technische Wetenschappen

The work in this thesis was supported by the SUM-BAT (SUPersizing Model-BASed Testing) project, funded by the STW grant 13859.

ISBN: 978-90-365-4844-1

ISSN: 2589-7721 (DSI Ph.D. Thesis Series No. 19-015)

DOI: 10.3990/1.9789036548441

Available online at <https://doi.org/10.3990/1.9789036548441>

Typeset with L^AT_EX

Printed by Ipskamp Printing

Cover design by Irene Meijer

© 2019 Jeroen Meijer, the Netherlands. All rights reserved. No parts of this thesis may be reproduced, stored in a retrieval system or transmitted in any form or by any means without permission of the author. Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd, in enige vorm of op enige wijze, zonder voorafgaande schriftelijke toestemming van de auteur.

Graduation Committee:

Chairman/secretary:

prof.dr. J.N. Kok

Supervisors:

prof.dr. J.C. van de Pol

prof.dr. M.I.A. Stoelinga

Members:

dr.ir. J.F. Broenink

University of Twente, the Netherlands

prof.dr. F.M. Howar

Technical University of Dortmund, Germany

prof.dr. M. Huisman

University of Twente, the Netherlands

prof.dr. F. Kordon

Sorbonne Université, France

prof.dr. K. Meinke

KTH Royal Institute of Technology, Sweden

dr.ir. R.IJ. de Vries

Malvern Panalytical B.V., the Netherlands

In loving memory of my dear sister

Joyce

Acknowledgments

Writing these acknowledgments marks the end of some very interesting and rewarding years. I have learned a lot, met amazing people and visited many interesting places, such as: Corfu, Haifa, London, Torún, Zaragoza, Reykjavik, Minneapolis, Santa Barbara, Newport News, Halmstad and Limassol. I want to thank the people that have supported me on this journey.

Joyce, while I was at the registration desk for the 2018 NASA Formal Methods Symposium (to present the work that is now Chapter 3) in Newport News, I received a goody bag with only a single NASA pin. Knowing that you would love to have one as well, I went back to the registration desk and managed to obtain a second pin. No surprise, you loved it, and wore the pin to the many festivals that you went to. Often, when you found a person wearing a NASA shirt, you would go to them and show this pin. You would say that you got the pin from your brother, who works for NASA. In one such instance, Liquicity 2018, that we went to together, you even made a very good friend. One of the amazing things about your personality. However, me working for NASA was an exaggeration of course — but a clear indication you were proud of me and the things that I accomplished. When I submitted my thesis you were also equally certain that it would be a success, and not being able to tell you in person that I was accepted for my defense last week has been heartbreaking. I will miss you very much, especially at the defense in four weeks, but I know you are proud of me becoming a doctor.

Irene, I really like the things that we do together, especially when we go skiing. The thesis cover that you designed turned out great — I like it very much, and the same holds for the invitations, they turned out great too. I want to thank you for the effort you put into this, especially because I know these past few weeks have been extremely difficult for you too.

Mom and dad, you have supported this journey that already started when I was very young. For instance, you bought me a LEGO Mindstorms set that got me acquainted with programming, a skill that has become crucial for finishing my PhD. You also helped me to build an enormous Saturn V model, a rocket that got people to the moon, launched by the NASA, the very same organization at which

I presented two of my research papers (what are now Chapters 3 and 6). I think it is safe to conclude these two investments in particular have paid themselves off now.

Oma Meijer en oma Scholte Lubberink, ik ben erg blij dat ik de speciale dag van mijn verdediging mag delen met jullie en dat ik de namen van beide opa's op een titelpagina kan zetten.

I also want to thank my very good friends: Andrea, Dominik, Gijs, Jeffrey, Karel, Lamar, Marcel, Marleen, Martijn, Martine, Merle, Nicole, Niek, Oliver, Pia, Pien, Pim, Tess, Vincent, Wouter and Rohan, especially for the enormous support I got from you over the past few weeks. This made it much easier to wrap up my thesis and prepare for the defense. I really enjoy the time that we spend together, especially when we go to festivals. They have always been a welcome distraction while working on this thesis for the past few years. Whenever I was talking about the science I was doing too excitedly, you were always interested. I look forward to going to Mysteryland with many of you this Saturday and seeing you at my defense. Dennis, Henry, Mark and Tim, although the Boshok times are sadly over, I am happy we are still very good friends and we always have a great time together, for instance at Mark's bachelor party last Saturday. I also look forward to seeing you at my defense. Anne, I am really glad you want to take the pictures at my defense. I am sure they will turn out great. Also, I am extremely honored to be your best man at your wedding, a week after my defense. Your wedding day is also something I look forward to a lot.

Benny and Rianne, you were also always interested when I was talking about the research I was doing. I look forward to seeing you at the defense and to hopefully celebrate with a diner and drinks afterwards.

My next thanks goes out to my (former) roommates. Vincent, talking about a range of topics with you is always very relaxing and fun. When it comes to science, and in particular some algorithms I was thinking about, you were always very interested and often had some good insight for me. Your work on integrating Spot and LTSmin have helped me tremendously as well. With this integration we are very competitive in the MCC and RERS competition. Your concurrent UFSCC algorithm together with Spot's Büchi automata are the defaults in the LearnLib's model checking module. Implementing monitoring based on your integration of Spot and LTSmin was only a few hours of work for me, and very fruitful for the results in Chapters 3 and 4. I have no doubt you will be a tremendous asset to Thales. Alfons, although we have only been roommates for a very short time, we kept working together on LTSmin. You spent a lot of time writing and improving code for LTSmin after you finished your PhD. This has also been very helpful for the scientific competitions I competed in and my thesis in general. Many results obtained in this thesis would not have been possible without your work, especially

the parts related to LTL model checking. Marcus, I also had a lot of fun with you as my roommate. Your mathematical skills have been very helpful to me throughout the years. Going to London, Halmstad and Washington with you have been very joyous times, especially when you got me into the world of Magic. I truly think you are a great teacher at the UT and reading your thesis has been very helpful for writing mine. Stefano, whenever you brought in your home baked macarons you managed to get them eaten by the FMT people in no time. Your baking skills are legendary, as well as the social activities you organized, such as the Fast Moving Team, and your sketches. This made working for FMT group some very fun years. Tom, ever since I started my PhD you have been very helpful to me. I learned a lot from you, especially good programming and GIT practices, as well as good scientific methods for addressing problems I tried to solve. I liked working on papers with you, and admire your efficiency while working on those. Now we are not PhD roommates anymore; I am happy you came back to FMT as an assistant professor. It is always ‘gezellig’ with you around. Arnd, although you were not my supervisor, you certainly have the skills to be one. Whenever I talked to you about issues I ran into, you could very quickly understand the problem I was trying to solve and give some very good pointers. The road trip we did in Iceland was really fun, especially when I found out we had a mutual interest in collecting cave rocks. Freark, you are no doubt the best software engineer I know. You were very helpful during the C++ metaprogramming course in particular. I also really appreciate the time you took helping me when I was debugging LTSmin. It is always fun times guaranteed when we have a few beers and play Rocket League or Supreme Commander. Your skills in these games I can only dream of having. David, you also bring a good amount of ‘gezelligheid’ to room ZI-3126, as well as during the BOCOM in the rappa. Our conversations, while enjoying the tasty FMT beer, always covered interesting topics.

Also a great thanks to my supervisors. Jaco, we both started at the UT in 2007. Since then, you have been a major influence on the person I am now. I am very grateful for the huge amount time that you have spent teaching me things and supervising the work I have been doing. I already miss the times walking into your office with problems I faced, and leaving relieved, knowing that I would only have to implement their solution, that we often found quickly, in Java or C. Your guidance has been essential throughout the years. The meetings I had with you, and later the many Skype calls, because you moved to Denmark, have always been a great motivation to continue my PhD. Mariëlle, as my second supervisor you were especially helpful in showing me how to write my thesis in a way that people would actually read it. My thesis has greatly improved when you started proofreading it, and I am happy with the end result because of this. Your excellent guidance during my internship at PANalytical in 2012 was probably the main booster for starting my PhD.

Gijs and Stefan, we worked closely together at the beginning of my PhD. We worked together on the paper that is now Chapter 5. Being a fresh MSc graduate, I learned the necessary skills from you to successfully publish more papers. Your good work on LTSmin has also contributed greatly to the nice results obtained in this thesis.

Ida, I truly appreciate that your door was always open. You were always very helpful with all kinds of matters I needed help with. Your organizational skills are amazing.

I also want to thank all my other colleagues at the FMT group. I loved the social activities and outings we did together, and really enjoyed the 5,5 years I worked together with all of you.

Vince, I am happy you want to be at my defense as well. We had some amazing times in London. I appreciate you looked into LTSmin to support your thesis. If LTSmin could have competed against PetriDotNet in the MCC, I am sure we would see an interesting outcome.

I also want to thank all my committee members for reading my thesis, and I look forward to the defense of it.

Enschede
August 2019

Abstract

In this thesis, we present techniques for more efficient learning and analysis of system behavior. The first part covers novel algorithms and tooling for testing systems based on active automata learning and Linear-time Temporal Logic (LTL) model checking, also called Learning-Based Testing (LBT). Next, we provide an improved learning algorithm that is able to deal with huge alphabets. These are commonly seen in large-scale industrial systems where input symbols contain data parameters. In the second part we discuss improvements for analyzing formal system specifications. We start out by looking at separated read, write and copy dependencies for symbolic model checking to speed up the verification of these specifications. Then, we show that bandwidth reduction techniques, originally designed for sparse matrix solvers, are very capable at reducing the memory footprint of the specifications' symbolic state space. Implementations of the presented algorithms are subjected to case studies and rigorous experimentation with scientific software competitions. Possible future improvements to these algorithms are discussed as well.

Part 1: Learning-Based Testing

The first contribution involves a design and implementation for LBT in the LearnLib, which is thoroughly documented and made freely available in the public domain. The LearnLib is a library that can be used to automatically infer the behavior of systems by means of active automata learning algorithms. By applying our implementations to the Rigorous Examination of Reactive Systems (RERS) verification competition we show how well learning algorithms such as L^* in the LearnLib perform in the context of LBT, which was previously unknown. An investigation into considering both the safety and liveness aspect of LTL properties separately is also done. We show that on the one hand, confirming counterexamples to safety properties on a running system is more straightforward than confirming lasso-shaped counterexamples to liveness properties. On the other hand, falsifying liveness properties seems to be more informative to learning algorithms than falsifying safety properties.

The second major contribution to LBT is a new learning algorithm that extends regular learning algorithms for finite state machines with automated refinement for partitioned alphabets. The key feature of our algorithm is a procedure that checks whether the partitioned alphabet, or the learned hypothesis automaton needs to be refined. The decision of the procedure is based on resolving non-determinism that may arise if the alphabet partitioning is too coarse. We subject the new algorithm again to the RERS challenge, in order to give software testers insight into the performance of learning algorithms available in the LearnLib: the ADT algorithm seems to be a good choice in general.

Part 2: Symbolic Model Checking

The first contribution to symbolic model checking exploits the separated read, write and copy dependencies between variables and expressions over them. By carefully looking at which variables are read, written or copied in assignments, we can reduce the number of NEXT-STATE function calls required to compute the state space of a formal system specification. The new NEXT-STATE function, an interface between a back-end storing the state space, and a language front-end, shows a reduced runtime for analyzing specifications written in the mCRL2 language in particular. We also show that analyzing Petri nets taken from the software competition called the Model Checking Contest (MCC) becomes more tractable with our improvements.

The second contribution to symbolic model checking, based on bandwidth reduction, relies on symmetrizing the matrix that encodes the dependency relation between variables and assignments in system specifications. We discuss several means to obtain a symmetric dependency matrix, as well as a method for de-symmetrizing the matrix. The latter method is necessary for obtaining separate optimized orders for both variables and assignments. Decades old bandwidth reduction algorithms can be run on symmetric dependency matrices, including Cuthill-McKee's and Sloan's algorithms. By experimenting with the MCC, we find that running Sloan's algorithm usually produces the best results; at least on par with the FORCE algorithm, which is the current state-of-the-art.

The improvements are implemented in the LTSmin model checker, and made available freely in the public domain. By interfacing LTSmin with the ProB model checker we show that specifications written in the B-method can now be verified more efficiently.

Contents

Acknowledgments	vii
Abstract	xi
Contents	xiii
List of Lists	xvii
List of Algorithms	xvii
List of Code Listings	xvii
List of Definitions	xvii
List of Examples	xix
List of Figures	xx
List of Tables	xxi
1 Introduction	1
1.1 Verification of Safety Critical Systems	2
1.2 Validation of Safety Critical Systems	2
1.3 Formal Methods for Uncovering Faults	3
1.3.1 Modeling	5
1.3.2 Model Checking	5
1.3.3 Model-Based Testing	6
1.3.4 Learning	7
1.3.5 Learning-Based Testing	9
1.4 Contribution to Learning-Based Testing	10
1.5 Contribution to Model Checking	12
1.6 The Value of Verification Competitions	15
1.6.1 Model Checking Contest (MCC)	15
1.6.2 Rigorous Examination of Reactive Systems (RERS)	16
1.7 Thesis Overview	17
2 Learning and Analysis from a Language Perspective	21
2.1 Introduction	21

2.2	Model Checking	22
2.3	Model-Based Testing	23
2.4	Active Automata Learning	24
2.5	Learning-Based Testing	26
3	Sound Learning-Based Testing in the LearnLib	29
3.1	Introduction	29
3.1.1	Contributions	31
3.2	Preliminaries	31
3.2.1	LTL Model Checking	34
3.2.2	Active Learning	36
3.2.3	Learning-Based Testing with Model Checking	38
3.3	Sound Learning-Based Testing in the LearnLib	41
3.3.1	Learning-Based Testing in the LearnLib	41
3.3.2	New Purposes for Queries	42
3.3.3	The LBT Algorithm and Strategies — Informally	42
3.3.4	Learning-Based Testing with Monitoring	44
3.3.5	Learning-Based Testing with Model Checking	45
3.3.6	The New API in the LearnLib	47
3.3.7	The Algorithms — Formally	51
3.4	Related Work	54
3.5	Experimental Results	56
3.5.1	Variables, Metrics and Constants	57
3.5.2	The RERS Challenge	58
3.5.3	Discussion of the Algorithms' Performance	61
3.6	Conclusion and Future Work	65
4	Automated Alphabet Partition Refinement for Learning-Based Testing	69
4.1	Introduction	70
4.1.1	Contribution	73
4.2	Related Work	73
4.3	Preliminaries	74
4.4	Approach	79
4.4.1	Abstraction and Concretization	80
4.4.2	Detecting Controllable Non-Determinism	82
4.4.3	Main algorithm	84
4.4.4	The New LearnLib API	86
4.5	Results	90
4.5.1	Variables, Metrics and Constants	90
4.5.2	Discussion of the Algorithm's Performance	91
4.6	Conclusion and Future Work	93

5	Partitioned Transition Systems and Dependencies	95
5.1	Introduction	95
5.2	Reachability Analysis	101
5.2.1	Transition Systems	103
5.3	The Partitioned Next-State Interface	103
5.3.1	The State Update Specification Language	106
5.4	State Slot Dependencies	108
5.4.1	The Read Dependency	108
5.4.2	The Write Dependency	110
5.5	Combining Transition Groups	112
5.6	Improved Symbolic Reachability Analysis	116
5.6.1	List Decision Diagrams	120
5.6.2	Set Operations for List Decision Diagrams	122
5.7	Results	123
5.8	Conclusion	125
6	Optimizing Variable Orders and Transition Group Orders	129
6.1	Introduction	129
6.2	Dependencies and Event Locality	132
6.3	Variable Ordering	133
6.4	Nodal Ordering for Sparse Matrix Solvers	136
6.4.1	Graph Metrics	136
6.4.2	Nodal Ordering	138
6.5	Problem and Solution	139
6.5.1	Symmetric Representations of Dependencies	140
6.5.2	De-symmetrization of Permuted Matrices	144
6.6	Experiments	147
6.7	Related and Future Work	154
6.8	Conclusion	155
7	Case Study: Symbolic Reachability Analysis for ProB	157
7.1	Introduction	157
7.2	Transition Systems for the B-Method	158
7.3	Symbolic Reachability Analysis for B	160
7.3.1	Symbolic Encoding of B-method States	161
7.3.2	Performance: NEXT-STATE Function	162
7.4	Technical Aspects and Implementation	163
7.5	Experiments	164
7.6	Related Work	167
7.7	Conclusion	168
8	Conclusion	169

8.1	Learning-Based Testing	169
8.1.1	Summary of the Contribution	169
8.1.2	Practical Validation with an ASML Case Study	171
8.1.3	Outlook	172
8.2	Symbolic Reachability Analysis	174
8.2.1	Summary of the Contribution	174
8.2.2	Validation	175
8.2.3	Outlook	176
A	ASML Case Study	177
B	Publications by the Author	181
	Bibliography	183
	Samenvatting	199

List of Algorithms

3.1	LBT with <i>model checking</i> , <code>DisproveFirstOracle</code> , TTT, pW-method	52
3.2	LBT with <i>monitoring</i> , <code>CExFirstOracle</code> , TTT, pW-method	53
4.1	Main algorithm for LBT with alphabet abstractions	86
5.1	REACH-BFS-PREV	119
5.2	LDD-NEXT	124
5.3	LDD-WRITE	124
5.4	LDD-COPY	124

List of Code Listings

3.1	API usage skeleton for learning-based testing	50
3.2	API usage for LBT with <i>model checking</i> , and <code>DisproveFirstOracle</code>	51
3.3	API usage for LBT with <i>monitoring</i> , and <code>CExFirstOracle</code>	51
3.4	RERS problem structure	59
3.5	RERS SUL implementation	59
7.1	<i>MutexSimple</i> B-Method machine example	159

List of Definitions

3.1	Definition (Edge Labeled Transition System)	32
3.2	Definition (Deterministic Finite Automaton)	32
3.3	Definition (Mealy Machine)	33
3.4	Definition (LTL)	34
3.5	Definition (Lasso)	35
3.6	Definition (Membership oracle)	44

3.7	Definition (ω -query)	45
3.8	Definition (ω -membership oracle)	46
4.1	Definition (Partition)	75
4.2	Definition (Partition refinement)	75
4.3	Definition (DFA output)	76
4.4	Definition (Mealy machine output)	76
4.5	Definition (Automaton initialization)	77
4.6	Definition (Automaton refinement)	78
4.7	Definition (Equivalence oracle)	79
4.8	Definition (Abstraction and concretization)	80
4.9	Definition (Approximating membership oracle)	81
4.10	Definition (Splitter oracle)	82
5.1	Definition (Transition System)	103
5.2	Definition (Partitioned Transition System)	104
5.3	Definition (State update specification)	107
5.4	Definition (Read independence)	108
5.5	Definition (Read Dependency Matrix)	109
5.6	Definition (Write independence)	110
5.7	Definition (Write Dependency Matrix)	111
5.8	Definition (Dependency Matrix)	112
5.9	Definition (Partial order for dependencies)	113
5.10	Definition (Row subsumption)	113
5.11	Definition (Projections)	116
5.12	Definition (Partitioned Next-State function)	117
5.13	Definition (NEXT)	118
5.14	Definition (LDD)	120
6.1	Definition (Order)	132
6.2	Definition (Graph)	132
6.3	Definition (Dependency Graph)	132
6.4	Definition (Span)	134
6.5	Definition (Weighted Span)	135
6.6	Definition (Bandwidth)	136
6.7	Definition (Wavefront)	137
6.8	Definition (Graph Permutation)	138
6.9	Definition (Symmetrization)	140
6.10	Definition (Total graph)	141
6.11	Definition (De-symmetrization)	144
6.12	Definition (Mean Standard Score)	148

List of Examples

3.1	Example (DFA)	33
3.2	Example (Mealy Machine)	34
3.3	Example (LTL for DFAs)	36
3.4	Example (Active Learning)	37
3.5	Example (Learning-Based Testing)	40
3.6	Example (Answering a query)	44
3.7	Example (Answering an ω -query)	46
4.1	Example (Partition)	75
4.2	Example (Partition refinement)	75
4.3	Example (DFA output)	76
4.4	Example (Automaton initialization)	77
4.5	Example (Automaton refinement)	78
4.6	Example (Equivalence oracle)	79
4.7	Example (Abstraction and concretization)	81
4.8	Example (Approximating membership oracle)	82
4.9	Example (Splitter oracle)	84
5.1	Example (Dependency analysis)	97
5.2	Example (Assignment with dynamic addressing)	99
5.3	Example (Reachability Analysis of a 1-safe Petri net)	102
5.4	Example (Transition System)	103
5.5	Example (Partitioned Transition System)	104
5.6	Example (Example List Decision Diagram (LDD))	105
5.7	Example (State Update Specification 1)	107
5.8	Example (State Update Specification 2)	108
5.9	Example (Read Dependency Matrix)	110
5.10	Example (Write Dependency Matrix)	111
5.11	Example (Dependency Matrix)	112
5.12	Example (Row subsumption)	114
5.13	Example (Partitioned Next-State function)	118
5.14	Example (Simple LDD)	121
5.15	Example (Transition relation LDD)	121
6.1	Example (Dependency Graph)	133
6.2	Example (Span)	135
6.3	Example (Weighted Span)	136
6.4	Example (Bandwidth)	137
6.5	Example (Wavefront)	138
6.6	Example (Symmetrization)	140

6.7	Example (Total graph)	142
6.8	Example (Nodal Ordering with Cuthill McKee)	142
6.9	Example (De-symmetrization)	145

List of Figures

1.1	Example coffee and tea machine	3
1.2	Example formal techniques	4
1.3	Learning the coffee machine	8
1.4	Minimally Adequate Teacher (MAT) setup	9
1.5	Example Binary Decision Diagram	13
1.6	Reading flows in this thesis	19
2.1	Venn diagram illustrating model checking	23
2.2	Venn diagram illustrating MBT with a failing test case.	24
2.3	Venn diagram illustration for active automata learning	25
2.4	Venn diagram illustration for learning-based testing	26
3.1	Active learning procedure	37
3.2	Sound learning-based testing procedure [PM18]	39
3.3	Learning-based testing algorithm in the LearnLib	43
3.4	LearnLib API extension	47
3.5	Legend for learning algorithms and LBT algorithms	61
3.6	Number of <i>learning</i> queries for <i>model checking</i> on problem 1	61
3.7	Number of <i>learning</i> queries for <i>monitoring</i> on problem 5	62
3.8	Number of <i>hypothesis refinements</i> for <i>monitoring</i> on problem 5	62
3.9	Length of counterexamples with <i>ADT</i> and <i>DisproveFirstOracle</i>	64
3.10	Legend for model checking and monitoring	64
3.11	Number of <i>learning</i> queries with <i>DisproveFirstOracle</i> on problem 1	65
4.1	Active learning procedure with alphabet abstractions	71
4.2	LearnLib API extension	87
4.3	The chain of membership oracles	89
4.4	Legend for learning algorithms and initial partitioning	91
4.5	Number of <i>total</i> queries on problem 1	91
4.6	Number of <i>total</i> queries on problem 5	92
5.1	Modular PINS architecture of LTSmin	96
5.2	Example transition system specification	97
5.3	Dependency matrix of $\mathcal{U}^{2'}$ with group $\mathbf{g}_4$	114
5.4	Dependency matrix of $\mathcal{U}^{2'}$ after row subsumption.	115

5.5	Projection without and with read-separation for Example 5.3. . . .	120
6.1	Reachable states as LDD with different orders	134
6.2	List of nodal ordering algorithms	139
6.3	Example de-symmetrized matrices	146
6.4	MSS for all languages ($ S = 785$)	150
6.5	MSS for B ($ S = 47$)	150
6.6	MSS for DVE ($ S = 264$)	151
6.7	MSS for mCRL2 ($ S = 142$)	151
6.8	MSS for PNML ($ S = 314$)	152
6.9	MSS for Promela ($ S = 18$)	152
6.10	NWES values	153
6.11	Saturation results ($ S = 106$)	156
7.1	<i>MutexSimple</i> statespace for MAXINT=1	160
7.2	Dependency Matrix for Listing 7.1	161
7.3	LDDs of the reachable states	162
7.4	High level design showing the integration of LTSmin and ProB . .	163
A.1	ASML holistic lithography	178
A.2	Happy flow between YieldStar and Litho Insight	179

List of Tables

1.1	Verification competitions participated in by the author	16
1.2	Legend for Table 1.1	17
4.1	Abstraction levels of components for LBT	85
5.1	NEXT-STATE based on combined dependencies (old situation). . . .	98
5.2	NEXT-STATE based on read-write separation.	99
5.3	NEXT-STATE with dynamic addressing (old situation).	100
5.4	NEXT-STATE with dynamic addressing and read-write separation. .	100
5.5	Transition table for g_1 and g_4	114
5.6	Transition table for g_1 and g_4 after row subsumption.	115
5.8	Symbols used in Table 5.9.	126
5.9	Highlighted experiment results.	126
7.1	The NEXT-STATE calls for sets of states	163
7.2	B and Event-B Machines, with BFS and deadlock detection	166

Introduction

During the writing of this thesis (around March 2019) several news outlets [Pos19, Tim19] reported that a Boeing 737 MAX 8 of Ethiopian airlines crashed with all 157 passengers' lives lost. Aside from this tragic loss of lives, in the days that followed, Boeing's shares plunged more than 9% in value, erasing \$32 billion dollar from the company's market value [Tim19]. The precise cause of the crash at this moment is still unknown. Investigators currently suspect it is similar to another crash [Tim18] that happened in October the year before, with the same type of airplane. In this earlier case, a faulty sensor fed erroneous data into the plane's flight-control system, sending the plane into a nose dive and leaving the pilots fighting to regain control over the aircraft in vain.

An airplane is just a single example where safety is critical; computer systems are pervasive throughout our society. They are also present in self-driving cars, railway intersections, pacemakers, nuclear reactors and rockets, where safety is also of clear importance. The field of formal methods offers a wide range of methods and tools to analyze the correctness of such systems, based on rigorous mathematical theory. Formal methods for learning are emerging and can be used to automatically infer behavior of systems [Mei18]. To automatically analyze the correctness of systems, formal methods for verification are employed [CW14]. This thesis improves the state of the art by presenting how to speed up learning and verification of systems automatically. We show how we extended both the LearnLib and LTSmin, as well how they can support each other. The LearnLib is a software library that implements several learning algorithms and LTSmin is a software tool set for verification. The increase in performance of both tools is shown by thorough experimentation, and their source code is made available to the public for free. Industry can apply our approach for learning and verification to a wide variety of systems where safety is critical.

1.1 Verification of Safety Critical Systems

Verification is the process of making sure software is being built right [Boe79]. In general, a “right” software product obeys some universal laws: it should not crash, it should not cause harm to human beings, it should be resistant to malicious attacks and it should not violate the privacy of its users [Hui+16]. These laws form either implicitly or explicitly part of the software’s specification. In this context a specification is a document containing the required behavior of the system that is being built. Had the plane’s flight-control system be properly verified against such a specification, the crash may not have happened.

Proper verification is however easier said than done — as the complexity of today’s software increases, so does the task of verifying it. Software testing is a form of verification where the software executes in its intended environment to determine whether it matches its specification [Whi00]. In other words, testing is applied to expose *faults* in software, that could lead to *failures* in systems that run the software. In this case, it is said in common language that such a system is not bug free. Testing can be done manually or automatically. In case of manual testing, a person interacts with the running system to see if it responds according to the specification. In case of automatic testing, programs execute test cases by interacting with a system and assert whether its software responds as specified. In this thesis we go one step further; we also automatically generate the test cases, based on the system’s specification. Through learning millions of test cases can be generated automatically [Mei18]. Automated generation and execution of test cases allows testers to accomplish the task of dealing with the increasing complexity of verification.

1.2 Validation of Safety Critical Systems

Verifying software depends on specifications that meet the users’ expectations of the software. Validation is the process of asserting that the right software is being built [Boe79]. Validation is typically done by relating the specification to its requirements. The difference between a requirement and specification can be characterized [BK08] as *what* should the system do? And *how* the system should do that? To specify what the system should do, a requirements engineer must adhere to four basic criteria for requirements: completeness, consistency, feasibility, and testability [Boe79]:

- Requirements must be *complete*, e.g. a reference to a button on the system should not be made if the function of the button is not properly defined.
- A requirement is *consistent* if it does not contradict another one.
- *Feasible* requirements are those that can be realized effectively; the benefits

exceed the costs of it to the system.

- A requirement is *testable* when a test case can be practically realized to determine whether or not the system meets it.

Based on these solid criteria, testers can appropriately judge whether a system meets its requirements.

Manual checking a system against its requirements is prone to human error however. Especially in the case of systems where safety is critical, validation benefits from automation too. Starting with the work from Clarke and Emerson [CE81], computer aided verification has been shown to be able to validate systems against its requirements as well, by means of model checking. This has the advantage that no mistakes in verdicts are made. By integrating the LearnLib and LTSmin, this thesis leverages the combination of learning and model checking to automate both verification and validation of systems. Ultimately with the goal of taking manual labor off the hands of testers, and speed up the overall process.

1.3 Formal Methods for Uncovering Faults

In order to automate the process of verification and validation we need to be rigorous and precise. To this end we use formal methods, which are mathematically based languages, techniques and tools [CW96]. Languages are mathematical abstractions of real-world systems. To help understand what a language is, consider an example coffee and tea machine in Figure 1.1.

Figure 1.1: Example coffee and tea machine

We call this mathematical abstraction of a coffee machine an automaton. This example has three states that indicate the amount of Euro that has been inserted

into the machine. The automaton does not accept more than two euro. When the machine is turned on we enter the initial state 0, indicated by the incoming arrow on the top of the automaton. Each transition between states has an input symbol suffixed with ‘?’ and an output symbol suffixed with ‘!’. Inputs and outputs are separated by a ‘/’. The inputs are the **tea?** and **coffee?** button, as well inserting a **euro?** coin. The outputs of the machine are **tea!**, and **coffee!** as well as a display indicating either how much money has been inserted, or the remaining cost of getting tea or coffee: **1!** and **2!**. The *language* of an automaton is the set of all its runs. A run is a sequence of inputs and outputs that are collected following transitions starting from the automaton’s initial state. For example, a run that is in the language of the automaton that symbolizes getting coffee is: **euro? / 1!**, **euro? / 2!**, **coffee? / coffee!**. A run that is not in the language of the automaton is **coffee? / tea!**. Formal techniques are based on language theory in order to verify and validate real-world systems.

Figure 1.2 illustrates the approach in this thesis based on automata to facilitate system verification and validation. Automata can be manually *modeled* and verified automatically by *model checking* based on requirements. By automatically interacting with a real system these automata can be used for *model-based testing* and obtained through *learning*. An integrated approach of these four techniques is called *learning-based testing*. The contribution of this thesis focuses on tool support for model checking and learning-based testing, hence they are highlighted in blue and emphasized.

Figure 1.2: Example formal techniques

1.3.1 Modeling

Modeling is a process that can be used for specifying the intended behavior of a system. It is usually performed manually based on the requirements, and often already reveals inconsistencies and requirements that are not feasible [Har15, Sij+11]. Usually modeling is done with higher level specification languages, i.e. those languages implicitly describe an automaton. For example the Textual Automaton Format (TAF) can be used to specify the coffee machine in Figure 1.1. Depending on the desired abstraction level and the properties of interest of the system, an appropriate language is chosen. In case the interest lies in concurrency, one may choose to model a Petri net in PNML [Bil+03]. To model non-determinism modelers can use mCRL2 [Cra+13] and in case time is of interest an appropriate formalism is UPPAAL [BDL04].

Once modelers are done specifying the automaton, they may assume they capture all intended behavior of the system correctly. However, potential modeling errors should never be ignored and thus model checking should be applied to the automaton, to assert it is indeed a correct model of the intended system.

1.3.2 Model Checking

Model checking is the process of relating the requirements of the system to (a modeled) automaton. Throughout this thesis we assume the requirements are formalized in a temporal logic. A temporal logic expresses which properties should hold over certain states or transitions in the automaton. Two basic operators in temporal logic are $\Diamond$, e.g. now or *eventually* the machine outputs coffee, and $\Box$, e.g. now and *always* the water temperature is 70°. Model checking [CGP01] is a formal method for verification and is independently developed by Clarke and Emerson [CE81], and Queille and Sifakis [QS82]. The procedure checks whether an automaton is a model with respect to the properties specified in the temporal logic. Model checking is an exhaustive technique that usually explores every state in the automaton to verify the property. When the property can not be verified, model checkers provide a counterexample that is a run showing why the property can not be verified. The counterexample can be used as a guide to improve the requirements or the modeled automaton, ensuring the quality of the design phase.

Formalization of requirements is categorized. This is done with respect to the kind of temporal logic that is most suitable for the requirement of interest. The model checking procedure is different and optimized for each kind of temporal logic. The categories are identified as follows.

Reachability analysis involves checking whether a property is always true. Two subcategories here are invariant violation detection and deadlock detection. With invariant violation detection the task is to assert a Boolean expression is true for

every state or transition in the automaton. With deadlock detection the task is to assert there is no state in the automaton with no outgoing transitions. Deadlock detection is important because a deadlock in an actual system typically signifies a software crash.

Linear Temporal Logic (LTL) [Pnu77] can be used to assert a property holds considering all runs expressed by the automaton. Here, the validity of runs can be checked by only looking at individual runs and without switching to other runs. At any point in time of a run its future is viewed linearly. LTL is ideal for specifying liveness (something good eventually happens) and safety (something bad never happens). For the coffee machine, pouring coffee eventually is something good, whereas a too low water temperature is something bad. In case a liveness property can not be met, the model checker provides an infinite run as a counterexample. Such a counterexample resembles a lasso which can be split into a finite part and a loop. The finite part resembles the initialization of a system and moving it to a particular state. From this state the system can execute the loop forever, meaning the “good” behavior never happens. In case a safety property can not be met, the model checker provides a counterexample that is finite. The finite run indicates exactly where the “bad” behavior happens. Part of this thesis shows how liveness and safety can be properly checked using learning algorithms.

Computation Tree Logic (CTL) [BPM83] can be used to assert a certain branch or all branches of runs expressed by the automaton satisfies a property. Here, at any point in time, either one future is investigated, or all possible futures. As the name suggests, CTL properties can be used to investigate the computation tree of a system. There are requirements that can be exclusively expressed in either LTL or CTL, and also some in neither, for a full comparison see [Var01]. More expressive logics than LTL and CTL are CTL* and μ -calculus, but these are not considered further in this thesis.

Prominent tools for model checking are: NuSMV [Cim+02], SPIN [Hol04] and LTSmin [Kan+15]. With clever algorithms, these model checkers can already investigate billions of states automatically. In this thesis we present a novel approach to this technique, that can be applied to problems where many trillions of states need to be searched. This huge amount of states is common for example in software with concurrency. Checking reachability properties of many interleaved computations is a difficult task that we address in this thesis.

1.3.3 Model-Based Testing

After the modeling phase, the actual system can be verified. The form of verification considered in this chapter is called Model-Based Testing (MBT) [TBS11]. The model in this context is a verified automaton, or verified Finite State Machine (FSM), and the system here is called a System Under Test (SUT). MBT based on

FSMs can be used for checking observation equivalence [LY96]. A more powerful form of MBT is based on IOCO [Tre08]. IOCO is capable of dealing with for example non-determinism and quiescence (i.e. the absence of output). Testing based on IOCO is out of the scope of this thesis, because how to learn IOLTSSs (instead of modeling them) is still largely an open research question, some initial work has been done in this direction [AV10].

To form a bridge between the abstract input and output symbols from the automaton, an adapter between the MBT tool and the SUT needs to be built. An adapter is usually a program and for example, needs to be able to send concrete inputs and outputs over some networking protocol to and from the system. When testers build these adapters, which is usually not hard, automatically generated test cases from MBT tools can be used to verify a real system.

With FSM testing, test cases are derived from the automaton. Test cases are sequences of inputs and expected outputs. When such test cases are executed on a SUT, they are usually labeled *pass* or *fail*, indicating whether they match the expected output. For example, suppose we execute the test case: `euro? / 1!, tea? / tea!`, where the SUT is a real coffee machine. Then we need an adapter that translates the input symbols `euro?` and `tea?` such that a Euro coin is inserted, and the tea button is pressed. If the coffee machine pours coffee after the sequence of inputs from our example test case, then it gets labeled *fail*, because tea was expected.

A test case that fails indicates that either one or more requirements are incorrect, or there is indeed a fault in the SUT. From passing all test cases one may not assume the system is fault-free however. Something that the famous Dutch computer scientist Edsger Dijkstra already stated in 1969: “*Program testing can be used to show the presence of bugs, but never to show their absence!*” [Dij70].

The form of FSM testing explained here is similar to Learning-Based Testing (LBT), that is explained in more detail later. With LBT the automaton is not modeled by hand, but it is learned automatically by interacting with the SUT.

1.3.4 Learning

Automata can be learned by interacting with a system [SHM11], and learning automata is thus an alternative to modeling them. In the context of learning, such a system is usually referred to as the System Under Learning (SUL). Modeling has two disadvantages over learning. Firstly, models of the system need to be continuously updated while the SUL evolves and requirements change over time. Secondly, system developers may not be trained in modeling with high level specification languages. Both issues are addressed with learning; learning is an automated process and writing programs that form an adapter between a learn-

Figure 1.3: Learning the coffee machine

ing tool and the system are not hard to build [Vaa17]. In fact, the adapter for a SUT and a SUL are similar as they need to be able to translate between abstract symbols in the automaton and concrete interactions with a real system.

In this thesis the learning algorithms are “active”: they iterate between learning an hypothesis automaton and verifying it against a running system. Opposed to these so called Active Automata Learning (AAL) algorithms that are central in this thesis, there are also “passive” learning algorithms. Passive learning algorithms construct automata based on logs, or execution traces of the system. Here we enter the field of *process mining* [Aal11], which lies outside the scope of this thesis however.

Learned automata need to be verified against the system. This is because learning algorithms produce automata that are hypothetical; they can be incorrect. This is illustrated in Figure 1.3, where a learning algorithm could first learn the automaton in Figure 1.3a by trying out input sequences of length one. A learning algorithm approximates this behavior as if the SUL always responds this way, hence there are loops on state 0. The concept of a teacher is used to verify the learned automata; it is an algorithm that also interacts with the SUL. When the learner asks the teacher whether its initial hypothesis is correct, it will provide a counterexample of at least length two, such as *euro? / 1!, tea? / tea!*. The learner then uses this information to refine the hypothesis, consults the teacher again, and repeats this until Figure 1.3b is reached. Separating the concept of a learner and teacher is from both a theoretical as well as a algorithmic perspective very practical.

The construct of using learners and teachers is developed in the seminal work by

Angluin [Ang87] and is illustrated in Figure 1.4.

Figure 1.4: Minimally Adequate Teacher (MAT) setup

This figure shows that the learner can ask two kinds of questions. It can pose Membership Queries (MQs), i.e. it asks the teacher whether the system executes a particular run. The answer to such a question is simply yes or no. The second kind are Equivalence Queries (EQs), i.e. the learner asks the teacher whether its constructed hypothesis is correct. The answer to such a question is either yes or no. In the latter case the teacher also gives a CounterExample (CE) that indicates why the constructed hypothesis is incorrect. Thus the learner will use MQs to form hypothesis automata and EQs to improve the hypotheses when the teacher says the current one is incorrect. By means of an adapter, the teacher answers both kinds of queries by interacting with the system, i.e. by sending inputs (I) and receiving outputs (O).

The purpose of learning algorithms is to infer the behavior of a running system, including possible faults in its software. This means a learned automaton can not immediately be used for testing.

1.3.5 Learning-Based Testing

To make the learning setup suitable for testing based on requirements, two ingredients are added to the setup [PVY02, Mei18]. The teacher is equipped with a model checker and a capability for executing test cases. These extensions to learning encompass the technique called Learning-Based Testing (LBT). LBT algorithms apply the model checker to each automaton obtained through learning. Whenever the model checker provides a counterexample it may be a false negative: although the counterexample indicates a requirement does not hold, the SUT may not actually behave as the counterexample shows. This is because the learned automata on which the requirements are verified are only hypothetical and thus the counterexample must be tested on the SUT by the teacher. If the teacher confirms the SUT behaves as the counterexample indicates, the counterexample is a true

negative and the test fails. This means the SUT does not meet the requirement, and identical to MBT, a requirement needs to be improved, or the SUT's software contains a fault that needs to be fixed. On the other hand, if the test reveals the counterexample is a false negative, the hypothesis is incorrect and the automaton needs to be refined. In this case, the test passes because current information of the SUT is insufficient to determine a requirement does not hold.

Due to the model checking involved, test case generation is guided by the requirements, and testing with LBT as a whole is seen as directed towards the requirements. As an added bonus, LBT fits nicely in an agile development setting [Mei18], this is because the behavior of systems can be continuously learned while they evolve over time.

1.4 Contribution to Learning-Based Testing

The concept of LBT has been around since 2002 and was originally named black-box checking [PVY02]. Yet, our contribution is still quite significant and consists of the first implementation based on the LearnLib. In addition to the implementation design, we explain how safety and liveness properties affect the LearnLib's learning and testing procedure.

The LearnLib implements many different learning algorithms, starting with Angluin's L^* to the more recent ADT [Fro15] algorithm. The performance of recent learning implementations in the LearnLib in particular, has not been thoroughly investigated in the context of LBT. In Chapter 3 we show their performance in terms of membership queries vastly differs, so the first research question is stated as:

Research Question 1: Learning-based testing with the LearnLib

How do learning algorithms in the LearnLib perform in the context of testing, and how should testers apply them to real-world systems?

Regarding the performance aspect of the different learning algorithms, we can be more precise. Namely, given an unknown system language S and n requirements formalized in temporal formulae $\phi_1, \dots, \phi_n$. Then for each formula $1 \leq i \leq n$ the question is: if $S \models \phi_i$ does not hold, how many membership queries does it take the learning algorithm to show that? Where fewer membership queries is better and $S \models \phi_i$ may be interpreted as "the system is correct with respect to requirement i ".

Regarding how testers should apply the extensions to LBT; we shall answer this

by discussing what a good Application Programming Interface (API) looks like for LBT in the LearnLib.

The performance aspect is also investigated when a formalized requirement is encoded as a monitor [TRV12] or as a Büchi automaton [Büc62]; both have advantages. Monitors can be used to verify the “safety part” of a requirement, while Büchi automata can encode both safety and liveness. The advantage of monitors is that counterexamples derived from them are finite and can be confirmed on the SUT in a straightforward manner. Counterexamples derived from Büchi automata have the advantage that they are more informative to the learner than those from monitors. However, since counterexamples for liveness properties are infinite, confirming them on the SUT becomes more difficult.

Research Question 2: A sound LBT approach for safety and liveness

How should a testing approach based on monitors and Büchi automata be integrated in the LearnLib, and how does it affect learning performance?

We will suggest to first investigate the “safety part” by using monitors. When liveness needs to be considered, a solution is presented that involves answering ω -queries over infinite runs. In both cases we improve upon the state of the art. For liveness in particular, the initial approach was to assume an upper-bound on the number of states in the SUT [PVY02]. By dropping this assumption and unrolling the loop of a lasso a fixed number of times, only a “warning” could be given [Mei18]. In case of both safety and liveness, the LearnLib can provide a definitive “no” answer.

We first published our work on LBT in [MP18, MP19] which now forms the main content of Chapter 3. The implementation of LBT in the LearnLib can be found online freely at <https://learnlib.de/>. The contribution amounts to ≈ 15.000 lines of code [Git19a] in the LearnLib.

Another way we improve the state of the art is by an approach for LBT that can be applied to systems with a huge number of input symbols. In particular, parameterized inputs that are seen among many real world systems [Aar14]. To illustrate this, consider again the coffee machine in Figure 1.1. Here the real system only has three input symbols (`euro?`, `tea?` and `coffee?`). Learning an automaton of such a system with a low number of inputs can be done quickly in practice. Now suppose however, there exists another coffee machine that parameterizes the inputs `tea?` and `coffee?` with the amount of milk and sugar, on a scale from one to five. This machine has 51 inputs instead of three: `euro?`, `tea(1...5,1...5)?` and `coffee(1...5,1...5)?`. This amount of inputs significantly increases the amount of membership queries that the learning algorithm has to perform to get hypothesis

automata. Assuming the real system still outputs `1!`, `2!`, `tea!` and `coffee!`, the learning algorithm may perform unnecessary work. That is, whenever all inputs `tea(..., ...)?` are sent, the system may still output only `tea!`, so we could have sent input `tea(1,1)?` only, and conclude it outputs `tea!` for any `tea(..., ...)?`. If this conclusion is incorrect the teacher will show why (e.g. the answer to `tea(5,5)?` may be `coffee!`). The information from the teacher is then used by the learner to refine the partition of input symbols. This concept is known as controllable non-determinism, initially presented by Howar, Steffen, and Merten [HSM11]. Our contribution is showing that the task of verifying requirements on automata where parameterized inputs are seen as equivalent, does not change the model checking procedure. So from the perspective of the tester, the approach to LBT we present simply runs more efficiently. The central research question for Chapter 4 is:

Research Question 3: Alphabet abstraction

To what extent do learning algorithms in the LearnLib, in an LBT context, benefit from partitioning input symbols?

This research question is answered similar to Research Question 1, i.e. in terms of number of membership queries. In the experimental section (Section 4.5) we see improvements when the number of inputs grows bigger. In particular, within a timeout of three hours, more requirements are shown falsifiable when a partitioning scheme for input symbols is applied.

Published work that is closest related to Chapter 4 can be found in [Aar14, Mei18]. Key in those publications is that the adapter (i.e. mapper or wrapper) that forms the bridge between the learning algorithm and real system, takes care of abstracting the input symbols. Our approach on the other hand follows [HSM11] where the learning algorithm itself is responsible for making the abstractions.

1.5 Contribution to Model Checking

The purpose of the improvements to reachability analysis developed in Chapters 5 and 6, is to handle the state space explosion problem [Val96]. This problem arises in systems typically due to several processes running in parallel, and synchronizing only rarely. As a consequence, the possible interleaving of process executions explodes the size of the automaton representing their behavior. It is said that these systems behave asynchronously [BCL91] opposed to synchronously. Model checkers are increasingly successful in dealing with the state space explosion problem [Cla08]. For example, LTSmin implements Partial Order Reduction (POR) [BLL09] so that only a restricted number of runs in the automaton are

Figure 1.5: Example Binary Decision Diagram

explored and the property that is analyzed can still be verified. Another approach that LTSmin takes is to store the states in the automaton symbolically, which is the underlying principle for Chapters 5 and 6.

Our contribution to symbolic reachability analysis can be explained as follows. Recall our example coffee machine in Figure 1.1 has three states, representing the amount of money it has accepted (0, 1 and 2 Euro, more than 2 Euro is not accepted) and we need to store these states in some form of data structure. An efficient way is to store them in decision diagrams, of which a binary [Bry86] form is most easy to illustrate. A binary representation of states mandates we need to encode the amount of Euro into bits; two bits are enough to encode three states. An example Binary Decision Diagram (BDD) is shown in Figure 1.5. This (non-reduced) BDD encodes exactly the set of (reachable) states $\{0, 1, 2\}$ in their binary form $\{00, 01, 10\}$. More precisely, a state starts at bit x_1 and goes along two edges down to a terminal 1 (true), or 0 (false). A dashed edge indicates the bit is 0 and a solid edge indicates the bit is 1. A state that goes to terminal 1 indicates the state is reachable, one that goes to terminal 0 is not reachable, indeed state 11 goes to terminal 0 as it is not reachable.

Symbolic reachability algorithms explore the state space in order to assert invariants for each state, or check if they deadlock. To this end the algorithms first store the initial state (0 Euro) in a BDD, then add the 1 Euro state and then the 2 Euro state. Whenever a state is added, requirements for reachability properties can be verified on the BDD. To do this more efficiently, Chapter 5 investigates how the individual bits are changed when new states are discovered. This is done by carefully looking at assignments of variables in the specification of the automaton (e.g. the coffee machine) that is analyzed. This process of analyzing dependencies between variables and assignments is called *dependency analysis*. Symbolic exploration of the state space is done using a so called NEXT-STATE [CMS06] function. This function serves as a bridge between specification languages and state storage (using decision diagrams). With the aim of reducing the runtime of state space

exploration, the main research question of Chapter 5 is stated as:

Research Question 4: Dependency analysis

To what extent can we use dependency analysis for specification languages to reduce the number of NEXT-STATE calls?

The contribution of Chapter 5 extends the state of the art by separating the dependencies in so called read, write and copy dependencies. The current state of the art does not distinguish between reading and writing [BP08]. We first published this work in [Mei+14], where we showed that the distinction reduces the number of NEXT-STATE calls and hence the runtime over the current state of the art.

The second contribution can be understood by observing that the BDD shown is ordered: bit x_1 is “smaller” than bit x_2 because x_1 always appears above x_2 . In general, the ordering of variables (bits) is very crucial, especially when the number of states grows to many trillions. A good variable order can significantly reduce the size of the BDD and hence the memory usage of the model checking procedure. In Chapter 6 we show how structural information in the specification modeling languages can be used to optimize the variable order of BDDs. We show that optimizing the variable order can be done with classical bandwidth reduction [CM69] algorithms. With the aim of reducing the memory footprint of state space exploration, the main research question of Chapter 6 is stated as:

Research Question 5: Variable ordering

To what extent can bandwidth reduction algorithms be used for reducing the size of decision diagrams?

Our novel approach was first published in [MP15, MP16], where we showed bandwidth reduction is very competitive to the state of the art, i.e. model checkers that use the FORCE [AMS03] algorithm for optimizing the variable order. Subsequent work by other authors presented in [Amp+17, Amp+18] investigates how variable ordering can be improved even further based on our work on bandwidth reduction.

In Chapter 7 we apply our findings of Research Questions 4 and 5, to an actual case study. The subject under study is the ProB model checker, and here our research question is:

Research Question 6: ProB case study

How well can we apply *dependency analysis* and *variable ordering* to the ProB model checker?

The implementation of the techniques discussed in Chapters 5 to 7 can be found online freely at <http://ltsmin.utwente.nl/>; they amount to ≈ 300.000 lines of code [Git19b] in the LTSmin model checker.

1.6 The Value of Verification Competitions

Scientific verification competitions have formed a major part in the realization of this thesis. Their purpose is to evaluate capabilities of software tools to various fields of validation and verification. The main scientific use for them in this thesis is in the experimental sections of chapters to validate our results. We have obtained requirements, specifications and actual software from these competitions and use them to benchmark the performance of the developed algorithms.

To scientists verification competitions are also valuable because they are fun, rewarding, increase the reliability of the tools they develop and the competitions bring the community together [Bar+19]. The reward can take three forms; a sense of accomplishment, actual prize money and review papers that are often highly cited by participants of the competition. The reliability of tools is increased because participants are usually punished, sometimes even exponentially hard, for every incorrect answer [Jas+17].

To highlight the importance of verification competitions, the first TOOLympics was held during the 25th edition of TACAS, in 2019 [Bar+19]. TACAS is one of the most highly regarded conferences for computer scientists specializing in automated verification. The TOOLympics hosts 16 different competitions, each with its own characteristics and types of problems to solve. The thesis' author has participated in both the Model Checking Contest (MCC) [Kor+18b, Amp+19], and the Rigorous Examination of Reactive Systems (RERS) [Jas+17, Jas+19]. The MCC and RERS competition are part of the 2019 TOOLympics and they were held separately in several years before as well. The characteristics of both challenges are shown in Tables 1.1 and 1.2.

1.6.1 Model Checking Contest (MCC)

The MCC is a yearly recurring competition where the task is to answer formal verification queries for Petri nets. These Petri nets are specified in PNML, so to compete with LTSmin in the MCC, PNML support was added. LTSmin is

able to compete in every *category* as can be seen in Table 1.1. Over multiple participated years both the reliability and performance of LTSmin has increased. For example, to improve reliability, LTSmin’s type system is extended and a type checker is implemented, which is not covered further in this thesis. The technology developed in Chapters 5 and 6 however, contributed a major part to LTSmin’s improved performance. Of particular interest is LTSmin’s performance in the LTL category, because in Chapter 3 we claim that LTL model checking queries in the context of LBT are cheap. This claim is supported by the fact that LTSmin ranked *first* place in the LTL category in 2016 [Kor+16] and second place in the next two years [Kor+17, Kor+18a].

1.6.2 Rigorous Examination of Reactive Systems (RERS)

The RERS challenge consists of two types of problems, different from the MCC. The nature of these two is that they are either synchronous or asynchronous. The asynchronous problems are very similar to the problems in the MCC. They are provided as Petri nets as well.

The synchronous problems are quite different in nature, and very suitable for a learning approach. The main motivation to start participating in the RERS challenge was to show the applicability of LTSmin to the “synchronous” category. After some years this resulted in a modular and finished integration of the LearnLib and LTSmin. This result is presented in Chapters 3 and 4.

RERS				MCC		
Report	[Jas+17]			[Kor+18b], [Kor+15], [Kor+16], [Kor+17] [Amp+19], [Kor+18a], [Kor+19]		
Research Input Nature	Java synchronous		[Ste+17] PNML asynchronous	PNML asynchronous		
Method	Explicit			Symbolic		Explicit
Tool Support	LearnLib	LearnLib + LTSmin Chapters 3 and 4	LTSmin	LTSmin Chapters 5 and 6		
Algorithm	learning	LBT	model checking	reachability	model checking	
Year	2016 – 2018		2017 – 2018	2015 – 2019	2016 – 2018	
Category	reachability		LTL	reachability	CTL	LTL

Table 1.1: Verification competitions participated in by the author

Report	Available reports with details of the competition, where the thesis' author is involved.
Research	The scientific paper we have contributed to, in order to improve the competition itself.
Input	The <i>input</i> specification language we used to solve the problem.
Nature	<i>Nature</i> of the problem to solve. Can be <i>synchronous</i> or <i>asynchronous</i> .
Method	Whether we used an <i>explicit</i> -state or <i>symbolic</i> method to store the state space.
Tool	The <i>tool</i> we used to tackle the problem.
Support	The chapters <i>supported</i> by the competition; typically the experimental sections.
Algorithm	The <i>algorithm</i> we used to tackle the problem.
Year	The <i>years</i> we have participated in the competition.
Category	The <i>category</i> of the model checking problem.

Table 1.2: Legend for Table 1.1

1.7 Thesis Overview

Aside from this introductory chapter and the conclusion later, this thesis consists of two main parts. In Chapters 3 and 4 we improve upon the field of learning-based testing as discussed in Section 1.4. The next part (Chapters 5 and 6) improves upon the field of model checking, discussed in Section 1.5. The scientific research method applied to both parts follows the same pattern. First we properly introduce the subject, as well as discuss related work, the current state of the art and the remaining open problem that is addressed. Next, mathematical preliminaries are covered that form a necessary basis to explain the developed formal method. The method then presents either new theory or new algorithms, or both. The algorithms are consequently subjected to rigorous experimentation based on the presented software competitions. Lastly, we conclude our findings of these benchmarks as to get an indication which algorithms may work best in practice. A separate chapter (Chapter 7) is then devoted to applying the developed methods from Chapters 5 and 6 to a case study with the ProB model checker. A brief overview of each chapter is as follows.

Chapter 2 provides a brief overview of the formal techniques in this thesis. The chapter explains the goals of model checking, model-based testing, active automata learning and learning-based testing, rather than their underlying algorithms. The explanation is done by means of Venn diagrams and serves as a starting point for the rest of the chapters.

Chapter 3 shows how we extended the LearnLib with sound methods for learning-based testing. We first discuss the theory necessary for applying LTL model checking in the context of active automata learning. Next we present two strategies for a learning-based testing algorithm based on this theory. Afterwards we show how we designed the extension for the LearnLib and how well it works on the RERS challenge. This chapter covers Research Questions 1 and 2 and is based on the following publications:

- Jeroen Meijer and Jaco van de Pol. “Sound Black-Box Checking in the LearnLib”. In: *NFM*. vol. 10811. Lecture Notes in Computer Science. Springer, 2018, pp. 349–366,
- Jeroen Meijer and Jaco van de Pol. “Sound black-box checking in the LearnLib”. In: *Innovations in Systems and Software Engineering* 15.3 (Sept. 2019), pp. 267–287. ISSN: 1614-5054.

Chapter 4 extends the learning-based testing approach from Chapter 3 with a partitioning strategy for input symbols. We first cover the underlying theory for the partitioning strategy and show how we designed a new learning algorithm. We repeat all the experiments from Chapter 3 and show particular cases from the RERS challenge where the partitioning strategy really pays off. This contribution to learning-based testing is first published in Chapter 4 and covers Research Question 3.

Chapter 5 shifts the focus from learning-based testing towards symbolic reachability analysis, a particular form of model checking. We show how careful analysis of dependencies between expressions and variables for several specification languages can speed up state space generation. This chapter covers Research Question 4 and is based on the following publication:

- Jeroen Meijer, Gijs Kant, Stefan Blom, and Jaco van de Pol. “Read, Write and Copy Dependencies for Symbolic Model Checking”. In: *Haifa Verification Conference*. Vol. 8855. Lecture Notes in Computer Science. Springer, 2014, pp. 204–219.

Chapter 6 presents a novel method for variable ordering, based on the foundation laid out in Chapter 5. The novelty to variable ordering is based on bandwidth reduction and we show how well it compares to the state of the art. This chapter covers Research Question 5 and is based on the following publications:

- Jeroen Meijer and Jaco van de Pol. “Bandwidth and Wavefront Reduction

for Static Variable Ordering in Symbolic Reachability Analysis”. In: *NFM*. vol. 9690. Lecture Notes in Computer Science. Springer, 2016, pp. 255–271.

Chapter 7 presents a case study we have done with the ProB model checker. We first show how the ProB model checker is integrated with LTSmin by using sockets for inter-process communication. Next we experiment with several specification languages supported by ProB and show that the methods presented in Chapters 5 and 6 are applicable to ProB as well. This chapter covers Research Question 6 and is based on the following publication:

- Jens Bendisposto, Philipp Körner, Michael Leuschel, Jeroen Meijer, Jaco van de Pol, Helen Treharne, and Jorden Whitefield. “Symbolic Reachability Analysis of B Through ProB and LTSmin”. In: *IFM*. vol. 9681. Lecture Notes in Computer Science. Springer, 2016, pp. 275–291.

The parts on learning-based testing and symbolic reachability analysis in this thesis can be read individually. For completeness, the reading flows are shown in Figure 1.6. A complete list of publications by the thesis’ author can be found in Appendix B.

Figure 1.6: Reading flows in this thesis

Learning and Analysis from a Language Perspective

2.1 Introduction

In this chapter we provide a brief overview of formal techniques for learning and analysis of system behavior. Specifically, we will not discuss how their underlying algorithms work, but rather what their goal is. By using a language perspective, we familiarize the interested reader with some concepts and mathematical notations that are used throughout the rest of this thesis.

The techniques that will be illustrated are model checking, Model-Based Testing (MBT), Active Automata Learning (AAL) and Learning-Based Testing (LBT). Model checking is the process of verifying an abstraction of a real system (i.e. automaton) is correct with respect to certain requirements. MBT and LBT are testing techniques based on automata for verifying a real system. Model checking is usually done as a preliminary step towards MBT, AAL depends on MBT to verify learned hypotheses, and LBT is a technique that integrates AAL and model checking. In case of MBT and LBT, the System Under Test (SUT) is viewed as a black-box. This perspective allows testing algorithms to input something into the black-box and observe how it responds. This works regardless of the code attributes or used data structures in the software of the SUT [Whi00]. In the explanation that follows, we show how the techniques compare to each other while the complete behavior of the SUT is unknown.

At the core of these techniques lie the requirements, automaton and system, previously illustrated in Figure 1.2. To each of these concepts we associate languages. These associations have distinct purposes.

- The requirements specify *what* the SUT should do. This means that the language associated to the requirements limits the runs the SUT may exhibit.

In the interest of system verification, runs that lead to failures in the SUT should not be in the language associated to the requirements.

- The automaton specifies *how* the SUT should behave. This means that all runs in the language associated to the automaton are potential executions on the system.
- To the system that is to be validated and verified, a language can be associated too. Because the system is viewed as a black-box, its language is *unknown* and an adapter is necessary that forms a bridge between the MBT and LBT tool. By interacting with the system through the adapter, its language can be observed. In general however, it is not possible to fully deduce what the language exactly is.

2.2 Model Checking

Requirements are assumed to be formalized in some temporal logic. A temporal logic allows us to see all the requirements as a single one, by taking the logical conjunction of all their formulae. This means that the languages of all formulae can be seen as a single language by taking their intersection and hence, it seems as if there is one global requirement with a single language. In practice, combining all requirements into a global one is infeasible¹. Viewing all requirements as a single one is however ideally suited for illustration purposes.

The Venn diagram in Figure 2.1 is used to illustrate model checking where the automaton has at least one run that does not meet the requirements. Every point in this Venn diagram is a run. There are as many points as there are (im)possibilities for interacting with a SUT through its adapter. The following concepts can be observed in the Venn diagram.

- The language of the requirements, indicated with the blue circle (R).
- The language of the automaton, indicated with the green circle (A).
- A counterexample which is a single run indicated with $\star$.

In case the automaton is incorrect with respect to the requirements (i.e. the case that is shown), then the task of a model checker tool is to find $\star$. The run $\star$ is called a counterexample, because it disproves the claim that A is correct.

¹Formalizing a combined requirement is considered too hard and a Büchi automaton of a single LTL formula is generally too big.

Figure 2.1: Venn diagram illustrating model checking

Model checking is the process of showing that A lies fully within R . Using a mathematical notation, model checking is the process of deciding whether $A \subseteq R$ is true, i.e. an inclusion check. Alternatively we say that model checkers perform an emptiness check, i.e. whether $A \cap \overline{R} = \emptyset$ is true, which is equivalent and more commonly used. The emptiness check here means there is no run in A that lies outside R ². If the checks fails, then A needs to be remodeled, or the requirements need to be adjusted. In both cases $\star$ may be used as a guide.

2.3 Model-Based Testing

Model-Based Testing (MBT) is a formal technique based on automata to derive test cases. These test cases are both derived automatically, as well as executed automatically on a real system by the MBT tool³. Automata used by MBT are usually modeled by hand and may contain modeling errors. Therefore model checking is seen as a preliminary step towards MBT. Test cases are runs that can be executed on the SUT through an adapter. They are labeled either *fail* or *pass* indicating whether the system responds as specified by the automaton.

The Venn diagram in Figure 2.2 is used to illustrate MBT, with at least one failing test case. The following concepts can be observed in the Venn diagram.

- The language of the automaton, indicated with the green circle (A).
- The language of the system, indicated with the red circle (S). The exact runs that lie within S are unknown, because the system is viewed as a black-box.
- Failing test cases, indicated with $\triangleleft$ and $\triangleright$. Test case $\triangleleft$ means the system might be missing behavior, and $\triangleright$ means the system has potentially incorrect behavior.

²Chapters 5 to 7 are dedicated to solving the emptiness check efficiently.

³In this thesis we assume the system is input-enabled

Figure 2.2: Venn diagram illustrating MBT with a failing test case.

If $\triangleleft$ or $\triangleright$ exists (i.e. the case that is shown), then the task of a MBT tool is to find either. In other words, MBT can be used to show the inequivalence of A and S . A possible strategy for MBT tools is to generate random runs [Bel14] to find a failing test case. These generated test cases are executed on the SUT through its adapter by sending inputs and observing outputs. The response of the SUT is compared to how the system should respond according to the automaton. A failing test case (i.e. the system does not respond as specified) is proof that A and S are not equivalent, i.e. $A = S$ is not true. Here, the tester should compare the failing test case with the requirements to verify whether it is actually met. If the failing test case is valid, then the tester remodels A to make it compatible with the actual system behavior. If indeed the failing test case is invalid, then the tester concludes the system does not meet the requirement. In this case the requirement must be improved or a fault in the SUT's software fixed.

The limits of MBT can be related to Edsger Dijkstra's famous quote: "*Program testing can be used to show the presence of bugs, but never to show their absence!*" [Dij70]. Because the language of the system (the red circle S) is unknown, MBT can not show that all runs from A are included in S (i.e. showing that $A \subseteq S$ is true). In other words, from all passing test cases we may not conclude the system meets the requirements, and thus MBT can not be used to show the system is bug free.

Both model checking and MBT are fully automated and as such, still, a powerful formal method for testers for locating potential faults in software.

2.4 Active Automata Learning

Active Automata Learning (AAL) is the process of learning an automaton by interacting with a system. Automata obtained through AAL are hypothetical, and teachers are used to show whether the hypotheses learned are correct. In case

Figure 2.3: Venn diagram illustration for active automata learning

an hypothesis is incorrect, a teacher provides a run that is a counterexample to the hypothesis. In case of learning, such a run is called a counterexample because it disproves the claim the automaton correctly captures the behavior of the system. Counterexamples lie in the symmetric difference of the language of the hypothesis automaton and system. Learning algorithms use these counterexamples to form increasingly better hypotheses, until the teacher says one is correct. This process is illustrated with Venn diagrams as well.

AAL is illustrated in Figure 2.3. In this Venn diagram the following can be observed.

- The language of a hypothesis automaton, indicated with the light green circle (H_n).
- The language of a next hypothesis automaton, indicated with the dark green circle (H_{n+1}).
- The language of the system, indicated with the red circle (S). Again, the exact runs that lie within S are unknown, because the system is viewed as a black-box.
- Two counterexamples for H_n , indicated with $\triangleleft$ and $\triangleright$.

If $\triangleleft$ or $\triangleright$ exists (i.e. the case that is shown), then the task of a teacher is to find either (not both). In case the teacher provides $\triangleleft$ then the algorithm learned something incorrect. In case the teacher provides $\triangleright$ then the learner is given new information. Observe that here we also try to show the inequivalence of a known language of an automaton (H_n) and an unknown system language (S). This is exactly what MBT tools are designed to do (see Figure 2.2). In practice, teachers are equipped with MBT algorithms, and active automata learning is therefore synonymous with *test-based modeling* [Tre11].

The task of a learning algorithm is to produce H_n and improve upon it by using either $\triangleleft$ or $\triangleright$. The learner ensures that $\triangleleft$ is not in the language of H_{n+1} anymore, while $\triangleright$ will be in the language of H_{n+1} . Hypothesis H_{n+1} is better than H_n in

the sense that the intersection of H_{n+1} with S is bigger than the intersection of H_n with S . In other words H_{n+1} covers more runs of the actual system than H_n ⁴. Based on the runs $\triangleleft$ or $\triangleright$ from the teacher, a next hypothesis is usually much better than just a single $\triangleleft$ or $\triangleright$. This is because learning algorithms approximate the behavior of a system. Approximating the behavior means that faults in software can be predicted rather than always directly observed [Mei18].

2.5 Learning-Based Testing

Learning-Based Testing (LBT) combines model checking with Active Automata Learning (AAL). This is done by running a model checker on each hypothesis automaton obtained through AAL. In case the model checker gives a counterexample, i.e. the automaton does not satisfy the requirements, either the hypothesis needs to be improved, or a failing test case is reported.

The possible outcomes of applying a model checker are illustrated in Figure 2.4.

Figure 2.4: Venn diagram illustration for learning-based testing

The following concepts can be observed in each Venn diagram.

⁴For a precise notion of “better” and “bigger” we refer the reader to [SHM11], especially parts regarding the Myhill-Nerode theorem [HMu03, Ner58].

- The language of the requirements, indicated with a blue circle (R).
- The language of a hypothesis automaton, indicated with a green circle (H_n).
- The language of the system, indicated with a red circle (S). Again, the exact runs that lie within S are unknown, because the system is viewed as a black-box.

The cases where the model checker produces a counterexample or not are distinguished in a negative and positive column. The true and false cases indicate the quality of the hypothesis.

- A true negative means the system does not meet the requirements. A counterexample ($\star$) from the model checker indicates a failing test case.
- A false negative means the system meets the requirements, but the hypothesis is incorrect. A counterexample ($\triangleleft$) indicates why the hypothesis is incorrect. The run $\triangleleft$ is seen as a passing test case.
- A true positive means both the hypothesis and the system satisfy the requirements.
- A false positive means the hypothesis satisfies the requirements, but the system does not.

Whenever a $\triangleleft$ is found, then the hypothesis automaton is refined by the learner. Note that this case is identical to the $\triangleleft$ run in Figure 2.3 which is used to produce H_{n+1} . On the other hand, note that a run that is in S and outside A (e.g. $\triangleright$ in Figure 2.3) is never found through model checking. Failing test cases found with LBT (e.g. $\star$) and MBT are processed by the tester in similar fashion. The two positive cases are practically irrelevant in Figure 2.4. This is because the system's language is unknown and whenever the model checker says the hypothesis satisfies the requirements, we can not conclude the system's software is fault free. This limitation is equivalent to MBT's limitation: testing can not be used to show software is free of faults.

When verifying systems, LBT is different from MBT in the sense that learning algorithms will learn faults that are in the SUT's software, whereas faults are not modeled in automata that are used by MBT algorithms. To illustrate the latter, observe that when model checking is successfully applied prior to MBT, the false positive case in Figure 2.4 is equivalent to Figure 2.2, yet failing test cases are not found with the false positive case. The power of LBT algorithms is that $\triangleleft$ is used by the underlying AAL algorithm to improve the hypothesis automaton automatically, whereas in case of MBT the automaton needs to be remodeled by hand.

Part of the contribution to LBT in Chapter 3 is an investigation into which learning algorithms are most efficient at finding $\star$. Furthermore in the same chapter, we present an architectural design in the LearnLib so that AAL algorithms can

automatically refine hypotheses based on $\triangleleft$. Part of the contribution in Chapter 4 to LBT is a novel approach that partitions the input symbols so that any learning algorithm can find $\star$ even more efficiently.

Sound Learning-Based Testing in the LearnLib

Abstract

In Learning-Based Testing (LBT) incremental hypotheses on the behavior of a system are learned in the form of finite automata, using information from a given set of requirements, specified in Linear-time Temporal Logic (LTL). The LTL formulae are checked on intermediate automata and potential counterexamples are validated on the actual system. Spurious counterexamples are used by the learner to refine these automata. We improve LBT in two directions. First, we improve checking lasso-like counterexamples by assuming a check for state equivalence. This provides a sound method without knowing an upper-bound on the number of states in the system. Second, we propose to check the safety portion of an LTL property first, by deriving simple counterexamples using monitors.

We have extended LearnLib’s system-under-learning API to make our methods accessible, using LTSmin as model checker under the hood. We illustrate how LearnLib’s most recent active learning algorithms can be used for LBT in practice. Using the RERS 2017 challenge we provide experimental results on the performance of all LearnLib’s active learning algorithms when applied in an LBT setting. We will show that the novel incremental algorithms TTT and ADT perform the best. We also provide experiments on the efficiency of various LBT strategies.

3.1 Introduction

From a formal methods perspective both liveness (something good eventually happens), and safety (something bad never happens) are essential to the functional

reliability of safety critical systems. It is key for testers and developers to have easily usable tooling available to investigate those liveness and safety properties. We expand the formal method toolbox of testers and developers. This is done by contributing the first Free and Open-Source Software (FOSS) implementation of Learning-Based Testing (LBT) [MS11] to the LearnLib.

The contributed method is based on learning and is used as a formal approach to testing systems. The LBT method was initially developed and named Black-Box Checking (BBC) by Peled, Vardi, and Yannakakis [PVY02]. In LBT requirements are checked on an automaton of the system that is automatically learned from observations. In particular, the requirements are applied to intermediate learned hypotheses in order to speed up the learning process, or tested on the system.

We introduce a state equivalence check, to obtain a sound method without assuming an upper-bound on the number of states in the System Under Learning (SUL). Additionally, we present a novel method for the safety portion of the requirements by means of so called monitors. We performed extensive experiments comparing both methods under several LBT strategies, to show how well they perform on an actual case study. One of the main results is that the ADT [Fro15] learning algorithm usually performs best.

In general, the LBT procedure requires checking that an infinite length counterexample can be executed on the SUT. An infinite counterexample is represented by a lasso of the form uv^ω , where u represents the initialization of the system, and v models an infinite loop. These lassos are provided by model checkers that check the automata by means of the LTL properties. If an LTL property can not be verified, the model checker must provide a counterexample in the form of such a lasso. The soundness of the original LBT procedure [PVY02] assumed guessing an upper-bound on the number of states in the system. This could be either dangerous (if the guess is too low), or inefficient (if the guess is too high). We resolve this by allowing the LearnLib to check for state equivalence in the SUL by implementing so-called ω -queries. In this work we also present a novel approach that uses monitors. The advantage of this new approach is that when a system can not be correctly monitored, a finite counterexample is produced. Validating such a finite counterexample from a model checker by simply testing whether the SUL accepts it is inherently sound. For safety properties, soundness is preserved without the need of checking for state equivalence. For arbitrary LTL properties, we propose to first check if their “safety portion” can be used, before resorting to the general procedure using lasso-like counterexamples.

3.1.1 Contributions

To summarize the contributions¹: we will revisit the work done on lasso shaped counterexamples initially described in [MP18]. The novel concept of monitoring and its implementation in the LearnLib is also covered. The various Active Automata Learning (AAL) algorithms and LBT strategies that are now available are subjected to rigorous experiments. Concretely, we contribute the following.

- Two variations of learning-based testing algorithms.
- A sound learning-based testing approach that uses state equivalences, instead of an upper-bound on the number of states in the SUL.
- A novel sound learning-based testing approach that uses monitors that may provide counterexamples for safety properties.
- A modular design, allowing new model checkers or active learning algorithms to be added easily, or smarter strategies to be implemented for detecting spurious counterexamples.
- A thorough reproducible experimental setup, with several combinations of automaton types, AAL algorithms and LBT strategies.

The rest of the chapter is structured as follows. Section 3.2 provides preliminary definitions and procedures for LTL model checking, active learning and learning-based testing. Section 3.3 describes monitoring, how one can check whether a SUL accepts an infinite lasso-shaped word, and how this is implemented in the LearnLib. In Section 3.4 we discuss related work, such as other model checkers, active learning algorithms, and LBT approaches. Section 3.5 details the result of our case study, and Section 3.6 concludes our work and discusses possibilities for future research.

3.2 Preliminaries

The LearnLib mainly contains AAL algorithms for Deterministic Finite Automata (DFAs) and Mealy machines. We provide a definition for both, and a definition for Labeled Transition Systems (LTSSs) where multiple labels per edge are allowed. Typically, model checkers verify LTL properties on LTSSs. Hence we provide LTL semantics for LTSSs, and provide straightforward translations from DFAs and Mealy machines to LTSSs. Implementations of these translations have been added to the LearnLib. Furthermore, this section gives a short introduction to active learning, and learning-based testing.

¹This chapter is based on the journal article [MP19].

Definition 3.1: Edge Labeled Transition System

An edge Labeled Transition System (LTS) is defined as a tuple $\mathcal{L} = \langle S, s_0, \delta, L, T, \lambda \rangle$, where

- S is a finite nonempty set of states,
- $s_0 \in S$ is the initial state,
- $\delta : S \rightarrow 2^S$ is the transition function,
- L is the set of edge labels,
- T is the set of edge label types, and
- $\lambda : S \times S \times T \rightarrow L$ is the edge labeling function.

We use the shorthand notation $\lambda(s, s') = \{(t, \lambda(s, s', t)) \mid t \in T\}$ to obtain all edge labels of a transition. Furthermore, a *path* in $\mathcal{L}$ is an infinite sequence of states beginning in s_0 . The set of paths is $\text{Paths}(\mathcal{L}) = \{s_0 s_1 \dots \in S^\omega \mid \forall i > 0: s_i \in \delta(s_{i-1})\}$. A *trace* is an infinite sequence of sets of tuples of labels:

$$\text{Traces}(\mathcal{L}) = \{\lambda(s_0, s_1)\lambda(s_1, s_2) \dots \in (2^{T \times L})^\omega \mid s \in \text{Paths}(\mathcal{L})\}.$$

The set of *finite prefixes* of $\mathcal{L}$ is:

$$\text{Pref}(\mathcal{L}) = \{p \in (2^{T \times L})^* \mid \exists s \in (2^{T \times L})^\omega : ps \in \text{Traces}(\mathcal{L})\}.$$

Note that as a consequence of the definition of $\text{Traces}(\mathcal{L})$, prefixes that lead to a deadlock state are *not* in $\text{Pref}(\mathcal{L})$. Consequently, prefixes that lead to a deadlock state can never serve as counterexamples to LTL properties. These semantics are appropriate for the RERS challenge in the experimental section later.

Definition 3.2: Deterministic Finite Automaton

A Deterministic Finite Automaton (DFA) is defined as a tuple $\mathcal{D} = \langle S, s_0, \Sigma, \delta, F \rangle$, where

- S is a finite nonempty set of states,
- $s_0 \in S$ is the initial state,
- Σ is a finite alphabet,
- $\delta : S \times \Sigma \rightarrow S$ is the *total* transition function,
- $F \subseteq S$ is the set of accepting states.

The language of $\mathcal{D}$ is denoted $L(\mathcal{D})$. A DFA is *prefix-closed* iff $\forall s \in S, \forall i \in \Sigma: \delta(s, i) \in F \implies s \in F$. This implies that $\forall \sigma_1 \dots \sigma_n \in L(\mathcal{D}): \sigma_1 \dots \sigma_{n-1} \in L(\mathcal{D})$. The LTS associated to a non-empty, prefix-closed DFA $\mathcal{D}$ is $\mathcal{L}_{\mathcal{D}} = \langle F, s_0, \delta_{\mathcal{L}}, \Sigma, \{\text{letter}\}, \lambda_{\mathcal{L}} \rangle$, where:

- $\delta_{\mathcal{L}}(s) = \bigcup_{i \in \Sigma} \delta(s, i)$, and
- $\lambda_{\mathcal{L}}(s, s') = \{(letter, l) \mid l \in \Sigma \wedge \delta(s, l) = s'\}$.

Example 3.1: DFA

An example prefix-closed DFA for the regular expression $(ab)^*(a?)$ is given below (left). Its LTS is shown on the right. This LTS has only a single trace:

$$\{(letter, a)\}\{(letter, b)\}\dots$$

The set of finite prefixes is the set:

$$\{\{(letter, a)\}, \{(letter, a)\}\{(letter, b)\}, \dots\}.$$

Definition 3.3: Mealy Machine

A Mealy machine is defined as a tuple $\mathcal{M} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$, where

- S is a finite nonempty set of states,
- $s_0 \in S$ is the initial state,
- Σ is a finite input alphabet,
- Ω is a finite output alphabet,
- $\delta : S \times \Sigma \rightarrow S$ is the *total* transition function, and
- $\lambda : S \times \Sigma \rightarrow \Omega$ is the *total* output function.

The LTS of $\mathcal{M}$ is $\mathcal{L}_{\mathcal{M}} = \langle S, s_0, \delta_{\mathcal{L}}, \Sigma \cup \Omega, \{input, output\}, \lambda_{\mathcal{L}} \rangle$, where:

- $\delta_{\mathcal{L}}(s) = \bigcup_{i \in \Sigma} \delta(s, i)$, and
- $\lambda_{\mathcal{L}}(s, s') = \{\{(input, i), (output, o)\} \mid i \in \Sigma \wedge \delta(s, i) = s' \wedge o \in \Omega \wedge \lambda(s, i) = o\}$.

Example 3.2: Mealy Machine

An example Mealy machine is given below (left). Its LTS is shown on the right. The only trace of this LTS is:

$$\{(input, a), (output, 1)\} \{(input, a), (output, 2)\} \dots$$

Throughout this chapter the following assumptions are made.

- We only consider DFAs that reject the empty language (otherwise their LTS is not defined).
- We only consider prefix-closed DFAs (Mealy machines are by definition prefix-closed).
- We only consider minimal DFAs and Mealy machines (automata constructed through active learning are always minimal; our definition of prefix-closed only holds on minimal automata).
- We assume that the SUL is deterministic.

3.2.1 LTL Model Checking

An LTL formula expresses a property that should hold over all infinite runs of a system. This means that if a system does not satisfy an LTL property, there generally exists a counterexample that is an infinite word which exhibits a lasso structure.

Definition 3.4: LTL

Given an LTS $\mathcal{L} = \langle S, s_0, \delta, L, T, \lambda \rangle$, LTL formulae over $\mathcal{L}$ adhere to the following grammar:^a

$$\phi ::= \text{true} \mid \phi_1 \wedge \phi_2 \mid \neg \phi \mid \mathbf{X} \phi \mid \phi_1 \mathbf{U} \phi_2 \mid t = l,$$

where $t \in T$, and $l \in L$. Given an LTL formula ϕ , all infinite words that satisfy

ϕ are given by the set:

$$\text{Words}(\phi) = \{\sigma \in (2^{T \times L})^\omega \mid \sigma \models \phi\}.$$

The satisfaction relation $\models \subseteq (T \times L)^\omega \times \text{LTL}$ is defined inductively over ϕ by the following properties. Let $\sigma = A_0 A_1 A_2 \dots \in (2^{T \times L})^\omega$, and $\sigma[j] = A_j A_{j+1} A_{j+2} \dots$:

$$\sigma \models \begin{cases} \text{true}, \\ \phi_1 \wedge \phi_2 & \text{iff } \sigma \models \phi_1 \wedge \sigma \models \phi_2, \\ \neg \phi & \text{iff } \sigma \not\models \phi, \\ \mathbf{X} \phi & \text{iff } \sigma[1] \models \phi, \\ \phi_1 \mathbf{U} \phi_2 & \text{iff } \exists j: \sigma[j] \models \phi_2 \wedge \forall i < j: \sigma[i] \models \phi_1, \\ t = l & \text{iff } (t, l) \in A_0. \end{cases}$$

Furthermore, the set of *finite prefixes* of the words that satisfy ϕ is:

$$\text{Pref}(\phi) = \{p \in (2^{T \times L})^* \mid \exists s \in (2^{T \times L})^\omega: ps \in \text{Words}(\phi)\}.$$

We say that:

- $\mathcal{L}$ *satisfies* ϕ iff $\text{Traces}(\mathcal{L}) \subseteq \text{Words}(\phi)$, and
- ϕ *monitors* $\mathcal{L}$ iff $\text{Pref}(\mathcal{L}) \subseteq \text{Pref}(\phi)$.

Both “ $\mathcal{L}$ satisfies ϕ ” and “ ϕ monitors $\mathcal{L}$ ” can be checked with model checkers, so for clarity we refer to them as procedures named *model checking* and *monitoring* respectively. Note that for safety properties: $\mathcal{L}$ satisfies $\phi \iff \phi$ monitors $\mathcal{L}$. For arbitrary LTL properties we only have implication: $\mathcal{L}$ satisfies $\phi \implies \phi$ monitors $\mathcal{L}$. So we can use monitoring to check at least the “safety portion” of ϕ .

^aExtensions and equivalences may be defined as in [BK08] (such as implication: $\implies$, globally: G , and future: F).

The practical advantage of using monitoring instead of model checking is that monitoring provides finite counterexamples, whereas our definition of model checking requires infinite counterexamples, here represented as lassos.

Definition 3.5: Lasso

Given an LTS $\mathcal{L}$, a trace $\sigma \in \text{Traces}(\mathcal{L})$ is a *lasso* if it can be split in a finite *prefix* p and a finite *loop* q , such that $pq^\omega = \sigma$.

Example 3.3: LTL for DFAs

An example LTL formula that is satisfied by the LTS $\mathcal{L}$ in Example 3.1 is: $\phi = \mathbf{x}(\text{letter} = b)$. All the words that satisfy the formula are in the set:

$$\text{Words}(\phi) = \{ \{(letter, a)\}\{(letter, b)\} \dots, \{(letter, b)\}\{(letter, b)\} \dots \}.$$

Clearly, $\text{Traces}(\mathcal{L}) \subseteq \text{Words}(\phi)$, so $\mathcal{L}$ satisfies ϕ (and also ϕ monitors $\mathcal{L}$, because $\text{Pref}(\mathcal{L}) \subseteq \text{Pref}(\phi)$). Note that ϕ is a safety property, and if we make a small change: $\phi' = \mathbf{x}(\text{letter} = a)$, it can not be monitored. A *finite* counterexample then is $\{(letter, a)\}\{(letter, b)\}$. A lasso to show that ϕ' can not be satisfied is $\epsilon(\{(letter, a)\}\{(letter, b)\})^\omega$.

An example for Mealy machines is analogous. For checking whether a formula ϕ monitors, or is satisfied by an LTS $\mathcal{L}$, a traditional model checker can be used. It will perform the emptiness check $\text{Pref}(\mathcal{L}) \cap \overline{\text{Pref}(\phi)} = \emptyset$ and the emptiness check $\text{Traces}(\mathcal{L}) \cap \overline{\text{Words}(\phi)} = \emptyset$, respectively. Following the automata based approach, this is done by computing the Cartesian product of their respective automata representations. A monitor automaton can recognize $\text{Pref}(\phi)$ and a Büchi automaton can recognize $\text{Traces}(\phi)$. Note that instead of performing emptiness checks, one can also perform the original inclusion check ($\text{Pref}(\mathcal{L}) \subseteq \text{Pref}(\phi)$) by checking the invariant that all reachable states in the monitor are accepting. We used LTSmin [Kan+15] to implement the inclusion check and emptiness check. Spot [Dur+16] is used to compute monitors and Büchi automata.

3.2.2 Active Learning

For our purposes, active learning is the process of learning a sequence of hypotheses $\mathcal{H}_1 \mathcal{H}_2 \dots \mathcal{H}_F$, such that their behavior converges to some target automaton (DFA, or Mealy machine). The key components are illustrated in Figure 3.1.

Learner: an algorithm that can form hypotheses based on queries and counterexamples. A learning algorithm will pose queries until it has *complete* information to form a hypothesis. The notion of completeness depends on the applied learning algorithm, e.g. the L^* algorithm will perform queries until its observation table is closed and consistent.

Equivalence oracle ($=$): an oracle that decides whether two languages are equal. The oracle decides between the language of the current hypothesis of the learner, and the language of the SUL. If the languages are not equivalent the oracle will provide a counterexample that distinguishes both languages. Here, the language of the SUL is a set of finite traces. Note that the distinguishing word is either in the language of the hypothesis, or the language of the SUL, but not both.

Figure 3.1: Active learning procedure

Membership oracle ($\in$): an oracle that decides whether or not a word is a member of the language of the SUL.

SUL: In the case an active learning algorithm is applied to an actual system, a SUL interface is used that can *step* through a system in order to answer membership queries. In the LearnLib, the SUL interface exposes the methods **pre** and **post** that can reset a system (i.e. put it back to the initial state), **step** that stimulates the system with one input symbol and returns the corresponding output, **canFork** and **fork** that may fork a SUL, i.e. provide some copy (that behaves identically to) the forked system. In active learning, this copy is used to pose queries in parallel. We will show its usefulness for performing state equivalence checks in the context of LBT too. More information about the SUL interface can be found in Figure 3.4 on page 47.

Example 3.4: Active Learning

Given an alphabet $\Sigma = \{a, b\}$, and a DFA $\mathcal{D}$ to be learned such that $L(\mathcal{D}) = (ab)^*(a?)$ (Example 3.1), an active learning algorithm could first produce the hypothesis $\mathcal{D}_1$ shown below on the left, where the language accepted is $L(\mathcal{D}_1) = a^*$. At some point the equivalence oracle generates $aa \in \Sigma^*$, and performs the membership query $aa \in \mathcal{L}?$ = no. The equivalence oracle recognizes that $aa \in L(\mathcal{D}_1)$, and concludes it found a counterexample to $\mathcal{D}_1$. The learner re-

finds $\mathcal{D}_1$, and produces the final hypothesis shown below on the right and we are done learning. Note that this example hides the complexity of actually refining the hypothesis. In the LearnLib refining a hypothesis is done with the method `Learner#refineHypothesis()` that accepts a query (counterexample) and subsequently poses additional membership queries. More details on refining hypotheses are outside of the scope of this chapter; they can be found in e.g. [Ang87, SHM11].

Hypothesis $\mathcal{D}_1$: Hypothesis $\mathcal{D}_F$:

Finding a counterexample to the current hypothesis by means of an equivalence oracle can be very time-consuming. In the worst-case the equivalence oracle has to try out all words of maximum length N in $\Sigma^{0 \dots N}$. Some smart equivalence oracles (e.g. ones using the partial W-method [Fuj+91]) can find a counterexample quite quickly, if there is one. However, the number of membership queries to find the counterexample is still orders of magnitudes larger than the size of the hypothesis. E.g. any word of maximum length $N = 2$ that could serve as a counterexample for the first hypothesis in Example 3.4 is in $\{\epsilon, a, b, aa, ab, ba, bb\}$. When hypotheses grow larger, the set of possible counterexamples grows with an even larger degree. Learning-based testing alleviates this problem by using LTL properties to restrict the search space of such counterexamples.

3.2.3 Learning-Based Testing with Model Checking

The sound approach to learning-based testing is illustrated in Figure 3.2; it builds upon the active learning algorithm. The difference however is that hypotheses are not immediately forwarded to an equivalence oracle, but first subjected to a model checker to check them against a number of predefined LTL formulae.

When the model checker provides a counterexample (ce) to a property, then ce needs to be confirmed on the SUL. If the counterexample can be simulated on the system, we found a violation of the property. If not, a prefix of the counterexample (w) is provided to the learner, saving one expensive equivalence check.

A complication is that the counterexample provided by the model checker is an

Figure 3.2: Sound learning-based testing procedure [PM18]

infinite trace, presented by a lasso xy^ω . In principle, one can only check finite unrollings xy^n on the system. However, this yields an unsound method, unless one knows an upper bound on the number of states of the SUL.

An initial sound approach to learning-based testing was proposed in [MP18]. Membership of infinite words ($\in^\omega$) is checked, by assuming that one can additionally save states of the SUL and check their equivalence. So we test the word and save intermediate states $x(s_0)y(s_1)y(s_2), \dots, y(s_n)$. As soon as we find that $s_k = s_j$ for some $0 \leq k < j \leq n$, we definitely know that xy^ω is a valid counterexample, and report a violation. If the path can not be continued, we have found a finite prefix w for the learner. Otherwise, we don't know (dk) if xy^ω holds, and we proceed to the tester.

The adapted procedure is sound, in the sense that it only reports true violations. However, it may miss some violations, so it is incomplete: first, the final hypothesis may still not reflect all system behavior. Second, the model checker may have detected a lasso that could not be confirmed within the bound.

This work presents an additional sound approach in Section 3.3.4. Before checking whether lassos are accepted, we first apply monitoring. If monitoring reveals that the safety portion of the property does not hold on the hypothesis, there exists a finite counterexample that needs to be confirmed on the SUL. Naturally, this is much more practicable than having to check lasso-like counterexamples for every property.

Example 3.5: Learning-Based Testing

Consider again the first hypothesis $\mathcal{D}_1$, produced by an active learning algorithm from Example 3.4, that accepts the language a^* , and the LTL formula $\phi = \mathbf{x}(\text{letter} = b)$, from Example 3.3. An LTL model checker checks whether the LTS of $\mathcal{D}_1$ satisfies formula ϕ ($\mathcal{L}_{\mathcal{D}_1} \models \phi$). The model checker concludes $\mathcal{L}_{\mathcal{D}_1}$ does not satisfy ϕ , and produces the lasso a^ω as a counterexample. The lasso from the model checker is verified on the SUL by means of the membership oracle that can perform omega queries, i.e. $\in^\omega$ performs the query $a^\omega \in L^\omega?$ = no. This means that with hypothesis $\mathcal{D}_1$ the property can not be disproved. Let us assume that $\in^\omega$ unrolled a^ω twice when performing the membership query, then $aa \notin L(\mathcal{D}_1)$ is provided as a counterexample to the learner.

In practice, the number of times the loop of the lasso is unrolled depends on the size of the hypothesis. The essence of the current example is that the first hypothesis in Example 3.4, can be refined without performing any equivalence query. This example (like Example 3.4 about active learning) hides the complexity of refining a hypothesis too. Refining a hypothesis in the LearnLib in the

context of LBT can also be done with `Learner#refineHypothesis()`.

3.3 Sound Learning-Based Testing in the LearnLib

This section provides the detailed description and some extensions of our sound LBT approach introduced in [MP18], including the LBT algorithm and its integration in the LearnLib [IHS15]. The main new contributions are an extension of sound LBT with finite counterexamples from monitors, the description of various strategies to interleave property checking and hypothesis refinement, and a detailed overview of the design in LearnLib's API.

3.3.1 Learning-Based Testing in the LearnLib

We implemented a more general procedure for learning-based testing in the LearnLib than presented in Section 3.2.3. First, the result from the model checker is generalized to a language representing a set of counterexamples to a property, instead of just a single word. This follows from our observation that some counterexamples returned by the model checker are uninformative or spurious, since hypotheses may be incorrect. So instead of a single membership query to validate counterexamples from the model checker an *emptiness oracle* checks whether the intersection between the counterexample language returned by the model checker and the language of the SUL is empty. If it is not empty, an example in the intersection serves as a counterexample to the property. Secondly, instead of checking just a single property, our extension to the LearnLib can check a set of properties. This check with properties is implemented as a loop, in which more and more properties are disproved. Having to check a set of properties gives rise to multiple strategies. One strategy tries to disprove as many properties as possible, before providing a counterexample to refine the current hypothesis. The other strategy tries to disprove a single property; if it can not be disproved it tries to refine the current hypothesis, before continuing with the next property. In Section 3.3.6 we will detail how the user of the LearnLib can pick either of those two strategies, and in Section 3.5 we will investigate the efficiency of these strategies. The following components were added to the LearnLib:

Model checker ($\models$): an algorithm that checks whether an LTL property is satisfied or is monitored by a hypothesis. If the check fails the result is a set of counterexamples, which is in fact a subset of the language of the checked hypothesis.

Emptiness oracle ($\emptyset$): an oracle that decides whether the intersection of two languages is empty. The oracle decides between the language of the counterexamples given by the model checker, and the language of the SUL. If the intersection is not empty it will provide a counterexample, which is a word in the intersection and as

such, a counterexample to the property checked by the model checker.

Inclusion oracle ($\sqsubseteq$): an oracle that decides whether one language is included in another. The oracle decides whether the language of the counterexamples given by the model checker is included in the language of the SUL. If the language is not included, the oracle will provide a counterexample outside the language of the SUL, and thus a counterexample to the current hypothesis. One can view the combination of the *model checker*, *emptiness oracle*, and *inclusion oracle* as a *property oracle*, and a set of property oracles as a *black-box oracle*.

3.3.2 New Purposes for Queries

In traditional active learning there are two kinds of membership queries; *learning queries* (by the learner) and *equivalence queries* (by the equivalence oracle). With LBT there are two more types of queries; *inclusion queries* (done by the inclusion oracle), and *emptiness queries* (done by the emptiness oracle). The decision between performing inclusion queries and emptiness queries depends on whether the property can be falsified with the current hypothesis. We generalize both to *model checking queries*. The key observation why adding properties to verify to the learning algorithm can be useful, follows from the observation that model checking queries are very cheap compared to equivalence queries. In other words, given an alphabet Σ , a naive equivalence oracle has to perform arbitrary membership queries for words in Σ^* , while the black-box oracle has to perform only membership queries for a subset of the language of the current hypothesis.

3.3.3 The LBT Algorithm and Strategies — Informally

Now, given that model checking queries are much cheaper than equivalence queries we present the learning-based testing algorithm in the LearnLib in Figure 3.3. Note that black numbers (e.g. ❷) represent learning-based testing steps, and white numbers (e.g. ②) active learning steps. Also note that steps can have multiple alternatives (e.g. ❶). The LBT procedure is now as follows. Initially (❶) the learner constructs an hypothesis $\mathcal{H}$ using membership queries (❷). This hypothesis is, together with a set of properties $P_1 \dots P_n$, given to the model checker (❷). If the model checker finds counterexamples for a property and the current hypothesis, then the counterexamples (CEs) are given to the emptiness oracle (❸). The emptiness oracle performs membership queries (❹) on the SUL to try to find a counterexample from the model checker that is not spurious. If a real counterexample for a property is found, it is reported to the user (❺), and the property is not considered for future hypotheses. Otherwise, to identify spurious counterexamples, the set of counterexamples is forwarded to the inclusion oracle. The inclusion oracle performs membership queries (❻) to find a counterexample for the

Figure 3.3: Learning-based testing algorithm in the LearnLib

current hypothesis (⑦). This is given back to the learner, which continues performing membership queries (②) to complete the next hypothesis. If the hypothesis is refined, the black-box oracle repeats steps (②, ..., ⑦) until the model checker can not find any new counterexample. In the latter case we enter the traditional active learning loop (cf. steps ②, ..., ⑤ and Figure 3.1): the equivalence oracle tries to find a counterexample for the current hypothesis (③) using membership queries (④). If a counterexample is found (⑤) the learner will construct the next hypothesis using membership queries (②) and the black-box oracle is put back to work. If the equivalence oracle does not find a counterexample (④) the final hypothesis is reported to the user.

We can now better illustrate the two black-box oracle strategies. Either, the black-box oracle can first try to find a counterexample for every property before finding a refinement for the current hypothesis. Or, it finds a counterexample for a single property and if such a counterexample does not exist, it searches for a counterexample to the current hypothesis, before checking the next property. The first implementation is more efficient if there is a high chance a property can be disproved with the current hypothesis, or if refining the current hypothesis becomes quite expensive. The experimental section later shows that the first strategy generally works better when applied to the RERS challenge.

3.3.4 Learning-Based Testing with Monitoring

In traditional active learning the concept of a *query* provides the main interface between the learner or equivalence oracle and the SUL. In this work, the emptiness oracle also uses queries to validate counterexamples from the model checker. A *query* is denoted as an input word $q \in \Sigma^*$ that can be answered by a membership oracle.

Definition 3.6: Membership oracle

Given a set of queries Q , the set of Booleans $\mathbb{B} = \{\perp, \top\}$, and a SUL S that is an unknown DFA, a *membership oracle* is a function:

$$\in: Q \rightarrow \mathbb{B}, \text{ such that } \in(q) \iff q \in L(S).$$

A membership oracle for Mealy machines can be defined similarly, see [SG09].

Example 3.6: Answering a query

Consider the example safety property $\phi = (\text{letter} = b)$. Then ϕ does not hold on the LTS $\mathcal{L}$ associated to the final DFA $\mathcal{D}_F$ in Example 3.4. When the model

checker checks whether ϕ monitors $\mathcal{L}$ it could provide the singleton language $\{a\}$ as a counterexample to ϕ . The emptiness oracle will ask the membership oracle to answer the query $q = a$ so that the counterexample can be validated. Since a is accepted by the SUL (i.e. $a \in L(S)$) the membership oracle will answer $\in(q) = \top$, and the emptiness oracle will report the answered query as a valid counterexample to ϕ .

Practically, and in case of monitoring, the product of an LTS $\mathcal{L}$ and a monitor automaton of $\overline{\text{Pref}(\phi)}$ is computed, while checking for a witness that visits an accepting state once. Finding such a witness can be achieved on-the-fly with any reachability algorithm. In this work the automaton accepting the language of the formula is created by Spot [Dur+16], while computing the product with the LTS and search for witness is done by LTSmin [Kan+15].

3.3.5 Learning-Based Testing with Model Checking

In case we want to know whether an LTS satisfies a formula ϕ , the product of an LTS and a Büchi automaton of $\text{Words}(\phi)$ is computed, while checking for a witness that visits an accepting state in the Büchi automaton infinitely often. Searching for such a witness can be done on-the-fly [Cou99] with (concurrent) nested depth-first search [Cou+92, Laa+11] or SCC-based approaches [Tar72, BP16].

Making the LBT procedure sound involves checking whether infinite lasso-shaped words given as counterexamples by the model checker are accepted by the SUL. Obviously, checking whether a SUL accepts an infinite word in practice is impossible. However, this can be resolved if one considers what goes on inside a black-box system. We need to check if the SUL also exhibits a particular lasso through its state space when stimulated with a finite word (that also produces the same output as given by the model checker). This can be achieved by observing particular states the SUL evolves through when stimulated. Note that this view of a SUL is still quite a black-box view; we only record the states, we do not *enforce* the SUL to move to a particular state.

We introduce a new notion of a *query*, named an ω -*query*, which in addition to the input word and output of the SUL also contains the periodicity of the loop of the lasso. An ω -query serves as the interface between the emptiness oracle and SUL.

Definition 3.7: ω -query

Given an alphabet Σ , an ω -query is a tuple $q^\omega = (p, l, r) \in \Sigma^* \times \Sigma^+ \times \mathbb{N}$, where

- p is the *prefix* of the lasso to check,
- l the *loop* of the lasso to check,

- $r \geq 1$ the maximum number of times the loop may be unrolled.

Following Definition 3.6, an ω -membership oracle is used to answer ω -queries and is defined as follows.

Definition 3.8: ω -membership oracle

Given a set of ω -queries Q^ω over an alphabet Σ , a SUL S that is an unknown DFA with a set of internal states Z , and a function $\text{States}_S: \Sigma^* \rightarrow Z$, such that $\text{States}_S(\sigma) = z$ gives the internal state z in S after input σ , an ω -membership oracle is a function:

$\in^\omega: Q^\omega \rightarrow \mathbb{N}$, such that $\in^\omega(q^\omega = (p, l, r)) = n$, where

- $r \geq n$ indicates the periodicity of l , and
- $n > 0 \iff \exists i < n: \text{States}_S(pl^i) = \text{States}_S(pl^n) \wedge pl^n \in L(S)$.

We can also illustrate an answered ω -query with $n > 0$ as:

$$p \xrightarrow{\cdot} \cdot \xrightarrow{\cdot} \cdot \xrightarrow{\cdot} \cdot \xrightarrow{\cdot} \cdot \xrightarrow{\cdot} s \xrightarrow{\cdot} \cdot \xrightarrow{\cdot} \cdot \xrightarrow{\cdot} s', \text{ where } s = s'.$$

$\overbrace{\hspace{10em}}^n$
 $\underbrace{\hspace{4em}}_i$

So in case $n > 0$ we generalize $pl^n \in L(S)$ to $pl^\omega \in L^\omega(S)$, because $pl^i(l^{n-i})^\omega = pl^\omega$. A definition for an ω -membership oracle for Mealy machines is similar. For learning a Mealy machine with output alphabet Ω an ω -membership oracle is a function $\in^\omega: Q^\omega \rightarrow \Omega^+ \times \mathbb{N}$, such that $\in^\omega(q^\omega = (p, l, r)) = (o, n)$ and iff $n > 0$, o is the output string of input pl^n . In Section 3.3.6 we will explain how the above state equivalence check is implemented in the LearnLib. We will also detail how checking for state equivalence is done *on-the-fly* when states can be serialized.

Example 3.7: Answering an ω -query

An example property that does not hold on the LTS $\mathcal{L}$ associated to the final DFA in Example 3.4 is $\phi = (\text{letter} = b)$. Whenever a model checker determines whether $\mathcal{L}$ satisfies ϕ , it may give the lasso $\ell = a(ba)^\omega$ as a potential counterexample to ϕ . The language $\{\ell\}$ is given to the emptiness oracle, with a limit to unroll the loop at most 3 times for instance, and asks the ω -membership oracle to answer the query $q^\omega = (a, ba, 3)$. When stimulating the SUL with $a(ba)^1$, the SUL cycles through state s_1 . Hence, the ω -membership oracle will answer $\in^\omega(q^\omega) = 1$. From the answer to the query, it becomes clear that the SUL accepts the infinite lasso-shaped word ℓ with periodicity 1, and the emptiness oracle will

report the answered ω -query as a valid counterexample to ϕ . In practice the LearnLib will determine the maximum number of unrolls relative to the number of states in the hypothesis automaton on which ϕ is checked.

3.3.6 The New API in the LearnLib

Figure 3.4: LearnLib API: classes are red, interfaces are blue, LBT extensions are darker. Solid arrows depict interface refinements, and dashed arrows implementations.

We extend the API (Application Programmer’s Interface) of the LearnLib following Figure 3.3 on page 43. Among others we add the concept of a model checker ($\models$) as an interface named `ModelChecker` and similar goes for inclusion oracles ($\subseteq$) and emptiness oracles ($\emptyset$). Other new first-class citizens are derived

from Definitions 3.7 and 3.8 and named accordingly; these establish the form of communication between the emptiness oracle and system. This form of communication is similar to the interface between the learner and system in the original AAL setting.

LearnLib’s API extension is illustrated in a class diagram in Figure 3.4. Note that for illustration purposes we do not show association arrows between classes, instead associations are represented as class attributes. If an association has a multiplicity greater or equal to zero, we suffix the attribute with the array notation “[]”. Furthermore all attributes of a class indicate required parameters of its constructor. A description of the new API is as follows.

ObservableSUL: The SUL interface is extended with methods `getState() : S` returning the current state of the SUL, `boolean deepCopies()` indicating whether the object returned by `getState()` is a deep copy, and a refinement of `fork()`.

ModelChecker: A `ModelChecker` may find a counterexample to a property and hypothesis. The counterexamples are finite words and are a subset of the language of the hypothesis. `LTSmin` [Kan+15, BP16] is an available implementation of a `ModelChecker` for Monitors in the LearnLib. A `ModelCheckerLasso` is a refinement of a `ModelChecker` that uses Büchi automata and where the counterexamples are lassos instead of finite words. The implementations are named `LTSminMonitor` and `LTSminLTL` respectively.

OmegaQuery: An `OmegaQuery` following Definition 3.7 is similar to a `Query`. An answered `OmegaQuery` stores information about whether an infinite word is in the language of the SUL. Note that type `D` denotes the output *domain* type for different automaton types that are learned.

EmptinessOracle: This oracle generates words that are in a given automaton, and tests whether those words are also in the SUL. The implementation `BfEmptinessOracle`, generates words in a breadth-first manner. A limit can be placed on the maximum number of words by supplying a *multiplier* that is used to limit the number of queries performed. This limit is computed by multiplying the size of the given automaton by the specified *multiplier*. An `EmptinessOracle` is used by `PropertyOracles` to check whether any word in the language given as a counterexample by the `ModelChecker` is present in the SUL. A specialization of an `EmptinessOracle` is a `LassoEmptinessOracle` that uses `OmegaQueries` to check whether infinite lasso-shaped words are not in the SUL.

InclusionOracle: Similar to the `EmptinessOracle`; it generates a limited number of words in a breadth-first manner, but checks whether words are *in* the language of the SUL. Note that both of these oracles may perform the same queries; this is a practical issue and is usually resolved by using a `SULCache` so that in case of a cache-hit the SUL is not stimulated. The `InclusionOracle`, and `EmptinessOracle`

may have different strategies (BFS vs. DFS), and hence are not merged together into a single oracle. Separation of concerns (finding a counterexample to the current hypothesis, vs. finding a counterexample to a property), is also considered a good design principle.

PropertyOracle: A **PropertyOracle** is an oracle for a property of a black-box system. Such an oracle tries to *disprove* the property or finds a *counterexample* to the current hypothesis. To this end, implementations require a **ModelChecker**, **EmptinessOracle**, **InclusionOracle**, and the property itself. In case the property should be model checked with a monitor one should construct a **FinitePropertyOracle**, in case the property should be checked with a Büchi automaton one should construct a **LassoPropertyOracle**.

BlackBoxOracle: An oracle that disproves a set of properties by means of multiple **PropertyOracles**, or finds a counterexample to the current hypothesis in the same collection of **PropertyOracles**. Currently, there are two implementations available.

1. **DisproveFirstOracle:** Iterates over the set of properties that are still unknown, and tries to disprove all of them before refining the current hypothesis.
2. **CExFirstOracle:** This implementation iterates over the set of properties that are still unknown, and before disproving a next property it first tries to refine the current hypothesis with the current property.

Both implementations execute a loop, trying to disprove as many properties as possible. The pseudocode of these strategies is provided in Algorithms 3.1 and 3.2. Both implementations will be evaluated later by the experiments in Section 3.5.

OmegaMembershipOracle: An oracle that decides if an infinite word is in the language of the SUL, see Definition 3.8. To this end it poses **OmegaQueries**. There are several implementations available; one that simulates DFAs and Mealy machines directly, and one that wraps around an **ObservableSUL**, by means of **ShallowCopySULOmegaOracle** and **DeepCopySULOmegaOracle**. Based on the implementation of **ObservableSUL#deepCopies()** one can decide to construct a **ShallowCopySULOmegaOracle** or a **DeepCopySULOmegaOracle**. If an **ObservableSUL** does not make a deep copy of the state of the SUL it could be the case that when **SUL#step()** is executed, a previously obtained state with **ObservableSUL#getState()** would also be modified, e.g. the assertion in the following Java snippet may not hold.

```

1 ObservableSUL oSUL = ...;
2 S s = oSUL.getState();
3 int hc = s.hashCode();
4 oSUL.step(...);
5 assert s.hashCode() == hc;

```

To resolve this; if `ObservableSUL#deepCopies()` does not hold, then `SUL#forkable()` must hold. `ShallowCopySULOmegaOracle` will use two instances of a `ObservableSUL`, i.e. one regular instance, and a forked instance to compare two states. More specifically, a `ShallowCopySULOmegaOracle` in fact uses hash codes of states, and if the hash codes of two states are equal, then it will step one instance of the `ObservableSUL` through the access sequence of one state, and the forked instance of the `ObservableSUL` through the access sequence of the second state. When both `ObservableSULs` are in the desired state `OmegaMembershipOracle#isSameState()` is issued to check if the queried word is in fact a lasso.

In case `ObservableSUL#deepCopies()` *does* hold, checking equality of two states is straightforward. `DeepCopySULOmegaOracle` simply invokes `Object.equals()` on the two states.

Concluding this section; we want to show how conveniently one can set up a LBT experiment with the class `Experiment`. This class needs to be constructed with a `Learner` implementation (e.g. TTT, ADT, etc.) And a *chain* of `EquivalenceOracles`. In our experiment, this chain has two elements: first a `BlackBoxOracle` and then a more complete `EquivalenceOracle`, for instance one that applies the W-method or tries random words. Listing 3.1 shows exactly how the running example can be implemented in the LearnLib using the chain of `EquivalenceOracles`. In Section 3.5 we show how one can learn a Mealy machine by implementing LearnLib's SUL interface.

Listing 3.1: API usage skeleton for learning-based testing

```

1 // define the alphabet
2 Alphabet sigma = Alphabets.characters('a', 'b');
3 // create the running example DFA
4 DFA dfa = AutomatonBuilders.newDFA(sigma).
5     withInitial("q0").withAccepting("q0").withAccepting("q1").
6     from("q0").on('a').to("q1").from("q1").on('b').to("q0").create();
7 // create an omega membership oracle, that simulates the DFA
8 OmegaMembershipOracle oMO = new SimulatorOmegaOracle(dfa);
9 // create a regular membership oracle
10 MembershipOracle mO = oMO.getMembershipOracle();
11 // create an equivalence oracle that uses the partial W-method
12 EquivalenceOracle eqO = new WpMethodEQOracle(3, mO);
13 // create a TTT learner
14 Learner learner = new TTTLearnerDFA(sigma, mO, LINEAR_FWD);
15 // create an inclusion oracle, with multiplier 1.0
16 InclusionOracle inO = new DFABFInclusionOracle(mO, 1.0);
17
18 /*****
19 *** insert code here from one of the next two Listings ***/
20 *****/
21

```

```

22 // modify the equivalence oracle by prepending the black-box oracle
23 eq0 = new EQOracleChain(bbo, eq0);
24 // create an experiment
25 Experiment e = new Experiment(learner, eq0, sigma);
26 // run the experiment
27 DFA finalHypothesis = e.run();
28 // assert we have the correct result
29 assert findSeparatingWord(dfa, finalHypothesis, sigma) == null;

```

Listing 3.2: API usage for learning-based testing with *model checking*, and the **DisproveFirstOracle** strategy

```

1 // create a parser that translates data between LTSmin and the
  LearnLib
2 Function<String, Character> s2c = s -> s.charAt(0);
3 // create an LTSmin model checker
4 ModelCheckerLasso checker = new
  LTSminLTLDFABuilder().withString2Input(s2c).create();
5 // create an emptiness oracle for lassos
6 LassoEmptinessOracle emO = new DFALassoEmptinessOracle(oMO);
7 // create the black-box property from the running example
8 PropertyOracle po = new DFALassoPropertyOracle("X letter==\"b\"",
  inO, emO, checker);
9 // create the black-box oracle with the singleton set of properties
10 BlackBoxOracle bbo = new DisproveFirstOracle(po);

```

Listing 3.3: API usage for learning-based testing with *monitoring*, and the **CExFirstOracle** strategy

```

1 // create a parser that translates data between LTSmin and the
  LearnLib
2 Function<String, Character> s2c = s -> s.charAt(0);
3 // create an LTSmin model checker
4 ModelChecker checker = new
  LTSminMonitorDFABuilder().withString2Input(s2c).create();
5 // create an emptiness oracle
6 EmptinessOracle emO = new DFABFEmptinessOracle(mO);
7 // create the black-box property from the running example
8 PropertyOracle po = new DFAFinitePropertyOracle("X letter==\"b\"",
  inO, emO, checker);
9 // create the black-box oracle with the singleton set of properties
10 BlackBoxOracle bbo = new CExFirstOracle(po);

```

3.3.7 The Algorithms — Formally

Java code Listings 3.1 to 3.3 show how one can set up learning-based testing experiments in the LearnLib. The code illustrates how one can setup monitoring

with the `CExFirstOracle`, and how to setup model checking with the `DisproveFirstOracle`. The code `Experiment#run()` in Listings 3.1 and 3.2 will run exactly the algorithm presented in Algorithm 3.1, while `Experiment#run()` in Listings 3.1 and 3.3 runs exactly the algorithm presented in Algorithm 3.2. Generics are omitted to improve the presentation of the code listings. The input to both algorithms is a set of LTL properties, an alphabet, and membership oracle(s). Additionally, Algorithm 3.1 requires a *multiplier* to restrict the maximum number of unrolls of each lasso. Note that when we write $|H|$, we mean the size of automaton H , i.e. the number of states in H . The output of both algorithms is the final hypothesis learned. Additionally, both algorithms will report which properties have been disproved, and what the counterexamples to those properties are.

Algorithm 3.1: Learning-based testing with *model checking*, `DisproveFirstOracle`, TTT, and partial W-method

Input : set of LTL properties P , alphabet Σ , membership oracle $\in$,
 ω -membership oracle $\in^\omega$, and multiplier M

Output: final hypothesis H

```

1   $P' \leftarrow \emptyset$                                 ▷ initialize previous set
2   $H \leftarrow \text{TTT-INIT}(\Sigma, \in)$            ▷ initialize hypothesis with the TTT learning algorithm
3  while  $P' \neq P$  do                             ▷ least fixed-point loop
4       $P' \leftarrow P$                              ▷ save set of properties to prove
5      for  $\phi \in P$  do                             ▷ try to disprove  $\phi$ 
6           $\Omega \leftarrow \text{Traces}(\mathcal{L}_H) \cap \overline{\text{Words}(\phi)}$            ▷ model check  $\phi$ 
7          for  $pq^\omega \in \Omega$  do                     ▷ do emptiness check
8              for  $1 \leq i \leq |H| * M$  do             ▷ compute number of unrolls  $i$ 
9                   $q^\omega \leftarrow (p, q, |H| * M)$            ▷ create the  $\omega$ -query
10                 if  $\in^\omega(q^\omega) > 0$  then           ▷ answer  $q^\omega$  and test if ultimately periodic
11                     REPORT( $\phi, pq^\omega$ )           ▷ report  $pq^\omega$  as a counterexample to  $\phi$ 
12                      $P \leftarrow P \setminus \{\phi\}$        ▷ remove  $\phi$  so that it is not checked again
13                     goto Line 5                 ▷ continue with next property
14             for  $\phi \in P$  do                             ▷ try to disprove  $H$ 
15                  $\Omega \leftarrow \text{Traces}(\mathcal{L}_H) \cap \overline{\text{Words}(\phi)}$            ▷ model check  $\phi$ 
16                 for  $pq^\omega \in \Omega$  do                     ▷ do inclusion check
17                      $q \leftarrow pq^{|H|*M}$              ▷ create the query
18                     if  $\in(q) = \perp$  then                 ▷ answer  $q$  and test if  $q$  can refine  $H$ 
19                          $H \leftarrow \text{TTT-REFINE}(H, \Sigma, \in, q, \perp)$            ▷ refine  $H$  with  $q$ 
20                         goto Line 3                 ▷ continue with the next hypothesis
21  $(q, b) \leftarrow \text{wp-EQUIV}(H, \Sigma, \in)$            ▷ do equivalence check with the partial W-method
22 if  $b \neq q \in L(H)$  then                             ▷ test if query  $q$  with answer  $b$  can refine  $H$ 
23      $H \leftarrow \text{TTT-REFINE}(H, \Sigma, \in, q, b)$            ▷ refine  $H$  with  $q$  and  $b$ 
24     goto Line 3                                     ▷ continue with the next hypothesis
25 return  $H$ 

```

Algorithm 3.2: Learning-based testing with *monitoring*, CExFirstOracle, TTT, and partial W-method

Input : set of LTL properties P , alphabet Σ , membership oracle $\in$

Output: final hypothesis H

```

1   $P' \leftarrow \emptyset$                                  $\triangleright$  initialize previous set
2   $H \leftarrow \text{TTT-INIT}(\Sigma, \in)$            $\triangleright$  initialize hypothesis with the TTT learning algorithm
3  while  $P' \neq P$  do                             $\triangleright$  least fixed-point loop
4       $P' \leftarrow P$                                  $\triangleright$  save set of properties to prove
5      for  $\phi \in P$  do                             $\triangleright$  iterate over all properties
6           $T \leftarrow \text{Pref}(\mathcal{L}_H) \cap \overline{\text{Pref}(\phi)}$                  $\triangleright$  monitor  $\phi$ 
7          for  $q \in T$  do                             $\triangleright$  do emptiness check
8              if  $\in(q) = \top$  then                     $\triangleright$  answer query  $q$  and test if  $q$  falsifies  $\phi$ 
9                   $\text{REPORT}(\phi, q)$                  $\triangleright$  report  $q$  as a counterexample to  $\phi$ 
10                  $P \leftarrow P \setminus \{\phi\}$          $\triangleright$  remove  $\phi$  so that it is not checked again
11                 goto Line 5                         $\triangleright$  continue with next property
12             for  $q \in T$  do                             $\triangleright$  do inclusion check
13                 if  $\in(q) = \perp$  then                 $\triangleright$  answer  $q$  and test if  $q$  can refine  $H$ 
14                      $H \leftarrow \text{TTT-REFINE}(H, \Sigma, \in, q, \perp)$      $\triangleright$  refine  $H$  with  $q$ 
15                     goto Line 3                     $\triangleright$  continue with the next hypothesis
16  $(q, b) \leftarrow \text{wp-EQUIV}(H, \Sigma, \in)$      $\triangleright$  do equivalence check with the partial W-method
17 if  $b \neq q \in L(H)$  then                         $\triangleright$  test if query  $q$  with answer  $b$  can refine  $H$ 
18      $H \leftarrow \text{TTT-REFINE}(H, \Sigma, \in, q, b)$      $\triangleright$  refine  $H$  with  $q$  and  $b$ 
19     goto Line 3                                 $\triangleright$  continue with the next hypothesis
20 return  $H$ 

```

To be more complete, we compare the original sketch of the LBT algorithm in Figure 3.3 on page 43 to executions of Algorithm 3.2 (Algorithm 3.1 is simply a variant). To start the *Learner* component uses the alphabet (①) and membership oracle ($\in$) to initialize the first hypothesis by answering membership queries (②) on line 2. The main loop on line 3 covers all steps in Algorithm 3.2 except returning the final hypothesis (⑤). In the main loop we iterate over (line 5) every property (②) that has not been disproved. This second iteration over properties consists of three parts. The first part at line 6 involves checking whether a property (ϕ) can be monitored by means of the model checker ($\models$). If the property can not be monitored a non-empty language (i.e. action ③ and variable T) is provided in which every word is a counterexample to ϕ . The second part (lines 7–11) covers the emptiness check that involves the emptiness oracle ($\emptyset$) to disprove properties. The role of the emptiness oracle is to decide no word in T is accepted by the system (④). When this can not be established (the intersection of T and the language of the system is not empty) a counterexample q is reported as a counterexample to ϕ . This is illustrated as output of the algorithm in step ⑤. The third part (lines 12–15) covers the alternative to step ⑤ where the emptiness of the intersection is established. The inclusion check involves the inclusion oracle ($\subseteq$) and its role is to disprove hypotheses. To this end, it decides whether every word in T is included in the language of the system by means of membership queries (⑥). When however, there is a word ($q \in T$) that is not included in the language of the system, the hypothesis is disproved and is refined with q at line 14 and illustrated in step ⑦. The remainder is the traditional equivalence check illustrated with steps ③–⑤ and executed at lines 17–19. This check is executed when there is no property that is able to disprove the current hypothesis.

The crucial difference between Algorithms 3.1 and 3.2 is the concept of a **CEx-FirstOracle** versus a **DisproveFirstOracle**. The first oracle admits an immediate alternative at step ⑤. At this step, if a property is disproved but no counterexample is accepted by the system, the language (i.e. the set of counterexamples) is given to the inclusion oracle which will find a counterexample to refine the hypothesis. A **DisproveFirstOracle** on the other hand will first try to disprove every property before refining the hypothesis. Experimental results later show that it is better to use a **DisproveFirstOracle** since it reduces the number of membership queries posed by the learner (②).

3.4 Related Work

Related work can be found in several areas. First, there is related work on LBT itself: [Raf+09, MS13, MP18]. Second, we mention monitoring [TV10], adaptive learning [GPY06, HMP18] and learning ω -regular languages [MP95, AF16, Li+19] as related research directions with a different focus. Third, other than the Learn-

Lib there is another active learning framework called libalf [Bol+10]. Finally, aside from LTSmin there are other model checkers such as NuSMV [Cim+02], and SPIN [Hol04]. Readers interested more in AAL are referred to an extensive review by Howar and Steffen [HS18].

The work in this chapter is based on [MP19], which is an extension of [MP18]. In the latter we extended and implemented the seminal learning-based testing approach by [PVY02]. We introduced the notion of ω -query based on loop detection, to obtain soundness of learning-based testing. We also experimented extensively with LBT in the context of the more recent algorithms ADT and TTT. Here, we improve our method by first checking the safety part of the LTL properties using monitors before using Büchi automata, to circumvent expensive loop checking when possible. We also added extensive experiments, comparing the approaches using monitors and using Büchi automata. Also, we added experiments to evaluate different strategies on interleaving property checking and hypothesis refinement. Finally, we incorporated more technical details, like algorithms in pseudocode and an overview of the software design, integrating LTSmin into the LearnLib for LBT.

The concept of *monitoring* in this work is similar to the concept of monitoring in the field of Runtime Verification (RV). In RV a monitor deadlocks when an illegal action may not be performed by the system. Thus in RV, a monitor prevents the system from performing an illegal action. We build *complete* monitors that explicitly reject finite traces if infinite extensions of those traces would violate the original formula. In our work a monitor does not prevent a system from performing an illegal action, we use monitors to check if there are illegal (finite) traces in the hypothesis.

LBT is not entirely new to the LearnLib; several years ago a similar study was performed, named *dynamic testing* [Raf+09]. However, since then new active learning algorithms such as ADT [Fro15], and TTT [IHS14b] have been added to the LearnLib. Their performance in the context of LBT was still unknown, and was first investigated in [MP18]. Both ADT and TTT are comparable to the Incremental Kripke Learning algorithm (IKL) [MS11] in LBTest, which is a so-called incremental learning algorithm. Incremental learning algorithms try to produce new hypotheses more quickly, in order to reduce the number of learning queries. Traditional active learning algorithms, such as L^* produce fewer hypotheses, but each new hypothesis requires more learning queries. The latter makes sense in the context of active learning, because this minimizes the number of equivalence queries necessary. In the context of active learning, incremental learning algorithms may actually degrade performance; while they may perform well in the number of learning queries, they may require more equivalence queries to refine the hypotheses, resulting in longer run times, see [Isb15, Section 5.5]. In LBT, model checking queries can be used to refine hypotheses. Model checking queries

induce negligible overhead compared to equivalence queries [MS11], making the ADT, and TTT algorithms excellent candidates for a LBT study.

Adaptive learning [GPY06, HMP18] is another paradigm that tries to improve the efficiency and applicability of active automata learning. Adaptive learning is suitable in the context of regression testing, where subsequent versions of the System under Learning are similar. Adaptive learning tries to reuse the learned automaton of the previous system, in order to boost learning the new system.

Active learning for ω -regular languages is a related topic, but it has a different focus. There the goal is to learn automata to recognize ω -regular languages, based on ω -queries. Several representations have been suggested, in particular ultimately periodic words [MP95] and *families of DFAs* [AF16].

The main reason for our technology choices is related to the availability of the underlying software tools. Currently, LBTest is not Free and Open Source Software (FOSS). The LearnLib on the other hand is licensed under the Apache 2 license and thus freely available, even for commercial use. This is interesting because LBT is very successful when applied to industrial critical systems [Mei17, KMR17]. Our new implementation in the LearnLib is also licensed under the Apache 2 license. The reason to implement LBT in the LearnLib instead of libalf is that LearnLib is actively maintained, while libalf is not.

We have chosen to select the LTSmin [Kan+15] model checker, because LTSmin, similar to the LearnLib has a liberal BSD license, and is still actively maintained. For Mealy machines LTSmin implements both *synchronous* and *alternating* trace semantics, which is detailed in [PM18]. In this work (in particular Section 3.2) we cover the synchronous trace semantics, because these are more intuitive than alternating trace semantics. Compared to NuSMV, LTSmin has an explicit-state model checker for LTL, while NuSMV is a symbolic model checker using BDDs. In principle, NuSMV would also suffice as a model checker in this work. We have designed our LBT approach in such a way that in the future integrating NuSMV with the LearnLib is easy; one can simply implement the `ModelChecker` interface. Another popular model checker is SPIN. A disadvantage of using the SPIN model checker is that the counterexamples it produces are state-based, while active learning algorithms require action-based counterexamples [Sin13].

3.5 Experimental Results

LBT in the presence of a good amount of LTL formulae can greatly reduce the number of learning and equivalence queries required to disprove the LTL formulae compared to active learning. In this section, we will quantify the improvement by an experiment. Note that, although LBT introduces additional model checking

queries (performed by the emptiness oracle or inclusion oracle), the number of model checking queries is dwarfed by the amount of equivalence and learning queries. We will thus refrain from reporting the amount of model checking queries here (they can be found online², alongside reproduction instructions). What we will show is the amount of learning queries and hypothesis refinements required by various learning algorithms and LBT strategies to disprove as many LTL formulae as possible. We have to be selective in displaying result graphs, since their possible variation is enormous, due to our modular design.

3.5.1 Variables, Metrics and Constants

The variables of our experiment (Learning algorithm, LBT strategy and Automaton type) are as follows.

- Eight learning algorithms: ADT [Fro15, Fro19], DHC [Mer+11], Discrimination Tree [How12], L^* [Ang87], Kearns & Vazirani [KV94], Maler & Pnueli [MP95], Rivest & Schapire [RS93], and TTT [IHS14b].
- Three learning-based testing algorithms: CExFirstOracle, DisproveFirstOracle, and – none –.
- Three automaton types: monitor, Büchi automaton, and both monitor and Büchi automaton

Note that the learning-based testing algorithm – *none* – computes the final hypothesis before checking any properties. So this strategy represents a traditional active learning experiment, in which properties are only checked once the final hypothesis has been learned.

To measure the performance of the above variations, we measure the following each time a property is disproved.

1. The number of states in the hypothesis.
2. The number of queries in total.
3. The number of learning queries.
4. The number of equivalence queries.
5. The number of emptiness queries.
6. The number of inclusion queries.
7. The aggregated length of each of above query types i.e. the total number of symbols.
8. The length of the counterexample that disproves the property.
9. The number of times the hypothesis has been refined.

The constants in the experiments are nine RERS problems with each 100 LTL formulae. Those RERS problems will be introduced in Section 3.5.2. Typically

²<https://github.com/Meijuh/NFM-ISSE-2018>

half of the 100 LTL formulae can be falsified. In total we have $(8 * 3 * 3 =) 96$ variations, and $(96 * 9 =) 864$ experiments (in a single experiment all LTL properties are checked). These experiments are run on the University of Twente CTIT compute cluster with a time-out of one hour.

The variations and metrics allow us to answer the following research questions.

1. What is the best *learning* algorithm applied in the context of learning-based testing? This question will be answered by showing which learning algorithm performs the least amount of learning queries.
2. What is the best *learning-based testing* strategy? (i.e. `DisproveFirstOracle` or `CExFirstOracle`) This question will also be answered by showing which learning-based testing algorithm performs the least amount of learning queries.
3. What is the *length* of counterexamples when model checking and monitoring? Since, in case of monitoring there is no need to unroll loops of lassos, counterexamples to properties are expected to be much shorter. We will compare lengths of counterexamples to properties in scatter plots.
4. Do we see a *difference* in performance between classical learning algorithms like L^* and modern *incremental* learning algorithms like TTT?
5. How do both model checking and monitoring *affect learning performance*? Model checkers will give different counterexamples to properties for both procedures. Furthermore, in case of monitoring, counterexamples to properties are much shorter. When these counterexamples to properties are spurious (i.e. they can be used to refine hypotheses) the performance of learning algorithms is affected. To investigate how the performance is affected, we will also show the number of learning queries performed for each.

3.5.2 The RERS Challenge

The Rigorous Examination of Reactive Systems (RERS) challenge³ is a yearly recurring verification challenge [Jas+17]. There are two main categories in this challenge. In one category one has to solve properties expressed in LTL and CTL for problems which are parallel in nature [Ste+17]. The other category involves sequential problems [Ste+14] that includes solving reachability and LTL properties. We are interested in these sequential problems. The RERS sequential problems are provided in Java (among others); the Java problem structure is given in Listing 3.4.

³<http://rers-challenge.org>

Listing 3.4: RERS problem structure

```

1 @EqualsAndHashCode(exclude = {"inputs"})
2 public class Problem {
3     ...
4     public String[] inputs = {"B", "E", "C", "A", "D"};
5
6     private int a175 = 6;
7     private int a52 = 9;
8     private int a176 = 7;
9     private String a166 = "e";
10    private String a167 = "e";
11    private String a62 = "f";
12
13    public String calculateOutput(String i) {...}
14
15    public void reset() {...}
16    ...
17 }

```

Our implementation of the `ObservableSUL` interface called `ProblemSUL` is shown in Listing 3.5. Class `ProblemSUL` connects the RERS problem instances to the `LearnLib`, these problem instances are given as part of the challenge.

Listing 3.5: RERS SUL implementation

```

1 public class ProblemSUL implements ObservableSUL {
2     final private Problem problem = new Problem();
3
4     @Override
5     public void pre() {
6     }
7
8     @Override
9     public void post() {
10        problem.reset();
11    }
12
13    @Override
14    public String step(String input) {
15        return problem.calculateOutput(input);
16    }
17
18    @Override
19    public boolean canFork() {
20        return true;
21    }
22 }

```

```

23     @Override
24     public ObservableSUL fork() {
25         return new ProblemSUL();
26     }
27
28     @Override
29     public boolean deepCopies() {
30         return false;
31     }
32
33     @Override
34     public Problem getState() {
35         return problem;
36     }
37 }

```

One can see that it is straightforward to actively learn a Mealy machine from a `Problem` instance. The *input alphabet* (that is part of a `Problem` instance) is specified with the field `String[] inputs` in Listing 3.4. The *state* of a `Problem` instance is determined by the valuations of some of its instance variables (`a175`, `a52`, `a176`, `a166`, `a167`, and `a62`). An *input* can be given to the `calculateOutput` method, which returns an *output* `String`. The problem instance can be *reset* with the `reset()` method. A SUL implementation of a RERS `Problem` is easy: `SUL#post()` resets the problem instance, and `SUL#step()` applies one input to the problem instance.

With model checking we must be able to retrieve the current state of a `Problem` instance. That is because we need to be able to perform ω -queries. Hence `ProblemSUL` implements `ObservableSUL` instead of only `SUL`. We choose not to make deep copies of a state of a `Problem`, hence `ObservableSUL#deepCopies()` does not hold. We have made this decision, because stepping through a problem instance is cheaper than serializing an entire problem state, which would be necessary if `deepCopies()` would return `true`. A serialized state for larger problem instances could be as big as several kilo bytes. Because we have decided that `deepCopies()` returns `false`, an `OmegaMembershipOracle` implementation, must use `Problem#hashCode()`, and `Problem#equals()`. These methods are generated with project Lombok⁴, by annotating the `Problem` class with `@EqualsAndHashCode`. Lastly, the `ObservableSUL` can be forked by creating a new `ProblemSUL` instance. Note that `ObservableSUL` mandates that it must be *forkable* if `Problem` instances are not deep copies.

⁴<https://projectlombok.org>

Figure 3.5: Legend for learning algorithms and learning-based testing algorithms

Figure 3.6: Number of *learning* queries for *model checking* (with Büchi automata) on problem instance 1, comparing various AAL algorithms and LBT strategies

3.5.3 Discussion of the Algorithms' Performance

We can now answer the first two research questions, i.e. what the best learning and learning-based testing algorithms are. We will illustrate the performance of these algorithms with cactus plots. The best algorithms for model checking can be read off from Figure 3.6. The figure shows the number of learning queries on RERS problem instance 1 only. The x-axis shows the number of queries required to disprove a certain number of properties. The y-axis shows the amount of properties that are disproved. The line color differentiates the learning algorithms and the type of line differentiates the learning-based testing strategies. A legend of these colors and line types is shown in Figure 3.5. The further a line appears to the left; the better the algorithms. Figure 3.7 displays the results for monitoring, but on the larger RERS problem instance 5; it shows which learning algorithms and strategies are superior for the monitoring case. Figure 3.8 shows the number of hypothesis refinements required for monitoring on the same problem. Note however, that a

Figure 3.7: Number of *learning* queries for *monitoring* on problem instance 5, comparing various AAL algorithms and LBT strategies

Figure 3.8: Number of *hypothesis refinements* for *monitoring* on problem instance 5, comparing various AAL algorithms and LBT strategies

better algorithm in Figure 3.8 does not necessarily appear further to the left. The number of equivalence queries when using LBT algorithms is zero for smaller RERS problems, hence we do not show such a graph. In fact, we would not be able to show such a graph, because the x-axes on the cactus plots shown have logarithmic scales. Lines with *Black-box oracle=none* are always only vertical, because active learning algorithms do not disprove properties on-the-fly (i.e. the same number of queries is required to disprove all properties). In the cases where LBT algorithms are applied, properties are disproved on-the-fly and result in lines that are strictly monotonic increasing. Interestingly, Figure 3.6 shows almost all algorithms use fewer learning queries to disprove some properties when used in the context of LBT. Figure 3.8 also shows that (as suspected) the incremental TTT, and ADT algorithms ultimately produce more hypotheses than classical algorithms like L^* and Rivest & Schapire. This observation answers research question three.

The performance of the eight algorithms is quite consistent throughout the larger problem instances. The ADT algorithm seems to perform really well, but the TTT algorithm is quite competitive too. This is most visible in the larger RERS problems. The great performance of ADT is particularly interesting, due to it being recently developed for AAL — not for LBT. The ADT algorithm is developed to reduce the number of resets of the SUL, and is seen as an improvement over TTT. Now ADT seems to be the best choice for LBT too, among the benchmarked algorithms and RERS problem instances.

Continuing with the fourth research question we will show the length of counterexamples and their differences between model checking and monitoring. Figure 3.9 shows the difference in length of counterexamples between those produced with Büchi automata and monitors. If a circle appears below the line $x = y$ than the counterexample with monitoring is shorter. However, if a dot appears below the line $y = 0$ the property could not be disproved with monitors. Counterexamples produced with monitors are almost always *much* shorter. This is because these are counterexamples to safety properties, where lassos do not need to be validated on the SUL, only finite prefixes. The counterexamples that are produced with Büchi automata are finite prefixes of their lassos. This means the lengths of counterexamples shown on the x-axis are also the shortest prefixes for which the emptiness oracle was able to deduce their lasso would be accepted by the SUL as well. These shortest prefixes are still quite long, they grow even larger on larger RERS problem instances, while the lengths of counterexamples produced with monitors remain quite small. One can see there are 7 properties that could only be disproved with Büchi automata. These 7 properties are liveness properties, not safety properties.

Counterexamples from both monitors and Büchi automata also influence the performance of learning algorithms differently. Figures 3.10 and 3.11 show how many learning queries are required to disprove a certain number of properties and an-

Figure 3.9: Length of counterexamples with *ADT* and *DisproveFirstOracle* on problem instance 1, comparing various Automata types (monitors, Büchi automata)

Figure 3.10: Legend for model checking and monitoring

Figure 3.11: Number of *learning* queries with `DisproveFirstOracle` on problem instance 1, comparing various Automata types (monitors, Büchi automata, or both)

swers the fifth research question. In these figures each learning algorithm has three line types. One where Büchi automata are used, one where monitors are used and one where both monitors and Büchi automata are used. The lines where only monitors are used, are not as high as those where Büchi automata are used. Indeed, all those lines are missing exactly those counterexamples that are below the line $y = 0$ in Figure 3.9. Further observations that can be made are that it is often better to try to disprove properties first with monitors and that the combination of monitors and Büchi automata is better than solely using Büchi automata and monitors.

With all this data we can conclude that it is best to use the *ADT* learning algorithm, the `DisproveFirstOracle` learning-based testing strategy and to use monitors. Furthermore, model checking with Büchi automata should be used as a complementary method to further improve the completeness of the method in the presence of liveness properties.

3.6 Conclusion and Future Work

We have presented a sound learning-based testing method and an implementation with the `LearnLib` and `LTSmin`. The bottom line is that learning-based testing exploits the knowledge in the given LTL properties to speedup the learning process.

We presented novel sound approaches for liveness LTL properties, where we can check if a system-under-learning accepts an infinite lasso-shaped word. Our method for liveness contrasts the original proposal where a (hard to guess) upper-bound on the number of states of the system-under-learning is assumed. Additionally one can now make use of monitors that may provide finite length counterexamples to safety properties. Using monitors further improves the speed of learning algorithms in terms of learning queries. Our implementation is available under a liberal free and open source license, such that it can be put to practice quite easily. Software testers now have a free ease-of-use *sound* learning-based testing implementation available for industrial use cases.

Our experimental results reveal three conclusions: (1) the recent learning algorithms ADT, and TTT perform the best in a learning-based testing setting. Furthermore, (2) trying to disprove all properties on a hypothesis before refining the hypothesis is better than trying to refine hypotheses when a counterexample is spurious. Finally, (3) the use of monitoring before full LTL model checking leads to a considerable reduction in the number of learning queries.

In contrast to some other learning algorithms in the LearnLib, ADT, and TTT are incremental learning algorithms, meaning they construct more hypotheses while using fewer learning queries. In an active learning setting this may degrade performance, because more equivalence queries are required. In a learning-based testing setting this appeared to be an advantage, because model checking queries replace expensive equivalence queries.

Future work includes performing experiments on a larger class of applications. Furthermore, another application of our work could be to use our notion of ω -queries with loop detection to implement the ω -queries in the theory of ω -regular learning [MP95, AF16]. Another direction is more practical; a usability study in an industrial context should be done, to assess how appropriate the extension to the LearnLib is there (see Section 8.1.2 for an initial step in this direction).

There are still several potential improvements of our method that could be explored. First, experiments could be performed to compare ADT and TTT with the IKL algorithm in LBTest. Similarly, one could try other model checkers such as NuSMV and SPIN, to investigate the effects of their counterexample generation. Another strategy could be to generate *all* counterexamples to a given property, and use many spurious counterexamples to refine the hypothesis. Currently, the learning-based testing approach is inherently asymmetric, in the sense that model checking only provides counterexamples that are in the language of the hypothesis but not in the SUL. It would be interesting if it were possible to use the LTL properties for detecting words that are in the SUL but missing in the hypothesis. Finally, in our case study we applied a perfect state equivalence function to the RERS problems, but it would be interesting to apply our approach to cases where

only part of the state can be observed, or when the SUL is hardware, instead of software.

Automated Alphabet Partition Refinement for Learning-Based Testing

Abstract

In Learning-Based Testing (LBT) incremental hypotheses on the behavior of a system are learned in the form of finite automata. Requirements formalized in Linear-time Temporal Logic (LTL) are analyzed on these automata. Counterexamples to the LTL formulae are either used to refine and improve the learned automata, or reported as violations of certain requirements. Both the learning of automata and formalization of requirements is done based on an alphabet that specifies the possible interactions with a running system. In complex industrial systems these interactions may contain data parameters, meaning that the alphabet can be huge. We present an LBT approach that can deal with a huge alphabet by partitioning it, as well as automatically refining it. This means hypotheses can be learned and analyzed by posing fewer queries on the system.

We show how LearnLib's API extension can be used to learn abstract hypotheses, resolve non-determinism that arises due to the given abstraction and model check LTL formulae on concretized hypotheses. Using the RERS 2017 challenge we provide experimental results and give some insight into the performance of several existing active learning algorithms when applied in an LBT setting with alphabet abstractions.

4.1 Introduction

In this chapter we present an algorithm for Learning-Based Testing (LBT) that can deal with alphabets that are large. By large, we mean the alphabet is too big for a learning algorithm for Finite State Machines (FSMs) to enumerate over in reasonable time. Usually such an alphabet is associated to systems where input symbols have data parameters [MM15] and partitioning those input symbols into a finite amount of groups may increase the efficiency of the Active Automata Learning (AAL) algorithms. We investigate such a partitioning strategy in the context of LBT.

LBT involves verifying requirements on a running system. These requirements are formalized in Linear Temporal Logic (LTL). LBT algorithms are fundamentally based on AAL algorithms. These AAL algorithms produce hypotheses, i.e. FSMs that capture the behavior of a system. Since these FSMs are hypothetical, they must be verified to be correct. This is done through a so called equivalence check. As the name suggest, it checks the equivalence of the language of the learned FSM, and the system. Whenever the equivalence check finds a word that is in their symmetric difference, the AAL algorithm knows it must refine its hypothesis FSM.

LBT extends this AAL approach, by first evaluating a given set of LTL formulae on the learned hypotheses. Whenever an LTL formula can be falsified, the algorithm investigates whether the current hypothesis is incorrect, or the system is indeed incorrect with respect to the requirement the formula formalizes.

In order for AAL algorithms to learn the hypothesis FSMs, they pose so called membership queries to the system. The work presented by Howar, Steffen, and Merten [HSM11] shows that these FSMs can be learned with fewer membership queries if the alphabet is partitioned into equivalence classes. Following the approach based on controllable non-determinism in [HSM11], we investigate whether such a partitioning strategy works well in the context of LBT too.

Controllable non-determinism arises when the hypothesis learned over an abstract (partitioned) alphabet is incompatible with the system. In particular, there may be equivalent words (their abstraction is the same) that produce the same output on the concretization of the hypothesis, but produce different output on the actual system. This non-determinism simply arises because the alphabet is too coarse, and becomes apparent whenever an equivalence check is done between the concretized hypothesis and system language. The non-determinism is controllable, because it can always be resolved by splitting an equivalence class in the partitioned alphabet.

The approach we present consists of a “general” algorithm, in the sense that it takes

Figure 4.1: Active learning procedure with alphabet abstractions

as input a partitioned alphabet as well as any existing learning algorithm for DFAs or Mealy machines, such as L^* [Ang87, SG09] or ADT [Fro15]. The algorithm we present makes its input learning algorithm learn over its partitioned input alphabet. Whenever the input learning algorithm forms an abstract hypothesis, our algorithm concretizes the hypothesis and uses it to verify the LTL formulae on the system. We implemented the new learning algorithm in the LearnLib [IHS15], a framework for AAL. Together with the work presented in [MP18, MP19] and Chapter 3, the algorithm can be applied in the context of LBT as well.

We have illustrated the approach for AAL with alphabet abstractions in Figure 4.1. With respect to Figure 3.1, Figure 4.1 shows the algorithm is augmented with the ability to:

1. answer queries of abstract words over a partitioned alphabet Π (an abstract word is shown as w^π),
2. *concretize* learned hypotheses,

3. abstract a concrete word w into w^π (w is in the Cartesian product of w^π) and
4. *resolve* controllable *non-determinism* with a *splitter oracle*.

The behavior of the augmented algorithm is as follows. The learning algorithm poses abstract queries over partitioned alphabet Π . The abstract queries are posed by means of the membership oracle $\in^\pi$ that can answer abstract queries. For each abstract query a representative concrete query is posed on the SUL, by means of membership oracle $\in$. Abstract queries are posed until the learner forms a complete (abstract) hypothesis H^π . Next, the hypothesis is concretized (H) and provided to the equivalence oracle ($=$). The equivalence oracle tests the hypothesis as is usual for equivalence oracles (equivalence oracles do not test abstract hypotheses). If the language of the hypothesis is found to be equal to the language of the SUL, the algorithm terminates with output H . However, if the equivalence oracle provides w as a counterexample (indicating the languages are not equal) then w is given to the splitter oracle. The splitter oracle abstracts w to w^π and investigates whether there is controllable non-determinism. This is done by trying a concrete word instance $\bar{w}$ of w^π , for which the output is different from the output of w . If there is no controllable non-determinism (there is no $\bar{w}$ for which the output differs), then the learner uses w^π to refine the hypothesis and the learning continues. However, if there is controllable non-determinism (the output of w and $\bar{w}$ differ), then an equivalence class in Π is split resulting in a refined alphabet Π' . Refined alphabet Π' is given to the learner, and the learner restarts with the new alphabet. Note that if there are LTL formulae that need to be analyzed (meaning the AAL algorithm becomes part of a LBT setup) then the behavior can be seen in Figure 3.2. The latter figure shows that before a concrete hypothesis (H) is given to the equivalence oracle ($=$), the model checker will first analyze one or more LTL properties (ϕ) on H . Violations of ϕ are either reported or used to detect controllable non-determinism. In the latter case w is given to the splitter oracle. Summarizing the behavior of the algorithm; hypotheses are learned on the abstract level, while LTL properties are model checked on the concrete level.

Our experimental section shows that the improvement to LBT with alphabet abstractions ranges from modest to excellent. In particular, modest improvements are seen when starting with a coarsest initial partitioning does not degrade performance for small alphabets. For larger alphabets the results are better, especially when using ADT. Many LTL formulae can be falsified with considerably fewer membership queries. Even more impressive is that within a timeout of three hours, our novel approach is capable of falsifying LTL formulae that were not falsifiable before within that time limit. That is, starting with the finest initial partitioning of the alphabet (which is equivalent to not partitioning the alphabet).

The rest of the chapter is structured as follows. First, we explicitly present the

contribution in Section 4.1.1. Next, we present related work in Section 4.2. In particular, alternative approaches we could have taken. The preliminaries presented in Section 4.3 follow the same approach taken in [HSM11], where they show how one can abstract away from the chosen FSM type that is learned, i.e. a Deterministic Finite Automaton (DFA), or Mealy machine. In the approach we present next, we again follow the foundations in [HSM11] based on controllable non-determinism, but present it in a different way. In particular, we present a new implementation of automated abstraction refinement for alphabets that can be used for LBT. This implementation was not available before in the new open-source version of the LearnLib [IHS15]. The implementation is based on the concept of a splitter oracle, which in essence, resolves the controllable non-determinism if it happens. After defining the behavior of a splitter oracle we conclude the section with the main novel algorithm. Lastly, in Section 4.5 we present the efficiency of the algorithm when applied to the 2017 Rigorous Examination of Reactive Systems (RERS) challenge [Jas+17] and conclude our work in Section 4.6.

4.1.1 Contribution

We present an approach that is completely built on top of the approach presented in Chapter 3. In particular, partitioning the alphabet works in combination with:

- Two variations of learning-based testing strategies: `DisproveFirstOracle`, and `CExFirstOracle`.
- Monitoring with monitors, and model checking with Büchi automata.
- Any existing active learning algorithm for FSMs, such as L^* and ADT.

Furthermore we contribute:

- A thorough experimental setup that shows which active automata learning algorithm works best in the context of automated alphabet abstraction refinement.
- A modular design allowing better splitter oracles and membership oracles (for answering queries over equivalence classes) to be implemented in the future. This modular design even allows one to apply the presented algorithm in a context free of any LTL formulae, which is equivalent to a traditional active automata learning setup.

4.2 Related Work

In order to deal with large (too big to enumerate over), or even infinite size alphabets, there are two main approaches in the literature. One either improves the learning algorithm so that more expressive types of automata can be learned, or one partitions the alphabet.

In this chapter we restrict ourselves to Finite State Machines (FSMs), in the form of Deterministic Finite Automaton (DFAs) and Mealy machines. In Chapter 3 we show how LTSmin is integrated in the LearnLib. We want to reuse this integration, and because LTSmin is capable of only model checking FSMs, we are left with the only approach of partitioning the alphabet. This is not necessarily a problem, because partitioning the alphabet allows one to reuse existing learning algorithms, which is part of the approach we present in this chapter.

Approaches for AAL based on more expressive automata include for instance: visibly pushdown automata [Isb15], symbolic automata [AD18, DD17, MM17, Men17, MM15], extended Mealy machines [Mei18, BJR08] and register automata [SHI12, IHS14a, Cas15, Cas+18]. Large support for learning register automata is available in RALib [CHJ15]. Learning symbolic automata is supported by SVPALib [SVP19]. A case study on TCP’s sliding window behavior, where learned register automata are checked for correctness is done in [FH17]. Automated verification of properties using model checking was not part of their approach however.

When partitioning the alphabet, there are two options, either one takes a state-local approach [IHS13], where each state in the FSM forms its own partition of the alphabet. Or alternatively, one can have a global partition of the alphabet. When partitioning the alphabet globally, there are two approaches too. Either, one constructs a mapper or wrapper [Aar+12, Aar14, Mei18]. In this case, the mapper takes care of the abstraction, and the learning algorithm never deals with concrete alphabet symbols. On the other hand, one can make the learning algorithm responsible for making the alphabet abstractions. The difference is explained in more detail in [HSM11, Figure 1] and a comparison in the context of AAL is done in [Aar+14]. In this chapter we take the latter approach, where the learner is responsible for making the abstractions, and we believe we are the first to do so in the context of LBT.

4.3 Preliminaries

Because this work extends active learning with partitioned alphabets, we start out by defining partition (refinement). The next step is to define, based on the output automata produce, the behavior of learning algorithms. This behavior is defined in terms of three operations on automata: *initialization*, *refinement* and *equivalence*. This is also done in the LearnLib, in the interfaces **LearningAlgorithm**, and **EquivalenceOracle** (see Chapter 3). Existing active learning algorithms such as L^* , TTT, etc. adhere to the “contract” of these operations. Our contributed algorithm (Section 4.4) also implements the three presented operations. It is a general algorithm in the sense that it requires an existing algorithm, such as L^* , TTT, etc. and it is capable of learning over partitioned alphabets.

To start, Definition 4.1 allows us to partition the alphabet over which is learned.

Definition 4.1: Partition

Given a set S , Π is a partition of S iff:

- $\emptyset \notin \Pi$, and
- $\bigcup_{B \in \Pi} B = S$, and
- $\forall B, B' \in \Pi: B \neq B' \implies B \cap B' = \emptyset$.

We say that every element of a partition is called a block or group. Furthermore, let Π, Π' be partitions of S , we denote:

- $\Pi \leq \Pi' \iff \forall B \in \Pi: \exists B' \in \Pi': B \subseteq B'$, and
- $\Pi < \Pi' \iff \Pi \neq \Pi' \wedge \Pi \leq \Pi'$.

Example 4.1: Partition

The *coarsest* partition of alphabet $\{a, b\}$ is $\{\{a, b\}\}$. The *finest* partition is $\{\{a\}, \{b\}\}$. And we have $\{\{a\}, \{b\}\} < \{\{a, b\}\}$.

Next, Definition 4.2 allows us to refine the partitioned alphabet if at some point in the learning process it is considered too coarse.

Definition 4.2: Partition refinement

Let $P \subseteq S$, and $\Pi = \{B_1, \dots, B_n\}$ be a partition of S , the *refinement* of Π according to *pivot* P is:

$$\text{Refine}(\Pi, P) = \{B_1 \cap P, B_1 \setminus P, \dots, B_n \cap P, B_n \setminus P\} \setminus \{\emptyset\}.$$

Partition refinement has the following properties:

- $\text{Refine}(\Pi, P) \leq \Pi$, and
- $\text{Refine}(\Pi, \emptyset) = \Pi$, and
- $\text{Refine}(\Pi, P) = \text{Refine}(\Pi, \overline{P})$.

Example 4.2: Partition refinement

Some example partition refinements are:

- $\text{Refine}(\{\{a, b\}\}, \{a\}) = \text{Refine}(\{\{a, b\}\}, \{b\}) = \{\{a\}, \{b\}\},$

- $\text{Refine}(\{\{a, b\}\}, \{a, b\}) = \text{Refine}(\{\{a, b\}\}, \emptyset) = \{\{a, b\}\}.$

To abstract away from the automaton type that is learned (i.e. DFA, or Mealy machine) we introduce a so called output function that is defined for both types of automata. This follows the approach presented in [HSM11, Isb15]. Note that we will *not* partition the set of output symbols, only the set of input symbols.

Definition 4.3: DFA output

Let $\mathcal{D} = \langle S, s_0, \Sigma, \delta, F \rangle$ be a DFA (Definition 3.2) and $\mathbb{B} = \{\top, \perp\}$ the set of Booleans, then we first extend δ to the total function:

$\delta^*: S \times \Sigma^* \rightarrow S$, such that:

- $\delta^*(s, \epsilon) = s$, and
- $\delta^*(s, pq) = \delta^*(\delta(s, p), q)$, if $p \in \Sigma$, and $q \in \Sigma^*$.

Then, $\mathcal{D}$'s output is a total function:

$\lambda_{\mathcal{D}}^*: \Sigma^* \rightarrow \mathbb{B}$, such that $\lambda_{\mathcal{D}}^*(\sigma) \iff \delta^*(s_0, \sigma) \in F$.

We say that $\lambda_{\mathcal{D}}^*$'s co-domain is $\mathcal{D}$'s output domain, i.e. $D = \mathbb{B}$.

Example 4.3: DFA output

Consider the following DFA $\mathcal{D}$ over alphabet $\Sigma = \{a, b\}$ that accepts the language a^* . For $\mathcal{D}$ we have $\lambda_{\mathcal{D}}^*(a) = \top$, $\lambda_{\mathcal{D}}^*(b) = \perp$, $\lambda_{\mathcal{D}}^*(aa) = \top$, etc.

Definition 4.4: Mealy machine output

Let $\mathcal{M} = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$ be a Mealy machine (Definition 3.3), then we first

extend λ to the total function:

$\lambda': S \times \Sigma^* \rightarrow \Omega^*$, such that:

- $\lambda'(s, \epsilon) = \epsilon$, and
- $\lambda'(s, pq) = \lambda(s, p)\lambda'(\delta(s, p), q)$, if $p \in \Sigma$, and $q \in \Sigma^*$.

Then, $\mathcal{M}$'s output is a total function:

$\lambda_{\mathcal{M}}^*: \Sigma^* \rightarrow \Omega^*$, such that $\lambda_{\mathcal{M}}^*(\sigma) = \lambda'(s_0, \sigma)$.

We say that $\lambda_{\mathcal{M}}^*$'s co-domain is $\mathcal{M}$'s output domain, i.e. $D = \Omega^*$.

Furthermore, whenever we say an input alphabet Σ , an output domain D and output function $\lambda_A^*: \Sigma^* \rightarrow D$ are given, then we also assume that (exclusively) either a DFA A with input alphabet Σ is given, such that $D = \mathbb{B}$, or that a Mealy machine A with input alphabet Σ and output alphabet Ω , such that $D = \Omega^*$ is given. For both λ_A^* is defined. We refer to this DFA or Mealy machine named A as the “automaton”.

Now we define the three operations for active learning from an automaton output perspective. Init is the first operation; it gives us the first hypothesis (H_0).

Definition 4.5: Automaton initialization

Given an alphabet Σ , an output domain D and a membership oracle $\in: \Sigma^* \rightarrow D$, a correct initial automaton satisfies:

$A = \text{Init}(\Sigma, \in)$, such that A 's output function is given by:

$$\lambda_A^*(\epsilon) = \in(\epsilon).$$

Note that because input ϵ is properly defined for A 's output function means that A has at least one state. Furthermore, since A 's transition function is total, A typically defines λ^* properly for one letter strings as well (i.e. $\lambda_A^*(\sigma) = \in(\sigma)$, where $\sigma \in \Sigma$), this depends on the learning algorithm used and is not required for our algorithm that is presented later to be sound or terminate.

Example 4.4: Automaton initialization

An example target DFA over which $\in$ answers queries is given below on the right. A valid initialization is given on the left. This initialization is valid, because it

correctly accepts ϵ .

valid initialization:

target DFA:

Whenever a hypothesis automaton is considered incorrect it can be refined according to Definition 4.6 as follows.

Definition 4.6: Automaton refinement

Given an input alphabet Σ , an output domain D , an initial automaton $A_0 = \text{Init}(\Sigma, \in)$ with output function $\lambda_{A_0}^* : \Sigma^* \rightarrow D$, a membership oracle $\in : \Sigma^* \rightarrow D$, queries $q_0, \dots, q_n \in \Sigma^*$, then for all $0 \leq i \leq n$, a correct refinement automaton for A_i satisfies:

$A_{i+1} = \text{Refine}(A_i, \Sigma, \in, q_i)$, with output function:

$\lambda_{A_{i+1}}^* : \Sigma^* \rightarrow D$ associated to A_{i+1} , such that $\lambda_{A_{i+1}}^* = \lambda_{A_i}^* \iff \lambda_{A_i}^*(q_i) = \in(q_i)$.

However, if $\lambda_{A_i}^*(q_i) \neq \in(q_i)$, then:

$$\lambda_{A_{i+1}}^*(\epsilon) = \in(\epsilon) \wedge \forall 0 \leq j \leq i: \lambda_{A_{i+1}}^*(q_j) = \in(q_j).$$

In essence, Definition 4.6 says that a refinement of A_i should at least be consistent with all observations for $q_0 \dots q_i$ and the empty string. This means refinements of hypotheses may only be as good as their counterexamples. Furthermore, note that by definition all refinements must be closed under their transition function. $\text{Refine}(A_i, \Sigma, \in, q_i)$ thus defines the set of possible refinements of A_i .

Example 4.5: Automaton refinement

Let $\mathcal{D}_1$ be the initial DFA on the left in Example 4.4, $\mathcal{D}_2$ the target DFA on the right, $\Sigma = \{a, b\}$ the alphabet of both DFAs and $\in : \Sigma^* \rightarrow \mathbb{B}$ the membership

oracle that answers queries for $\mathcal{D}_2$. Then we have:

- $\mathcal{D}_1 = \text{Refine}(\mathcal{D}_1, \Sigma, \in, \epsilon) = \text{Refine}(\mathcal{D}_1, \Sigma, \in, a)$, and
- $\mathcal{D}_2 = \text{Refine}(\mathcal{D}_1, \Sigma, \in, b)$.

To establish whether a hypothesis automaton needs to be refined, an equivalence oracle is queried; it is defined as follows.

Definition 4.7: Equivalence oracle

Given an alphabet Σ , an output domain D , a membership oracle $\in: \Sigma^* \rightarrow D$, and an automaton A with output function $\lambda_A^*: \Sigma^* \rightarrow D$, an *equivalence oracle* for A gives a query:

$$q = \text{Equiv}(A, \Sigma, \in), \text{ such that } q \in \Sigma^*.$$

However, if $\exists \sigma \in \Sigma^*: \lambda_A^*(\sigma) \neq \in(\sigma)$, then $\lambda_A^*(q) \neq \in(q)$.

Definition 4.7 says q should highlight some difference in output between the hypothesis and system. Note that Equiv is an oracle, since establishing equivalence of two languages in a black-box context is undecidable. Implementations of equivalence oracles are usually based on random words, or the partial W-method. How they are implemented precisely is not important for the soundness of the algorithm we present in this chapter, we assume the implementations match the definition above.

Example 4.6: Equivalence oracle

Let again $\mathcal{D}_1$ be the initial DFA on the left in Example 4.4, $\mathcal{D}_2$ the target DFA on the right, $\Sigma = \{a, b\}$ the alphabet of both DFAs and $\in: \Sigma^* \rightarrow \mathbb{B}$ the membership oracle that answers queries for $\mathcal{D}_2$. Then the following is a correct equivalence oracle:

- $b = \text{Equiv}(\mathcal{D}_1, \Sigma, \in)$, and
- $\epsilon = \text{Equiv}(\mathcal{D}_2, \Sigma, \in)$, or
- $a = \text{Equiv}(\mathcal{D}_2, \Sigma, \in)$, etc.

4.4 Approach

In the previous section we presented three operations for learning automata: initialization, refinement and equivalence. In this section we will show how these automata can be learned over abstract alphabets. Recall that interacting with

the System Under Learning (SUL) happens at the concrete level. LTL properties are expressed on the concrete level as well. This means that we have to define what concretizations of abstract words are so that the LTL properties can be analyzed. Abstraction and concretization will be defined first (Section 4.4.1). Next, in Section 4.4.2 we will explain how controllable non-determinism can be resolved with a splitter oracle. This is a sufficient basis to present the main algorithm in Section 4.4.3. In Section 4.4.4 we will discuss the API extensions to the LearnLib that implement the presented algorithm.

4.4.1 Abstraction and Concretization

We start by defining abstraction and concretization of words and automata in Definition 4.8, an example is given in Example 4.7. In Definition 4.9 and Example 4.8 we show how queries over abstract words can be answered.

Definition 4.8: Abstraction and concretization

Given a partitioned alphabet $\Pi = \{B_1, \dots, B_n\}$ of Σ , we define the *abstraction* of a word as a function $\alpha: \Sigma^* \rightarrow \Pi^*$, such that:

$$\alpha(b_1 \dots b_n) = B_1 \dots B_n \iff b_1 \in B_1 \wedge \dots \wedge b_n \in B_n.$$

Let $a \in \Pi^*$, the *concretization* of word a is:

$$\gamma(a) = \{c \in \Sigma^* \mid \alpha(c) = a\}.$$

Furthermore, let $\mathcal{D}^\pi = \langle S, s_0, \Pi, \delta^\pi, F \rangle$, be a DFA. Then we define its concretization as:

$$\Gamma(\mathcal{D}^\pi) = \langle S, s_0, \Sigma, \delta, F \rangle, \text{ where:}$$

$$\delta(s, i) = s' \iff \exists B \in \Pi: \delta^\pi(s, B) = s' \wedge i \in B.$$

And analogous for Mealy machines; let $\mathcal{M}^\pi = \langle S, s_0, \Pi, \Omega, \delta^\pi, \lambda^\pi \rangle$, be a Mealy machine. Then we define its concretization as:

$$\Gamma(\mathcal{M}^\pi) = \langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle, \text{ where:}$$

- $\delta(s, i) = s' \iff \exists B \in \Pi: \delta^\pi(s, B) = s' \wedge i \in B$, and
- $\lambda(s, i) = o \iff \exists B \in \Pi: \lambda^\pi(s, B) = o \wedge i \in B$.

Example 4.7: Abstraction and concretization

Let $\Sigma = \{a, b\}$ be an alphabet, $\Pi = \{\{a, b\}\}$ a partition of Σ , then we have:

- $\alpha(a) = \{a, b\}$, and
- $\alpha(ab) = \{a, b\}\{a, b\}$, and
- $\gamma(\{a, b\}) = \{a, b\}$, and
- $\gamma(\{a, b\}\{a, b\}) = \{aa, ab, ba, bb\} (= \{a, b\} \times \{a, b\})$.

Furthermore consider DFA $\mathcal{D}^\pi$ on the left with the language Π^* recognized. Its concretization $\mathcal{D} = \Gamma(\mathcal{D}^\pi)$ is given on the right; $\mathcal{D}$ recognizes Σ^* .

abstract DFA:

concrete DFA:

In Example 4.7 it can be observed that concretization of automata simply involves adding transitions for each concrete alphabet symbol. Concretization does not blow up the size of the automaton. Concretization does however blow up the size of the language of the automaton. This means some of the effort for analyzing LTL formulae is moved from the learning algorithm to the model checker: the learning algorithm has to learn a smaller language, whereas the model checker has to analyze a larger language. We see this as a desired effect, because model checking is a form of static analysis that can be run on automata with huge languages. Learning algorithms have to deal with running systems and answering queries takes time, hence posing fewer queries is better. To be able to answer abstract queries (i.e. words over equivalence classes) we need a capable membership oracle, defined next.

Definition 4.9: Approximating membership oracle

Given an alphabet Σ , an output domain D , a (regular) membership oracle $\in: \Sigma^* \rightarrow D$, and a partitioned alphabet Π of Σ , an *approximating membership oracle* is a membership oracle that answers abstract queries, i.e. it is a function

$\in^\pi: \Pi^* \rightarrow D$, such that:

$$\in^\pi(a) = d \iff \exists c \in \gamma(a): \in(c) = d.$$

As is usual in a Minimally Adequate Teacher (MAT) setup [Ang87], $\in^\pi$ returns the answer for an arbitrary but fixed concrete instance.

Example 4.8: Approximating membership oracle

Let $\Sigma = \{a, b\}$ be an alphabet, $\Pi = \{\{a, b\}\}$ a partition of Σ , $\mathbb{B} = \{\top, \perp\}$ the set of Booleans and $\in^\pi: \Pi^* \rightarrow \mathbb{B}$ an *approximating membership oracle* for the DFA below, then we can have either:

- $\in^\pi(\{a, b\}) = \in^\pi(\{a, b\}\{a, b\}) = \top$, or
- $\in^\pi(\{a, b\}) = \in^\pi(\{a, b\}\{a, b\}) = \perp$.

It is exactly because $\in^\pi$ can answer abstract queries arbitrarily, we at some point need to refine Π . At which point partitioned alphabet Π needs to be refined is discussed next.

4.4.2 Detecting Controllable Non-Determinism

As a last new ingredient for the new learning algorithm we need to be able to compute a pivot that is capable of refining the partitioned alphabet if there is controllable non-determinism. This ingredient is called a *splitter oracle* and computes the pivot based on a concrete query.

Definition 4.10: Splitter oracle

Given an alphabet Σ , an output domain D , a partitioned alphabet Π of Σ , a membership oracle $\in: \Sigma^* \rightarrow D$ and a query $q \in \Sigma^*$, a *splitter oracle* computes a pivot for Π :

$$P = \text{Split}(\Pi, q, \in), \text{ such that:}$$

$$\emptyset \subseteq P \subseteq \Sigma \wedge (\exists \bar{q} \in \gamma(\alpha(q)): \in(q) \neq \in(\bar{q}) \implies \text{Refine}(\Pi, P) < \Pi).$$

Definition 4.10 says that if there is no controllable non-determinism, then Π may be refined arbitrarily, or not at all. However, if there is controllable non-determinism, then the pivot that is computed by $\text{Split}(\Pi, q, \in)$ must refine Π . Always refining arbitrarily is fine, because this will ultimately produce a finest partition of the alphabet for which no controllable non-determinism can arise. Note that if there is controllable non-determinism then the test $\exists \bar{q} \in \gamma(\alpha(q)): \in(q) \neq \in(\bar{q})$ is guaranteed to succeed [HSM11]. In case $\bar{q}$ exists, then in practice it can be found by iterating over the Cartesian product of $\alpha(q)$ until $\in(q) \neq \in(\bar{q})$ succeeds. The definition of a splitter oracle gives rise to many possible implementations. We chose to implement a splitter oracle in the LearnLib called **AcexSplitterOracle**, and it behaves according to the following two cases.

1. In case $\bar{q}$ does *not* exist, then simply $P = \emptyset$.
2. In case $\bar{q}$ *exists*, select $pcd = q, \bar{p}\bar{c}\bar{d} = \bar{q}$, such that:

$$c, \bar{c} \in \Sigma \wedge \alpha(p) = \alpha(\bar{p}) \wedge \in(pcd) \neq \in(\bar{p}\bar{c}\bar{d}), \text{ then:}$$

$$P = \{x \in \alpha(c) \mid \in(pxd) = \in(q) \wedge \in(\bar{p}\bar{x}\bar{d}) \neq \in(\bar{q})\}.$$

The pivot computed by **AcexSplitterOracle** will not refine Π if there is no controllable non-determinism (case 1.). In the other case (2.), **AcexSplitterOracle** will resolve the controllable non-determinism by providing a pivot that splits a block (i.e. $\alpha(c)$) that causes it. To locate block $\alpha(c)$, **AcexSplitterOracle** performs a linear, binary or exponential search strategy, starting from the first symbol in q , or the last symbol in q ¹. The search (and search only) is done based on the Abstract CounterEXample analyzer framework (ACEX) presented in [IS14] (hence the name **AcexSplitterOracle**). In essence, based on the given search strategy and test, the ACEX framework allows one to scan through given queries. We use the framework to scan through q and $\bar{q}$ until the test $\in(pcd) \neq \in(\bar{p}\bar{c}\bar{d})$ succeeds (this test is also guaranteed to succeed [HSM11, Figure 2])^{2,3}. When the test succeeds, the framework provides an index i , such that the length of p and $\bar{p}$ is equal to i . Index i can then be used to compute P . In the end, P will have all symbols in $\alpha(c)$ that behave identical to q , but no symbol that behaves as $\bar{q}$. This way, P resolves the controllable non-determinism. In other words, let $\Pi' = \text{Refine}(\Pi, P)$ be the new partition of Σ , $\alpha': \Sigma^* \rightarrow \Pi'^*$ be the new abstraction function and $\in^{\pi'}: \Pi'^* \rightarrow D$ the new approximating membership oracle, then we guarantee

¹By default, **AcexSplitterOracle** performs a binary search starting from the last symbol in q .

²Scanning until $\in(\bar{p}\bar{c}\bar{d}) \neq \in(pcd)$ can be done as well, but that is not what we implemented.

³Different search strategies we implemented with the ACEX framework [IS14] is novel and not discussed in [HSM11]. In [HSM11, Figure 2] only a linear forward search is illustrated.

$\in^{\pi'}(\alpha'(q)) \neq \in^{\pi'}(\alpha'(\bar{q}))$. This means that learning algorithms that form hypotheses with $\in^{\pi'}$ will not encounter the controllable non-determinism that arose with the old $\in^{\pi}$.

Example 4.9: Splitter oracle

Let $\Sigma = \{a, b\}$ be an alphabet, $\Pi = \{\{a, b\}\}$ a partition of Σ , $\mathbb{B} = \{\top, \perp\}$ the set of Booleans, and $\in: \Sigma^* \rightarrow \mathbb{B}$ the membership oracle for the target DFA below, on the right.

The DFA below, on the left, is an abstract hypothesis DFA for the target DFA on the right. Let us assume the equivalence oracle provides $q = b$, based on the concretized DFA in the middle as a counterexample for the hypothesis, then **AcexSplitterOracle** will compute the pivot $\{b\} = \text{Split}(\Pi, b)$ as follows. First, controllable non-determinism is observed, because $\bar{q} = a$ exists (because $\in(b) \neq \in(a)$). Next, **AcexSplitterOracle** will scan through $\bar{q}$ until $\in(pcd) \neq \in(p\bar{c}\bar{d})$ succeeds; this will reveal that $\alpha(c)$ is located at index $i = 0$. Block $\alpha(c)$ contains both $\{a, b\}$, but only b behaves as $\in(q)$ and not as $\in(\bar{q})$, hence the pivot computed is $\{b\}$ that refines Π .

abstract DFA:

concrete DFA:

target DFA:

4.4.3 Main algorithm

The main algorithm is now explained as follows. First we relate Algorithm 4.1 to Algorithm 3.2 and explain how both differ. A key element in which they differ is the apparent absence of LTL formulae in the new algorithm; Algorithm 4.1 presents a generalized Active Automata Learning (AAL) algorithm that is extended with partition refinement for alphabets. We can present it in such a way, because in Chapter 3 we show that an LBT algorithm can be implemented as an extension of AAL: model checking with LTL formulae and testing for spurious counterexamples to them is implemented by refining equivalence oracles (cf. Figure 3.4 and listing 3.1). This means the equivalence check at line 6 in Algorithm 4.1 covers all the model checking and testing. This leaves us to explain how

queries from equivalence oracles are used to either refine the abstract hypothesis, or the partitioned alphabet, which is the other way both algorithms differ.

component	level
SUL interface	concrete
LTl formula	concrete
model checker	concrete
equivalence oracle	concrete
membership oracle	concrete
existing learning algorithm	abstract
splitter oracle	concrete
approximating membership oracle	abstract

Table 4.1: Abstraction levels of components for LBT

Table 4.1 summarizes at which abstraction level certain components for LBT operate in our setup. The bottom two components are new and only existing learning algorithms (like L^* and TTT) and the approximating membership oracle operate on the abstract level. All other components operate on the concrete level. For instance, the System Under Learning (SUL) interface operates on the concrete level, as an actual system only accepts concrete inputs. LTL formulae are also written over concrete alphabet symbols, hence the model checker also operates on the concrete level. Controllable non-determinism can only be observed on the concrete level, this means the equivalence oracle operates on the concrete level. To resolve the controllable non-determinism, the splitter oracle also operates on the concrete level. In Algorithm 4.1 it can be seen at which points the algorithm properly abstracts queries and concretizes hypotheses. For instance, the hypothesis that is learned with the existing learning algorithm is concretized (line 5) before doing the equivalence check and counterexamples are abstracted (line 14) by the presented algorithm before refining the abstract hypothesis.

The rest of Algorithm 4.1 behaves as follows. Instead of immediately refining the hypothesis, we must first check for controllable non-determinism. This is done by the splitter oracle. Whenever the splitter oracle returns a pivot, we check whether it refines the alphabet (line 8). If it does not refine the alphabet, this means there is no controllable non-determinism and the query is abstracted at line 14, subsequently saved at line 15 and the hypothesis is refined at line 16. However, if the alphabet needs to be refined, then first the partitioned alphabet is refined at line 9, and only legal counterexamples are retained at line 10. Because we essentially have a new alphabet, the learning is restarted at line 11 and counterexamples are reused at line 12. Restarting the learning process from scratch may seem dras-

tic and inefficient. We resolve the inefficiency by using a memoization cache for $\in$. This cache is explained in more detail later. Learning needs to be restarted because learning algorithms in the LearnLib can not deal with removing symbols from an alphabet (which is the case for alphabet refinement).

Algorithm 4.1: Main algorithm for LBT with alphabet abstractions

Input : Alphabet Σ , membership oracles $\in, \in^\pi$
Output: the final hypothesis

```

1  $\Pi \leftarrow \{\Sigma\}$  ▷ initialize (coarsest) partitioned alphabet
2  $H^\pi \leftarrow \text{Init}(\Pi, \in^\pi)$  ▷ initialize approximation
3  $X \leftarrow \{\}$  ▷ initialize set of counterexamples
4 do ▷ main loop for refining hypotheses
5    $H \leftarrow \Gamma(H^\pi)$  ▷ concretize hypothesis
6    $q \leftarrow \text{Equiv}(H, \Sigma, \in)$  ▷ do equivalence check
7    $P \leftarrow \text{Split}(\Pi, q, \in)$  ▷ compute possible pivot based on query
8   if  $\text{Refine}(\Pi, P) < \Pi$  then ▷ check if alphabet must be refined
9      $\Pi \leftarrow \text{Refine}(\Pi, P)$  ▷ refine alphabet
10     $X \leftarrow \{x \in X \mid x \in \Pi^*\}$  ▷ retain only legal counterexamples
11     $H^\pi \leftarrow \text{Init}(\Pi, \in^\pi)$  ▷ re-initialize hypothesis
12    for  $x \in X$  do  $H^\pi = \text{Refine}(H^\pi, \Pi, \in^\pi, x)$  ▷ reuse queries
13  else ▷ no controllable non-determinism
14     $q^\pi \leftarrow \alpha(q)$  ▷ make query abstract
15     $X \leftarrow X \cup \{q^\pi\}$  ▷ save counterexample
16     $H^\pi \leftarrow \text{Refine}(H^\pi, \Pi, \in^\pi, q^\pi)$  ▷ refine hypothesis
17 while  $\lambda_H^*(q) \neq \in(q)$  ▷ compare outputs of query
18 return  $\Gamma(H^\pi)$  ▷ return the final concretized hypothesis

```

4.4.4 The New LearnLib API

We now present the extension to the LearnLib API that is used to implement Algorithm 4.1 with Java. Figure 4.2 shows all interfaces and classes we added to the LearnLib. Some interfaces and classes that were already available are shown as well. Classes are shown in red and interfaces in blue. Classes and interfaces that are new are darker, while existing classes and interfaces are lighter. We will cover the behavior of all classes and how they relate to the presented algorithm.

PartedAlphabet is an interface that specifies the behavior of a partitioned alphabet according to Definitions 4.1 and 4.2. Note that implementations of learning algorithms (such as **TTT**, **LStar**, etc.) require an **Alphabet** instance for learning. Interface **PartedAlphabet** refines **Alphabet** so that **TTT**, **LStar**, etc. can learn over a partitioned alphabet as well.

Figure 4.2: LearnLib API: classes are red, interfaces are blue, extensions are darker. Solid arrows depict interface refinements, and dashed arrows implementations.

TreePartedAlphabet is an implementation of **PartedAlphabet**, where maintaining blocks as a **TreeSet** is its key feature. Class **TreeSet** guarantees an iteration order on its elements. In our case this is the lexicographic order on alphabet symbols. Cartesian products that are computed based on this alphabet also contain elements in lexicographic order. This means for instance that membership oracles for abstract words behave deterministically. Observe **TreePartedAlphabet** has an attribute named **finest**. When **TreePartedAlphabet** is constructed with **finest=true**, then it is constructed as the finest partitioning. This method is used later in the experiments to simulate learning with no partitioning of the alphabet. Class **TreePartedAlphabet** also has an attribute **Collection** that contains the original **Alphabet**.

PiMQOracle is a simple refinement of **MembershipOracle**; instead of answering queries over concrete symbols, it answers queries over blocks of symbols, following Definition 4.9. The fact that **PiMQOracle** is a refinement of **MembershipOracle** means regular learning algorithms (TTT, LStar, etc.) can learn an approximation based on the alphabet of a system.

ProdMQOracle is an implementation of **PiMQOracle** that will first compute the Cartesian product of blocks from the abstract query. Next it will use a regular **MembershipOracle** to pick the first element in the Cartesian product and use it to answer the abstract query. Note that the Cartesian product is computed on-the-fly, meaning that concrete instances of queries will only be computed when requested.

PiLearner: This class implements a large part of Algorithm 4.1. In particular this class executes the initialization of the actual learning algorithm at lines 2–3; these lines form the implementation of the **Learner#init()** method. Lines 5–16 of the main while loop are executed as well; they form the implementation of **Learner#refine(Query)** method. Observe that the attribute **PartedAlphabet** is the partitioned alphabet Π that is initialized at line 1. Furthermore attribute **Learner** is the actual learning algorithm that will learn an automaton (DFA, or Mealy machine) over Π . To this end, **init()** of this **Learner** is called at lines 2 and 11. Method **Learner#refine(Query)** of the underlying learner is called at lines 12 and 16. The method **getHyp()** of **PiLearner** is called at lines 5 and 18 and actually uses **Learner#getHyp()** of the underlying algorithm and an implementation of the Γ function in Definition 4.8. Instances of this underlying learning algorithm (i.e. attribute **Learner**) can be TTT, LStar, or any other existing learning algorithm for DFAs or Mealy machines that is implemented according to Definitions 4.5 and 4.6. Attribute **SplitterOracle** is consulted at line 7.

SplitterOracle: specifies one method named **findCEx(PartedAlphabet, Query)**. As the name suggests, it finds a counterexample to the claim there is no controllable non-determinism. If there is controllable non-determinism, a counterexample (i.e.

a pivot) is returned which is a set of concrete alphabet symbols that refines the alphabet. The query, that is the second parameter, should aid implementations for computing a “good” pivot. It is called an oracle as it is inspired by the MAT framework by Angluin. For one, oracles interact with the system that is being learned. The second reason is because one can imagine computing a pivot is undecidable when blocks in the partition are of infinite size.

AcexSplitterOracle is an implementation of **SplitterOracle**, based on the Abstract CounterEXample analyzer (ACEX) framework in [IS14]. Attribute **MembershipOracle** is used to resolve the controllable non-determinism, while attribute **AcexAnalyzer** implements the search strategy: linear, binary or exponential, and in a backward or forward fashion.

CounterMQOracle is an (existing) implementation of **MembershipOracle**; it simply counts the number of queries performed as well as the total aggregated length of these queries.

CacheMQOracle is another (existing) implementation of **MembershipOracle**; it memoizes answers to previous queries. This cache is a key ingredient for the performance of Algorithm 4.1. The performance follows from the design of Algorithm 4.1, specifically lines 9–12. At these lines the alphabet is refined, the learning is restarted and previous counterexamples reused. To prevent learning algorithms from querying the system for what should already be known the memoization cache is used.

To count only the number of unique queries on the SUL, we use a chain of membership oracles, shown in Figure 4.3.

Figure 4.3: The chain of membership oracles

First, recall that learning algorithms need answers to abstract queries (hence **ProdMQOracle** is used, whereas splitter oracles and equivalence oracles need answers for concrete queries (based on a concretized hypothesis automaton). Queries from learning algorithms first pass through the **ProdMQOracle**. For splitter oracles and equivalence oracles the flow of queries starts at the cache, then the counter and lastly the system. This means always only unique queries performed on the

system are counted.

4.5 Results

We benchmarked on problems from the 2017 Rigorous Examination of Reactive Systems (RERS), with the main purpose of showing that our new algorithm can disprove LTL formulae faster (i.e. with less unique queries) than the algorithms in Chapter 3. This means we can follow the same experimental approach as in Section 3.5. We thus use the exact same implementation for interfacing with the RERS problem instances as in Section 3.5.2 and show similar graphs as in Section 3.5.3. However, we fix the variables from Section 3.5.1 to using the `DisproveFirstOracle` and using a combination of both Büchi automata and monitors. This is because we already know from the results in Chapter 3 this works best. Our new variable simply becomes whether we start with the *finest* or *coarsest* initial alphabet partitioning. As a last note, this means we still apply the same eight learning algorithms, and measure the performance identically to how it is discussed in Section 3.5.1.

4.5.1 Variables, Metrics and Constants

Concretely, we adapt the experimental setup from Section 3.5.1 as follows. The variables of our experiment (learning algorithm and initial partitioning strategy) are as follows.

- Eight learning algorithms: ADT [Fro15, Fro19], DHC [Mer+11], Discrimination Tree [How12], L^* [Ang87], Kearns & Vazirani [KV94], Maler & Pnueli [MP95], Rivest & Schapire [RS93], and TTT [IHS14b].
- Two initial partitioning strategies: *coarsest* and *finest*.

To measure the performance of the above variations, we measure the following each time a property is disproved.

1. The number of states in the hypothesis.
2. The number of queries in total.
3. The number of learning queries.
4. The number of equivalence queries.
5. The number of emptiness queries.
6. The number of inclusion queries.
7. The aggregated length of each of above query types i.e. the total number of symbols.
8. The length of the counterexample that disproves the property.

The constants in the experiments are 9 RERS problems with each 100 LTL for-

Figure 4.4: Legend for learning algorithms and initial partitioning

Figure 4.5: Number of *total* queries on problem instance 1 (5 alphabet symbols), comparing various AAL algorithms and initial partitioning strategies

mulae. Those RERS problems are discussed in Section 3.5.2. Typically half of the 100 LTL formulae can be falsified. In total we have $(8 * 2 =) 16$ variations, and $(16 * 9 =) 144$ experiments (in a single experiment all LTL properties are checked). These experiments are run on the University of Twente CTIT compute cluster with a time-out of three hours.

In this chapter there is one main research question we want to answer: *does automated partition refinement of the alphabet contribute to the efficiency for falsifying LTL formulae?*

4.5.2 Discussion of the Algorithm's Performance

The performance of Algorithm 4.1 is illustrated with cactus plots. The legend of these plots is shown in Figure 4.4. Figure 4.5 shows the number of total queries on RERS problem instance 1 only. Problem instance 1's alphabet has 5 symbols.

Figure 4.6: Number of *total* queries on problem instance 5 (15 alphabet symbols), comparing various AAL algorithms and initial partitioning strategies

The x-axis shows the number of queries required to disprove a certain number of properties. The y-axis shows the amount of properties that are disproved. The line color differentiates the learning algorithms and the type of line differentiates the initial partitioning strategies. The further a line appears to the left; the better the learning algorithm and partitioning strategy.

The cactus plot of problem 1 shows which partitioning strategy works best fluctuates between both *coarsest* and *finest*, depending on the amount of formulae that are disproved. This fluctuation is seen among several other RERS problem instances as well (which are not shown). The gain can increase with the size of the alphabet however; illustrated in Figure 4.6. Here the result for RERS problem instance 5 is shown, which has 15 alphabet symbols (10 more than problem instance 1). In the latter figure it shows that especially the ADT algorithm together with the *coarsest* partitioning strategy wins for disproving most of the LTL formulae. This shows because the solid line of ADT appears overall further to the left than the dashed line of ADT. More crucially, within the timeout of three hours, many algorithms with the *coarsest* initial partitioning were able to disprove more LTL formulae. This can be seen in Figure 4.6 because the solid lines ultimately appear higher.

4.6 Conclusion and Future Work

We have presented a learning algorithm that is able to deal increasingly better with problems where alphabets grow bigger. In particular, we show that the ADT algorithm in combination with alphabet abstractions can be applied to large problems from the 2017 RERS challenge. Here we found that there are certain RERS problem instances where we can reduce the number of unique queries necessary to falsify LTL formulae, compared to traditional learning algorithms without alphabet abstractions. In some cases we are even able to falsify formulae that were not falsifiable before.

Future work includes applying the presented approach to real-world case studies where alphabet symbols with data parameters play a crucial role. For example, promising applications are in the context of testing web applications [Bai+16], testing SSH implementations [Fit+17] and testing TCP implementations [FJV16, FH17]. Furthermore, in the future we would also like to compare the approach presented in this chapter to a mapper approach [Aar14], or state-local abstractions [IHS13].

Another practical use case where we foresee our modular API design pays off is in the context of analyzing LTL formulae on Petri nets from the Model Checking Contest (MCC) [Amp+19]. LBT for Petri nets is straightforward, because their marking graph is simply a DFA with one rejecting state. The alphabet in this setup contains all transitions of the Petri net, and allows one to learn the marking graph. These marking graphs can have a huge amount of states however, and without alphabet abstractions analyzing LTL formulae may not be tractable.

We like to investigate whether custom heuristics for an approximating membership oracle (cf. `PiMQOracle` in Figure 4.2 and Definition 4.9) are suitable for Petri nets. The current implementation of `PiMQOracle`, called `ProdMQOracle` simply returns the output of the first element in the Cartesian product of an abstract query. When this naive heuristic outputs “false”, then this means a whole class of words is seen as rejected. This is not really useful in the model checking step, where only accepting words are considered. We like to see a new implementation of `PiMQOracle` for DFAs with a heuristic that favors accepting words over rejecting words. With this heuristic we can investigate how well our approach performs in the MCC. There are currently no approaches that use learning-based testing in the MCC verification competition.

Partitioned Transition Systems and Dependencies

Abstract

This chapter aims at improving symbolic reachability for explicit state modeling languages, e.g., Promela, DVE and mCRL2. The modular PINS architecture of LTSmin used to support a notion of event locality, by merely indicating for each event on which variables it depends. However, one could distinguish three separate dependencies: *read*, *write* and *copy*. In this chapter, we introduce these notions in a language-independent manner. In particular, specifications with arrays need to distinguish overwriting and copying of values.

We also adapt the symbolic reachability algorithms to exploit the refined dependency information. We have implemented refined dependency matrices for Promela, DVE and mCRL2, in order to compare our new algorithms to the original version of LTSmin. The results show that the amount of successor computations and memory footprint can be reduced significantly.

5.1 Introduction

Model checking [Cla08] is a technique to verify the correctness of systems. Often these systems are made up of several processes running in parallel. Examining all possible execution paths of the system is hard, because of the well known state space explosion problem: because of the interleaving of the processes, the possible number of states is exponential in the number of processes. Symbolic model checking [McM93, Bur+92] has proven to be very effective in dealing with that problem. Symbolic here means storing sets of vectors and relations between vectors as decision diagrams, such as Binary Decision Diagrams (BDDs) or Multi-Value Decision Diagrams (MDDs). Implementations of decision diagrams (e.g.

Figure 5.1: Modular PINS architecture of LTSmin

BuDDy, CUDD and Sylvan [DP17, Dij16]) support several set operations such as union, intersection and relational product to facilitate state space generation. A well known symbolic model checker is NuSMV [Cim+00], where systems are specified in the SMV language, directly describing transition relations.

We use the LTSmin toolset [BPW10, Kan+15], which also provides a symbolic model checker, but is different from NuSMV in several ways. LTSmin provides a language independent interface, called PINS, to communicate states and transitions, and learns the partitioned transition relation on-the-fly, as in, e.g., [CMS03, BP08]. New transitions are learned through an explicit NEXT-STATE [CMS06] function, which is the language specific part of LTSmin. Currently the languages that have been implemented on top of PINS include Promela [BL13], DiVinE, mCRL2, PNML, ProB and UPPAAL. In [BP08, BPW09], the PINS interface and underlying symbolic core algorithms of LTSmin are described. An overview of the architecture is given in Figure 5.1. Highlighted in blue are the components that are improved in this chapter. The (green) PNML front-end is added to support the experimental section of Chapter 6. The improvements for languages supported by the ProB (red) model checker are discussed in Chapter 7.

In PINS, states are represented as fixed-length *vectors* of values. Transitions are distinguished in separate disjunctive *transition groups*. A generalized definition of systems that is compatible with PINS is given in Section 5.3. Between slots of the state vector and the transition groups there can be dependencies, i.e. a transition group can be dependent on the value of a state variable for a condition to be true, or a transition may change the value of a state variable. The dependencies between transition groups and state slots are captured in a *dependency matrix*, which can be determined by static analysis of the specification. Often it is the case that a transition group depends on a limited number of slots, which is known as *event locality*. This is the basis of many optimizations in symbolic model checking, as

presented in, e.g., [Cim+00, BCL91, CY05]. For symbolic state space generation it is best when the dependency matrix is sparse, i.e. when transition groups have a relatively local footprint, for the following reasons. First, a sparse matrix means that the transition relations for the transition groups depend on few variables and can be quite small. Also, because of the on-the-fly nature of LTSmin, there will be fewer redundant calls to NEXT-STATE.

To further benefit from dependencies in the input specification, in this chapter we refine the notion of dependency and distinguish three types of dependencies: *read dependence*, *write dependence* and *copy dependence*. The latter form is merely the absence of read dependence and write dependence. To illustrate the multiple forms of dependence, we use a transition system specification with three variables (i, a_0, a_1), where i can be used to access the array a (with length 2) at index 0 or 1¹. The specification has two transition groups:

$$\begin{array}{ll} 1 & : i = 1 \vee a_1 = 0 \quad \rightarrow a_0 := 1, i := 0 \\ 2 & : a_0 = 1 \quad \rightarrow a_1 := 0, i := 1 \end{array}$$

(a) Two transition groups

$$\begin{array}{ccc} & i & a_0 & a_1 \\ 1 & \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \\ 2 & \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \end{array}$$

(b) Dependency matrix

$$\begin{array}{ccc} & i & a_0 & a_1 \\ 1 & \begin{bmatrix} 1 & 0 & 1 \end{bmatrix} \\ 2 & \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \end{array}$$

(c) Read matrix

$$\begin{array}{ccc} & i & a_0 & a_1 \\ 1 & \begin{bmatrix} 1 & 1 & 0 \end{bmatrix} \\ 2 & \begin{bmatrix} 1 & 0 & 1 \end{bmatrix} \end{array}$$

(d) Write matrix

Figure 5.2: Example transition system specification

In Figure 5.2b, the dependency matrix indicates no event locality, but if read and write dependencies are considered separately, as in Figures 5.2c and 5.2d, then we see the transition groups depend on most variables only for reading or only for writing. Separating reads and writes helps in reducing the number of NEXT-STATE calls, the size of the transition relation, and the size of its symbolic representation. For instance, when a transition group is not read dependent on variable a_0 , then the previous value of a_0 can be discarded in computing the successors of a state. The redundant number of successor computations a NEXT-STATE function would do based on the dependency matrix in Figure 5.2b is illustrated in Example 5.1.

Example 5.1: Dependency analysis

Table 5.1 below shows all 10 transitions specified in Figure 5.2 that can be

¹ a_i may also be read as $a[i]$, used commonly in computer programming languages.

taken from all eight states, assuming that variables i, a_0, a_1 are Booleans. The column $\#$ indicates transition numbers, columns with unprimed variables (i, a_0, a_1) denote valuations that form current states, columns with primed variables (i', a'_0, a'_1) form successor states and column $\rightarrow$ denotes the first or second transition group in Figure 5.2a. Note that state $\langle 0, 0, 1 \rangle$ is a special case that has no successors.

#	i	a_0	a_1	i'	a'_0	a'_1	$\rightarrow$
1	0	0	0	0	1	0	1
—	0	0	1	—			
2	0	1	0	0	1	0	1
3	0	1	0	1	1	0	2
4	0	1	1	1	1	0	2
5	1	0	0	0	1	0	1
6	1	0	1	0	1	1	1
7	1	1	0	0	1	0	1
8	1	1	0	1	1	0	2
9	1	1	1	0	1	1	1
10	1	1	1	1	1	0	2

Table 5.1: NEXT-STATE based on combined dependencies (old situation). Dashed lines distinguish transition groups, solid lines distinguish current states.

A NEXT-STATE function that is based on a dependency matrix without read and write separation (e.g. Figure 5.2b) would have to make 10 distinct successor computations. We have highlighted the values that are actually read (in the unprimed columns), and those that are actually overwritten (in the primed columns). By projecting the states down to the highlighted values, one can identify equivalent transitions. For transition group $\rightarrow = 1$, these are:

$$\{\{1, 2\}, \{5, 7\}, \{6, 9\}\}.$$

For transition group $\rightarrow = 2$, these are:

$$\{\{3, 4, 8, 10\}\}.$$

Because certain transitions can be seen as equivalent, we devise a NEXT-STATE function based on read and write dependency matrices (e.g. Figures 5.2c and 5.2d). The four necessary successor computations are illustrated in Table 5.2.

#	i	a_0	a_1	i'	a'_0	a'_1	$\rightarrow$
1,2	0	▲	0	0	1	▲	1
3,4,8,10	▲	1	▲	1	▲	0	2
5,7	1	▲	0	0	1	▲	1
6,9	1	▲	1	0	1	▲	1

Table 5.2: NEXT-STATE based on read and write separation (new situation).

Column # now lists the equivalent transitions and ▲ serves as a wildcard. A ▲ in the unprimed columns indicates the actual value is ignored. A ▲ in the primed columns indicates the actual value is copied from the respective unprimed column. It thus takes less time to compute Table 5.2 than Table 5.1. Tables like Table 5.1 from other problem instances can always be inferred from tables like Table 5.2 as well.

In Table 5.2 it can be seen that transition group 1 never writes to a_1 and group 2 never writes to a_0 . In general however it is not trivial to statically determine whether a transition group writes to a state slot. Languages such as Promela, DVE and C support dynamic addressing of variables. Assignments with dynamic addressing come in the form of for instance array subscripting or conditional assignments. A conditional assignment is of the form:

$$\text{if } i = 0 \text{ then } a_0 := i \text{ else } a_1 := i.$$

In the expression above, the array index that is written depends on the value of i (the value that is written also depends on i). Array subscripting is only syntactically different from conditional assignment. Assignment with dynamic addressing is further elaborated with array subscripting in Example 5.2.

Example 5.2: Assignment with dynamic addressing

Suppose we extend Figure 5.2a with a third group:

$$3 : a_i := i.$$

This group writes to either a_0 or a_1 exclusively, depending on the value of i , and the value that is written also depends on i . This group has the following associated matrices:

- dependency matrix:
$$3 \begin{bmatrix} i & a_0 & a_1 \\ 1 & 1 & 1 \end{bmatrix},$$

- read matrix:
$$3 \begin{bmatrix} i & a_0 & a_1 \\ 1 & 0 & 0 \end{bmatrix},$$
- write matrix:
$$3 \begin{bmatrix} i & a_0 & a_1 \\ 0 & 1 & 1 \end{bmatrix}.$$

Note that the dependency matrix and write matrix are special cases, we know that either a_0 or a_1 is written, so we need to set both entries to 1. Without read and write separation, the transition table for group 3 is given in Table 5.3.

#	i	a_0	a_1	i'	a'_0	a'_1	$\rightarrow$
11	0	0	0	0	0	0	3
12	0	0	1	0	0	1	3
13	0	1	0	0	0	0	3
14	0	1	1	0	0	1	3
15	1	0	0	1	0	1	3
16	1	0	1	1	0	1	3
17	1	1	0	1	1	1	3
18	1	1	1	1	1	1	3

Table 5.3: NEXT-STATE with dynamic addressing (old situation).

When we project the states down to columns that have highlighted values (i.e. those that have been read or written), the equivalent groups are:

$$\{\{11, 12, 13, 14\}, \{15, 16, 17, 18\}\}.$$

Based on the groups that are equivalent, the necessary successor computations are illustrated in Table 5.4.

#	i	a_0	a_1	i'	a'_0	a'_1	$\rightarrow$
11,12,13,14	0	▲	▲	▲	0	▲	3
15,16,17,18	1	▲	▲	▲	▲	1	3

Table 5.4: NEXT-STATE with dynamic addressing and read and write separation (new situation).

Similar to Table 5.2, symbol ▲ has the same meaning: for unprimed columns the value is ignored, and for primed columns the value is copied. Furthermore, Table 5.4 can be computed in less time than Table 5.3 and the latter can be inferred from the first.

Note the crucial difference between Tables 5.2 and 5.4: in the first table, groups 1 and 2 always write the correct value. Group 3 in some cases copies the value, and sometimes writes the correct value. In other words, if a group in Table 5.2 has one $\blacktriangle$ for a primed column, then the group always has a $\blacktriangle$ for that primed column. Similarly, if a group in Table 5.2 writes a value, then it always writes a value. In Table 5.4 this is not true; assignment of variables is resolved dynamically (not statically), hence instances of $\blacktriangle$ need to be stored at *runtime*. Dynamic assignment is one of the key issues addressed in this chapter. The goal of this chapter is to provide a framework for on-the-fly symbolic reachability analysis with separate read and (dynamic) write dependencies. One key aspect of this framework is that it is language agnostic and thus abstracts away from language specifics (e.g., array subscripting and conditional assignments are generalized to dynamic assignment).

In [CY05], a similar distinction is made between types of dependence. The main difference with this work is that we deal with dynamic variable addressing, both in the definitions of dependency and in the symbolic algorithms, where we use a special symbol ($\blacktriangle$) in transition relations to denote that a value should be copied.

The dependencies and the associated matrices are described in detail in Section 5.4. In Section 5.5 the *row subsumption* on dependency matrices is discussed. This technique can improve the efficiency of the read-write distinction even more. In Section 5.6, we provide an adapted symbolic reachability algorithm that exploits the read and write dependencies.

We have benchmarked our work with the whole BEEM database and many Promela and mCRL2 specifications. There are many specifications that benefit from the distinction between read and write dependencies, but also several that do not. In Section 5.7, we highlight the results for six specifications. For mCRL2, BEEM and Promela the number of NEXT-STATE calls is reduced. The NEXT-STATE function for mCRL2 is relatively slow, due to the term rewriter that was introduced to provide very expressive data types. This means the runtime of reachability algorithm is improved drastically for mCRL2 in particular.

5.2 Reachability Analysis

Reachability analysis involves analyzing whether or not a system can enter a particular state. LTSmin supports reachability analysis for a large class of languages, such as PNML, Promela, DVE, mCRL2, etc. In the rest of the chapter we continue with two running examples: the specification in pseudo code in Figure 5.2, and its extension of a third group in Example 5.2. The pseudo code with dynamic assignment is used to illustrate particular cases, indicating why our formal notion of dependencies is correct. The other running example is a Petri net that is given

in Example 5.3. The Petri net is used to illustrate reachability analysis and also serves as the running example in Chapter 6 to illustrate variable ordering².

Example 5.3: Reachability Analysis of a 1-safe Petri net

Given below is an example 1-safe Petri net with 5 places and 6 transitions.

A Petri net is a bipartite graph, where vertices represent places (circles) or transitions (squares). Places contain a non-negative number of tokens (dots). The edges in the graph are called arcs. On places an outgoing arc means that tokens will be consumed and an incoming arc means that tokens will be produced. A state (or marking) of a Petri net is a vector with the number of tokens in each place. For example, the marking of the Petri net shown above is $\langle 0, 0, 1, 0, 0 \rangle$ for places p_1, p_2, p_3, p_4, p_5 respectively. In the Petri net above, after transition t_1 fires, p_3 will have no token, while both p_1 and p_2 get one token. In total the Petri net in question has five reachable states, and 10 reachable transitions that can fire. In case a given Petri net is 1-safe, then its enabled transitions produce only a single new token in each target place. A reachability question might be whether or not p_5 will eventually have a token, which it will, after firing t_1 followed by t_4 .

Petri nets model many kinds of systems, like distributed protocols, control and data flow in concurrent software, business processes, or even biological systems. A formal notion that describes a Petri net's marking graph is called a transition system.

²The Petri net can be specified in PNML, and since PNML is an input language for LTSmin, we can automatically generate the decision diagrams for the Petri net (with different variable orders).

5.2.1 Transition Systems

A state of a Petri net is a marking, and firing a transition produces a new marking of a Petri net. Such states and transitions form the key objects of transition systems. Other specification languages also describe transition systems; we formalize the concept as follows.

Definition 5.1: Transition System

A Transition System (TS) is a structure $\langle S, \rightarrow, s^0 \rangle$, where

- S is a set of states, and
- $\rightarrow \subseteq S \times S$ is a transition relation, and
- $s^0 \in S$ is the initial state.

Furthermore let $\rightarrow^*$ be the reflexive and transitive closure of $\rightarrow$, then the set of reachable states is $R = \{s \in S \mid s^0 \rightarrow^* s\}$.

Example 5.4: Transition System

For the transition system of Example 5.3 we have $|S| = 2^5$, $|R| = 5$, $|\rightarrow| \geq 10$ and s^0 is the initial marking shown.

Computing a huge set of reachable states R is very time and memory consuming. Many techniques exist to alleviate this problem. In this chapter, we focus on symbolic reachability analysis. Storing the set of reachable states symbolically involves using Boolean expressions to represent this set. Symbolic reachability analysis works well when there is a high locality of events. To precisely describe event locality, we introduce a more fine-grained view of a transition system called a partitioned transition system that is compatible with the PINS interface.

5.3 The Partitioned Next-State Interface

The starting point of our approach is a generalized structure for systems, called the *Partitioned Interface to the Next-State function* (PINS), which supports several specification languages within a single framework, without exposing language details to the underlying algorithms.

In PINS, states are vectors of N values. We write $\langle x_1, \dots, x_N \rangle$, or simply $\mathbf{x}$, for vector variables. Each slot of the vector has a unique identifier, which is used in the language front ends to specify conditions and updates. Every language module furthermore, has a NEXT-STATE function which computes the successor(s)

of a state. This function is *disjunctively* partitioned in K transition groups, such that

$$\text{NEXT-STATE}(\mathbf{x}) = \bigcup_{1 \leq i \leq K} \text{NEXT-STATE}_i(\mathbf{x}).$$

A specification available through PINS, gives rise to a partitioned transition system, defined as follows.

Definition 5.2: Partitioned Transition System

A *Partitioned Transition System* (PTS) [BPW09] is a structure $\mathcal{P} = \langle \langle S_1, \dots, S_N \rangle, \langle \rightarrow_1, \dots, \rightarrow_K \rangle, \langle s_1^0, \dots, s_N^0 \rangle \rangle$, where

- $\langle S_1, \dots, S_N \rangle$ defines the set of states $S_{\mathcal{P}} = S_1 \times \dots \times S_N$, i.e., we assume that the set of states is a *Cartesian product*, and
- the *transition groups* $\rightarrow_i \subseteq S_{\mathcal{P}} \times S_{\mathcal{P}}$ (for $1 \leq i \leq K$) define the transition relation $\rightarrow_{\mathcal{P}} = \bigcup_{i=1}^K \rightarrow_i$, and
- the initial state is $\mathbf{s}^0 = \langle s_1^0, \dots, s_N^0 \rangle \in S_{\mathcal{P}}$.

The defined TS of $\mathcal{P}$ is $\langle S_{\mathcal{P}}, \rightarrow_{\mathcal{P}}, \mathbf{s}^0 \rangle$. We write $\mathbf{s} \rightarrow_i \mathbf{t}$ when $\langle \mathbf{s}, \mathbf{t} \rangle \in \rightarrow_i$ for some $1 \leq i \leq K$. Also we write $\mathbf{s} \rightarrow_{\mathcal{P}} \mathbf{t}$ when $\langle \mathbf{s}, \mathbf{t} \rangle \in \rightarrow_{\mathcal{P}}$.

Example 5.5: Partitioned Transition System

Below is illustrated a possible (but common) interpretation of the Petri net in Example 5.3 as PTS.

The three objects of the PTS are as follows. Each state is a vector of five slots, i.e. $N = 5$. Each vector element indicates the number of tokens in its respective place. This means a state (or marking) is an element in $\mathbb{N}^N$. We implicitly assign each Petri net transition t_i to group i , i.e. $K = 6$. This means two transitions in the PTS are assigned to each groups $t_2 \dots t_5$, and one transition in the PTS to groups t_1 and t_6 . The initial state is $s^0 = \langle 0, 0, 1, 0, 0 \rangle$.

In the next example we illustrate how a set of reachable states can be encoded as a List Decision Diagram (LDD).

Example 5.6: Example List Decision Diagram (LDD)

The following figure is the LDD of all reachable states in Example 5.5, with the lexicographic order on variables $p_1 \dots p_5$. Every path from the top left node (with value 0) to $\{\epsilon\}$ forms a vector and is a state in the PTS of Example 5.5 (and vice versa).

LDDs can store states in a very compact manner, because of sharing between nodes (e.g. some nodes in Example 5.6 have multiple parent nodes). A sketch of a symbolic reachability algorithm that computes a set of reachable states with decision diagrams is as follows. The algorithm maintains a set of current states

$\mathcal{R}$, that initially contains the initial state. In the first step of the main loop new states are discovered by the NEXT-STATE_i functions. However, this is done in an optimized manner; NEXT-STATE_i only considers *dependent* variables. For example, in Example 5.3 the next state function for transition t_1 only has to consider variables for p_1 , p_2 and p_3 . The relation between current states and successor states that are discovered is stored in a relation for group i , based on the dependent variables only. In the second step all successor states are computed by applying all expanded relations to $\mathcal{R}$. A relation for group i that is applied to $\mathcal{R}$ gives all successor states in one go. For example, even though a relation for group t_1 only stores the changes for variables p_1 , p_2 and p_3 , it is known that the values for p_4 and p_5 can be copied. All successor states are added to $\mathcal{R}$, and the main loop continues until $\mathcal{R}$ does not change anymore. Because the change specified by a relation for group i is local, only few new nodes are created in the LDD that encodes $\mathcal{R}$. This key aspect of decision diagrams enables the speed up that is often seen in symbolic reachability algorithms over explicit-state reachability algorithms. In Section 5.6 we provide a new algorithm based on read-write separation, that in many cases speeds up reachability analysis even further.

5.3.1 The State Update Specification Language

The partitioning of the state vector into *slots* and the partitioning of the transition relations into *transition groups*, enables specifying the *dependencies* between the two, i.e. which transition groups touch which slots of the vector. The definition of these dependencies will be given in Section 5.4. Here we give an abstract description of how the variables in the state vector are read from and written to by the transition groups.

For every language module reading and writing is implemented differently, but there is a common pattern. In all of the supported languages, the specification of a transition is of the form:

$$\text{NEXT-STATE}_i(\mathbf{x}) = \text{cond}_i \rightarrow \text{action}_i . \text{update}_i(\mathbf{x}).$$

The expression cond_i is the condition that guards an action and may read variables from $\mathbf{x}$. The symbol ‘ action_i ’ specifies the name of the action that is performed, i.e., the transition label. The expression $\text{update}_i(\mathbf{x})$ defines the state after the action. The update is an atomic assignment to the variables in the vector. However, these variables may be assigned to dynamically, e.g. they may be references to a location in an array.

We define state updates formally in a language called State Update Specification (SUS). The SUS language supports dynamic assignments for instance, and is used

to illustrate how one should implement dependency matrices, in general. These dependency matrices capture the different dependency relations.

Definition 5.3: State update specification

The syntax of a state update of transition group i is as follows: $\sigma_i ::= c_i \rightarrow a_i \cdot \langle v_{i,1} := t_{i,1}, \dots, v_{i,L_i} := t_{i,L_i} \rangle$, where $L_i \leq N$ and c_i , $v_{i,j}$, and $t_{i,j}$ are expressions over variables $x_1, \dots, x_N$. The conditions c_i are Boolean expressions and the left hand sides $v_{i,j}$ evaluate to variables in $\{x_1, \dots, x_N\}$. The semantics of this state update is defined as the successor states after applying the update:

$$s \rightarrow_i t \iff \llbracket c_i \rrbracket_s \wedge t = s[\llbracket v_{i,1} \rrbracket_s := \llbracket t_{i,1} \rrbracket_s, \dots, \llbracket v_{i,L_i} \rrbracket_s := \llbracket t_{i,L_i} \rrbracket_s].$$

A *State Update Specification* (SUS) is a structure $\mathcal{U} = \langle \langle x_1, \dots, x_N \rangle, \{\sigma_1, \dots, \sigma_K\}, s^0 \rangle$, containing a vector of state variables x_j for $1 \leq j \leq N$, a set of *state updates* σ_i with $1 \leq i \leq K$, and an initial state s^0 .

Example SUSs are given in Examples 5.7 and 5.8.

Example 5.7: State Update Specification 1

In the listing below a SUS of the 1-safe Petri net in Example 5.3 is given.

```

1  init  ⟨true, false, false, false, false⟩
2  sus   ⟨p1, p2, p3, p4, p5 ∈ ℬ⟩ = ⟨
3      p3 →
4          t1 . ⟨p3 := false, p1 := true, p2 := true⟩,
5      p1 →
6          t2 . ⟨p1 := false, p4 := true⟩,
7      p4 →
8          t3 . ⟨p4 := false, p1 := true⟩,
9      p2 →
10         t4 . ⟨p2 := false, p5 := true⟩,
11     p5 →
12         t5 . ⟨p5 := false, p2 := true⟩,
13     (p4 ∧ p5) →
14         t6 . ⟨p4 := false, p5 := false, p3 := true⟩⟩

```

Example 5.8: State Update Specification 2

The following listing is a SUS of the specification in Figure 5.2, and its extension of a third group in Example 5.2. The initial state and action labels are chosen somewhat arbitrarily.

```

1 init   $\langle 0, 0, 0 \rangle$ 
2 sus   $\langle i, a_0, a_1 \in \{0, 1\} \rangle = \langle$ 
3       $i = 1 \vee a_1 = 0 \rightarrow$ 
4       $g_1 \cdot \langle a_0 := 1, i := 0 \rangle,$ 
5       $a_0 = 1 \rightarrow$ 
6       $g_2 \cdot \langle a_1 := 0, i := 1 \rangle,$ 
7       $\text{true} \rightarrow$ 
8       $g_3 \cdot \langle a_i := i \rangle$ 

```

5.4 State Slot Dependencies

We exploit the notion of event locality by statically (a priori, before exploring any states) approximating dependencies between transition groups and state slots. We distinguish three types of dependencies: *read dependence* (whether the value of a state slot influences transitions), *write dependence* (whether a state slot is written to dynamically), and *copy dependence* (the absence of read and write dependence). We provide formal definitions for read dependence, write dependence and dependency matrices on a semantic level for PTSs. For each concrete specification language these have to be approximated and implemented. We illustrate how this can be done for SUSs, in particular for Examples 5.7 and 5.8.

5.4.1 The Read Dependency

The notion of read independence was first defined in [Mei+14] and later strengthened in [Ben+16a]. Definition 5.4 is copied from [Mei+14] and thus weaker than the latter one.

Definition 5.4: Read independence

Given a PTS $\mathcal{P} = \langle S_{\mathcal{P}}, \rightarrow_{\mathcal{P}}, \mathbf{s}^0 \rangle$, transition group i is *read independent* from state slot j if: for all $\mathbf{s}, \mathbf{t} \in S_{\mathcal{P}}$: whenever $\langle s_1, \dots, s_j, \dots, s_N \rangle \rightarrow_i \langle t_1, \dots, t_j, \dots, t_N \rangle$, it holds that

- either $(s_j = t_j) \wedge \forall r_j \in S_j: \langle s_1, \dots, r_j, \dots, s_N \rangle \rightarrow_i \langle t_1, \dots, r_j, \dots, t_N \rangle$, or

- $\forall r_j \in S_j: \langle s_1, \dots, r_j, \dots, s_N \rangle \rightarrow_i \langle t_1, \dots, t_j, \dots, t_N \rangle,$

i.e. the values t_k for $k \neq j$ do not depend on the value of slot j and either the value of state slot j is always copied, or always the value t_j is written, regardless of the value of s_j . In both cases the specific value of s_j is not relevant in transition group i .

The definition used here allows groups to be more easily marked as read independent from state slots. With the stronger one, the target languages are those that are supported by the ProB model checker. These languages do not require support for copying values (dynamic addressing), hence the stronger definition of read independence is appropriate. The exact difference is the universal quantification of groups over the two cases when values are copied or overwritten. In particular, with the definition above, a group can have a transition that copies a value, and another transition that overwrites a value. With the definition in [Ben+16a] transitions in a group must either always copy a value *or* overwrite a value. In this chapter we continue with the weaker notion of read independence as in [Mei+14] because:

1. it allows for faster symbolic reachability analysis for languages that support dynamic variable addressing, and
2. it is required for languages where the stronger notion of read independence would suffice, but are extended with the concept of row subsumption. Row subsumption will be covered in Section 5.5.

In Chapter 7 we illustrate why the stronger notion of read independence is sufficient for ProB. The next definition captures the concept of read independence for SUSs.

Definition 5.5: Read Dependency Matrix

For a PTS $\mathcal{P}$, the *Read Dependency Matrix* (RDM) is a $K \times N$ matrix $RDM(\mathcal{P}) = RM_{K \times N}^{\mathcal{P}} \in \{0, 1\}^{K \times N}$ such that $(RM_{i,j} = 0) \implies$ transition group i is *read independent* from state slot j . For a SUS $\mathcal{U}$, the read dependency matrix $RDM(\mathcal{U})$ is defined as $RM_{K \times N}^{\mathcal{U}}$ with^a:

$$RM_{i,j}^{\mathcal{U}} = \begin{cases} 1 & \text{if } (x_j \text{ occurs in } c_i), \text{ or} & (a) \\ & \text{if } \exists_{1 \leq k \leq L_i} : (x_j \text{ occurs in } t_{i,k}) \wedge (v_{i,k} \neq x_j \vee t_{i,k} \neq x_j), \text{ or} & (b) \\ & \text{if } \exists_{1 \leq k \leq L_i} : (x_j \text{ occurs in } v_{i,k}) \wedge (v_{i,k} \neq x_j), & (c) \\ 0 & \text{otherwise.} & (d) \end{cases}$$

^aWhenever we write x_j we mean the variable name of x_j , not its value.

In case (a), condition c_i depends on x_j . In case (b), the Right Hand Side (RHS) of the update depends on x_j , but the assignment is not merely a copy. In case (c), variable x_j is used to dynamically determine a state slot for an assignment, but x_j is not directly used as Left Hand Side (LHS). For instance, update $a_{x_j} := 1$ reads x_j . In that case x_j is marked as read dependent, because it influences how the state vector is updated. In all other cases x_j is read independent. By static analysis of a SUS, all necessary entries in its read dependency matrix are marked, e.g. a static check $(v_{i,k} \neq x_j \vee t_{i,k} \neq x_j)$ in case (b) is the best we can do to check if a value is not simply copied. Example read dependency matrices are given next.

Example 5.9: Read Dependency Matrix

Let $\mathcal{U}^1$ be the SUS in Example 5.7 and $\mathcal{U}^2$ be the SUS in Example 5.8, their RDMs are:

$$RDM(\mathcal{U}^1) = \begin{array}{c} \begin{matrix} & p_1 & p_2 & p_3 & p_4 & p_5 \end{matrix} \\ \begin{matrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{matrix} \begin{bmatrix} 0 & 0 & \mathbf{1} & 0 & 0 \\ \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} & 0 \\ 0 & \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & \mathbf{1} & \mathbf{1} \end{bmatrix} \end{array}, \quad RDM(\mathcal{U}^2) = \begin{array}{c} \begin{matrix} i & a_0 & a_1 \end{matrix} \\ \begin{matrix} g_1 \\ g_2 \\ g_3 \end{matrix} \begin{bmatrix} \mathbf{1} & 0 & \mathbf{1} \\ 0 & \mathbf{1} & 0 \\ \mathbf{1} & 0 & 0 \end{bmatrix} \end{array}.$$

The example above illustrates how transition groups require read dependencies on state slots to determine when actions are enabled, or read when used as LHS of an assignment. To accommodate updating state vectors we cover the notion of write dependencies next.

5.4.2 The Write Dependency

The definition for write independence is given below in Definition 5.6.

Definition 5.6: Write independence

Given a PTS $\mathcal{P} = \langle S_{\mathcal{P}}, \rightarrow_{\mathcal{P}}, s^0 \rangle$, transition group i is *write independent* from state slot j if:

$$\forall \mathbf{s}, \mathbf{t} \in S_{\mathcal{P}}: \langle s_1, \dots, s_j, \dots, s_N \rangle \rightarrow_i \langle t_1, \dots, t_j, \dots, t_N \rangle \implies (s_j = t_j),$$

i.e. state slot j is never modified in transition group i .

Thus, if transition group i is write-dependent on state slot j , then there are some

states $\mathbf{s}, \mathbf{t}$ and a transition $\mathbf{s} \rightarrow_i \mathbf{t}$, where the value in state slot j is changed: $s_j \neq t_j$. Now, it is necessary to capture writing with dynamic variable resolution in a matrix for SUSs.

Definition 5.7: Write Dependency Matrix

For a PTS $\mathcal{P}$, the *Write Dependency Matrix* (WDM) is a $K \times N$ matrix $WDM(\mathcal{P}) = WM_{K \times N}^{\mathcal{P}} \in \{0, 1\}^{K \times N}$ such that $(WM_{i,j} = 0) \implies$ transition group i is *write independent* from state slot j . For a SUS $\mathcal{U}$, the write dependency matrix $WDM(\mathcal{U})$ is defined as $WM_{K \times N}^{\mathcal{U}}$ with^a:

$$WM_{i,j}^{\mathcal{U}} = \begin{cases} 0 & \text{if } \forall 1 \leq k \leq L_i, \exists 1 \leq l \leq N: l \neq j \wedge v_{i,k} = x_l \\ 1 & \text{otherwise.} \end{cases} \quad \begin{matrix} \text{(a)} \\ \text{(b)} \end{matrix}$$

^aWhere we write x_l we mean the variable name of x_l , not its value.

In case (a), if all updates immediately assign to a variable other than x_j we know group i is independent from x_j , otherwise (b) we do not know and must set the entry in the matrix to dependent. It is clear that at least every necessary entry $WM_{i,j}^{\mathcal{U}}$ is set to 1. Note that in practice, the bounds of a LHS expression $v_{i,k}$ can be approximated quite well. For instance, in Example 5.8 we know that variable i is bounded to $\{0, 1\}$ and thus transition group 3 ($a_i := i$) never assigns to variable i . With this knowledge we give example write matrices for SUSs below.

Example 5.10: Write Dependency Matrix

Let $\mathcal{U}^1$ be the SUS in Example 5.7 and $\mathcal{U}^2$ be the SUS in Example 5.8, their WDMs are:

$$WDM(\mathcal{U}^1) = \begin{matrix} & p_1 & p_2 & p_3 & p_4 & p_5 \\ \begin{matrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{matrix} & \begin{bmatrix} \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} \end{bmatrix} \end{matrix}, \quad WDM(\mathcal{U}^2) = \begin{matrix} & i & a_0 & a_1 \\ \begin{matrix} g_1 \\ g_2 \\ g_3 \end{matrix} & \begin{bmatrix} \mathbf{1} & \mathbf{1} & \mathbf{0} \\ \mathbf{1} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{1} & \mathbf{1} \end{bmatrix} \end{matrix}.$$

For convenience we define a matrix that combines both previously defined matrices.

Definition 5.8: Dependency Matrix

For a PTS $\mathcal{P}$, the *dependency matrix* (DM) is a $K \times N$ matrix $DM_{K \times N}^{\mathcal{P}} \in \{-, \mathbf{r}, \mathbf{w}, +\}^{K \times N}$ with the elements $DM_{i,j}^{\mathcal{P}}$ as specified in the following table:

$DM_{i,j}^{\mathcal{P}}$		$RM_{i,j}^{\mathcal{P}}$	$WM_{i,j}^{\mathcal{P}}$
-	(copy)	0	0
$\mathbf{r}$	(read)	1	0
$\mathbf{w}$	(dyn. write)	0	1
+	(read/write)	1	1

Again, example matrices are given below.

Example 5.11: Dependency Matrix

Let $\mathcal{U}^1$ be the SUS in Example 5.7 and $\mathcal{U}^2$ be the SUS in Example 5.8, their DMs are:

$$DM(\mathcal{U}^1) = \begin{array}{c} \begin{matrix} & p_1 & p_2 & p_3 & p_4 & p_5 \\ \begin{matrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{matrix} & \begin{bmatrix} \mathbf{w} & \mathbf{w} & + & - & - \\ + & - & - & \mathbf{w} & - \\ \mathbf{w} & - & - & + & - \\ - & + & - & - & \mathbf{w} \\ - & \mathbf{w} & - & - & + \\ - & - & \mathbf{w} & + & + \end{bmatrix} \end{matrix} \end{array}, \quad DM(\mathcal{U}^2) = \begin{array}{c} \begin{matrix} & i & a_0 & a_1 \\ \begin{matrix} g_1 \\ g_2 \\ g_3 \end{matrix} & \begin{bmatrix} + & \mathbf{w} & \mathbf{r} \\ \mathbf{w} & \mathbf{r} & \mathbf{w} \\ \mathbf{r} & \mathbf{w} & \mathbf{w} \end{bmatrix} \end{matrix} \end{array}.$$

The notion of combined dependencies is convenient for illustration purposes (e.g. to the person operating the LTSmin model checker), as well as for explaining row subsumption, that is explained next.

5.5 Combining Transition Groups

Row subsumption on the dependency matrix is a heuristic for combining transition groups. Combining transition groups by taking the union of their respective transition relations can in specific cases speed up reachability analysis. A row can subsume another row if each entry in the first is at least “as large” as the corresponding entry in the latter. The old notion of row subsumption in LTSmin did not consider read and write dependencies and is based on the (reflexive) total order “- < +”. This order makes sense, because a ‘+’ dependency can copy (‘-’)

values simply by reading values in the current state, and writing the same value in the successor state.

Consider dependency matrix $\text{DM}(\mathcal{U}^1)$ in Example 5.11, in the old situation every non ‘-’ dependency would be a ‘+’ dependency. This means that group t_2 can subsume t_3 and vice versa, and similar goes for t_4 and t_5 . In the old situation, every group in $\text{DM}(\mathcal{U}^2)$ can subsume another. In the new situation we run into a problem, because a read dependency (‘r’) can not write a value, and a write dependency (‘w’) can not read a value. Following the order approach on dependencies, this gives rise to a partial one: reading is not smaller than writing and writing is not smaller than reading. The order that supports read and write dependencies is given in Definition 5.9.

Definition 5.9: Partial order for dependencies

The partial order on dependencies is given below.

The definition shows that we still have $- < +$, but $r \not< w$, nor $w \not< r$. However, we do have $- < r$, and $- < w$. And for completeness, we also have $r < +$, and $w < +$. Using the partial order $\leq$, row subsumption is defined as follows.

Definition 5.10: Row subsumption

For two distinct matrix rows $\mathbf{m}, \mathbf{m}' \in M^N$ of length N , the *row subsumption* operator $\sqsubseteq: M^N \times M^N \rightarrow \mathbb{B}$ is defined as follows:

$$\mathbf{m} \sqsubseteq \mathbf{m}' \iff \forall_{1 \leq j \leq N}: m_j \leq m'_j.$$

If a row $\mathbf{m}_i$ is *subsumed* by row $\mathbf{m}_k$, ($\mathbf{m}_i \sqsubseteq \mathbf{m}_k$), the corresponding transition groups i and k can be merged and the combined matrix row becomes the larger one: $\mathbf{m}_k$. In general, any two rows could be merged by taking their point-wise least upper bound. The result is that there are fewer transition groups and less applications of a transition relation, but the transition relations might be larger.

To aid front-end developers for PINS, the partial order shows which dependencies are *safe* to over-approximate, e.g. it is safe to set an actual ‘w’ to ‘+’, but it is illegal to set an actual ‘r’ to ‘-’. Note that the partial order follows from the directions

of ‘ $\implies$ ’ in Definitions 5.5 and 5.7, i.e. those of the form “ $(\{R, W\}M_{i,j} = 0) \implies$ transition group i is $\{\text{read}, \text{write}\}$ independent from state slot j ”.

We illustrate our extension of row subsumption on the SUS in Example 5.8 as follows.

Example 5.12: Row subsumption

As a start, observe that no row in the dependency matrix $DM(\mathcal{U}^2)$ in Example 5.11 can subsume another. Hence, for illustration purposes, we add a transition group to the SUS of Example 5.8: $a_1 = 1 \rightarrow g_4$. Transition group g_4 expresses a self-loop when a_1 equals 1. The new group is completely independent from i and a_0 , the new dependency matrix is:

$$DM(\mathcal{U}^{2'}) = \begin{matrix} & i & a_0 & a_1 \\ \begin{matrix} g_1 \\ g_2 \\ g_3 \\ g_4 \end{matrix} & \begin{bmatrix} + & w & r \\ w & r & w \\ r & w & w \\ - & - & r \end{bmatrix} \end{matrix}.$$

Figure 5.3: Dependency matrix of $\mathcal{U}^{2'}$ with group g_4 .

The corresponding transition table for $DM(\mathcal{U}^{2'})$ is shown below (groups g_2 and g_3 have been omitted, because they are irrelevant for this example).

i	a_0	a_1	i'	a'_0	a'_1	$\rightarrow$
0	▲	0	0	1	▲	g_1
1	▲	0	0	1	▲	g_1
1	▲	1	0	1	▲	g_1
▲	▲	0	▲	▲	▲	g_4
▲	▲	1	▲	▲	▲	g_4

Table 5.5: Transition table for g_1 and g_4 .

Identical to Example 5.1, a ▲ in unprimed columns means the value is ignored, and a ▲ in primed columns means the value is copied. Column $\rightarrow$ denotes the name of the transition group. Table 5.5 expresses five symbolic transitions, three for group g_1 and two for group g_4 .

By applying row subsumption to $DM(\mathcal{U}^{2'})$ we get:

$$\begin{matrix} & i & a_0 & a_1 \\ \begin{matrix} g_1, g_4 \\ g_2 \\ g_3 \end{matrix} & \begin{bmatrix} + & w & r \\ w & r & w \\ r & w & w \end{bmatrix} \end{matrix}.$$

Figure 5.4: Dependency matrix of $\mathcal{U}^{2'}$ after row subsumption.

The matrix above shows that group g_1 subsumes g_4 , meaning that their transition relations are merged. The result of the union is shown in the table below.

i	a_0	a_1	i'	a'_0	a'_1	$\rightarrow$
0	▲	0	0	1	▲	g_1, g_4
1	▲	0	0	1	▲	g_1, g_4
1	▲	1	0	1	▲	g_1, g_4
0	▲	0	0	▲	▲	g_1, g_4
0	▲	1	0	▲	▲	g_1, g_4
1	▲	0	1	▲	▲	g_1, g_4
1	▲	1	1	▲	▲	g_1, g_4

Table 5.6: Transition table for g_1 and g_4 after row subsumption.

Table 5.6 contains a single transition relation called “ g_1, g_4 ”, at the expense of two more symbolic transitions compared to Table 5.5. The extra symbolic transitions arises due to the fact that the dependency matrix specifies i is both read and written for group g_4 (because it has been merged into g_1). Both Tables 5.5 and 5.6 are equivalent in the sense that they will produce the same state space, starting from some initial state.

Transition relations that have been merged can thus speed up symbolic reachability analysis, because the relational product has to be applied fewer times. In other words, having larger transition relations can be favorable to having more of them. We have implemented row subsumption based on Definition 5.9, extending LTSmin’s previous subsumption capabilities, based on the now obsolete total order: $- < +$.

5.6 Improved Symbolic Reachability Analysis

To allow symbolic reachability analysis with read, write and copy dependencies we provide three definitions. The first contains projections on dependency matrices. Secondly, we provide a definition of the restricted NEXT-STATE function from read-projected states to their write-projected successors according to transition group $\rightarrow_i$, as it is used in PINS for on-the-fly reachability analysis. This technique is language independent, but depends essentially on the PINS-architecture, based on state vectors, disjunctive transition partitioning and read and write dependency matrices. Lastly, we provide a symbolic definition of NEXT that formalizes the transition relation and the application of the transition relation on a set of states.

Notation For convenience, we introduce the function *ind*; the column *indices* of the cells that contain a **1** in row M_i :

$$\text{ind}(M_i) = \{j \mid 1 \leq j \leq |M_i| \wedge M_{i,j} = 1\}.$$

Given a vector $\mathbf{s}$ and a set of indices I , the notation $(s_j)_{j \in I}$ is used to project $\mathbf{s}$ and represent the *subvector* $\langle s_{\bar{I}_1}, \dots, s_{\bar{I}_\ell} \rangle$ of length $\ell = |I|$, where $\bar{I}$ is the sorted list of elements from I .

Definition 5.11: Projections

For any vector set:

$$S = \prod_{1 \leq j \leq N} S_j,$$

transition group $1 \leq i \leq K$ and $K \times N$ matrix M , we define the projection

$$\pi_i^M: S \rightarrow \prod_{j \in \text{ind}(M_i)} S_j, \text{ such that}$$

$$\pi_i^M(\mathbf{x}) = (x_j)_{j \in \text{ind}(M_i)},$$

i.e. the subvector of $\mathbf{x}$ that contains the elements at indices in $\text{ind}(M_i)$, the indices that are marked **1** in row i of matrix M . The projection function is extended to apply to a set S in a straightforward way:

$$\pi_i^M(S) = \{\pi_i^M(\mathbf{x}) \mid \mathbf{x} \in S\}.$$

We use the shorthand notation π_i^r for π_i^{RM} and π_i^w for π_i^{WM} , where RM and WM are defined in Definitions 5.5 and 5.7 respectively.

Using these read and write projections, we can define how the read and write dependency matrices can be used to compute the successor states for a transition

group, using only the dependent variables. We define the function NEXT-STATE_i^P that takes as input a read projected vector, and computes for transition group i the set of write projected successor vectors. The input read projected vector may match a set of input states, and each of the output projected successor vectors may represent a set of successor states. In the case a variable is write dependent, but not changed, the symbol $\blacktriangle$ is used to mark that the variable should be copied from the input vector. This can occur, e.g., in the case an entire array $a_{1..10}$ is marked write dependent, because of an assignment $a_z := e$. If $z = 5$, the position a_5 is written to and all positions a_j with $j \neq 5$ are marked with $\blacktriangle$. We use $S_j^\blacktriangle$ for $S_j \cup \{\blacktriangle\}$ and $S_P^\blacktriangle$ for the set $S_1^\blacktriangle \times \dots \times S_N^\blacktriangle$.

Definition 5.12: Partitioned Next-State function

The *Partitioned Next-State function* is a function:

$$\text{NEXT-STATE}_i^P: \pi_i^r(S_P) \rightarrow 2^{\pi_i^w(S_P^\blacktriangle)}, \text{ such that}$$

$$\text{NEXT-STATE}_i^P(\mathbf{s} = (s_j)_{j \in \text{ind}(RM_i)}) = \left\{ \begin{array}{l} \pi_i^w(\mathbf{t}) \mid \exists \mathbf{s}', \mathbf{t}' \in S_P, \exists \mathbf{t} \in S_P^\blacktriangle: \pi_i^r(\mathbf{s}') = \mathbf{s} \wedge \mathbf{s}' \rightarrow_i \mathbf{t}' \wedge \forall_{1 \leq j \leq N}: t_j = \\ \left\{ \begin{array}{l} \blacktriangle \text{ if } s'_j = t'_j \wedge \forall r_j \in S_j: \langle s'_1, \dots, r_j, \dots, s'_N \rangle \rightarrow_i \langle t'_1, \dots, r_j, \dots, t'_N \rangle, \\ t'_j \text{ otherwise} \end{array} \right. \end{array} \right\}.$$

The behavior of NEXT-STATE_i^P can be visualized as:

$$\mathbf{s} \xrightarrow{\text{expand}} \mathbf{s}' \xrightarrow{\text{transit}} \{\mathbf{t}'\} \xrightarrow{\text{copy/write}} \{\mathbf{t}\} \xrightarrow{\text{project}} \{\mathbf{p}\}, \text{ where:}$$

- “expand” is the step where a read projected vector $\mathbf{s}$ is expanded by PINS to a full length vector $\mathbf{s}'$ (according to $\pi_i^r(\mathbf{s}') = \mathbf{s}$) that can be consumed by a PINS language front-end,
- “transit” is the step where a PINS language front-end computes the set of full length successor(s) $\{\mathbf{t}'\}$ of $\mathbf{s}'$ (according to $\mathbf{s}' \rightarrow_i \mathbf{t}'$),
- “copy/write” is the step where values are copied or written from any $\mathbf{t}'$ into a $\mathbf{t}$ by the front-end,
- “project” is the step where any full length successor $\mathbf{t}$ vector is write projected by PINS to a short vector $\mathbf{p}$ ($\{\mathbf{p}\}$ is the result of $\pi_i^w(\mathbf{t})$).

Note that the step “transit” can be non-deterministic for PINS language front-ends, hence $\mathbf{s}'$ can have multiple successors $\mathbf{t}'$. Furthermore, this non-deterministic behavior can also arise due to row subsumption. For a State Update Specification (SUS) $\mathcal{U}$ with variables $\langle x_1, \dots, x_N \rangle$, NEXT-STATE_i^P follows from the semantics in

Definition 5.3. We note however that in practice, determining whether state slot j gets value $\blacktriangle$ (i.e. $t_j = \blacktriangle$) in state $\mathbf{s}$ is done by checking there is no LHS in state update σ_i in $\mathcal{U}$ that evaluates to x_j , i.e. for all $1 \leq l \leq L_i$, $\llbracket v_{i,l} \rrbracket_{\mathbf{s}}$ never evaluates to variable x_j . In case there is no assignment to x_j , then the value for state slot j is certainly copied.

Example 5.13: Partitioned Next-State function

Illustrated by the transition table in Table 5.4, two example calls for group g_3 ($\text{true} \rightarrow g_3. \langle a_i = i \rangle$) of SUS $\mathcal{U}$ in Example 5.8 are as follows.

1. $\text{NEXT-STATE}_{g_3}^P(\langle 0 \rangle) = \{\langle 0, \blacktriangle \rangle\}$,
2. $\text{NEXT-STATE}_{g_3}^P(\langle 1 \rangle) = \{\langle \blacktriangle, 1 \rangle\}$.

In both calls the condition in group g_3 (obviously) enables the transitions. Furthermore, the calls can be illustrated as:

1. $\langle 0 \rangle \xrightarrow{\text{expand}} \langle 0, 0, 0 \rangle \xrightarrow{\text{transit}} \{\langle 0, 0, 0 \rangle\} \xrightarrow{\text{copy/write}} \{\langle \blacktriangle, 0, \blacktriangle \rangle\} \xrightarrow{\text{project}} \{\langle 0, \blacktriangle \rangle\}$,
2. $\langle 1 \rangle \xrightarrow{\text{expand}} \langle 1, 0, 0 \rangle \xrightarrow{\text{transit}} \{\langle 1, 0, 1 \rangle\} \xrightarrow{\text{copy/write}} \{\langle \blacktriangle, \blacktriangle, 1 \rangle\} \xrightarrow{\text{project}} \{\langle \blacktriangle, 1 \rangle\}$.

In both cases we expand the read projected vectors based on values in the initial state in $\mathcal{U}$ ($\langle 0, 0, 0 \rangle$), this is also the default behavior in PINS. For completeness, we note that for call 1. $\llbracket a_i := i \rrbracket_{\langle 0, 0, 0 \rangle}$ evaluates to $a_0 := 0$, and for call 2. $\llbracket a_i := i \rrbracket_{\langle 1, 0, 0 \rangle}$ evaluates to $a_1 := 1$. Based on both evaluations of the LHS in $a_i := i$, we can infer which values need to be copied, and hence which state slots get $\blacktriangle$.

The example above shows that transitions based on short vectors give rise to many transitions based on long vectors. This provides the power of symbolic reachability analysis, even in the presence of an explicit NEXT-STATE function. After a call to NEXT-STATE_i^P the result vectors $(t_j)_{j \in \text{ind}(WM_i)}$, combined with the input vectors $(s_j)_{j \in \text{ind}(RM_i)}$ are stored in a partial transition relation $\hookrightarrow_i^P$ for group i .

Definition 5.13: NEXT

We define the function:

$$\text{NEXT}: 2^{S^P} \times (\pi^r(S^P) \times \pi^w(S_{\mathcal{P}}^{\blacktriangle}) \rightarrow \mathbb{B}) \times M_N \times M_N \rightarrow 2^{S^P},$$

which applies a partial transition relation to a set of states as follows. Given a set S , a partial transition relation $\hookrightarrow^P$, a read matrix row $\mathbf{r}$ and a write matrix row $\mathbf{w}$, i.e.

$$\begin{aligned} \text{NEXT}(S, \hookrightarrow^P, \mathbf{r}, \mathbf{w}) = \\ \left\{ \mathbf{y} \in S_{\mathcal{P}} \mid \exists \mathbf{x} \in S, \mathbf{z} \in S_{\mathcal{P}}^{\blacktriangle} : \hookrightarrow^P(\pi^{\mathbf{r}}(\mathbf{x}), \pi^{\mathbf{w}}(\mathbf{z})) \wedge \right. \\ \left. \bigwedge_{j \in \text{ind}(\mathbf{w})} \left(y_j = \begin{cases} x_j & \text{if } z_j = \blacktriangle, \\ z_j & \text{otherwise} \end{cases} \right) \wedge \bigwedge_{j \notin \text{ind}(\mathbf{w})} (y_j = z_j) \right\}. \end{aligned}$$

The symbolic reachability algorithm that uses the functions NEXT-STATE and NEXT is given in Algorithm 5.1. The algorithm is an extension of the symbolic reachability algorithm in [BP08, Table 6].

Algorithm 5.1: REACH-BFS-PREV

Input : $s^0 \in S_{\mathcal{P}}, K \in \mathbb{N}, RM, WM$

Output: The set of reachable states $\mathcal{R}$

```

1  $\mathcal{R} \leftarrow \{s^0\}$ 
2  $\mathcal{L} \leftarrow \mathcal{R}$ 
3 for  $1 \leq i \leq K$  do
4    $\mathcal{R}_i^P \leftarrow \emptyset$ 
5    $\hookrightarrow_i^P \leftarrow \emptyset$ 
6 while  $\mathcal{L} \neq \emptyset$  do
7   for  $1 \leq i \leq K$  do ▷ Learn new transitions
8      $\mathcal{L}^P \leftarrow \pi_i^{\mathbf{r}}(\mathcal{L})$ 
9     for  $s^P \in \mathcal{L}^P \setminus \mathcal{R}_i^P$  do
10       $\hookrightarrow_i^P \leftarrow \hookrightarrow_i^P \cup \{\langle s^P, d^P \rangle \mid d^P \in \text{NEXT-STATE}_i(s^P)\}$ 
11       $\mathcal{R}_i^P \leftarrow \mathcal{R}_i^P \cup \mathcal{L}^P$ 
12    $\mathcal{N} \leftarrow \emptyset$ 
13   for  $1 \leq i \leq K$  do ▷ Compute new states
14      $\mathcal{N} \leftarrow \mathcal{N} \cup \text{NEXT}(\mathcal{L}, \hookrightarrow_i^P, RM_i, WM_i)$ 
15    $\mathcal{L} \leftarrow \mathcal{N} \setminus \mathcal{R}$ 
16    $\mathcal{R} \leftarrow \mathcal{R} \cup \mathcal{N}$ 
17 return  $\mathcal{R}$ 

```

A description of the algorithm is as follows. Variable $\mathcal{R}$ maintains the set of reachable states so far, while $\mathcal{L}$ stores the current BFS level. After initialization (lines 1–5), the next level $\mathcal{N}$ will be continuously computed and added, until the current level is empty (lines 6–16). In each iteration, first the new transitions must be learned (lines 7–11). The next level is computed by calling NEXT for each transition group (line 14). An implementation of NEXT for List Decision Diagrams (LDDs) is given later in Algorithm 5.2.

Our extension includes three subtle modifications compared to [BP08, Table 6],

when growing the transition relations on-the-fly (lines 7–11). *First*, the state is read-projected on line 8. The benefit being that fewer calls to NEXT-STATE are needed. *Secondly*, the vectors added to the partial transition relation in line 10 may contain the special value $\blacktriangle$. This allows dynamic assignments to be resolved efficiently at a lower (symbolic) level. *Thirdly*, the transition relation is applied using both the read and the write dependency matrices (line 14). That way, fewer levels of the underlying decision diagrams are affected.

Figure 5.5: Projection without and with read-separation for Example 5.3.

Figure 5.5 clearly shows the difference between using the previously used projections (to the left) and using read-projections (to the right). Both figures show decision trees that encode the read projected sets of all reachable states in $\mathcal{R}$ after termination of Algorithm 5.1. Both can be used to compute successors for the states in Example 5.3, but when using read-projections, the function $\rightarrow_i$ is applied to only one of the four states with $p_3 = 0$, instead of to all.

5.6.1 List Decision Diagrams

To investigate the effects of separating dependencies, we have implemented the transition relation and its application from Definition 5.13 in LTSmin’s native LDD library. An LDD is a form of Multi-way Decision Diagram (MDD) which was initially described in [BP08, Section 5]. The definition is as follows.

Definition 5.14: LDD

A *List Decision Diagram* (LDD) is a *Directed Acyclic Graph* (DAG) with three types of nodes:

- $\{\epsilon\}$, which encodes the empty vector of length 0 and has no successors,

- $\emptyset$, which encodes the empty set and has no successors,
- $\langle v, \text{down}, \text{right} \rangle$, a tuple with label v and two successors: *down* and *right*.

The semantics $\llbracket s \rrbracket$ of an LDD node s is a set of vectors, as follows:

- $\llbracket \{\epsilon\} \rrbracket = \{\epsilon\}$,
- $\llbracket \emptyset \rrbracket = \emptyset$,
- $\llbracket \langle v, \text{down}, \text{right} \rangle \rrbracket = \{vw \mid w \in \llbracket \text{down} \rrbracket\} \cup \llbracket \text{right} \rrbracket$, where vw denotes the concatenation of vectors v and w .

For some node $n = \langle v, \text{down}, \text{right} \rangle$, we use n_v , n_d and n_r to denote its elements v , *down* and *right*, respectively.

We assume (and enforce) in the implementation that the sequence of values in a level is ordered from small to large. E.g., $\langle 0, d_1, \langle 1, d_2, \emptyset \rangle \rangle$ is a valid node, but $\langle 1, d_2, \langle 0, d_1, \emptyset \rangle \rangle$ is not. We define $\blacktriangle$ to always be smallest.

A single vector $\mathbf{x} = \langle x_1, \dots, x_N \rangle$ (or singleton set $\{\mathbf{x}\}$) can be represented as an LDD node as $\langle x_1, \langle x_2, \dots, \emptyset \rangle, \emptyset \rangle$. Note that for vector $\mathbf{x}$, encoded as LDD node x , the LDD node of the subvector $\mathbf{x}_{1 < j \leq |\mathbf{x}|}$, i.e., the vector $\mathbf{x}$ with the first element removed, equals x_d (i.e. the down node of x). An example LDD is provided below.

Example 5.14: Simple LDD

The LDD with two variables x_0 and x_1 that encodes the set $\{\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle\}$ ($= \{0, 1\} \times \{0, 1\}$) is as follows.

In the LDD above, every path from the top left node (with value 0) to $\{\epsilon\}$ forms a vector element in $\{0, 1\}^2$.

As a last example for LDDs, we show how a transition relation is encoded.

Example 5.15: Transition relation LDD

The following figure shows the LDD which encodes the transitions of groups

g_1 and g_4 after row subsumption in Table 5.6. For visualization purposes the terminal node $\emptyset$ (and edges to it) have been omitted.

In the figure above the following can be observed.

- Based on the dependency matrix in Figure 5.4 the variable levels can be seen, e.g. i is both read and written, hence it has two levels, one for the current state (unprimed) and one for the next state (primed).
- The variable for the next state (primed) appears on the level below its unprimed counterpart, e.g. i' appears below i . This is true for any LDD that encodes a transition relation: any variable v appears immediately above v' .
- On the variable level of a'_0 there are nodes with value $\blacktriangle$, that specify values need to be copied.
- On every variable level the lists are sorted as $\blacktriangle < 0 < 1$.

What now remains to be shown is how set operations on LDDs are extended with read-write separation.

5.6.2 Set Operations for List Decision Diagrams

In the implementation of the application of the transition relation NEXT in Definition 5.13, we use LDDs to encode the set S_P , the relation $\hookrightarrow^P$ and the matrix rows $\mathbf{r}$ and $\mathbf{w}$. Specifically, S_P is encoded as an LDD of depth N and $\hookrightarrow^P$ as an LDD of depth $|\text{ind}(\mathbf{r})| + |\text{ind}(\mathbf{w})|$. The rows $\mathbf{r}$ and $\mathbf{w}$ are encoded as LDD representations of the sorted vectors with the indices of dependent variables $\text{ind}(\mathbf{r})$ and $\text{ind}(\mathbf{w})$, respectively. In Example 5.15, the LDD of $\mathbf{r}$ encodes the vector $\langle 0, 2 \rangle$, and the LDD of $\mathbf{w}$ encodes the vector $\langle 0, 1 \rangle$. The algorithm using LDDs for NEXT given in Algorithms 5.2 to 5.4 recursively traverses the LDDs level by level, maintaining a counter $0 \leq l < N$ to keep track of the current level, initially set to 0. For

convenience, we denote the depth of node s as:

$$\text{depth}(s) = \begin{cases} 0 & , \text{ if } s = \{\epsilon\} \\ \text{depth}(s_d) + 1 & , \text{ otherwise.} \end{cases}$$

Lines 1–4 handle a few base cases. Line 4 in particular asserts that the depth of $\hookrightarrow^P$ is equal to the depth of $\mathbf{r}$ plus the depth of $\mathbf{w}$. If their depths are not equal, then an LDD is constructed incorrectly, which is exceptional behavior. In case the current level l (variable x_l) is both *read and write dependent* (lines 5–8), first a (read) value is matched (s_v and $\hookrightarrow_v^P$) and then each value from the next level of the relation is written using LDD-WRITE. The resulting node is united with all other values we may need to write. If the level is *read dependent* only (lines 9–12), then we first find a matching value and then create a new node with two recursive elements: downward matching the other levels and to the right; other nodes on the current level that may match the relation. If the level is *write dependent* only (lines 13–15), then for each value in the set we create a new node, where we either copy the value s_v or write the value from the relation. If the level has no read or write dependency (line 16), then a new node is created with the *down* and *right* nodes computed recursively with LDD-COPY. LDD-WRITE writes all values from the relation reachable on the current level. However, it needs to unite all the nodes with these values because they may occur in the wrong order. The unions are computed in the standard way for decision diagrams.

5.7 Results

To evaluate our work we have benchmarked with 266 DVE models from the BEEM database [Pel07], 38 mCRL2 models, mostly from mCRL2’s distribution, and 60 Promela models from different sources. To compare our results to both the current version of LTSmin we implemented the option `r2+` and `w2+`. These two options over-approximate read or write to both read and write, thus simulating the situation without read-write separation. Every experiment is run three times in both setups to determine the effect of our work. The machine we used has an Intel Xeon E5520 CPU, with 24 GB of memory. We have restricted the runtime of each experiment to 30 minutes.

Overall, we see that the mCRL2 specifications benefit from read-write separation the most, because of the reduced amount of NEXT-STATE calls. This is due to the fact that a NEXT-STATE call for mCRL2 is rather time consuming because of the term rewriting involved. The DVE and Promela front-ends run optimized C code. For these languages, the overhead of many unnecessary NEXT-STATE calls in the current version of LTSmin is less noticeable. Of all experiments which

Algorithm 5.2: LDD-NEXT

Input: LDD nodes $s, \hookrightarrow^P, \mathbf{r}, \mathbf{w}$ and level $0 \leq l < N$

```

1 if  $s = \emptyset \vee \hookrightarrow^P = \emptyset$  then return  $\emptyset$ 
2 if  $\mathbf{r} = \{\epsilon\} \wedge \mathbf{w} = \{\epsilon\}$  then return  $s$ 
3 assert  $s \neq \{\epsilon\} \wedge \hookrightarrow^P \neq \{\epsilon\}$ 
4 assert  $\text{depth}(s) = N - l \wedge \text{depth}(\hookrightarrow^P) = \text{depth}(\mathbf{r}) + \text{depth}(\mathbf{w})$ 
5 if  $\mathbf{r}_v = l \wedge \mathbf{w}_v = l$  then ▷ Read and write dependent
6   if  $\hookrightarrow_v^P < s_v$  then return LDD-NEXT( $s, \hookrightarrow_r^P, \mathbf{r}, \mathbf{w}, l$ )
7   else if  $\hookrightarrow_v^P > s_v$  then return LDD-NEXT( $s_r, \hookrightarrow^P, \mathbf{r}, \mathbf{w}, l$ )
8   else return
      LDD-WRITE( $s, \hookrightarrow_d^P, \mathbf{r}_d, \mathbf{w}, l$ )  $\cup$  LDD-NEXT( $s_r, \hookrightarrow_r^P, \mathbf{r}, \mathbf{w}, l$ )
9 else if  $\mathbf{r}_v = l$  then ▷ Only read dependent
10  if  $\hookrightarrow_v^P < s_v$  then return LDD-NEXT( $s, \hookrightarrow_r^P, \mathbf{r}, \mathbf{w}, l$ )
11  else if  $\hookrightarrow_v^P > s_v$  then return LDD-NEXT( $s_r, \hookrightarrow^P, \mathbf{r}, \mathbf{w}, l$ )
12  else return
       $\langle s_v, \text{LDD-NEXT}(s_d, \hookrightarrow_d^P, \mathbf{r}_d, \mathbf{w}, l + 1), \text{LDD-NEXT}(s_r, \hookrightarrow_r^P, \mathbf{r}, \mathbf{w}, l) \rangle$ 
13 else if  $\mathbf{w}_v = l$  then ▷ Only write dependent
14  if  $\hookrightarrow_v^P = \blacktriangle$  then return
      LDD-COPY( $s, \hookrightarrow_d^P, \mathbf{r}, \mathbf{w}_d, l$ )  $\cup$  LDD-WRITE( $s, \hookrightarrow_r^P, \mathbf{r}, \mathbf{w}, l$ )
15  else return
      LDD-WRITE( $s, \hookrightarrow^P, \mathbf{r}, \mathbf{w}, l$ )  $\cup$  LDD-WRITE( $s_r, \hookrightarrow^P, \mathbf{r}, \mathbf{w}, l$ )
16 else return LDD-COPY( $s, \hookrightarrow^P, \mathbf{r}, \mathbf{w}, l$ ) ▷ Copy

```

Algorithm 5.3: LDD-WRITE

Input: LDD nodes $s, \hookrightarrow^P, \mathbf{r}, \mathbf{w}$ and $0 \leq l < N$

```

1 assert  $\mathbf{w} \neq \{\epsilon\}$ 
2 assert  $\text{depth}(s) = N - l \wedge \text{depth}(\hookrightarrow^P) = \text{depth}(\mathbf{r}) + \text{depth}(\mathbf{w})$ 
3 assert  $(\mathbf{r} = \{\epsilon\} \vee \mathbf{r}_v > l) \wedge \mathbf{w}_v = l$ 
4 if  $\hookrightarrow^P = \emptyset$  then return  $\emptyset$ 
5 return  $\langle \hookrightarrow_v^P, \text{LDD-NEXT}(s_d, \hookrightarrow_d^P, \mathbf{r}, \mathbf{w}_d, l + 1), \emptyset \rangle$ 
       $\cup \text{LDD-WRITE}(s, \hookrightarrow_r^P, \mathbf{r}, \mathbf{w}, l)$ 

```

Algorithm 5.4: LDD-COPY

Input: LDD nodes $s, \hookrightarrow^P, \mathbf{r}, \mathbf{w}$ and $0 \leq l < N$

```

1 assert  $\text{depth}(s) = N - l \wedge \text{depth}(\hookrightarrow^P) = \text{depth}(\mathbf{r}) + \text{depth}(\mathbf{w})$ 
2 assert  $(\mathbf{r} = \{\epsilon\} \vee \mathbf{r}_v > l) \wedge (\mathbf{w} = \{\epsilon\} \vee \mathbf{w}_v > l)$ 
3 if  $s = \emptyset$  then return  $\emptyset$ 
4 return  $\langle s_v, \text{LDD-NEXT}(s_d, \hookrightarrow^P, \mathbf{r}, \mathbf{w}, l + 1),$ 
      LDD-COPY( $s_r, \hookrightarrow^P, \mathbf{r}, \mathbf{w}, l) \rangle$ 

```

have a run time longer than one second, 101 out of 167 are faster with read-write separation. We have highlighted six interesting experiments with relevant information in Table 5.9, of which a legend can be found in Table 5.8. Experiments that are run *with* read-write separation are shown in **bold**.

With DVE specifications we see improvements in memory footprint in general. Runtime improvements are not always the case, for example in **anderson.6** and **telephony.7.dve**. **blocks.3** is a specification where we obtained very good results. The state space of this specification contains 695,418 states, there are 26 groups and 18 state slots. Because **blocks.3** contains many write only dependencies on these slots we are able to greatly reduce the amount of NEXT-STATE calls. Furthermore, the amount of nodes in the node table is reduced significantly. Both factors attribute to the improved runtime. The PTS of specification **telephony.7.dve** has 21,960,308 states, 24 state slots and 120 groups. Similar to **anderson.6** we see a slow down when we use separated dependencies. The **anderson.6** specification has 18,206,917 states, 36 groups and 19 state slots. In this specification we see no speedup, because it is hard to find a good variable order. The bad variable order results in more recursive LDD-NEXT calls which slows down the reachability analysis. This poor result was one of the motivations to systematically investigate how to obtain better variable orders, discussed in Chapter 6.

In the first two mCRL2 specifications (**1394-fin** and **lift3-final**) we can see that the amount of reduced NEXT-STATE calls correspond closely to the speedup attained. The specification **1394-fin** has 188,596 states, 34 state slots and 1,069 transition groups. The second, **lift3-final**, has 4,312 states, 30 state slots and 60 transition groups. We obtained the most interesting result with the **vasy** specification, a 1-safe Petri net submitted to the Petri net mailing list in 2003 [Kor+13] by Hubert Garavel. The marking graph has 9.79×10^{21} states, 776 transition groups and 485 state slots. With our work we have managed to make reachability analysis for this Petri net tractable for LTSmin. Special about this Petri net is the first transition, which removes the token from the initial place to several other places (like in Example 5.3). Without read-write separation, this required exponentially many NEXT-STATE calls for this transition: $\leq 2^{61}$ calls, because there are 61 dependent state slots of Boolean type. With our improvements it is identified that only one state slot is read, resulting in only 2^1 NEXT-STATE calls.

5.8 Conclusion

Separating dependencies into distinct read and write dependencies can speed-up symbolic reachability considerably. To do so, we had to solve one key problem. A *copy* dependency can in general not be approximated to a write dependency in a straightforward manner by static analysis. Therefore we introduced the definition

specification	name of the specification
$\overline{rt}$	average reachability time in seconds
$\overline{mem}$	average peak memory usage in kilobytes
#NS	number of NEXT-STATE calls
#LDD	number of calls to LDD-NEXT (Algorithm 5.2)
$ \mathcal{R} $	number of nodes of the set of reachable states
$ \pi(\mathcal{R}) $	approx. number of nodes in the projections
$ \hookrightarrow $	approx. number of nodes of the whole transition relation

Table 5.8: Symbols used in Table 5.9.

specification	$\overline{rt}$	$\overline{mem}$	#NS	#LDD	$ \mathcal{R} $	$ \pi(\mathcal{R}) $	$ \hookrightarrow $
anderson.6.dve	25.4	439,076	7,464	64,034,383	50,120	2,442	2,064
anderson.6.dve	29.9	144,216	4,080	109,711,771	41,079	1,533	1,386
blocks.3.dve	31.0	239,293	6,559,927	69,695,086	39,522	375,603	269,996
blocks.3.dve	10.9	144,064	262,543	62,467,909	39,522	12,314	1,604
telephony.7.dve	107.6	1,111,840	918,817	231,808,995	284,449	36,951	6,038
telephony.7.dve	123.7	696,188	730,841	393,099,843	284,449	31,473	5,337
1394-fin.mcrl2	22.6	208,084	3,372,554	1,995,202	7,384	870,142	12,505
1394-fin.mcrl2	3.4	188,944	443,813	1,800,912	7,384	229,251	8,399
lift3-final.mcrl2	5.3	184,624	190,347	313,868	5,452	162,956	7,023
lift3-final.mcrl2	2.5	181,372	79,941	378,179	5,452	54,249	5,496
vasy.mcrl2	-	-	-	-	-	-	-
vasy.mcrl2	152.6	1,149,592	2,694	241,432,226	9,387	4,340	5,444

Table 5.9: Selected experiment results. Specification names in **bold** are experiments run *with* read-write separation.

of write dependence with dynamic updates. This notion is used when it can not be statically determined whether a value needs to be written to or copied. This behavior is often exhibited by languages such as Promela and DVE, because they allow dynamic assignment to positions in an array.

We have presented the theoretical foundation underlying LTSmin’s extended NEXT-STATE function that supports read-write separation. The NEXT-STATE function is an explicit-state function in PINS, enabling symbolic reachability analysis by means of two dependency matrices. The read matrix specifies which state variables the function depends on in the current state, while the write matrix specifies which values may have changed in the successor state. When a language implementing the NEXT-STATE function resolves assignments dynamically, e.g. in case of array assignment, it needs to specify which array positions have not been written to. This is done by indicating those positions with a special $\blacktriangle$ value, which is stored at runtime in decision diagrams. The extended NEXT-STATE function is implemented in LTSmin and we have shown that improvements can be seen at least among Promela, DVE and mCRL2. Due to LTSmin’s language agnostic architecture of PINS, other specification languages can be added in a straightforward manner. In Chapter 7 we show how this can be done for languages supported by the ProB model checker and report the speed ups attained with read-write separation for those languages.

Specifications for the Promela and DVE language front-ends for PINS are both highly optimized C programs. Thus a NEXT-STATE call is relatively fast compared to symbolic operations of the back-end. This means only in some cases we see a speed up for Promela and DVE with read-write separation. On the other hand, computing the state space of mCRL2 specifications involves the term rewriter of mCRL2. The increased expressiveness has a prize: term rewriting is a lot slower than the optimized C programs for Promela and DVE. So symbolic operations are relatively fast compared to a NEXT-STATE call to the mCRL2 language front-end. This means a speed up is seen for many mCRL2 specifications.

Overall, we conclude that separating dependencies in the transition relation by default in LTSmin is a good idea. We have observed only a few cases with a slow-down, and this slow-down is rather minimal. The observed speed-ups on the other hand were considerable, and in some cases necessary to make problems tractable for LTSmin, e.g. for the **vasy** specification.

In the future we hope to generalize the notion of dynamic dependencies further. In particular, we suspect that dynamic reads may reduce the number of NEXT-STATE calls even further.

Optimizing Variable Orders and Transition Group Orders

Abstract

We investigate the use of bandwidth and wavefront reduction algorithms to determine a static BDD variable ordering. The aim is to reduce the size of BDDs arising in symbolic reachability. Previous work showed that minimizing the (weighted) event span of the variable dependency graph yields small BDDs. The bandwidth and wavefront of symmetric matrices are well studied metrics, used in sparse matrix solvers, and many bandwidth and wavefront reduction algorithms are readily available in libraries like Boost and ViennaCL.

In this chapter, we transform the dependency matrix to a symmetric matrix and apply various bandwidth and wavefront reduction algorithms, measuring their influence on the (weighted) event span. We show that Sloan's algorithm, executed on the total graph of the dependency matrix, yields a variable order with minimal event span. We demonstrate this on a large benchmark of Petri nets, DVE, Promela, B, and mCRL2 models. As a result, good static variable orders can now be determined in milliseconds by using algorithms initially developed for sparse matrix solvers.

6.1 Introduction

Reachability analysis is an approach for investigating properties of reachable states of computer programs. Some type of computer programs allow efficient storage of their set of reachable states by means of decision diagrams, this technique is known as symbolic reachability analysis. Storing sets of states symbolically entails storing

sets of integer vectors as binary formulas in for example Binary Decision Diagrams (BDDs) [Bry86]. One major issue with this approach is the ordering of variables in Decision Diagrams (DDs) representing the formula. Improving variable order is known to be NP-complete [BW96]. Therefore, heuristics for DD variable ordering has been extensively studied [Rud93, AMS03, SC06, GLM03]. Dynamic reordering modifies variable orders during computations, while static variable ordering precomputes a total order based on some structure of the input. In the latter case, several metrics have been proposed that lead to small DDs [RK08].

Systematic research on good algorithms to obtain orders with low values for such metrics is sparse. Since this work was first published in [MP16] additional research has been done by Amparore et al. [Amp+17, Amp+18]. However there the authors limit to the PNML specification language whereas here we present an approach for specifications of asynchronous systems in general. An existing algorithm for static variable ordering is Noack's algorithm [Noa99], but Noack's algorithm is only applicable to Petri nets. The only existing algorithm that can compete with our proposed ones is FORCE [AMS03]. We will show that four well known algorithms used in sparse matrix solvers can drastically and very quickly improve variable orders for any specification language, just like FORCE can¹. The novel contribution in this chapter is a systematic benchmark for Noack, FORCE and our proposed bandwidth and wavefront reduction algorithms executed on many specifications written in different languages.

Static variable ordering exploits the notion of event locality. Events, such as program statements or transitions in Petri nets are often local, i.e. they touch only a few variables or places and ordering these local variables near each other tends to significantly reduce the memory footprint of the DDs. An appropriate metric for indicating the quality of the variable order is Event Span (ES), by Siminiceanu and Ciardo [SC06]. The event span metric is used to measure the total distance between the minimum variable and maximum variable of all events. Additionally Siminiceanu and Ciardo introduce an extension of ES, called Weighted Event Span (WES). The weight of every event signifies the location of a span, i.e. whether the span changes the top or bottom of the DD. It is known that operations changing the bottom of the DD are cheaper [CMS06], which is beneficial for the saturation strategy in particular.

The quality of the variable order can be visualized using matrices. Such an approach is taken in [Mei+14] and Chapter 5, where a dependency matrix has rows as transition groups and columns as variables. A nonzero entry indicates that a transition depends on a variable. These dependency matrices tend to be sparse, hinting that traditional sparse matrix algorithms can be applied to these matrices.

¹We restrict ourselves to languages that induce a disjunctive transition relation.

A subcategory of sparse matrix algorithms are bandwidth reduction algorithms. One key example of a bandwidth reduction algorithm is by Cuthill and McKee developed in 1969 [CM69]. The goal of these algorithms is very similar to ES reduction algorithms. The bandwidth measures the distance of nonzeros to the diagonal of the matrix. Bandwidth is related to event span, because of the triangle inequality theorem, which states that event span is always smaller than twice the bandwidth.

Another popular algorithm is Sloan's [Slo89] algorithm, which optimizes total bandwidth (also called profile), and wavefront. The graph algorithm has a very low time complexity $\mathcal{O}(\log \hat{D} \cdot |E|)$, where $\hat{D}$ is the maximum degree, and E the set of edges in the adjacency graph. This results in runtimes of mere seconds when applied to matrices with a million rows and columns, or transitions and variables. Conveniently, Sloan's algorithm is freely available in Boost's graph library. Every model checker written in C/C++ or Python can be linked to Boost without much effort.

While bandwidth and wavefront reduction algorithms have proven themselves during the past decades, they only work on symmetric matrices. A dependency matrix (e.g. Example 5.11) is asymmetric because clearly, transition groups (rows) and variables (columns) are different objects and there exists no natural total order on the union of both. Reid et al. [RS06] discuss several methods of symmetrizing asymmetric matrices. With visualizations and experimental data we show that indeed, on an adjacency graph level, adding the inverse set of edges, and simply assigning some total order, that preserves the partial order on transitions and variables works extremely well.

We extensively benchmark the Cuthill McKee, Gibbs Poole Stockmeyer, King and Sloan nodal ordering algorithms implemented in Boost and ViennaCL [Rup+16]. The benchmark consists of 785 specifications, written in PNML, DVE, mCRL2, Promela and B. The model checker LTSmin [Kan+15] is already capable of handling all five input languages. By linking Boost and ViennaCL to LTSmin we can execute the nodal ordering algorithms and measure their influence on bandwidth, wavefront and (weighted) event span.

The rest of the chapter is structured as follows. In Sections 6.2 and 6.3 we explain what the role of the dependency matrix is related to variable ordering. Next, Section 6.4 explains the nodal ordering algorithms and why these can not be directly applied to the dependency matrix. The solution is given in Section 6.5; it involves symmetrizing the dependency matrix; permuting the matrix and desymmetrizing the matrix. An experimental evaluation is given in Section 6.6, in Section 6.7 we discuss related work and we conclude our findings in Section 6.8.

6.2 Dependencies and Event Locality

In Chapter 5 we have shown that distinguishing read and write dependencies is beneficial for symbolic reachability analysis, however this distinction will not be exploited in this chapter. We will restrict here to the combined notion of dependencies in Definition 5.8.

In order to better illustrate dependencies we introduce the notion of an ordered adjacency graph. The edges of this graph will represent the dependency relation captured by Definition 5.8.

Definition 6.1: Order

Given a set V , an *order* on V is a (reflexive, antisymmetric and transitive) relation $O \subseteq V \times V$. We write $a \leq b \iff \langle a, b \rangle \in O$. If $\forall a, b \in V: \langle a, b \rangle \in O \vee \langle b, a \rangle \in O$ then O is *total*, otherwise O is *partial*.

Definition 6.2: Graph

A *graph* is a pair $G = \langle V, E \rangle$, where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. If E is symmetric, then G is *undirected*, else G is *directed*. An order on the vertices of G is denoted $O \subseteq V \times V$. We subscript a vertex $v_i \in V$ to denote its position in a total order $\hat{O}$ on V : $i = |\{u \in V \mid \langle u, v_i \rangle \in \hat{O}\}|$.

Definition 6.3: Dependency Graph

Given a PTS $\mathcal{P} = \langle \langle S_1, \dots, S_N \rangle, \langle \rightarrow_1, \dots, \rightarrow_K \rangle, \mathbf{s}^0 \rangle$ (see Definition 5.2) and a dependency matrix $DM_{K \times N}^{\mathcal{P}}$ for $\mathcal{P}$ (see Definition 5.8), a *dependency graph* D is a *partially ordered, directed, bipartite graph* (or *bi-adjacency graph*) of $DM_{K \times N}^{\mathcal{P}}$, i.e.:

$$D = \langle R \cup C, E \rangle, \text{ where}$$

- $R = \{r_1, \dots, r_K\}$ are vertices with a mutual total order that represent the K transition groups, and
- $C = \{c_1, \dots, c_N\}$ are vertices with a mutual total order that represent the N state slots, and
- $E = \{\langle r_i, c_j \rangle \in R \times C \mid DM_{i,j} \neq -\}$ are the edges.

Example 6.1: Dependency Graph

The *dependency graph* of the Petri net in Example 5.3, with the lexicographic (partial) order $(t_1 < t_2 < t_3 < t_4 < t_5 < t_6 \cup p_1 < p_2 < p_3 < p_4 < p_5)$ is shown below.

The locality of events can be seen clearly in the graph on the left. Next to the graph its bi-adjacency matrix is shown. This matrix is the same matrix shown in Example 5.11. Observe that transition group t_1 does not depend on state slots p_4 and p_5 in any way. One can also observe that the positioning of the vertices follows from the partial order. Later we will show that a better partial order results in less crossing edges in the dependency graph.

6.3 Variable Ordering

Figure 6.1 shows the set of reachable states of the Petri net in Example 5.3 in a particular kind of DD, namely List Decision Diagrams (LDDs) [BP08], with different variable orders. Figure 6.1a is the same LDD from Example 5.6 while Figure 6.1b is an LDD with the same states, but with an order computed by the Cuthill McKee algorithm. In the next sections we will detail how this result is obtained.

Every path from the top left node (with value 0) to the $\{\epsilon\}$ node represents a reachable state. The value in a node indicates the number of tokens. One can see that Figure 6.1b, which variable order is computed using Cuthill McKee, has fewer nodes than Figure 6.1a with the default lexicographic variable order. Thus storing the decision diagram in Figure 6.1b requires less memory. Improving the variable ordering of decision diagrams is a classic NP-Complete [BW96] problem.

Existing algorithms for variable ordering are Noack [Noa99] and FORCE [AMS03]. Both algorithms optimize a heuristic called *span*, or *event span*. Noack's algorithm exploits the structure of a Petri net to order places close to each other. FORCE re-

Figure 6.1: Reachable states as LDD with different orders

peatedly (until there is no improvement) computes the so called *Center Of Gravity* (COG) of transition groups, to place state slots close to each other.

Span is an important metric for symbolic reachability, because it tells how close related variables are ordered near each other. Ordering related variables near each other results in smaller DDs. Span is defined on the dependency matrix and measures the distance between the leftmost and rightmost non ‘-’ column (representing a state slot) of row i (representing a transition group).

Definition 6.4: Span

Given an ordered graph $G = \langle V, E \rangle$, the *vertex span* is a function $s_G: V \rightarrow \mathbb{N}$, such that:

$$s_G(v_i) = \begin{cases} 0 & \text{if } \nexists v_j \in V: \langle v_i, v_j \rangle \in E, \\ \max_{\langle v_i, v_j \rangle \in E} j - \min_{\langle v_i, v_j \rangle \in E} j + 1 & \text{otherwise.} \end{cases} \quad (1)$$

(2)

The *span* or *event span* of a graph is:

$$ES_G = \sum_{v \in V} s_G(v).$$

Example values computed for the running example is as follows.

Example 6.2: Span

Let D be the graph in Example 6.1, the vertex span for t_1 is $s_D(t_1) = 4$, and for p_1 it is $s_D(p_1) = 0$. The event span of graph D is $ES_D = 22$.

We also introduce a version of span that assigns weights to rows of a matrix. These weights signify the location of spans in rows, following [SC06]. There Siminiceanu and Ciardo have shown that it is important that the leftmost non ‘-’ appears as far right as possible in the dependency matrix so that when a transition relation (of a row) is applied, the bottom of a decision diagram is changed rather than the top. Indeed, the leftmost column corresponds with the top of the decision diagram, and the rightmost with the bottom of the DD. Changing the bottom of the DD is usually cheaper than changing the top.

Definition 6.5: Weighted Span

Given an ordered graph $G = \langle V, E \rangle$, and $C = \{v \in V \mid \exists u \in V: \langle u, v \rangle \in E\}$, the *weighted span* or *weighted event span* of a graph is:

$$WES_G = \sum_{v_i \in V} s_G(v_i) \cdot \frac{|C| - m(v_i)}{|C|/2}, \text{ where}$$

$$m(v_i) = \begin{cases} 0 & \text{if } \nexists v_j \in V: \langle v_i, v_j \rangle \in E, \\ \min_{\langle v_i, v_j \rangle \in E} j & \text{otherwise.} \end{cases}$$

Normalization yields:

$$NWES_G = \frac{WES_G}{(|C| \cdot |V - C|)}.$$

If WES is measured on the dependency graph, then C contains only vertices that represent Columns in the dependency matrix (that represent state slots), and $m(v_i)$ gives the leftmost nonzero column number of row i in the dependency matrix. Normalization of WES allows us to compare matrices of different sizes with each other.

Example 6.3: Weighted Span

Let D again be the graph in Example 6.1, the weighted event span is $\text{WES}_D = 30$, and $\text{NWES}_D = 0.99$.

Optimizing (weighted) event span is well known [SC06] to work well for symbolic reachability analysis, but like improving variable orders it is also NP-Complete. We will show that algorithms that have been around for decades and are used in sparse matrix solvers are actually very capable at reducing (weighted) span.

6.4 Nodal Ordering for Sparse Matrix Solvers

Sparse matrix solvers solve a system of linear equations, and this system can be put in a symmetric matrix. As a preprocessing step it is necessary to order these equations in a particular way to limit memory and time usage during solving. Metrics that indicate the memory and time usage are bandwidth and wavefront respectively. Bandwidth measures the distance of nonzeros to the diagonal of a matrix. The wavefront of a row i measures the number of nonzero columns in all rows smaller or equal to i .

In this section we show how to apply nodal ordering algorithms on symmetric matrices to reduce bandwidth and wavefront. This immediately raises an issue, since the dependency matrix in the previous section is asymmetric. We will address this in the next section. As an example algorithm we explain Cuthill McKee [CM69], a simple but effective algorithm, developed in 1969.

6.4.1 Graph Metrics

The bandwidth of a row in a matrix measures the maximal distance of nonzeros in that row to the diagonal. Our conjecture is that bandwidth is important for symbolic reachability, because it tells how close related variables are ordered near the diagonal. We will substantiate this claim in the next sections. The difference between bandwidth and span reduction is that instead of moving a cluster of nonzeros towards an arbitrary column, nonzeros are always moved towards the diagonal. In other words, bandwidth reduction is a special case of span reduction. Bandwidth is formalized as follows.

Definition 6.6: Bandwidth

Given an ordered graph $G = \langle V, E \rangle$, the *vertex bandwidth* [Slo89] is a function

$b_G: V \rightarrow \mathbb{N}$, such that

$$b_G(v_i) = \begin{cases} 0, & \text{if } \nexists v_j \in V: \langle v_i, v_j \rangle \in E, \\ \max_{\langle v_i, v_j \rangle \in E} |i - j|, & \text{otherwise.} \end{cases} \quad (1)$$

$$(2)$$

The *bandwidth* of a graph is the maximum bandwidth of all vertices:

$$B_G = \max_{v \in V} b_G(v)$$

The *profile* of a graph is the sum of all vertex bandwidths:

$$P_G = \sum_{v \in V} b_G(v).$$

Example values for bandwidth computed on the running example are as follows.

Example 6.4: Bandwidth

For the dependency graph D in Example 6.1, we have $b_D(t_1) = 3$ for transition group t_1 , the graph bandwidth $B_D = 3$ and the graph profile $P_D = 14$ ($= 2 + 2 + 2 + 2 + 3 + 3$). Note that $\sum_{p \in \{p_1, \dots, p_5\}} b_D(p) = 0$, since D is directed and there are no outgoing edges from vertices that represent places.

The *wavefront* of a vertex v is the number of adjacent vertices of all vertices smaller or equal to v . Our conjecture is that wavefront is important for symbolic reachability analysis because the lower the wavefront is, the more nonzeros are located near the bottom right of the matrix, similar to the WES-metrics. The rightmost column corresponds to the bottom variable of the DD, so repeatedly applying the transition relations at the bottom rows in the matrix will correspond to saturating the bottom of the DD. This means that when the wavefront is low, less transitions will be fired at the top of the DD during saturation. Wavefront is formalized as follows.

Definition 6.7: Wavefront

Given an ordered graph $G = \langle V, E \rangle$, function $adj: 2^V \rightarrow 2^V$, is defined such that $adj(X) = \{y \in V \mid \exists x \in X: \langle x, y \rangle \in E\} \setminus X$, giving the adjacency set of a set X . The *vertex frontwidth* or *vertex wavefront* [Slo89, KP97] is a function

$f_G: V \rightarrow \mathbb{N}$, such that:

$$f_G(v_i) = |\{v_i\} \cup \text{adj}(\{v_1, \dots, v_i\})|.$$

The *average wavefront* of G is:

$$F_G = \frac{\sum_{v \in V} f_G(v)}{|V|}.$$

Again we compute the values on the running example Petri net, shown below.

Example 6.5: Wavefront

For the dependency graph D in Example 6.1, we have $f_D(t_2) = 5$, $f_D(p_1) = \dots = f_D(p_5) = 1$ and $F_D = 3.4$ ($= \frac{4+5+5+6+6+6+5*1}{11}$).

The next section will address how the previously given metrics may be reduced with heuristic algorithms.

6.4.2 Nodal Ordering

Nodal ordering is a method of applying a permutation to the order of vertices in a graph. We will illustrate this with an algorithm by Cuthill and McKee. The way to apply permutations is however identical for all algorithms in this chapter.

Definition 6.8: Graph Permutation

Given an ordered graph $G = (V, E)$ with order O , and a permutation, which is a bijective function $\pi: V \rightarrow V$, the permuted order O^π for G is:

$$a \leq_\pi b \iff \pi(a) \leq \pi(b).$$

Cuthill McKee is a nodal ordering algorithm for bandwidth reduction. The input to the algorithm is a totally ordered undirected graph and the initial vertices to visit for each disconnected component in the graph. Cuthill McKee is a simple breadth first graph traversal that visits neighbors of vertices in increasing order of degree. If there are vertices with the same degree, an arbitrary vertex may be chosen. The initial vertices for each component are usually computed by means of a pseudoperipheral pair heuristic. A pseudoperipheral vertex is one that has the longest path to any other vertex. The order in which vertices are visited is equivalent to the permutation the algorithm produces. The resulting permutation can be directly applied to the input graph.

Algorithm	Package	Time complexity	Reducing type
Cuthill McKee	Boost	$\mathcal{O}(\hat{D} \cdot \log \hat{D} \cdot V)$	bandwidth
King [Kin70]		$\mathcal{O}(\hat{D}^2 \cdot \log \hat{D} \cdot E)$	bandwidth, profile
Sloan [Slo89]		$\mathcal{O}(\log \hat{D} \cdot E)$	profile, wavefront
Cuthill McKee	ViennaCL	unknown	bandwidth
adv. Cuthill McKee		unknown	bandwidth
GPS [GPS76]		unknown	bandwidth, profile
Notation: $\hat{D}$ is the maximum degree over all vertices			

Figure 6.2: List of nodal ordering algorithms

The reason why Cuthill McKee does not work on asymmetric matrices can be seen in Example 6.1; the vertices $p_1 \dots p_5$ do not have outgoing edges, meaning that not all vertices will be visited. The solution to this problem is presented in Section 6.5. The solution involves adding extra edges to make the set of edges symmetric.

Later, we benchmark with six different implementations of nodal ordering algorithms, implemented in Boost and ViennaCL. The implementations are summarized in Figure 6.2. There are three categories of algorithms, those that reduce only bandwidth, reduce both bandwidth and profile, and those that reduce profile and wavefront. In both Boost and ViennaCL the Cuthill McKee algorithm is implemented.

Our results confirm that the Cuthill McKee implementations differ in both tools. Possibly because the initial set of vertices is different, or arbitrary vertices are chosen differently. The Gibbs Poole Stockmeyer (GPS) algorithm is only implemented in ViennaCL and the time complexity of algorithms in ViennaCL is not precisely known, but should be in the order of similar BFS algorithms.

6.5 Problem and Solution

The main problem with applying nodal ordering algorithms to the dependency graph is that the dependency graph is a directed graph, while nodal ordering algorithms only work on undirected graphs. In other words, the adjacency matrix of such a graph is asymmetric and nodal ordering algorithms only work on symmetric matrices. In this section we show how to symmetrize asymmetric matrices, how bandwidth relates to span and how to de-symmetrize matrices.

6.5.1 Symmetric Representations of Dependencies

Symmetrization [RS06] of a directed graph is defined as follows.

Definition 6.9: Symmetrization

Given an asymmetrix matrix $\mathbf{A} \in \{0, 1\}^{K \times N}$, its symmetrized matrix is:

$$\hat{\mathbf{A}} = \begin{bmatrix} \mathbf{0}^{K \times K} & \mathbf{A} \\ \mathbf{A}^T & \mathbf{0}^{N \times N} \end{bmatrix}, \text{ where}$$

$\mathbf{0}^{X \times X}$ is a square matrix of size X with only 0 entries and $\mathbf{A}^T$ is the transpose of $\mathbf{A}$.

On the graph level, this means that we add the inverse set of edges and assign a total order to the new set of edges, i.e. let $A = \langle V, E \rangle$ be the bi-adjacency graph of $\mathbf{A}$ with order O , and $\hat{A}$ the adjacency graph of $\hat{\mathbf{A}}$ with order $\hat{O}$, then $\hat{A} = \langle V, E \cup E^{-1} \rangle$ and the vertices of $\hat{A}$ are totally ordered, but constrained to $O \subseteq \hat{O}$.

We will illustrate how to apply symmetrization to the running example Petri net.

Example 6.6: Symmetrization

The symmetrized graph and matrix from Example 6.1 are shown below. Their associated metrics are $B_{\hat{D}} = 8$, $P_{\hat{D}} = 88$ and $F_{\hat{D}} = 4.27$. Note that the total order that follows from adding the transpose matrix is $t_1 < t_2 < t_3 < t_4 < t_5 < t_6 \leq p_1 < p_2 < p_3 < p_4 < p_5$.

	t ₁	t ₂	t ₃	t ₄	t ₅	t ₆	p ₁	p ₂	p ₃	p ₄	p ₅
t ₁	-	-	-	-	-	-	w	w	+	-	-
t ₂	-	-	-	-	-	-	+	-	-	w	-
t ₃	-	-	-	-	-	-	w	-	-	+	-
t ₄	-	-	-	-	-	-	-	+	-	-	w
t ₅	-	-	-	-	-	-	-	w	-	-	+
t ₆	-	-	-	-	-	-	-	-	w	+	+
p ₁	w	+	w	-	-	-	-	-	-	-	-
p ₂	w	-	-	+	w	-	-	-	-	-	-
p ₃	+	-	-	-	-	w	-	-	-	-	-
p ₄	-	w	+	-	-	+	-	-	-	-	-
p ₅	-	-	-	w	+	+	-	-	-	-	-

Nodal ordering algorithms can be run on any symmetric matrix. It is thus also possible to create a total graph of the symmetric dependency matrix. Kaveh [Kav06] hints that some nodal ordering algorithms produce even better permutations on the total graph. A graph that has been made total has more vertices and edges, hence the runtime of the algorithm may be longer. This means there is a trade-off between the desired quality of the permutation and runtime of the algorithm. Making a graph total involves transforming edges to vertices and connecting incident edges and adjacent vertices.

Definition 6.10: Total graph

Given a graph $G = \langle V, E \rangle$, a total graph of G is $G_T = \langle V_T, E_T \rangle$, where $V_T = V \cup E$ is the set of vertices, and

$$V_T \times V_T \supseteq E_T = \begin{cases} E \cup & (1) \\ \{\langle a, \langle a, b \rangle \rangle \mid \langle a, b \rangle \in E\} \cup & (2) \\ \{\langle \langle a, b \rangle, a \rangle \mid \langle a, b \rangle \in E\} \cup & (3) \\ \{\langle \langle a, c \rangle, \langle c, b \rangle \rangle \mid \{\langle a, c \rangle, \langle c, b \rangle\} \subseteq E\}. & (4) \end{cases}$$

Here, set (1) are the original edges, (2) are new incident edges from old vertices to new vertices, (3) are new incident edges from new vertices to old vertices, and (4) are adjacent edges between the new vertices.

We can compute metrics on the total graph as follows.

Example 6.7: Total graph

Let $\hat{D}$ be the directed graph in Example 6.6 and $\hat{D}_T$ its *total graph*, if we order the new vertices in $\hat{D}_T$ in lexicographic order we have $B_{\hat{D}_T} = 19$, $P_{\hat{D}_T} = 394$ and $F_{\hat{D}_T} = 10.8$.

We now have two type of graphs to represent the dependencies on which nodal ordering algorithms can be run.

Example 6.8: Nodal Ordering with Cuthill McKee

We can apply Cuthill McKee to reduce bandwidth as follows. A pseudo-peripheral vertex in dependency graph $\hat{D}$ in Example 6.6 is p_4 , so this vertex is visited first. Next, while resolving vertices with minimum degree arbitrarily, Cuthill McKee may visit vertices in the order:

$$p_4 < t_3 < t_2 < t_6 < p_1 < p_3 < p_5 < t_1 < t_5 < t_4 < p_2.$$

Applying this computed order to the symmetrized graph and matrix yields the following result.

	p_4	t_3	t_2	t_6	p_1	p_3	p_5	t_1	t_5	t_4	p_2
p_4	-	+	w	+	-	-	-	-	-	-	-
t_3	+	-	-	-	w	-	-	-	-	-	-
t_2	w	-	-	-	+	-	-	-	-	-	-
t_6	+	-	-	-	-	w	+	-	-	-	-
p_1	-	w	+	-	-	-	-	w	-	-	-
p_3	-	-	-	w	-	-	-	+	-	-	-
p_5	-	-	-	+	-	-	-	-	+	w	-
t_1	-	-	-	-	w	+	-	-	-	-	w
t_5	-	-	-	-	-	-	+	-	-	-	w
t_4	-	-	-	-	-	-	w	-	-	-	+
p_2	-	-	-	-	-	-	-	w	w	+	-

Observe that the *band* the algorithm produces is clearly visible in the matrix. Observe also that there are fewer crossing edges than in Example 6.6. Some metrics of the graph are $B_{\hat{D}^\pi} = 3$, $P_{\hat{D}^\pi} = 41$, $F_{\hat{D}^\pi} = 3.27$. This is a reduction in graph bandwidth of 5, graph profile of 47 and average wavefront of 1 compared to Example 6.6. If we permute the total graph with Boost's Cuthill McKee implementation we get $B_{\hat{D}_T^\pi} = 7$, $P_{\hat{D}_T^\pi} = 171$ and $F_{\hat{D}_T^\pi} = 5.04$. With the total graph we have a reduction in graph bandwidth of 12, graph profile of 223 and average wavefront of 5.76 compared to Example 6.7.

Why reducing bandwidth may also reduce span is because span is limited by twice the bandwidth, plus the diagonal.

Theorem 6.1: Bandwidth limits span

Given an ordered graph $G = \langle V, E \rangle$, we have $\forall v \in V: s_G(v) \leq 2 \cdot b_G(v) + 1$.

Proof. Take an arbitrary $v \in V$, let $n(v) = \{w \in V \mid \exists \langle v, w \rangle \in E\}$ denote the neighbors of v , then we have two cases for v :

Case 1 ($n(v) = \emptyset$). By the first cases in Definitions 6.4 and 6.6 we have $0 \leq 2 \cdot 0 + 1 \iff s_G(v) \leq 2 \cdot b_G(v) + 1$.

Case 2 ($|n(v)| > 0$). We have two symmetric cases:

$$\begin{aligned}
 \text{Case 2.1 } (&|\min_{w \in n(v)} p(w) - p(v)| \geq |p(v) - \max_{w \in n(v)} p(w)|). \\
 s_G(v) = &\max_{w \in n(v)} p(w) - \min_{w \in n(v)} p(w) + 1 = |\max_{w \in n(v)} p(w) - \min_{w \in n(v)} p(w)| + 1 \\
 = &|\max_{w \in n(v)} p(w) - p(v) + p(v) - \min_{w \in n(v)} p(w)| + 1 \\
 \leq &|\max_{w \in n(v)} p(w) - p(v)| + |p(v) - \min_{w \in n(v)} p(w)| + 1 \leq 2 \cdot |p(v) - \min_{w \in n(v)} p(w)| + 1 \\
 = &2 \cdot \max_{w \in n(v)} |p(v) - p(w)| + 1 = 2 \cdot b_G(v) + 1.
 \end{aligned}$$

Case 2.2 ($|p(v) - \max_{w \in n(v)} p(w)| \geq |\min_{w \in n(v)} p(w) - p(v)|$). Symmetric to *Case 2.1*.

Since v is arbitrary we are done. $\square$

Proposition 6.1: Span and symmetrization

Given a graph $G = \langle V, E \rangle$, and its symmetrized graph $\hat{G}$, we have $ES_{\hat{G}} = ES_G + ES_H$, where $H = \langle V, E^{-1} \rangle$.

If G represents the dependency relation, these results tell that the profile $P_{\hat{G}}$ limits the event span ES_G by a factor of two. Thus reducing the value $P_{\hat{G}}$ may reduce the value ES_G as well.

6.5.2 De-symmetrization of Permuted Matrices

The question that remains now is how to de-symmetrize the dependency matrix in Examples 6.6 and 6.7. This is essential, because if we would simply use the permuted total order we can incorrectly swap columns with rows and vice versa. De-symmetrization works as follows.

Definition 6.11: De-symmetrization

Given a partitioned transition system $\mathcal{P} = \langle \langle S_1, \dots, S_N \rangle, \langle \rightarrow_1, \dots, \rightarrow_K \rangle, \mathbf{s}^0 \rangle$, with a symmetrized dependency graph $\hat{D} = \langle R \cup C, E \rangle$ and a permuted total order $\hat{O}^\pi$, where R represents the transition groups $1 \dots K$ and C represents the state slots $1 \dots N$. The de-symmetrized matrix, or directed graph is:

$$D = \langle R \cup C, E \cap (R \times C) \rangle.$$

Its permuted partial order is:

$$O^\pi = (\hat{O}^\pi \cap (R \times R)) \cup (\hat{O}^\pi \cap (C \times C)).$$

If a nodal ordering algorithm is run on the total graph of the dependency graph with order O_T^π , the approach to obtain the partial order is identical:

$$O^\pi = (\hat{O}_T^\pi \cap (R \times R)) \cup (\hat{O}_T^\pi \cap (C \times C)).$$

The permutation for the dependency matrix follows from O^π .

Applying the new partial order to the dependency graph and matrix gives the obvious result as follows.

Example 6.9: De-symmetrization

Below we illustrate the de-symmetrized graph and matrix. Their partial order is $t_3 < t_2 < t_6 < t_1 < t_5 < t_4 \cup p_4 < p_1 < p_3 < p_5 < p_2$. Observe the following.

- The total order on the places $p_4 \dots p_2$ is the same order as the order on variables in the LDD shown in Figure 6.1b. This LDD has fewer nodes than the LDD in Figure 6.1a where no reordering is done.
- The *band* in the matrix below is still visible.
- Due to de-symmetrization, the small span of each event $t_3 \dots t_4$ is now visible too: the leftmost and rightmost non ‘-’ columns are close.

Now let D^π be the de-symmetrized graph, key are its reduced event span metrics:

$$ES_{D^\pi} = 16, \text{ and}$$

$$WES_{D^\pi} = 19.$$

We thus have a reduction in event span of 6 compared to the value computed in Example 6.2, and a reduction in weighted event span of 11 compared to Example 6.3. If we permute the total graph with Cuthill McKee in Boost we also get $ES_{D^\pi} = 16$ and $WES_{D^\pi} = 19$. *These reductions are the main contribution of this chapter.*

As an aside, note that we also have $B_{D^\pi} = 2$, $P_{D^\pi} = 9$ and $F_{D^\pi} = 3.1$, which is a reduction in graph bandwidth of 1, graph profile of 5 and average wavefront of 0.3 compared to Examples 6.4 and 6.5.

Figures 6.3a and 6.3b visualize some dependency matrices from real world examples, after applying some nodal reordering algorithms from Figure 6.2. Their NWES metrics are shown as well. The first two matrices are of a Petri net with 20 dining philosophers, one of the best results achieved in our benchmarks. Even on instances with 5000 philosophers (*25.000 variables*) we get very small weighted event span, and the permutation is computed within milliseconds. The matrices

Figure 6.3: Example de-symmetrized matrices

from the `database10UNFOLD.pnml` show the more typical structure of dependency matrices, e.g. the *band* produced by the GPS algorithm is clearly visible. In Figure 6.3b one can see the difference between Sloan and GPS. Sloan does not try to reduce bandwidth only profile: there are more outliers visible in the matrix permutation computed by Sloan compared to the order computed by GPS. The NWES value is lower in the Sloan case; in our experiments we also see that Sloan is more capable at reducing NWES than GPS.

We have set up a large number of experiments, in order to answer several relevant questions: First, which algorithm and which type of symmetric graph is best for reducing WES? Second, how well does this result compare to Noack and FORCE which are currently considered state of the art? Third, does the choice of best algorithm/graph depend on the chosen specification language? Fourth, can we quantify the performance of the best algorithm/graph?

6.6 Experiments

To show the applicability of nodal ordering algorithms to variable ordering, we benchmark with five different specification languages². There are 47 B specifications, collected through the B community [LB08]. We have 264 DVE specifications from the BEEM [Pel07] database. From the examples directory in the mCRL2 [Cra+13] distribution we collected 142 specifications. There are 314 Petri nets from the 2015 model checking contest [Kor+15]. Also, we have a collection of 18 Promela models.

For two reasons we can not always use complete sets of specifications, such as for the PNML language, where the complete set consists of 361 Petri nets. First, some total graph representations of the dependencies are too large to compute an order for within a time limit of one hour. Second, the implementation of Sloan in Boost crashes when run on a graph that has disconnected components. In our benchmarks we thus vary over a total of 785 specifications, two graph representations, and nine ordering algorithms. The nine algorithms consist of those in Figure 6.2 plus two variations of Noack's [Noa99] algorithm, and FORCE [AMS03].

Our benchmark generates a lot of data; to concisely present this data we use graphs instead of tables. Furthermore, to show which combination of algorithm and graph performs best, we compute *Mean Standard Scores* (MSSs). Scatter plots of these MSSs are shown later. These plots allow us to quantify the performance of the best algorithm/graph. In these graphs we use some abbreviations for certain algorithms:

- CMB: Cuthill McKee in Boost,

²Reproduction instructions at: <https://github.com/utwente-fmt/BW-NFM2016>

- aCM: advanced Cuthill McKee (in ViennaCL),
- K: King (in ViennaCL),
- GPS: Gibbs Poole Stockmeyer (in ViennaCL), and
- CMV: Cuthill McKee in ViennaCL.

Figures 6.4 to 6.9 show the MSSs for all five specification languages, plus the MSS for all languages combined. The MSS is defined as follows.

Definition 6.12: Mean Standard Score

Let $A \subset \{\text{FORCE, Sloan, GPS, CMB, King, aCM, CMV, none, Noack1, Noack2}\} \times \{\text{total, bipartite}\}$ be the set of appropriate combinations of algorithms and graph representations. We say appropriate, because e.g. Noack1 and Noack2 are not run on the total graph, and neither is running no (none) reordering algorithm. Let $S \subseteq *.pnml \cup *.mcr12 \cup *.B \cup *.dve \cup *.promela$ be a (sub) set of specifications, e.g. one possibly including a Petri net with 20 dining philosophers, or Petersons mutual exclusion algorithm in DVE. Given an appropriate combination of an algorithm and graph $a \in A$, the MSS for a is:

$$MSS_a = \frac{\sum_{s \in S} \frac{m(s,a) - \mu_{a' \in A} m(s,a')}{\sigma_{a' \in A} m(s,a')}}{|S|}, \text{ where}$$

- $m(s,a)$ denotes the value of metric m (e.g. WES) for a and a specification s , and
- $\mu_{a' \in A} m(s,a')$ denotes the mean for specification s over all A , and
- $\sigma_{a' \in A} m(s,a')$ denotes the standard deviation for specification s over all A .

In Figures 6.4 to 6.9 the appropriate values for A appear on the x-axes and the values for MSS_a appear on the y-axes³. An example MSS in Figure 6.4 for WES is $MSS_{\langle \text{Sloan}, \text{total} \rangle} = -0.57$. This means that Sloan, run on the total graph, is on average 0.57 standard deviations better than the average of all graphs and algorithms, run on all specifications. All graphs in Figures 6.4 to 6.9 are sorted from smallest to largest WES, meaning that the best algorithm (according to WES) appears on the left.

Looking at the figures we can observe that Sloan is overall second best (after FORCE) and using the total graph is better than using the bi-adjacency graph,

³There are three side notes. First, μ and σ for bandwidth, profile and wavefront are computed per graph type, because the bipartite and total graph have different sizes. Second, Noack1 and Noack2 can only be computed directly on Petri nets (PNML, Figure 6.8), so bandwidth, profile and wavefront are unknown. Third, when FORCE is executed or without reordering, bandwidth, profile and wavefront are not reported. The reason is that our symmetrization approach typically produces high values for those metrics. Event span does not have this problem.

except for B and mCRL2. For Promela the Sloan algorithm seems to be the winner. We explicitly note that Sloan beats Noack in minimizing WES (see Figure 6.8). This is interesting because Noack's algorithms are specifically designed for Petri nets, while Sloan is not.

Figure 6.4: MSS for all languages ($|S| = 785$) Figure 6.5: MSS for B ($|S| = 47$)

+ Weighted Event Span
 ■ Event Span
 ▲ bandwidth
 × profile
 ● average wavefront
 ✱ time

Figure 6.6: MSS for DVE ($|S| = 264$)

Figure 6.7: MSS for mCRL2 ($|S| = 142$)

Figure 6.8: MSS for PNML ($|S| = 314$) Figure 6.9: MSS for Promela ($|S| = 18$)

All graphs tell us in detail which algorithms and symmetric graphs perform better than others, e.g. we can conclude:

1. running any algorithm is better than none,
2. running GPS on the total graph is worse than running it on the bipartite graph,
3. King's algorithm does not perform well,
4. running algorithms on the total graph takes longer, and
5. running nodal ordering algorithms is beneficial for any specification language.

The question still remains, how good an MSS of -0.57 (WES obtained with Sloan on the total graph) actually is. Consider Figure 6.10, which shows the normalized weighted event span for all 785 specifications. A point below the line $x = y$ means that Sloan, run on the total graph, produces better NWES than running no reordering algorithm. There are only 69 specifications where the initial order that follows from the specification had a better WES than the ordering computed by Sloan. Some of the 716 improvements are extreme, showing an NWES reduction from ~ 1.0 to nearly 0.

Figure 6.10: NWES values

6.7 Related and Future Work

There are several branches of related work that also serves as a starting point for future work. The first one is to see whether bandwidth reduction algorithms can be applied to problems similar to symbolic reachability. The second is to try different graph representations aside from the total graph in Definition 6.10 and different approaches to symmetrizing an asymmetric graph. The third branch is to look at more recent bandwidth and wavefront reduction algorithms since the Sloan algorithm was already published in 1986.

Our approach works for disjunctive partitioning schemes. Yet, a remaining question is whether our method also works for conjunctive [BCL91] partitioning schemes, such as in NuSMV [Cim+02]. Furthermore bandwidth and wavefront reduction should be applicable to SAT/SMT solving, since also the FORCE algorithm is used in both symbolic reachability and SAT/SMT solving. The hypergraph used by FORCE is equivalent to our dependency graph: a hyperedge is equivalent to a vertex (including its edges) that represents a transition group.

Another branch of related work is to consider other graph representations and other bandwidth and wavefront reduction algorithms. Kaveh [Kav06] discusses many different graph transformations of the dependency graph on which nodal ordering algorithms can be run. We have only investigated the total graph. Reid and Scott [RS06] provide two more methods of symmetrizing an asymmetric matrix $\mathbf{A}$, namely $\mathbf{A} + \mathbf{A}^T$ and $\mathbf{A} \cdot \mathbf{A}^T$. Additionally the authors provide a modified Cuthill McKee algorithm that can be run on an asymmetric matrix directly, available in the HSL library.

After we first presented this work in [MP16], Sloan’s algorithm was also investigated by Amparore et al. [Amp+18]. Their approach is slightly different than ours. First, they symmetrize the matrix by means of $\mathbf{A}' = \mathbf{A} + \mathbf{A}^T$. Second, entries in $\mathbf{A}'$ are only set based on some threshold to avoid generating a dense $\mathbf{A}'$ in degenerate cases. Third, the Sloan algorithm accepts two weight parameters $\frac{w_1}{w_2}$ which form a fraction. These weight parameters affect the traversal of vertices. In Boost these weights are set to $\frac{1}{2}$ by default. Amparore et al. [Amp+17] suggest to change the fraction to $\frac{1}{16}$, which is now the default in LTSmin.

Recent surveys [Maf14, BO15] cover the state of the art in bandwidth reduction, including metaheuristic algorithms, of which many have been developed in the past decade. For example, advances have been made in parallelizing [Kar+14, RBC17] nodal ordering algorithms. Running nodal ordering algorithms on the total graph is considerably slower than on the default dependency graph, so running parallel algorithms on the total graph would provide a welcome speedup. Another improved profile minimization algorithm is published in [AKA19]. In this work profile minimization is formulated as an m -way ordered hypergraph partitioning

problem.

We can conclude that bandwidth and wavefront reduction is still a very active research area. There are many algorithms still to try that hopefully produce even better variable orders beyond those we can currently compute with Boost and ViennaCL.

6.8 Conclusion

We have shown that bandwidth and wavefront reduction algorithms are clearly applicable to symbolic reachability analysis for many specification formalisms. We demonstrated how they perform relative to each other and to other state-of-the-art algorithms. The best nodal ordering algorithm for variable reordering is Sloan; for all five tested specification languages Sloan is the clear winner when it comes to reduction of weighted event span. Furthermore Sloan's algorithm beats Noack's algorithms that are specifically designed for Petri nets, but FORCE performs even better on WES.

Figures 6.4 to 6.8 show that FORCE produces better WES than Sloan. However, on Promela specifications (Figure 6.9) Sloan's algorithm is the winner. Future work includes performing extensive benchmarks to see how well saturation algorithms in LTSmin actually perform with the variable orders computed by the presented nodal ordering algorithms. This direction of future work looks very promising because Amparore et al. [Amp+18] have already experimented with a saturation algorithm in GreatSPN [ABD14] that uses the Meddly [BM10] tool for Multi-way Decision Diagrams (MDDs).

Since this work on bandwidth reduction was first published in [MP16] its applicability to variable ordering is certainly confirmed by the results obtained with the GreatSPN model checker. We hope to see the algorithms presented in this chapter also be applied to other model checkers that apply saturation algorithms, such as SmArT [CMW09], MARCIE [HRS13] and ITS-Tools [Thi15].

We have done some preliminary experiments with a saturation-like algorithm in LTSmin as well, mainly to illustrate how Sloan affects WES, *peak nodes*, and reachability time. The peak nodes indicate the highest number of DD nodes to store the visited set during the whole computation, often much higher than the DD-size at the end of reachability. So peak nodes indicate the limiting factor in terms of memory usage. Keeping the number of peak nodes low is the main motivation to perform static variable ordering.

Figure 6.11 shows the Mean Standard Score (MSS) for peak nodes, WES and reachability time. The MSS values are sorted from smallest peak nodes to largest, showing the best algorithm on the left. We ran a benchmark on all 106 PNML

Figure 6.11: Saturation results ($|S| = 106$)

specifications for which the saturation algorithm completed within 30 minutes. For peak nodes and time, the best algorithm is Sloan, not FORCE. The difference in MSS between Sloan/total and FORCE for peak nodes is 0.13. Also, Noack appears even further to the right than in Figure 6.8. This suggests that Sloan has interesting properties, besides showing that bandwidth reduction is a good alternative to span reduction. We conjecture that Sloan is a good algorithm for full saturation since it also reduces wavefront (which is clearly visible in Figures 6.4 to 6.9). Moreover, nodal ordering algorithms also compute an order on the transition groups, unlike FORCE and Noack. It would be interesting to see how chaining and saturation benefit from the computed order on transition groups.

Case Study: Symbolic Reachability Analysis for ProB

Abstract

We present a symbolic reachability analysis approach for B that can provide a significant speedup over traditional explicit state model checking. The symbolic analysis is implemented by linking ProB to LTSmin. The link is achieved via LTSmin's PINS interface, allowing ProB to benefit from LTSmin's analysis algorithms, while only writing a few hundred lines of glue-code, along with a bridge between ProB and C using ØMQ. ProB supports model checking of several formal specification languages such as B, Event-B, Z and TLA⁺. Our experiments are based on a wide variety of B-Method and Event-B specifications to demonstrate the efficiency of the new link. Among the tested categories are state space generation and deadlock detection; but action detection and invariant checking are also feasible in principle. In many cases we observe speedups of several orders of magnitude. We thus provide a new scalable, symbolic analysis algorithm for the B-Method and Event-B, along with a platform to integrate other model checking improvements via LTSmin in the future.

7.1 Introduction

In this chapter we describe the process, technique and design decisions we made for integrating the two tooling sets: LTSmin and ProB. Use of the ProB tool in a rail system case study [Jam+13], where model checking large industrial sized complex specifications was performed, illustrated that there continues to be limitations with the tooling. Model checking CSP||B [ST05] specifications in ProB was the original

motivator for this research, and based on a promising initial exploration [Whi15], this chapter defines a systematic integration of the two tooling sets.

It is through PINS that we have been able to leverage the ProB tool, therefore allowing us to take advantage of LTSmin’s algorithmic back-ends. In this chapter we focus on the new ProB language front-end, the grouping of transitions, and the symbolic back-end. In Section 7.5 we also briefly discuss state variable orders.

ProB [LB08] is an animator and model checker for many different formal languages [PL10], including the classical B-Method [Abr05], Event-B [Abr10], CSP, CSP||B, Z and TLA⁺. ProB can perform automatic or step by step animation of B machines, and can be used to systematically verify the behaviour of machines. The verification can identify states which do not meet the invariants, do not satisfy assertions or that deadlock. At the heart of ProB is a constraint solver written in SICStus Prolog [Car+88], which enables the tool to animate and model check high-level specifications. The built-in model checker is a straightforward, explicit state model checker (albeit augmented with various features such as symmetry reduction [LM10] or partial order reduction [DL14]). The explicit state model checker TLC can also be used as a backend [HL16].

The purpose of this chapter is to make use of the advanced features of the LTSmin model checker, such as symbolic reachability analysis, by linking the ProB state exploration engine with LTSmin. This is achieved through a C programming interface [Ben15] within the ProB tool, allowing the representation of a state to be compatible for LTSmin’s consumption. In this chapter the integration focuses on what is required in order to perform symbolic reachability analysis of B-Method and Event-B specifications. The contribution of this research is a new tool integration and a discussion of its performance.

The rest of the chapter is structured as follows: Section 7.2 presents an overview of the B-Method, a running example and an illustration of definitions of transition systems used by LTSmin. Section 7.3 details the symbolic reachability analysis and Section 7.4 outlines the implementation details. Section 7.5 provides empirical results from performing reachability analysis benchmarking examples in ProB alone and using the new integration of the two tools. Related work is discussed in Section 7.6. The chapter concludes in Section 7.7.

7.2 Transition Systems for the B-Method

A B machine consists of a collection of clauses and a collection of operations. The **MACHINE** clause declares the abstract machine and gives it its name. The **VARIABLES** clause declares the variables that are used to carry the state information within the machine. The **INVARIANT** clause gives the type of

the variables, and more generally it also contains any other constraints on the allowable machine states. The **INITIALISATION** clause determines the initial state(s) of the machine. Operations in a machine are events that change the state of a machine and can have input parameters. Operations can be of the form **SELECT** P **THEN** S **END** where P is a guard and S is the action part of the operation. The predicate P must include the type of any input variables and also give conditions on when the operation can be performed. When the guard of an operation is true then the operation is enabled and can be performed. If the guard is the simple predicate **true** then the operation form is simplified to **BEGIN** S **END**. An operation can also be of the form **PRE** P **THEN** S **END** so that the predicate is a precondition and if the operation is invoked outside its precondition then this results in a divergence (we do not illustrate this in our running example). Finally, the action part of an operation is a *generalized substitution*, which consists of one or more assignment statements (in parallel) to update the state or assign to the output variables of an operation. Conditional statements and non-deterministic choice statements are also permitted in the body of the operation. The example in Listing 7.1 illustrates the *MutexSimple* machine with three variables and five operations. Its initial state is deterministic and *wait* is set to MAXINT. For MAXINT=1 we get 4 states; the state space constructed by ProB can be found in Figure 7.1¹. From the initial state only the guards for **Enter** and **Leave** are true. Following an **Enter** operation the value of the *cs* variable is true which means that the guard of the **CS_Active** operation is true and the system can indicate that it is in the critical section by performing the **CS_Active** operation. The example presented could also be considered as an *Event-B* example since it is a simple guarded system. We do not elaborate further on the notation of Event-B in this chapter but note that the results in the subsequent sections are also applicable to Event-B.

Listing 7.1: *MutexSimple* B-Method machine example

```

1 MACHINE MutexSimple
2 VARIABLES cs, wait, finished
3 INVARIANT
4   cs: BOOL & wait: NATURAL & finished: NATURAL
5 INITIALISATION cs := FALSE || wait := MAXINT || finished := 0
6 OPERATIONS
7   Enter      = SELECT cs = FALSE & wait > 0 THEN
8               cs := TRUE || wait := wait - 1 END;
9   Exit       = SELECT cs = TRUE THEN
10              cs := FALSE || finished := finished + 1 END;
11   Leave      = BEGIN cs := FALSE END;
12   CS_Active  = SELECT cs = TRUE THEN skip END;
```

¹One subtle issue is that LTSmin actually only supports a single initial state; this is solved by introducing the artificial *root* state linked to the initial states properly.


```

11   Restart  = SELECT finished > 0 THEN
        wait := wait + 1 || finished := finished - 1 END
12 END

```

Figure 7.1: *MutexSimple* statespace for $\text{MAXINT}=1$

A B-Method and Event-B specification induces a partitioned transition system following Definitions 5.1 and 5.2. Initial states are defined by the initialization clause (i.e. $s^0 = \langle \text{FALSE}, \text{MAXINT}, 0 \rangle$) and the individual operations together define the transition relation $\rightarrow$. The goal of this chapter is to show how the computation of all reachable states of the partitioned transition system can be sped up by using the symbolic method.

7.3 Symbolic Reachability Analysis for B

Computing the set of reachable states of a transition system can be done efficiently with symbolic algorithms if many transition groups touch only a few variables. This concept is known as event locality, see Chapter 6. Many specifications of transition systems in the B-Method employ event locality. In the B-Method event locality occurs in operations, where only a few variables are read from, or written to. For example in Listing 7.1 operation **CS_Active** only reads from **cs** and **Leave** only writes to **cs**. This event locality benefits the symbolic reachability analysis, so that the algorithm is capable of coping with the well known state space explosion problem. Since the B-Method employs event locality we build on the foundations of earlier work on LTSmin discussed in Chapters 5 and 6 and extend it to ProB. To perform symbolic reachability analysis of the B-Method, ProB should provide LTSmin with read matrices and write matrices. These matrices inform LTSmin about the locality of events in the B-Method.

The syntactic dependency analysis for the B-method is similar to the analysis for the state update specification in Definition 5.3. In fact, aside from implementation concerns, the dependency analysis for the B-method is easier because every write certainly does not copy values (the writes are not dynamic). For the example in Listing 7.1 the syntactic dependency information is as shown in Figure 7.2.

	cs	wait	finished
Enter	+	+	-
Exit	+	-	+
Leave	w	-	-
CS_Active	+	-	-
Restart	-	+	+

Figure 7.2: Dependency Matrix for Listing 7.1

We infer that:

- The transition group of **Enter** is only completely independent from **finished**: both **cs** and **wait** are read from in the guard and written to in the assignment statements.
- **Exit** is only completely independent from **wait**. This is similar to **Enter**, however **finished** is read from in the right-hand side of the second assignment.
- **Leave** is only write dependent from **cs**: **cs** is only assigned to in the simplified form of the operation **Leave**.
- **CS_Active** is only read dependent from **cs**: it is only read from in the guard.
- **Leave** is only completely independent from **cs**: **cs** does not appear anywhere in the operation.

In general the dependency information of B machines is used to improve the amount of NEXT-STATE calls. The NEXT-STATE function however, requires reasoning over a set of states by means of decision diagrams.

7.3.1 Symbolic Encoding of B-method States

The symbolic reachability algorithm in Algorithm 5.1 uses List Decision Diagrams (LDDs) to store the reachable states and transition relations. Due to the sharing of state vectors within an LDD, the memory usage can be very low, even for very large state spaces. Three example LDDs for the running example are given in Figure 7.3. The LDDs represent the set of reachable states $\mathcal{R}$ at each iteration of the **while** loop. In an LDD every path from the top left node to $\{\epsilon\}$ is a state, e.g., the initial state $\langle \text{FALSE}, 1, 0 \rangle$ in Figure 7.3b. A node in an LDD represents a unique set of (sub) vectors, e.g., $\{\epsilon\}$ represents the set of zero-

Figure 7.3: LDDs of the reachable states

length vectors and the right-most 0 of variable `wait` in Figure 7.3d encodes the set $\{\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 1, 0 \rangle\}$. Figure 7.3c shows that firing `Enter` will add $\langle \text{TRUE}, 0, 0 \rangle$ to R . In Figure 7.3d $\langle \text{FALSE}, 0, 0 \rangle$ and $\langle \text{FALSE}, 0, 1 \rangle$ are added to R , by firing `Leave` and `Exit` respectively. The benefit of using LDDs for state storage is due to the sharing of state vectors. For example, the subvector $\langle \text{FALSE} \rangle$ of the states $\{\langle \text{FALSE}, 0, 0 \rangle, \langle \text{FALSE}, 0, 1 \rangle, \langle \text{FALSE}, 1, 0 \rangle, \langle \text{FALSE}, 1, 1 \rangle\}$ in iteration 3 is encoded in the LDD with a single node. For bigger state spaces the sharing can be huge; resulting in a low memory footprint for the reachability algorithm.

7.3.2 Performance: NEXT-STATE Function

There are two big differences of Algorithm 5.1 with classical explicit state model checking as used by ProB [LB08]. First, the state space is represented using an LDD datastructure, which enables the sharing among states. Second, independence is used to apply the NEXT-STATE function not state by state, but for entire sets of states in one go. For each of the 4 states in Figure 7.1, the explicit model checking algorithm of ProB would check whether each of the 5 operations are enabled; resulting in 20 next-state calls. With LTSmin's symbolic reachability Algorithm 5.1, only 12 NEXT-STATE calls are made. This is shown in Table 7.1, where + means enabled, - means disabled, and C means that LTSmin has reused the results of a previous call to ProB².

If we initialize `wait` with `MAXINT = 500`, the state space has 251,002 states. The runtime with ProB is 70 seconds, with LTSmin+ProB 48 seconds and LTSmin performs only 6012 NEXT-STATE calls. The example does not have a lot of concur-

²Some results are reused from calls to the artificial `root` state.

State#	Variable			Operation				
	cs	wait	finished	Enter	Exit	Leave	CS_Active	Restart
1	FALSE	1	0	+	C	C	C	-
2	TRUE	0	0	-	+	+	+	-
3	FALSE	0	0	-	-	C	-	C
4	FALSE	0	1	C	-	C	C	+

Table 7.1: The NEXT-STATE calls for sets of states

Figure 7.4: High level design showing the integration of LTSmin and ProB

rency and uses only simple data structures (and thus the overhead of the LTSmin’s ProB front-end is more of a factor compared to the runtime of ProB for computing successor states); other examples will show greater speedups (see Section 7.5). The purpose of this example is to merely illustrate the principles.

7.4 Technical Aspects and Implementation

We used a distributed approach to integrate ProB and LTSmin. Both tools are stand-alone applications, so a direct integration, i.e., turning one of the tools into a shared library would require considerable effort. We therefore added extensions to both tools that convert the data formats and use sockets to communicate with each other. A high level view of the integration is shown in Figure 7.4. We use the ØMQ [Hin13] library for communication. ØMQ is oriented around message queues and can be used as both a networking library with very high throughput, and as a concurrency framework. We have chosen ØMQ because it worked very well in previous work [Ben15].

The communication is always initiated by LTSmin; it sends a message and blocks until it receives the answer from ProB. We usually run both tools on a single computer using interprocess (IPC) sockets, but it is only a matter of configuration to

run the tools on different machines using TCP sockets. We currently only support Linux and Mac OS. The communication protocol is straightforward. Reachability analysis is initiated from LTSmin by sending an initialisation packet. ProB answers with a message containing the relevant static information about the specification, such as the dependency matrices that LTSmin requires (see Section 7.3). Each matrix is encoded as a 2-dimensional array, which is not optimal for a sparse matrix but is not an issue because we only send the matrices once. The packet also contains the list of variables, their types, the list of transition groups, and the initial state.

States are represented as a list of so called *chunks* [BP08]. A chunk is one of the elements in the state vector according to Definition 5.2. In the case of B, each chunk is a value of one of the state variables. Because LTSmin will not look inside the chunks, we simply use the binary representation of ProB's Prolog term that represents the value of a variable. This has the advantage that ProB does not have to keep information about the state space. It can always recover a state from the chunks that are sent by LTSmin. The transition groups correspond to B operations as explained in Section 7.2. The names of transition groups will form the edge labels of the LTS that is instantiated by LTSmin. These labels are also encoded as chunks.

Once the initial setup is done, LTSmin will start to ask ProB for successor states for *specific* transition groups. It will send a next-state message containing a state and a transition group. The state that LTSmin sends, is a list of chunks and ProB's LTSmin extension can directly consume them and construct a Prolog term that internally represents a state. Using this constructed state and the transition group, the extension will then ask ProB for all successor states. The result is a list of Prolog terms, each representing a successor state. The extension transforms the list of states into a list of lists of chunks and sends them back to LTSmin. This is repeated until LTSmin has explored all necessary states and sends a termination signal.

The next-state messaging follows Definition 5.12, the read projection is achieved by replacing all read independent variables by a dummy value, in particular values from the initial state.

7.5 Experiments

To demonstrate that the combination of ProB and LTSmin improves, the performance of the reachability analysis and deadlock detection are compared to the standalone version of ProB. To this end we use a wide range of B and Event-B specifications. Our benchmark suite contains puzzles (e.g., towers of Hanoi) as well as specifications of protocols (e.g., Needham-Schroeder), algorithms (e.g.,

Simpson’s four slot algorithm) and industrial specifications (e.g., a choreography model by SAP, a cruise control system by Volvo and a fault tolerant automatic train protection system by Siemens)³.

The experiments are run on Ubuntu 15.10 64-bit, with 8 GB RAM, 120 GB SSD and an Intel Sandybridge Mobile i5 2520M 2.50 GHz Dual core. The version of ProB used in this chapter is “1.5.1-beta3”, and LTSmin git tag “LTSminProB-iFM2016”⁴.

Table 7.2 summarizes a selection of the experiments that we ran. The last two specifications are written in Event-B, the others are B machines. In these experiments we used Breadth-First Search (BFS) and looked for deadlocks. A deadlock was found only for the Philosophers model (this is also why there are no next state call statistics for this specification). The table also contains the number of next state calls for ProB reachability analysis on its own and when called from LTSmin’s symbolic reachability analysis algorithm (by means of the new integration) without deadlock checking. One can clearly see that we obtain a considerable reduction in wallclock time. The ProB time is the walltime of the ProB reachability analysis and initial state computation and does not include parsing and loading. The LTSmin CPU time column shows how much time is spent in the LTSmin side of the symbolic reachability analysis algorithm. The LTSmin wall time shows the total walltime, and this also contains the time spent in the communication layer and waiting for the ProB process to compute the next states. To compare the benefit of our new algorithm we compute the speedup of the walltime in the last column by dividing the ProB walltime from column 5 with the LTSmin walltime in column 7.

We can see that for some of the smaller instances the overhead of setting up LTSmin does not pay off. However, for all larger ones, except for the Train1_Lukas_POR model considerable speedups were obtained.

A major result we achieved with non default settings for LTSmin, is for elevator12.eventb. This specification is not listed in Table 7.2, because ProB runs out of memory on the hardware configuration used for this experiment. LTSmin computed in 34 seconds, with 96,523 NEXT-STATE calls, that the LTS has 1,852,655,841 states. As reachability algorithm we chose chaining [RCP95], and to compute a better variable order, we ran Sloan’s profile reduction algorithm (Chapter 6) on the dependency matrix.

As far as memory consumption is concerned; when performing reachability analysis on CAN_BUS, the ProB process consumes 370 MB real memory, while the LTSmin process consumes 633 MB, with the default settings. With the default

³More detailed descriptions can be found in [Ben+16b].

⁴Reproduction instructions at <https://github.com/utwente-fmt/ProB-LTSmin-iFM16>.

Benchmark	Events	States	Prob Next St Calls	LTsmin NextSt Calls	Prob Wall (ms)	LTsmin CPU (ms)	LTsmin Wall (ms)	Speedup
CAN_BUS	21	132600	2784560	3534	122850	660	1590	77.264
ConcurrentCounters	4	110813	443249	113032	21820	2760	13820	1.579
Cruise_finite1	26	1361	35361	1667	2900	100	1020	2.843
file_system	8	698	5577	1198	1900	180	4660	0.41
MutexSimple	5	10	46	26	10	10	190	0.053
Philosophers	5				480	40	590	0.814
SiemensMiniPilot_Abrial0	9	181	1621	182	100	20	260	0.385
Simpson_Four_Slot	9	46658	419906	2089	17310	200	860	20.128
Train1_Lukas_POR	8	24637	197082	101441	33660	6480	50260	0.670
nota	11	80719	887899	588	287970	130	660	436.318
pkeyprot2	10	4412	44111	2004	22190	210	1710	12.977
Ref_Switch_mch	38	29861	1134681	1281	160600	490	1260	127.460
obsw_M001	21	589279	12374779	23406	2051320	1620	12420	165.163

Table 7.2: B and Event-B Machines, with BFS and deadlock detection

settings LTSmin will allocate 2^{22} elements (≈ 100 MB) for the node table and 2^{24} elements (≈ 500 MB) for the operations cache. If we choose a smaller node table and operations cache for the LDD package (both 2^{18} elements), LTSmin consumes only 22 MB. The default settings for LTSmin are geared towards larger symbolic state spaces than that of CAN_BUS. The default node table and cache are too big for CAN_BUS, and thus not completely filled during reachability.

In summary, Table 7.2 shows that for several non-trivial B and Event-B instances, considerable improvements can be obtained using the symbolic reachability analysis technique described in this chapter.

7.6 Related Work

Other techniques for improving model checking for B-Method and Event-B specifications have been developed and evaluated in the recent years. The authors in [HL16] presented a translation from the B-Method to TLA^+ , with the goal of using the TLC model checker [YML99] as backend. TLC has no constraints solving capabilities, and as such it can only deal with lower level specifications. On the other hand, its execution can be considerably faster than ProB, and its explicit state model checking engine (which stores fingerprints) is very efficient. On the downside, there is a small probability that fingerprint collisions can occur. The experiments show that TLC does not deal well with benchmark programs which require constraint solving (graph isomorphism, JobsPuzzle, ...), running up to three orders of magnitude slower than ProB or LTSmin with ProB. However, it does deal very well with lower level specifications, e.g., it is faster than LTSmin for ConcurrentCounters. For many benchmark specifications however, even those not requiring constraint solving, our symbolic reachability analysis is faster. For example, for the `nota` example, TLC runs in about 10 seconds—faster than ProB without any optimization—but slower than using LTSmin.

Symmetry reduction [LM10] can be very useful; but exponential improvements usually occur only on academic examples. In [Ben+16a] we have experimented with the hash marker symmetry reduction, which is ProB's fastest symmetry method, but is generally not guaranteed to explore all states. The method gives the best results for certain models (e.g., `file_system`). But for several of the larger, industrial examples shown above, its benefit is not of the same scale as LTSmin.

We have also experimented with partial order reduction in [Ben+16a]. [DL14] uses a semantic preprocessing phase to determine independence (different from our purely syntactic determination in Section 7.3). As such, it can induce a slow down for some examples where this does not pay off (e.g., `file_system`). ProB's partial order reduction obtains the best times for certain specifications with a large degree of concurrency (ConcurrentCounters, SiemensMiniPilot_Abrial, and `nota`).

However, once we start doing invariant checking, [DL14] does not scale nearly as well (e.g., it takes 134 seconds instead of 0.5 seconds for LTSmin checking the *nota* model). But even without invariant checking, there are plenty of examples where the symbolic reachability analysis approach is better (e.g., *Cruise_finitel*, *Philosophers*, *Simpson_Four_Slot* and almost two orders of magnitude for *CAN_BUS*).

As an extension of the work presented in this chapter, the authors in [KLM18] experiment with a more complete model checking toolset based on LTSmin. Specifically, support for evaluating state labels is added and this allows for experiments such as symbolic invariant checking, LTL model checking, partial order reduction and multi-core explicit-state invariant checking. In all of these categories promising speed ups are seen. These speed ups are enabled by the presented integration of ProB and LTSmin in this chapter.

7.7 Conclusion

This chapter presents a stable extensible architectural link between ProB and LTSmin through *OMQ*. We show the benefits of applying symbolic reachability analysis implemented in LTSmin to specifications written in languages supported by the ProB model checker. Speed ups are achieved that are of several orders of magnitude. In an extreme case (*elevator12.eventb*) we show that using LTSmin can make reachability tractable, where it was not by solely using ProB. Specifically, we had to apply the read-write optimizations discussed in Chapter 5 and the Sloan algorithm from Chapter 6.

In summary, we have presented a new scalable, symbolic analysis algorithm for the B-Method and Event-B, along with a platform to integrate other model checking improvements via LTSmin in the future, of which [KLM18] is an example.

Automated techniques for validation and verification are crucial for making sure that safety critical systems are reliable. In this thesis we set out to improve the efficiency of the otherwise labor-intensive and error prone manual techniques for analyzing system behavior. In particular, we have shown how we enhanced automated procedures for learning and verifying system behavior, as well as for testing. The presented method enables automation of three key aspects of testing: generation of test cases, execution of test cases and deciding their verdicts (pass/fail).

Mathematical models describing the behavior of systems can either be specified by hand, or learned automatically through active automata learning. By means of model checking the behavior can be verified automatically against requirements expressing the desired behavior of the system. Learning-based testing combines both active learning and model checking in order to automate the three aspects of testing. The improvements to learning-based testing and model checking have been implemented in the LearnLib and LTSmin. Both toolsets facilitate system engineers to more efficiently analyze system behavior.

We conclude this thesis by summarizing the developed methods, discussing how they improve the state-of-the-art and by highlighting potential future research directions.

8.1 Learning-Based Testing

8.1.1 Summary of the Contribution

In the first half of this thesis we contributed a large amount of novelties to the field of learning-based testing. In essence, we were interested in finding out how classical as well as modern Active Automata Learning (AAL) algorithms perform when conjoined with algorithms for monitoring, model checking, Learning-Based

Testing (LBT) strategies and automated alphabet partitioning. We implemented a meticulously designed API in the LearnLib and benchmarked the algorithms on the Rigorous Examination for Reactive Systems (RERS) 2017 challenge.

The first result of the performance is given in Figure 3.6. Here we show that the ADT AAL algorithm in combination with the `DisproveFirstOracle` strategy for LBT works best for model checking liveness properties (expressing something good eventually happens). The graph shows that this specific combination requires the least amount of learning queries to disprove the largest amount of Linear-time Temporal Logic (LTL) properties. To achieve this result we developed a new method for querying a running system. We introduced so called ω -queries and related ω -membership oracles for answering ω -queries. These concepts can be used to decide whether infinite lasso-shaped counterexamples from model checkers are executable on the system that is analyzed. To be able to test whether the system enters a loop, we extended the SUL interface with a method for obtaining the current state of the system. The ω -membership oracle queries states of the system to perform state equivalence checks, to see whether the system entered a loop. This method is *sound* in the sense that every LTL formula that is falsified is indeed falsifiable. This is an improvement over the state-of-the-art, where either a hard to guess upper-bound on the number of states in the system had to be assumed, or a finite number of unrolls of the loop of the lasso had to be tested on the system. In the latter case only a warning could be given in case a finite prefix of the lasso is executable on the system.

Figure 3.7 shows the result of our novel contribution based on monitoring. Monitoring differs from model checking: monitors can only be used to analyze the safety part of LTL properties (expressing something bad never happens). Similar to the previous graph, we found that the ADT algorithm in combination with the `DisproveFirstOracle` works best. To achieve this result, we extended the LearnLib and LTSmin with monitoring capabilities. Soundness for monitoring is less involved than for model checking, because counterexamples for safety properties are finite. Finite words can simply be executed on a running system by regular membership oracles that were already available in the LearnLib.

In Figure 3.11 we show that monitoring and model checking can also be combined, to further improve performance. The combination of strategies involves first analyzing the safety part of the property by monitoring. If monitoring can not be used to falsify the property, then model checking is applied that also investigates the liveness part of the property. We again concluded that the ADT learning algorithm as well as the `DisproveFirstOracle` strategy works best.

Having established which AAL algorithm and LBT strategy are optimal for the RERS challenge, we investigated whether our contribution to LBT could be further extended with partitioning the alphabet over which is learned. The result of

the experiments we ran is given in Figures 4.5 and 4.6. For a small alphabet (5 symbols) the improvement in terms of queries posed on the system is small: the least amount of queries necessary to disprove certain LTL formulae fluctuates between starting out with the coarsest and finest partitioning. For a larger alphabet (15 symbols) the improvement is more significant: starting out with the coarsest partitioning of the alphabet, the ADT algorithm performs quite well, if not the best. To achieve this result we had to implement automated partition refinement based on so called controllable non-determinism that is compatible with LBT. As a result of this implementation, counterexamples from model checkers can be used in three ways. Firstly, to refine the alphabet if it is too coarse. Secondly, to refine the hypothesis automaton that is learned. Thirdly, to be reported as violations to requirements of the system. The algorithm that we developed automatically decides between the first two ways by means of a so called splitter oracle. Finding and reporting violations by means of alphabet abstractions is compatible with the techniques for monitoring and model checking discussed previously.

In conclusion, we have provided the novel theories and algorithms necessary for efficient learning and analysis of system behavior. While using the class diagrams in Figures 3.4 and 4.2 as reference and Listings 3.1 to 3.3 as a step-by-step guide, testers can use the off-the-shelf LBT implementations in the LearnLib to test their own systems.

8.1.2 Practical Validation with an ASML Case Study

At the time of writing we are in the process of validating our LBT algorithms on a practical case study at ASML. A description of the case study performed can be found in Appendix A. The case study involves a LBT approach using the LearnLib and the LTSmin model checker, applied to a metrology system for semiconductor lithography. We plan to setup experiments to measure *test session times*. The time of a test session is the total runtime an LBT algorithm takes to analyze all requirements on a black-box system. By trying different learning algorithms and different initial partitions of the alphabet, we hope to find the configuration that runs the fastest for the case study.

The focus with the case study is inspired by [Mei18, Section 6]: “The Querying Power of Model Checkers”. In this article, Meinke states more research has to be done on how test session times are affected by applying a model checker on each hypothesis inferred from a System Under Learning (SUL). This question has to be researched in the context of different systems, learning algorithms, requirements modeling languages and model checkers. The techniques we presented in Chapters 3 and 4 certainly provide new means to address this question. In particular, the LearnLib provides eight implementations of learning algorithms: ADT [Fro15, Fro19], DHC [Mer+11], Discrimination Tree [How12], L* [Ang87], Kearns & Vazi-

rani [KV94], Maler & Pnueli [MP95], Rivest & Schapire [RS93], and TTT [IHS14b]. In Chapter 4, we presented an additional learning algorithm that can deal with alphabet abstractions. All these learning algorithms have been integrated with LTSmin’s requirements modeling language and monitoring capabilities. This is a novel contribution to the field of automated testing, comparable with testing techniques that LBTest provides with the NuSMV model checker.

For validation of the contributed LBT techniques, we measured test session times in number of queries posed on the SUL (e.g. Figure 3.7), whereas measurements in seconds, minutes and hours are interesting to the tester as well. We have shown that certain learning algorithms (e.g. ADT) outperform others (e.g. L*) in number of queries, where fewer queries is better. Applying monitoring and alphabet abstractions can reduce the number of queries as well. Clearly, reducing the number of queries posed on the SUL results in lower runtimes. However, counting the number of queries only, hides the possible runtime overhead measured in seconds imposed by the model checker.

Meinke [Mei18] notes that the possible runtime overhead is due to “conventional off-the-shelf model checkers are forced to completely re-analyze each model M_i many times, without any re-use of previous results”, which is “highly inefficient”. We believe this is not going to be a significant issue with our ongoing case study at ASML, for two reasons. First, we have broad experience with applying our LTSmin model checker to the Linear-time Temporal Logic (LTL) subcategories from the Model Checking Contest (MCC) [Amp+19]. Second, resetting the SUL (from the case study) generally takes several seconds. From experience in the MCC we know that LTSmin can analyze tens of LTL requirements on hypotheses with thousands of states in a fraction of a second. Even though LTSmin can also not reuse previous results, the overhead should be negligible, since model checking is quick anyway. Furthermore, we can setup the LBT approach with the LearnLib and LTSmin in such a way that model checking and resetting the SUL happens in parallel. If indeed the model checking finishes before the reset is completed, then there should be no runtime overhead. By continuing the case study we hope to get further insight into actual test session times and to what extent model checking imposes runtime overhead.

8.1.3 Outlook

Although we believe the presented augmentation to the LearnLib for testing already provides a valuable addition to the toolset of testers, there are important research directions that can be investigated further.

The LBT approach in Chapters 3 and 4 is based on analyzing Finite State Machines (FSMs). This means values of data parameters are inherently part of alphabet

symbols. The method for partitioning alphabet symbols in Chapter 4 can be used to treat these values equivalently, to speedup the testing process. It is relevant to investigate whether state-local abstractions [IHS13], instead of global alphabet abstractions speedup the testing process even further. In [Cas+18] the authors show how register automata can be learned. They benchmark for instance with the connection establishment phase of TCP, and Java’s implementation of a priority queue. In these systems, data flow and their relation with control flow play an important role: increasing sequence numbers for TCP packets and in-order removal of elements in the priority queue. We expect that our presented improvements to LBT with FSMs can also be extended to LBT with register automata. At the current time of writing this requires a better integration of the LearnLib with the RALib (that facilitates learning register automata). Analyzing requirements on learned register automata could also be augmented with Quantified Temporal Logic (QTL) [HP18]. QTL enables expressing the allowed flow of values for data parameters and hopefully also decreases the effort of finding counterexamples to hypotheses expressed as register automata.

Adaptive Model Checking (AMC) is another technique to augment LBT. AMC involves informing learning algorithms with behavioral knowledge of previous versions of a system. This means that when faults are resolved or features added, the learning algorithm only has to learn these updates of the system. Currently, there are no off-the-shelf implementations of AMC available in the LearnLib. It is already known how to incorporate AMC with observation table based learners, such as L* and Rivest & Schapire [GPY06]. However, how to combine AMC with modern discrimination tree based algorithms, such as TTT and ADT is still an open research question [HMP18]. When implementations of AMC are available (for modern learners in particular), testers can benefit from a reduced number of queries necessary to continuously test their systems.

To adopt LBT on a large industrial scale it is important to have seamless integration with industrial tool chains. For instance, LBT should be part of nightly builds of software systems. To this end, the community should develop compatible plugins for continuous build integration services like: TeamCity, Jenkins, Bamboo, Travis or AppVeyor. These plugins should automatically start LBT algorithms that test the nightly builds. The LBT tool’s output should then be analyzed by the plugins and presented as an overview report to the tester. Such an overview report may contain: the number of executed queries, traces for failing test cases and to what extent the behavior of the system has changed with respect to the previous nightly build. This form of regression testing enables manufacturers to produce higher quality software products with a lower time to market.

8.2 Symbolic Reachability Analysis

8.2.1 Summary of the Contribution

The second half of this thesis is dedicated to symbolic reachability analysis. This formal method is a specific form of model checking that generates state spaces using decision diagrams. By considering sets of states simultaneously, invariants and possible deadlocks can be checked efficiently, in particular for asynchronous systems. We examined read and write dependencies between variables and assignments in these systems' specifications, in order to reduce the runtime and memory footprint of reachability algorithms. In Chapter 6, we found that bandwidth reduction algorithms can be used to reduce “distances” between variables to optimize their order in decision diagrams. The improvements with read-write separation and bandwidth reduction was thereafter subjected to a case study with the ProB model checker.

Table 5.9 shows the results we obtained by implementing read-write separation in the LTSmin model checker. We found that the memory footprint of the reachability algorithm can be reduced in general, and the runtime in particular for LTSmin's slow language front-ends, such as mCRL2. We achieved this result by first extending the general notion of dependencies in [BP08] to reading and writing separately. We had to solve one key issue for write dependencies: dynamic addressing of variables. Dynamic addressing arises for instance with conditional assignments or array subscripting, e.g. $a[i] := 1$. In this assignment, static analysis can not reveal to what position in the array is written, since this depends on the value of i . Our introduced notion of write dependence marks all positions in array a as write dependent. At runtime (the exploration of the state space), the values of array positions that must be copied are marked with a special value in decision diagrams that encode transition relations. We devised a language independent NEXT-STATE function that incorporates the refined notion of dependencies. As input, the function takes read projected state vectors, and outputs their write projected successors, possibly containing the special value that indicates which values need to be copied. We implemented dependency matrices and the NEXT-STATE function for Promela, DVE and mCRL2 that enabled us to benchmark system specifications written in those languages. We have also shown a straightforward language called System Update Specification (SUS). Running examples written in SUS can be used by language developers as a guide to connect their own languages to LTSmin. Newly developed language front-ends for LTSmin can then also benefit from the improvements based on read-write separation, that we implemented in LTSmin's symbolic back-end.

Inspired by the low time complexity (some even subquadratic) of nodal ordering algorithms that reduce bandwidth on symmetric matrices, we wondered if those

algorithms could also reduce event span. Event span is a measure for maximum distance between variables in asymmetric dependency matrices. Lower event span usually correlates with fewer nodes in decision diagrams. In order to benchmark with nodal ordering algorithms we linked Boost and ViennaCL (that implement nodal ordering algorithms) to LTSmin. The results in Figures 6.4 to 6.9 show that Sloan’s algorithm is very competitive to FORCE for event span reduction. For the Promela language, Sloan’s algorithm for profile reduction is even better than FORCE, which is considered the current state-of-the-art. To achieve this result, we developed a method for symmetrizing and de-symmetrizing dependency matrices that are asymmetric. In [RS06], we found that making them symmetric can be done by adding the transpose of the dependency matrix. After obtaining a permutation for the symmetric dependency matrix, we retrieve its individual permutations for rows and columns. These new permutations are then used to optimize variable orders in decision diagrams used by the reachability algorithms. We also experimented with permuting total adjacency graphs of dependency matrices, and applying those permutations to the symbolic reachability algorithm in LTSmin. The result we obtained here is shown in Figure 6.11. The graph shows that Sloan’s nodal ordering algorithm run on the total graph is superior to all other variable ordering algorithms in LTSmin. Since we first published this work in [MP16] good results with Sloan’s algorithm were also confirmed with the GreatSPN model checker [Amp+18].

8.2.2 Validation

Languages supported by the ProB model checker can model high level system specifications, similar to the mCRL2 language. ProB’s core is implemented with Prolog to model check languages such as Event-B and CSP||B. Using Prolog comes at a disadvantage: computing successor states during reachability analysis can be quite slow. We used read-write separation and bandwidth reduction from Chapters 5 and 6 to speedup symbolic reachability analysis for ProB. The results we obtained can be found in Table 7.2: significant reductions in runtime were achieved by implementing the new NEXT-STATE function, as well as by using Sloan’s nodal ordering algorithm. We used OMQ as a message passing interface between ProB and LTSmin. Messages sent between both model checkers initially contain static information such as the state variables involved, as well as read and write dependency matrices. During reachability analysis (successor) states are exchanged in the form of Prolog terms between both model checkers. Analyzing, for instance railway interlockings [Jam+14] specified in CSP||B, becomes more tractable with the new integration. Based on the good results we obtained for ProB, we invite language developers to contribute more front-ends for LTSmin. This allows harnessing the optimizations we have built in LTSmin’s back-end for even more specification languages.

8.2.3 Outlook

We want to investigate whether a generalized notion of dynamic addressing can be established: we suspect that dynamic reads can further reduce the number of NEXT-STATE calls, similar to dynamic writes. We also want to improve the effectiveness of bandwidth reduction algorithms by investigating different forms of symmetrization of the dependency matrix, for example by multiplying it with its transpose. More recent algorithms for bandwidth reduction should be tried as well, after all, Sloan’s algorithm was developed already in 1989. An overview of modern algorithms for bandwidth reduction can be found in [BO15]. Due to LTSmin’s language independent architecture of its PINS interface, these optimizations become available for a large class of languages, such as PNML, DVE, Promela, mCRL2 and those supported by the ProB model checker.

Model checking competitions like RERS, MCC and SV-COMP [Bey19], provide excellent test beds for high-performance model checking. We believe that the aforementioned techniques can bring considerable improvements in future editions of the RERS and MCC competitions. In order to also apply these techniques to SV-COMP requires an adaptation of LTSmin’s PINS architecture, to support the dynamic nature of software programs. To this end, a recursive PINS interface provides support for variable length state vectors. These vectors can encode growing and shrinking program stacks and heaps. This also requires dependencies to be resolved dynamically for variable length state vectors. Our method for dealing with dynamic addressing can be used as a starting point to achieve this. Another direction in which PINS can be improved is by adding more symbolic engines. Specifically, SMT-based engines that can deal with data in system specifications efficiently. Model checkers with such engines can be used for instance to verify register automata that are obtained through active learning. This gives testers a more powerful method for analyzing industrial systems where data plays an important role.

ASML Case Study

ASML provides chip makers with products to manufacture a wide range of semiconductor chips. These products include hardware, software and services. The key product ASML provides is the semiconductor lithography system, which is a very complex high-tech system used to expose patterns on a thin slice of silicon, called the wafer. For this purpose, light is projected through a reticle on the wafer that is coated with a light-sensitive chemical. The light interacts with the material, which results in a pattern printed on the wafer. This procedure is repeated many times on a single wafer to create the chips.

To stay competitive, chips are becoming smaller and smaller. Therefore, the typical ASML lithography systems are combined with other high-tech systems and advanced software to compensate for physical limitations. ASML's product portfolio has been extended with a metrology system, which, for instance, allows measurement of overlay and critical dimension of structures printed on the wafer. Moreover, advanced software has been developed to calculate corrections based on the measurements, which are fed back to the lithography system. This closed loop solution is called 'Holistic Lithography' and is depicted in Figure A.1.

To qualify the full holistic lithography solution, testing is done on different levels, e.g. unit, component, system, cross-system and fab level. Whenever a system is upgraded with new software, it has to be qualified that the communication with the other systems is still working. We have addressed this problem in a case study by automatically inferring the communication between two ASML products. For regression testing we can use active learning to verify that the inferred model is exactly the same for the old and new software. Moreover, we can verify bad weather behavior, i.e. reaction to unexpected inputs, which typically is tested on a smaller scale in standard functional and non-functional tests.

We have applied an active learning algorithm to infer the communication between a YieldStar metrology system and the Litho InSight software running on the litho

Figure A.1: ASML holistic lithography: links the lithography system (1) to the metrology system (2) and the litho computing platform (3).

computing platform. We have focused on a special setup of the systems, where the Litho InSight software returns the calculated corrections to the metrology tool instead of the lithography system. The communication and interfaces of both systems are explicitly specified, serving as an agreement between those systems. The happy flow of the interaction between YieldStar and Litho InSight is depicted in a sequence diagram in Figure A.2. At startup of YieldStar the system retrieves the definitions for reporting. Once the metrology system has finished measuring the wafers, it notifies the Litho InSight software. Next, the YieldStar sends several documents to Litho InSight that are requested by the Litho InSight software to calculate corrections. Finally, Litho InSight notifies the metrology system that the corrections are available, which then are retrieved by the YieldStar.

In our learning setup we used the LearnLib with an active learning algorithm that was initialized with the messages sent by YieldStar. The Litho InSight software served as System Under Learning (SUL) to be inferred. Note that the ‘get document 1’, ‘get document 2’ and ‘send notification (corrections calculated)’ are preceded by ‘wait’ inputs with fixed timeouts to infer a Mealy machine. However, note that we did not infer the timing between two inputs or inputs and outputs. Furthermore, we only used valid data in our experiments, i.e. we did not verify how the Litho InSight software reacts if a valid input with invalid data is sent, e.g. returning document 1 with invalid data.

Figure A.2: Sequence diagram showing the happy flow of the communication between the YieldStar metrology system and the Litho InSight software.

Publications by the Author

Jeroen Meijer, Gijs Kant, Stefan Blom, and Jaco van de Pol. “Read, Write and Copy Dependencies for Symbolic Model Checking”. In: *Haifa Verification Conference*. Vol. 8855. Lecture Notes in Computer Science. Springer, 2014, pp. 204–219.

Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. “LTSmin: High-Performance Language-Independent Model Checking”. In: *TACAS*. vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 692–707.

Jens Bendisposto, Philipp Körner, Michael Leuschel, Jeroen Meijer, Jaco van de Pol, Helen Treharne, and Jorden Whitefield. “Symbolic Reachability Analysis of B Through ProB and LTSmin”. In: *IFM*. vol. 9681. Lecture Notes in Computer Science. Springer, 2016, pp. 275–291.

Jeroen Meijer and Jaco van de Pol. “Bandwidth and Wavefront Reduction for Static Variable Ordering in Symbolic Reachability Analysis”. In: *NFM*. vol. 9690. Lecture Notes in Computer Science. Springer, 2016, pp. 255–271.

Bernhard Steffen, Marc Jasper, Jeroen Meijer, and Jaco van de Pol. “Property-Preserving Generation of Tailored Benchmark Petri Nets”. In: *ACSD*. IEEE Computer Society, 2017, pp. 1–8.

Marc Jasper, Maximilian Fecke, Bernhard Steffen, Markus Schordan, Jeroen Meijer, Jaco van de Pol, Falk Howar, and Stephen F. Siegel. “The RERS 2017 challenge and workshop (invited paper)”. In: *SPIN*. ACM, 2017, pp. 11–20.

Jeroen Meijer and Jaco van de Pol. “Sound Black-Box Checking in the LearnLib”. In: *NFM*. vol. 10811. Lecture Notes in Computer Science. Springer, 2018, pp. 349–366.

Philipp Körner, Michael Leuschel, and Jeroen Meijer. “State-of-the-Art Model Checking for B and Event-B Using ProB and LTSmin”. In: *IFM*. vol. 11023. Lecture Notes in Computer Science. Springer, 2018, pp. 275–295.

David Huistra, Jeroen Meijer, and Jaco van de Pol. “Adaptive Learning for Learn-Based Regression Testing”. In: *FMICS*. vol. 11119. Lecture Notes in Computer Science. Springer, 2018, pp. 162–177.

Jaco van de Pol and Jeroen Meijer. “Synchronous or Alternating? - LTL Black-Box Checking of Mealy Machines by Combining the LearnLib and LTSmin”. In: *Models, Mindsets, Meta*. Vol. 11200. Lecture Notes in Computer Science. Springer, 2018, pp. 417–430.

Fabrice Kordon, Hubert Garavel, Lom-Messan Hillah, Emmanuel Paviot-Adet, Loïc Jezequel, Francis Hulin-Hubard, Elvio Gilberto Amparore, Marco Beccuti, Bernard Berthomieu, Hugues Evrard, Peter Gjøøl Jensen, Didier Le Botlan, Torsten Liebke, Jeroen Meijer, Jiří Srba, Yann Thierry-Mieg, Jaco van de Pol, and Karsten Wolf. “MCC’2017 - The Seventh Model Checking Contest”. In: *T. Petri Nets and Other Models of Concurrency* 13 (2018), pp. 181–209.

Tom van Dijk, Jeroen Meijer, and Jaco van de Pol. “Multi-core On-The-Fly Saturation”. In: *TACAS (2)*. Vol. 11428. Lecture Notes in Computer Science. Springer, 2019, pp. 58–75.

Elvio Gilberto Amparore, Bernard Berthomieu, Gianfranco Ciardo, Silvano Dal-Zilio, Francesco Gallà, Lom-Messan Hillah, Francis Hulin-Hubard, Peter Gjøøl Jensen, Loïc Jezequel, Fabrice Kordon, Didier Le Botlan, Torsten Liebke, Jeroen Meijer, Andrew S. Miner, Emmanuel Paviot-Adet, Jiří Srba, Yann Thierry-Mieg, Tom van Dijk, and Karsten Wolf. “Presentation of the 9th Edition of the Model Checking Contest”. In: *TACAS (3)*. Vol. 11429. Lecture Notes in Computer Science. Springer, 2019, pp. 50–68.

Jeroen Meijer and Jaco van de Pol. “Sound black-box checking in the LearnLib”. In: *Innovations in Systems and Software Engineering* 15.3 (Sept. 2019), pp. 267–287. ISSN: 1614-5054.

Bibliography

- [Aal11] Wil M. P. van der Aalst. *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011 (cit. on p. 8).
- [Aar+12] Fides Aarts et al. “Automata Learning through Counterexample Guided Abstraction Refinement”. In: *FM*. Vol. 7436. Lecture Notes in Computer Science. Springer, 2012, pp. 10–27 (cit. on p. 74).
- [Aar+14] Fides Aarts et al. “Algorithms for Inferring Register Automata - A Comparison of Existing Approaches”. In: *ISoLA (1)*. Vol. 8802. Lecture Notes in Computer Science. Springer, 2014, pp. 202–219 (cit. on p. 74).
- [Aar14] Fides Aarts. *Tomte: bridging the gap between active learning and real-world systems*. 2014 (cit. on pp. 11, 12, 74, 93).
- [ABD14] Elvio Gilberto Amparore, Marco Beccuti, and Susanna Donatelli. “(Stochastic) Model Checking in GreatSPN”. In: *Petri Nets*. Vol. 8489. Lecture Notes in Computer Science. Springer, 2014, pp. 354–363 (cit. on p. 155).
- [Abr05] Jean-Raymond Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 2005 (cit. on p. 158).
- [Abr10] J-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010 (cit. on p. 158).
- [AD18] George Argyros and Loris D’Antoni. “The Learnability of Symbolic Automata”. In: *CAV (1)*. Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 427–445 (cit. on p. 74).
- [AF16] Dana Angluin and Dana Fisman. “Learning regular omega languages”. In: *Theor. Comput. Sci.* 650 (2016), pp. 57–72 (cit. on pp. 54, 56, 66).
- [AKA19] Seher Acer, Enver Kayaasian, and Cevdet Aykanat. “A Hypergraph Partitioning Model for Profile Minimization”. In: *SIAM J. Sci. Comput.* 41 (2019), pp. 83–108 (cit. on p. 154).

- [Amp+17] Elvio Gilberto Amparore et al. “Decision Diagrams for Petri Nets: Which Variable Ordering?” In: *PNSE @ Petri Nets*. Vol. 1846. CEUR Workshop Proceedings. CEUR-WS.org, 2017, pp. 31–50 (cit. on pp. 14, 130, 154).
- [Amp+18] Elvio Gilberto Amparore et al. “Decision Diagrams for Petri Nets: A Comparison of Variable Ordering Algorithms”. In: *T. Petri Nets and Other Models of Concurrency* 13 (2018), pp. 73–92 (cit. on pp. 14, 130, 154, 155, 175).
- [Amp+19] Elvio Gilberto Amparore et al. “Presentation of the 9th Edition of the Model Checking Contest”. In: *TACAS (3)*. Vol. 11429. Lecture Notes in Computer Science. Springer, 2019, pp. 50–68 (cit. on pp. 15, 16, 93, 172, 182).
- [AMS03] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. “FORCE: a fast and easy-to-implement variable-ordering heuristic”. In: *ACM Great Lakes Symposium on VLSI*. ACM, 2003, pp. 116–119 (cit. on pp. 14, 130, 133, 147).
- [Ang87] Dana Angluin. “Learning Regular Sets from Queries and Counterexamples”. In: *Inf. Comput.* 75.2 (1987), pp. 87–106 (cit. on pp. 9, 38, 57, 71, 82, 89, 90, 171).
- [AV10] Fides Aarts and Frits W. Vaandrager. “Learning I/O Automata”. In: *CONCUR*. Vol. 6269. Lecture Notes in Computer Science. Springer, 2010, pp. 71–85 (cit. on p. 7).
- [Bai+16] Alexander Bainczyk et al. “ALEX: Mixed-Mode Learning of Web Applications at Ease”. In: *ISoLA (2)*. Vol. 9953. Lecture Notes in Computer Science. 2016, pp. 655–671 (cit. on p. 93).
- [Bar+19] Ezio Bartocci et al. “TOOLympics 2019: An Overview of Competitions in Formal Methods”. In: *TACAS (3)*. Vol. 11429. Lecture Notes in Computer Science. Springer, 2019, pp. 3–24 (cit. on p. 15).
- [BCL91] Jerry R. Burch, Edmund M. Clarke, and David E. Long. “Symbolic Model Checking with Partitioned Transition Relations”. In: *VLSI*. 1991, pp. 49–58 (cit. on pp. 12, 97, 154).
- [BDL04] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. “A Tutorial on Uppaal”. In: *SFM*. Vol. 3185. Lecture Notes in Computer Science. Springer, 2004, pp. 200–236 (cit. on p. 5).
- [Bel14] Axel Belinfante. “JTorX: exploring model-based testing”. PhD thesis. University of Twente, Enschede, Netherlands, 2014 (cit. on p. 24).

- [Ben+16a] Jens Bendisposto et al. “Symbolic Reachability Analysis of B Through ProB and LTSmin”. In: *IFM*. Vol. 9681. Lecture Notes in Computer Science. Springer, 2016, pp. 275–291 (cit. on pp. 19, 108, 109, 167, 181).
- [Ben+16b] Jens Bendisposto et al. “Symbolic Reachability Analysis of B through ProB and LTSmin”. In: *CoRR* abs/1603.04401 (2016) (cit. on p. 165).
- [Ben15] Jens Marco Bendisposto. “Directed and Distributed Model Checking of B-Specifications”. PhD thesis. Heinrich Heine University Düsseldorf, 2015 (cit. on pp. 158, 163).
- [Bey19] Dirk Beyer. “Automatic Verification of C and Java Programs: SV-COMP 2019”. In: *TACAS (3)*. Vol. 11429. Lecture Notes in Computer Science. Springer, 2019, pp. 133–155 (cit. on p. 176).
- [Bil+03] Jonathan Billington et al. “The Petri Net Markup Language: Concepts, Technology, and Tools”. In: *ICATPN*. Vol. 2679. Lecture Notes in Computer Science. Springer, 2003, pp. 483–505 (cit. on p. 5).
- [BJR08] Therese Berg, Bengt Jonsson, and Harald Raffelt. “Regular Inference for State Machines Using Domains with Equality Tests”. In: *FASE*. Vol. 4961. Lecture Notes in Computer Science. Springer, 2008, pp. 317–331 (cit. on p. 74).
- [BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008 (cit. on pp. 2, 35).
- [BL13] Freark van der Berg and Alfons Laarman. “SpinS: Extending LTSmin with Promela through SpinJa”. In: *Electr. Notes Theor. Comput. Sci.* 296 (2013), pp. 95–105 (cit. on p. 96).
- [BLL09] Dragan Bosnacki, Stefan Leue, and Alberto Lluch-Lafuente. “Partial-order reduction for general state exploring algorithms”. In: *STTT* 11.1 (2009), pp. 39–51 (cit. on p. 12).
- [BM10] Junaid Babar and Andrew S. Miner. “Meddly: Multi-terminal and Edge-Valued Decision Diagram Library”. In: *QEST*. IEEE Computer Society, 2010, pp. 195–196 (cit. on p. 155).
- [BO15] Júnior Assis Barreto Bernardes and Sanderson L. Gonzaga de Oliveira. “A Systematic Review of Heuristics for Profile Reduction of Symmetric Matrices”. In: *Procedia Computer Science* 51 (2015). International Conference On Computational Science, ICCS 2015, pp. 221–230. ISSN: 1877-0509 (cit. on pp. 154, 176).
- [Boe79] B. W. Boehm. “Guidelines for Verifying and Validating Software Requirements and Design Specifications”. In: *Euro IFIP 79*. Ed. by P. A. Samet. North Holland, 1979, pp. 711–719 (cit. on p. 2).

- [Bol+10] Benedikt Bollig et al. “libalf: The Automata Learning Framework”. In: *CAV*. Vol. 6174. Lecture Notes in Computer Science. Springer, 2010, pp. 360–364 (cit. on p. 55).
- [BP08] Stefan Blom and Jaco van de Pol. “Symbolic Reachability for Process Algebras with Recursive Data Types”. In: *ICTAC*. Vol. 5160. Lecture Notes in Computer Science. Springer, 2008, pp. 81–95 (cit. on pp. 14, 96, 119, 120, 133, 164, 174).
- [BP16] Vincent Bloemen and Jaco van de Pol. “Multi-core SCC-Based LTL Model Checking”. In: *Haifa Verification Conference*. Vol. 10028. Lecture Notes in Computer Science. 2016, pp. 18–33 (cit. on pp. 45, 48).
- [BPM83] Mordechai Ben-Ari, Amir Pnueli, and Zohar Manna. “The Temporal Logic of Branching Time”. In: *Acta Inf.* 20 (1983), pp. 207–226 (cit. on p. 6).
- [BPW09] Stefan Blom, Jaco van de Pol, and Michael Weber. *Bridging the Gap between Enumerative and Symbolic Model Checkers*. Technical Report TR-CTIT-09-30. University of Twente, Enschede: CTIT, 2009 (cit. on pp. 96, 104).
- [BPW10] Stefan Blom, Jaco van de Pol, and Michael Weber. “LTSmin: Distributed and Symbolic Reachability”. In: *CAV*. Vol. 6174. Lecture Notes in Computer Science. Springer, 2010, pp. 354–359 (cit. on p. 96).
- [Bry86] Randal E. Bryant. “Graph-Based Algorithms for Boolean Function Manipulation”. In: *IEEE Trans. Computers* 35.8 (1986), pp. 677–691 (cit. on pp. 13, 130).
- [Büc62] J. Richard Büchi. “On a decision method in restricted second order arithmetic”. In: *International Congress on Logic, Methodology and Philosophy of Science* (1962), pp. 1–11 (cit. on p. 11).
- [Bur+92] Jerry R. Burch et al. “Symbolic Model Checking: 10^{20} States and Beyond”. In: *Inf. Comput.* 98.2 (1992), pp. 142–170 (cit. on p. 95).
- [BW96] Beate Bollig and Ingo Wegener. “Improving the Variable Ordering of OBDDs Is NP-Complete”. In: *IEEE Trans. Computers* 45.9 (1996), pp. 993–1002 (cit. on pp. 130, 133).
- [Car+88] Mats Carlsson et al. *SICStus Prolog user’s manual*. Swedish Institute of Computer Science Kista, 1988 (cit. on p. 158).
- [Cas+18] Sofia Cassel et al. “Extending Automata Learning to Extended Finite State Machines”. In: *Machine Learning for Dynamic Software Analysis*. Vol. 11026. Lecture Notes in Computer Science. Springer, 2018, pp. 149–177 (cit. on pp. 74, 173).

- [Cas15] Sofia Cassel. “Learning Component Behavior from Tests : Theory and Algorithms for Automata with Data”. PhD thesis. Uppsala University, Sweden, 2015 (cit. on p. 74).
- [CE81] Edmund M. Clarke and E. Allen Emerson. “Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic”. In: *Logic of Programs*. Vol. 131. Lecture Notes in Computer Science. Springer, 1981, pp. 52–71 (cit. on pp. 3, 5).
- [CGP01] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001 (cit. on p. 5).
- [CHJ15] Sofia Cassel, Falk Howar, and Bengt Jonsson. “RALib: A LearnLib extension for inferring EFSMs”. In: *DIFTS 5* (2015) (cit. on p. 74).
- [Cim+00] Alessandro Cimatti et al. “NUSMV: A New Symbolic Model Checker”. In: *STTT 2.4* (2000), pp. 410–425 (cit. on pp. 96, 97).
- [Cim+02] Alessandro Cimatti et al. “NuSMV 2: An OpenSource Tool for Symbolic Model Checking”. In: *CAV*. Vol. 2404. Lecture Notes in Computer Science. Springer, 2002, pp. 359–364 (cit. on pp. 6, 55, 154).
- [Cla08] Edmund M. Clarke. “The Birth of Model Checking”. In: *25 Years of Model Checking*. Vol. 5000. Lecture Notes in Computer Science. Springer, 2008, pp. 1–26 (cit. on pp. 12, 95).
- [CM69] E. Cuthill and J. McKee. “Reducing the Bandwidth of Sparse Symmetric Matrices”. In: *Proceedings 24th National Conference*. ACM. 1969, pp. 157–172 (cit. on pp. 14, 131, 136).
- [CMS03] Gianfranco Ciardo, Robert M. Marmorstein, and Radu Siminiceanu. “Saturation Unbound”. In: *TACAS*. Vol. 2619. Lecture Notes in Computer Science. Springer, 2003, pp. 379–393 (cit. on p. 96).
- [CMS06] Gianfranco Ciardo, Robert M. Marmorstein, and Radu Siminiceanu. “The saturation algorithm for symbolic state-space exploration”. In: *STTT 8.1* (2006), pp. 4–25 (cit. on pp. 13, 96, 130).
- [CMW09] Gianfranco Ciardo, Andrew S. Miner, and Min Wan. “Advanced features in SMART: the stochastic model checking analyzer for reliability and timing”. In: *SIGMETRICS Performance Evaluation Review 36.4* (2009), pp. 58–63 (cit. on p. 155).
- [Cou+92] Costas Courcoubetis et al. “Memory-Efficient Algorithms for the Verification of Temporal Properties”. In: *Formal Methods in System Design 1.2/3* (1992), pp. 275–288 (cit. on p. 45).
- [Cou99] Jean-Michel Couvreur. “On-the-Fly Verification of Linear Temporal Logic”. In: *World Congress on Formal Methods*. Vol. 1708. Lecture Notes in Computer Science. Springer, 1999, pp. 253–271 (cit. on p. 45).

- [Cra+13] Sjoerd Cranen et al. “An Overview of the mCRL2 Toolset and Its Recent Advances”. In: *TACAS*. Vol. 7795. Lecture Notes in Computer Science. Springer, 2013, pp. 199–213 (cit. on pp. 5, 147).
- [CW14] Edmund M. Clarke and Qinsi Wang. “2⁵ Years of Model Checking”. In: *Ershov Memorial Conference*. Vol. 8974. Lecture Notes in Computer Science. Springer, 2014, pp. 26–40 (cit. on p. 1).
- [CW96] Edmund M. Clarke and Jeannette M. Wing. “Formal Methods: State of the Art and Future Directions”. In: *ACM Comput. Surv.* 28.4 (1996), pp. 626–643 (cit. on p. 3).
- [CY05] Gianfranco Ciardo and Andy Jinqing Yu. “Saturation-Based Symbolic Reachability Analysis Using Conjunctive and Disjunctive Partitioning”. In: *CHARME*. Vol. 3725. Lecture Notes in Computer Science. Springer, 2005, pp. 146–161 (cit. on pp. 97, 101).
- [DD17] Samuel Drews and Loris D’Antoni. “Learning Symbolic Automata”. In: *TACAS (1)*. Vol. 10205. Lecture Notes in Computer Science. 2017, pp. 173–189 (cit. on p. 74).
- [Dij16] Tom van Dijk. “Sylvan: multi-core decision diagrams”. PhD thesis. University of Twente, Enschede, Netherlands, 2016 (cit. on p. 96).
- [Dij70] E.W. Dijkstra. *Notes on structured programming*. 2nd ed. EUT report. WSK, Dept. of Mathematics and Computing Science. Technische Hogeschool Eindhoven, 1970 (cit. on pp. 7, 24).
- [DL14] Iyavlo Dobrikov and Michael Leuschel. “Optimising the ProB Model Checker for B Using Partial Order Reduction”. In: *SEFM*. Vol. 8702. Lecture Notes in Computer Science. Springer, 2014, pp. 220–234 (cit. on pp. 158, 167, 168).
- [DMP19] Tom van Dijk, Jeroen Meijer, and Jaco van de Pol. “Multi-core On-The-Fly Saturation”. In: *TACAS (2)*. Vol. 11428. Lecture Notes in Computer Science. Springer, 2019, pp. 58–75 (cit. on p. 182).
- [DP17] Tom van Dijk and Jaco van de Pol. “Sylvan: multi-core framework for decision diagrams”. In: *STTT* 19.6 (2017), pp. 675–696 (cit. on p. 96).
- [Dur+16] Alexandre Duret-Lutz et al. “Spot 2.0 - A Framework for LTL and ω -Automata Manipulation”. In: *ATVA*. Vol. 9938. Lecture Notes in Computer Science. 2016, pp. 122–129 (cit. on pp. 36, 45).
- [FH17] Paul Fiterau-Brosteau and Falk Howar. “Learning-Based Testing the Sliding Window Behavior of TCP Implementations”. In: *FMICS-AVoCS*. Vol. 10471. Lecture Notes in Computer Science. Springer, 2017, pp. 185–200 (cit. on pp. 74, 93).

- [Fit+17] Paul Fiterau-Brostean et al. “Model learning and model checking of SSH implementations”. In: *SPIN*. ACM, 2017, pp. 142–151 (cit. on p. 93).
- [FJV16] Paul Fiterau-Brostean, Ramon Janssen, and Frits W. Vaandrager. “Combining Model Learning and Model Checking to Analyze TCP Implementations”. In: *CAV (2)*. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016, pp. 454–471 (cit. on p. 93).
- [Fro15] Markus Frohme. “Active Automata Learning with Adaptive Distinguishing Sequences”. MA thesis. Technische Universität Dortmund, 2015 (cit. on pp. 10, 30, 55, 57, 71, 90, 171).
- [Fro19] Markus Theo Frohme. “Active Automata Learning with Adaptive Distinguishing Sequences”. In: *CoRR* abs/1902.01139 (2019) (cit. on pp. 57, 90, 171).
- [Fuj+91] Susumu Fujiwara et al. “Test Selection Based on Finite State Models”. In: *IEEE Trans. Software Eng.* 17.6 (1991), pp. 591–603 (cit. on p. 38).
- [Git19a] GitHub. *Contributors to LearnLib/learnlib*. 2019. URL: <https://github.com/LearnLib/learnlib/graphs/contributors> (visited on 04/12/2019) (cit. on p. 11).
- [Git19b] GitHub. *Contributors to utwente-fmt/ltsmin*. 2019. URL: <https://github.com/utwente-fmt/ltsmin/graphs/contributors> (visited on 04/12/2019) (cit. on p. 15).
- [GLM03] Orna Grumberg, Shlomi Livne, and Shaul Markovitch. “Learning to Order BDD Variables in Verification”. In: *J. Artif. Intell. Res.* 18 (2003), pp. 83–116 (cit. on p. 130).
- [GPS76] Norman E Gibbs, William G Poole Jr, and Paul K Stockmeyer. “An algorithm for reducing the bandwidth and profile of a sparse matrix”. In: *SIAM Journal on Numerical Analysis* 13.2 (1976), pp. 236–250 (cit. on p. 139).
- [GPY06] Alex Groce, Doron A. Peled, and Mihalis Yannakakis. “Adaptive Model Checking”. In: *Logic Journal of the IGPL* 14.5 (2006), pp. 729–744 (cit. on pp. 54, 56, 173).
- [Har15] Arnd Hartmanns. “On the analysis of stochastic timed systems”. PhD thesis. Saarland University, 2015 (cit. on p. 5).
- [Hin13] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. O’Reilly Media, Inc., 2013 (cit. on p. 163).
- [HL16] Dominik Hansen and Michael Leuschel. “Translating B to TLA+ for validation with TLC”. In: *Sci. Comput. Program.* 131 (2016), pp. 109–125 (cit. on pp. 158, 167).

- [HMP18] David Huistra, Jeroen Meijer, and Jaco van de Pol. “Adaptive Learning for Learn-Based Regression Testing”. In: *FMICS*. Vol. 11119. Lecture Notes in Computer Science. Springer, 2018, pp. 162–177 (cit. on pp. 54, 56, 173, 182).
- [HMU03] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2nd ed.)*. Addison-Wesley, 2003 (cit. on p. 26).
- [Hol04] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004 (cit. on pp. 6, 55).
- [How12] Falk Howar. “Active learning of interface programs”. PhD thesis. Dortmund University of Technology, 2012 (cit. on pp. 57, 90, 171).
- [HP18] Klaus Havelund and Doron Peled. “BDDs on the Run”. In: *ISoLA (4)*. Vol. 11247. Lecture Notes in Computer Science. Springer, 2018, pp. 58–69 (cit. on p. 173).
- [HRS13] Monika Heiner, Christian Rohr, and Martin Schwarick. “MARCIE - Model Checking and Reachability Analysis Done Efficiently”. In: *Petri Nets*. Vol. 7927. Lecture Notes in Computer Science. Springer, 2013, pp. 389–399 (cit. on p. 155).
- [HS18] Falk Howar and Bernhard Steffen. “Active Automata Learning in Practice - An Annotated Bibliography of the Years 2011 to 2016”. In: *Machine Learning for Dynamic Software Analysis*. Vol. 11026. Lecture Notes in Computer Science. Springer, 2018, pp. 123–148 (cit. on p. 55).
- [HSM11] Falk Howar, Bernhard Steffen, and Maik Merten. “Automata Learning with Automated Alphabet Abstraction Refinement”. In: *VMCAI*. Vol. 6538. Lecture Notes in Computer Science. Springer, 2011, pp. 263–277 (cit. on pp. 12, 70, 73, 74, 76, 83).
- [Hui+16] Marieke Huisman et al. “Software that Meets Its Intent”. In: *ISoLA (2)*. Vol. 9953. Lecture Notes in Computer Science. 2016, pp. 609–625 (cit. on p. 2).
- [IHS13] Malte Isberner, Falk Howar, and Bernhard Steffen. “Inferring Automata with State-Local Alphabet Abstractions”. In: *NASA Formal Methods*. Vol. 7871. Lecture Notes in Computer Science. Springer, 2013, pp. 124–138 (cit. on pp. 74, 93, 173).
- [IHS14a] Malte Isberner, Falk Howar, and Bernhard Steffen. “Learning register automata: from languages to program structures”. In: *Machine Learning* 96.1-2 (2014), pp. 65–98 (cit. on p. 74).

- [IHS14b] Malte Isberner, Falk Howar, and Bernhard Steffen. “The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning”. In: *RV*. Vol. 8734. Lecture Notes in Computer Science. Springer, 2014, pp. 307–322 (cit. on pp. 55, 57, 90, 172).
- [IHS15] Malte Isberner, Falk Howar, and Bernhard Steffen. “The Open-Source LearnLib - A Framework for Active Automata Learning”. In: *CAV (1)*. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 487–495 (cit. on pp. 41, 71, 73).
- [IS14] Malte Isberner and Bernhard Steffen. “An Abstract Framework for Counterexample Analysis in Active Automata Learning”. In: *ICGI*. Vol. 34. JMLR Workshop and Conference Proceedings. JMLR.org, 2014, pp. 79–93 (cit. on pp. 83, 89).
- [Isb15] Malte Isberner. “Foundations of active automata learning: an algorithmic perspective”. PhD thesis. Technical University Dortmund, Germany, 2015 (cit. on pp. 55, 74, 76).
- [Jam+13] Phillip James et al. “Verification of Scheme Plans Using CSP || B”. In: *SEFM Workshops*. Vol. 8368. Lecture Notes in Computer Science. Springer, 2013, pp. 189–204 (cit. on p. 157).
- [Jam+14] Phillip James et al. “Techniques for modelling and verifying railway interlockings”. In: *STTT* 16.6 (2014), pp. 685–711 (cit. on p. 175).
- [Jas+17] Marc Jasper et al. “The RERS 2017 challenge and workshop (invited paper)”. In: *SPIN*. ACM, 2017, pp. 11–20 (cit. on pp. 15, 16, 58, 73, 181).
- [Jas+19] Marc Jasper et al. “RERS 2019: Combining Synthesis with Real-World Models”. In: *TACAS (3)*. Vol. 11429. Lecture Notes in Computer Science. Springer, 2019, pp. 101–115 (cit. on p. 15).
- [Kan+15] Gijs Kant et al. “LTSmin: High-Performance Language-Independent Model Checking”. In: *TACAS*. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 692–707 (cit. on pp. 6, 36, 45, 48, 56, 96, 131, 181).
- [Kar+14] Konstantinos I. Karantasis et al. “Parallelization of Reordering Algorithms for Bandwidth and Wavefront Reduction”. In: *SC*. IEEE Computer Society, 2014, pp. 921–932 (cit. on p. 154).
- [Kav06] A. Kaveh. “Ordering for Optimal Patterns of Structural Matrices”. In: *Optimal Structural Analysis*. John Wiley & Sons, Ltd, 2006, pp. 191–271. ISBN: 9780470033326 (cit. on pp. 141, 154).

- [Kin70] Ian P. King. “An automatic reordering scheme for simultaneous equations derived from network systems”. In: *International Journal for Numerical Methods in Engineering* 2.4 (1970), pp. 523–533. ISSN: 1097-0207 (cit. on p. 139).
- [KLM18] Philipp Körner, Michael Leuschel, and Jeroen Meijer. “State-of-the-Art Model Checking for B and Event-B Using ProB and LTSmin”. In: *IFM*. Vol. 11023. Lecture Notes in Computer Science. Springer, 2018, pp. 275–295 (cit. on pp. 168, 182).
- [KMR17] Hojat Khosrowjerdi, Karl Meinke, and Andreas Rasmusson. “Learning-Based Testing for Safety Critical Automotive Applications”. In: *IMBSA*. Vol. 10437. Lecture Notes in Computer Science. Springer, 2017, pp. 197–211 (cit. on p. 56).
- [Kor+13] Fabrice Kordon et al. “Model Checking Contest @ Petri Nets, Report on the 2013 edition”. In: *CoRR* abs/1309.2485 (2013) (cit. on p. 125).
- [Kor+15] F. Kordon et al. *Complete Results for the 2015 Edition of the Model Checking Contest*. <http://mcc.lip6.fr/2015/results.php>. 2015. (Visited on 2015) (cit. on pp. 16, 147).
- [Kor+16] F. Kordon et al. *Complete Results for the 2016 Edition of the Model Checking Contest*. <http://mcc.lip6.fr/2016/results.php>. June 2016. (Visited on 2016) (cit. on p. 16).
- [Kor+17] F. Kordon et al. *Complete Results for the 2017 Edition of the Model Checking Contest*. <http://mcc.lip6.fr/2017/results.php>. June 2017. (Visited on 2017) (cit. on p. 16).
- [Kor+18a] F. Kordon et al. *Complete Results for the 2018 Edition of the Model Checking Contest*. <http://mcc.lip6.fr/2018/results.php>. June 2018. (Visited on 2018) (cit. on p. 16).
- [Kor+18b] Fabrice Kordon et al. “MCC’2017 - The Seventh Model Checking Contest”. In: *T. Petri Nets and Other Models of Concurrency* 13 (2018), pp. 181–209 (cit. on pp. 15, 16, 182).
- [Kor+19] F. Kordon et al. *Complete Results for the 2019 Edition of the Model Checking Contest*. <http://mcc.lip6.fr/2019/results.php>. Apr. 2019. (Visited on 2019) (cit. on p. 16).
- [KP97] Gary Kumfert and Alex Pothén. “Two improved algorithms for envelope and wavefront reduction”. In: *BIT Numerical Mathematics* 37.3 (Sept. 1997), pp. 559–590. ISSN: 1572-9125 (cit. on p. 137).
- [KV94] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994 (cit. on pp. 57, 90, 172).

- [Laa+11] Alfons Laarman et al. “Multi-core Nested Depth-First Search”. In: *ATVA*. Vol. 6996. Lecture Notes in Computer Science. Springer, 2011, pp. 321–335 (cit. on p. 45).
- [LB08] Michael Leuschel and Michael J. Butler. “ProB: an automated analysis toolset for the B method”. In: *STTT* 10.2 (2008), pp. 185–203 (cit. on pp. 147, 158, 162).
- [Li+19] Yong Li et al. “ROLL 1.0: ω -Regular Language Learning Library”. In: *TACAS (I)*. Vol. 11427. Lecture Notes in Computer Science. Springer, 2019, pp. 365–371 (cit. on p. 54).
- [LM10] Michael Leuschel and Thierry Massart. “Efficient Approximate Verification of B via Symmetry Markers”. In: *Annals of Mathematics and Artificial Intelligence* 59.1 (2010), pp. 81–106 (cit. on pp. 158, 167).
- [LY96] David Lee and Mihalis Yannakakis. “Principles and methods of testing finite state machines-a survey”. In: *Proceedings of the IEEE* 84.8 (1996), pp. 1090–1123 (cit. on p. 7).
- [Maf14] Liviu Octavian Maftciu-Scai. “The Bandwidths of a Matrix. A Survey of Algorithms”. In: *Annals of West University of Timisoara-Mathematics* 52.2 (2014), pp. 183–223 (cit. on p. 154).
- [McM93] Kenneth L. McMillan. *Symbolic model checking*. Kluwer, 1993 (cit. on p. 95).
- [Mei+14] Jeroen Meijer et al. “Read, Write and Copy Dependencies for Symbolic Model Checking”. In: *Haifa Verification Conference*. Vol. 8855. Lecture Notes in Computer Science. Springer, 2014, pp. 204–219 (cit. on pp. 14, 18, 108, 109, 130, 181).
- [Mei17] Karl Meinke. “Learning-Based Testing of Cyber-Physical Systems-of-Systems: A Platooning Study”. In: *EPEW*. Vol. 10497. Lecture Notes in Computer Science. Springer, 2017, pp. 135–151 (cit. on p. 56).
- [Mei18] Karl Meinke. “Learning-Based Testing: Recent Progress and Future Prospects”. In: *Machine Learning for Dynamic Software Analysis*. Vol. 11026. Lecture Notes in Computer Science. Springer, 2018, pp. 53–73 (cit. on pp. 1, 2, 9–12, 26, 74, 171, 172).
- [Men17] Irini-Eleftheria Mens. “Learning regular languages over large alphabets. (Apprentissage de langages réguliers sur des alphabets de grandes tailles)”. PhD thesis. Grenoble Alpes University, France, 2017 (cit. on p. 74).
- [Mer+11] Maik Merten et al. “Automata Learning with On-the-Fly Direct Hypothesis Construction”. In: *ISoLA Workshops*. Vol. 336. Communications in Computer and Information Science. Springer, 2011, pp. 248–260 (cit. on pp. 57, 90, 171).

- [MM15] Irini-Eleftheria Mens and Oded Maler. “Learning Regular Languages over Large Ordered Alphabets”. In: *Logical Methods in Computer Science* 11.3 (2015) (cit. on pp. 70, 74).
- [MM17] Oded Maler and Irini-Eleftheria Mens. “A Generic Algorithm for Learning Symbolic Automata from Membership Queries”. In: *Models, Algorithms, Logics and Tools*. Vol. 10460. Lecture Notes in Computer Science. Springer, 2017, pp. 146–169 (cit. on p. 74).
- [MP15] Jeroen Meijer and Jaco van de Pol. “Bandwidth and Wavefront Reduction for Static Variable Ordering in Symbolic Model Checking”. In: *CoRR* abs/1511.08678 (2015) (cit. on p. 14).
- [MP16] Jeroen Meijer and Jaco van de Pol. “Bandwidth and Wavefront Reduction for Static Variable Ordering in Symbolic Reachability Analysis”. In: *NFM*. Vol. 9690. Lecture Notes in Computer Science. Springer, 2016, pp. 255–271 (cit. on pp. 14, 18, 130, 154, 155, 175, 181).
- [MP18] Jeroen Meijer and Jaco van de Pol. “Sound Black-Box Checking in the LearnLib”. In: *NFM*. Vol. 10811. Lecture Notes in Computer Science. Springer, 2018, pp. 349–366 (cit. on pp. 11, 18, 31, 40, 41, 54, 55, 71, 181).
- [MP19] Jeroen Meijer and Jaco van de Pol. “Sound black-box checking in the LearnLib”. In: *Innovations in Systems and Software Engineering* 15.3 (Sept. 2019), pp. 267–287. ISSN: 1614-5054 (cit. on pp. 11, 18, 31, 55, 71, 182).
- [MP95] Oded Maler and Amir Pnueli. “On the Learnability of Infinitary Regular Sets”. In: *Inf. Comput.* 118.2 (1995), pp. 316–326 (cit. on pp. 54, 56, 57, 66, 90, 172).
- [MS11] Karl Meinke and Muddassar A. Sindhu. “Incremental Learning-Based Testing for Reactive Systems”. In: *TAP*. Vol. 6706. Lecture Notes in Computer Science. Springer, 2011, pp. 134–151 (cit. on pp. 30, 55, 56).
- [MS13] Karl Meinke and Muddassar A. Sindhu. “LBTest: A Learning-Based Testing Tool for Reactive Systems”. In: *ICST*. IEEE Computer Society, 2013, pp. 447–454 (cit. on p. 54).
- [Ner58] A. Nerode. “Linear Automaton Transformations”. In: *Proceedings of the American Mathematical Society* 9.4 (1958), pp. 541–544 (cit. on p. 26).
- [Noa99] Andreas Noack. “A ZBDD package for efficient model checking of Petri nets”. In: *Forschungsbericht, Brandenburgische Technische Universität Cottbus* (1999) (cit. on pp. 130, 133, 147).

- [Pel07] Radek Pelánek. “BEEM: Benchmarks for Explicit Model Checkers”. In: *SPIN*. Vol. 4595. Lecture Notes in Computer Science. Springer, 2007, pp. 263–267 (cit. on pp. 123, 147).
- [PL10] Daniel Plagge and Michael Leuschel. “Seven at a stroke: LTL model checking for High-level Specifications in B, Z, CSP, and more”. In: *STTT* 11 (2010), pp. 9–21 (cit. on p. 158).
- [PM18] Jaco van de Pol and Jeroen Meijer. “Synchronous or Alternating? - LTL Black-Box Checking of Mealy Machines by Combining the LearnLib and LTSmin”. In: *Models, Mindsets, Meta*. Vol. 11200. Lecture Notes in Computer Science. Springer, 2018, pp. 417–430 (cit. on pp. 39, 56, 182).
- [Pnu77] Amir Pnueli. “The Temporal Logic of Programs”. In: *FOCS*. IEEE Computer Society, 1977, pp. 46–57 (cit. on p. 6).
- [Pos19] The Washington Post. *With Ethiopian crash, a second new Boeing 737 MAX 8 jet goes down*. 2019. URL: <https://www.washingtonpost.com/business/2019/03/10/with-ethiopian-airlines-crash-another-new-boeing-max-jet-goes-down/> (visited on 03/10/2019) (cit. on p. 1).
- [PVY02] Doron A. Peled, Moshe Y. Vardi, and Mihalis Yannakakis. “Black Box Checking”. In: *Journal of Automata, Languages and Combinatorics* 7.2 (2002), pp. 225–246 (cit. on pp. 9–11, 30, 55).
- [QS82] Jean-Pierre Queille and Joseph Sifakis. “Specification and verification of concurrent systems in CESAR”. In: *Symposium on Programming*. Vol. 137. Lecture Notes in Computer Science. Springer, 1982, pp. 337–351 (cit. on p. 5).
- [Raf+09] Harald Raffelt et al. “Dynamic testing via automata learning”. In: *STTT* 11.4 (2009), pp. 307–324 (cit. on pp. 54, 55).
- [RBC17] Thiago Nascimento Rodrigues, Maria Claudia Silva Boeres, and Lucia Catabriga. “A Parallel Sloan Algorithm for the Profile Minimization Problem of Sparse Matrices”. In: *49th Brazilian Symposium of Operational Research (SBPO)*. Blumenau, Brazil: Brazilian Society of Operational Research - SOBRAPO, Aug. 2017, pp. 4079–4090 (cit. on p. 154).
- [RCP95] Oriol Roig, Jordi Cortadella, and Enric Pastor. “Verification of Asynchronous Circuits by BDD-based Model Checking of Petri Nets”. In: *Application and Theory of Petri Nets*. Vol. 935. Lecture Notes in Computer Science. Springer, 1995, pp. 374–391 (cit. on p. 165).

- [RK08] Michael Rice and Sanjay Kulhari. “A survey of static variable ordering heuristics for efficient BDD/MDD construction”. In: *University of California, Tech. Rep* (2008) (cit. on p. 130).
- [RS06] J. K. Reid and Jennifer A. Scott. “Reducing the Total Bandwidth of a Sparse Unsymmetric Matrix”. In: *SIAM J. Matrix Analysis Applications* 28.3 (2006), pp. 805–821 (cit. on pp. 131, 140, 154, 175).
- [RS93] Ronald L. Rivest and Robert E. Schapire. “Inference of Finite Automata Using Homing Sequences”. In: *Inf. Comput.* 103.2 (1993), pp. 299–347 (cit. on pp. 57, 90, 172).
- [Rud93] Richard Rudell. “Dynamic variable ordering for ordered binary decision diagrams”. In: *ICCAD*. IEEE, 1993, pp. 42–47 (cit. on p. 130).
- [Rup+16] Karl Rupp et al. “ViennaCL - Linear Algebra Library for Multi- and Many-Core Architectures”. In: *SIAM J. Scientific Computing* 38.5 (2016) (cit. on p. 131).
- [SC06] Radu Siminiceanu and Gianfranco Ciardo. “New Metrics for Static Variable Ordering in Decision Diagrams”. In: *TACAS*. Vol. 3920. Lecture Notes in Computer Science. Springer, 2006, pp. 90–104 (cit. on pp. 130, 135, 136).
- [SG09] Muzammil Shahbaz and Roland Groz. “Inferring Mealy Machines”. In: *FM*. Vol. 5850. Lecture Notes in Computer Science. Springer, 2009, pp. 207–222 (cit. on pp. 44, 71).
- [SHI12] Bernhard Steffen, Falk Howar, and Malte Isberner. “Active Automata Learning: From DFAs to Interface Programs and Beyond”. In: *ICGI*. Vol. 21. JMLR Proceedings. JMLR.org, 2012, pp. 195–209 (cit. on p. 74).
- [SHM11] Bernhard Steffen, Falk Howar, and Maik Merten. “Introduction to Active Automata Learning from a Practical Perspective”. In: *SFM*. Vol. 6659. Lecture Notes in Computer Science. Springer, 2011, pp. 256–296 (cit. on pp. 7, 26, 38).
- [Sij+11] Marten Sijtema et al. “Experiences with Formal Engineering: Model-Based Specification, Implementation and Testing of a Software Bus at Neopost”. In: *FMICS*. Vol. 6959. Lecture Notes in Computer Science. Springer, 2011, pp. 117–133 (cit. on p. 5).
- [Sin13] Muddassar Azam Sindhu. “Algorithms and Tools for Learning-based Testing of Reactive Systems”. PhD thesis. KTH Royal Institute of Technology, Schweden, 2013 (cit. on p. 56).

- [Slo89] S. W. Sloan. “A FORTRAN program for profile and wavefront reduction”. In: *International Journal for Numerical Methods in Engineering* 28.11 (1989), pp. 2651–2679. ISSN: 1097-0207 (cit. on pp. 131, 136, 137, 139).
- [ST05] Steve Schneider and Helen Treharne. “CSP theorems for communicating B machines”. In: *Formal Asp. Comput.* 17.4 (2005), pp. 390–422 (cit. on p. 157).
- [Ste+14] Bernhard Steffen et al. “Property-driven benchmark generation: synthesizing programs of realistic structure”. In: *STTT* 16.5 (2014), pp. 465–479 (cit. on p. 58).
- [Ste+17] Bernhard Steffen et al. “Property-Preserving Generation of Tailored Benchmark Petri Nets”. In: *ACSD*. IEEE Computer Society, 2017, pp. 1–8 (cit. on pp. 16, 58, 181).
- [SVP19] SVPAlib. *Git repository*. 2019. URL: <https://github.com/lorisdanto/symbolicautomata/tree/master/SVPAlib> (visited on 04/15/2019) (cit. on p. 74).
- [Tar72] Robert Endre Tarjan. “Depth-First Search and Linear Graph Algorithms”. In: *SIAM J. Comput.* 1.2 (1972), pp. 146–160 (cit. on p. 45).
- [TBS11] Mark Timmer, Ed Brinksma, and Mariëlle Stoelinga. “Model-Based Testing”. In: *Software and Systems Safety - Specification and Verification*. Vol. 30. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2011, pp. 1–32 (cit. on p. 6).
- [Thi15] Yann Thierry-Mieg. “Symbolic Model-Checking Using ITS-Tools”. In: *TACAS*. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 231–237 (cit. on p. 155).
- [Tim18] The New York Times. *Indonesia Plane Crash Adds to Country’s Troubling Safety Record*. 2018. URL: <https://www.nytimes.com/2018/10/28/world/asia/indonesia-lion-air-plane-crash.html> (visited on 03/12/2019) (cit. on p. 1).
- [Tim19] The Economic Times. *Boeing shares plunge 10% after second 737 MAX crash*. 2019. URL: <https://economictimes.indiatimes.com/markets/stocks/news/boeing-shares-dive-after-second-deadly-737-max-8-crash/articleshow/68356316.cms> (visited on 03/12/2019) (cit. on p. 1).
- [Tre08] Jan Tretmans. “Model Based Testing with Labelled Transition Systems”. In: *Formal Methods and Testing*. Vol. 4949. Lecture Notes in Computer Science. Springer, 2008, pp. 1–38 (cit. on p. 7).

- [Tre11] Jan Tretmans. “Model-Based Testing and Some Steps towards Test-Based Modelling”. In: *SFM*. Vol. 6659. Lecture Notes in Computer Science. Springer, 2011, pp. 297–326 (cit. on p. 25).
- [TRV12] Deian Tabakov, Kristin Y. Rozier, and Moshe Y. Vardi. “Optimized temporal monitors for SystemC”. In: *Formal Methods in System Design* 41.3 (2012), pp. 236–268 (cit. on p. 11).
- [TV10] Deian Tabakov and Moshe Y. Vardi. “Optimized temporal monitors for SystemC”. In: *RV*. Vol. 6418. Lecture Notes in Computer Science. Springer, 2010, pp. 436–451 (cit. on p. 54).
- [Vaa17] Frits W. Vaandrager. “Model learning”. In: *Commun. ACM* 60.2 (2017), pp. 86–95 (cit. on p. 8).
- [Val96] Antti Valmari. “The State Explosion Problem”. In: *Petri Nets*. Vol. 1491. Lecture Notes in Computer Science. Springer, 1996, pp. 429–528 (cit. on p. 12).
- [Var01] Moshe Y. Vardi. “Branching vs. Linear Time: Final Showdown”. In: *TACAS*. Vol. 2031. Lecture Notes in Computer Science. Springer, 2001, pp. 1–22 (cit. on p. 6).
- [Whi00] J. A. Whittaker. “What is software testing? And why is it so hard?”. In: *IEEE Software* 17.1 (Jan. 2000), pp. 70–79 (cit. on pp. 2, 21).
- [Whi15] Jordan Whitefield. *Linking ProB and LTSmin*. Final Year Dissertation, University of Surrey. 2015 (cit. on p. 158).
- [YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. “Model Checking TLA+ Specifications”. In: *CHARME*. Vol. 1703. Lecture Notes in Computer Science. Springer, 1999, pp. 54–66 (cit. on p. 167).

In dit proefschrift presenteren we technieken voor efficiënter leren en analyseren van systeemgedrag. Het eerste deel behandelt nieuwe algoritmes en gereedschap voor het testen van systemen gebaseerd op actief leren van automaten en Lineaire-tijd Temporele Logica (LTL) model-checken, ook wel Leren-Gebaseerd Testen (LGT) genoemd. Vervolgens geven we een verbeterd leeralgoritme dat in staat is om met grote alfabetten om te gaan. Deze komen vaak voor in grootschalige industriële systemen waarbij invoersymbolen gegevensparameters bevatten. In het tweede deel bespreken we verbeteringen voor het analyseren van formele systeem-specificaties. We beginnen met het kijken naar gescheiden lees-, schrijf- en kopieer-afhankelijkheden voor symbolisch model-checken om de verificatie van deze specificaties te versnellen. Hierna laten we zien dat bandbreedte-reductietechnieken, die oorspronkelijk ontworpen zijn voor ijle-matrixoplossers, zeer geschikt zijn om het geheugengebruik van symbolische toestandsruimtes van specificaties te reduceren. Implementaties van de gepresenteerde algoritmes worden onderworpen aan casussen en rigoureuze experimenten met wetenschappelijke softwarewedstrijden. Mogelijke toekomstige verbeteringen voor deze algoritmes worden eveneens besproken.

Deel 1: Leren-Gebaseerd Testen

De eerste bijdrage betreft een ontwerp en implementatie voor LGT in de LearnLib, die grondig gedocumenteerd is en vrij beschikbaar is gemaakt in het publieke domein. De LearnLib is een bibliotheek die gebruikt kan worden om automatisch het gedrag van systemen af te leiden door middel van actieve automaat-leeralgoritmes. Door onze implementaties toe te passen op de 'Rigorous Examination of Reactive Systems (RERS)' verificatiewedstrijd laten we zien hoe goed leeralgoritmes zoals L^* in de LearnLib presteren in de context van LGT, hetgeen voorheen onbekend was. Ook wordt onderzocht of zowel het veiligheids- als het levendigheidaspect van LTL-eigenschappen afzonderlijk gebruikt kunnen worden. We laten enerzijds zien dat het bevestigen van tegenvoorbeelden voor veiligheidseigenschappen op een lopend systeem eenvoudiger is dan het bevestigen van lassovormige tegenvoorbeel-

den voor levendigheidseigenschappen. Aan de andere kant lijkt het weerleggen van levendigheidseigenschappen informatiever voor de leeralgoritmes dan het weerleggen van veiligheidseigenschappen.

De tweede belangrijke bijdrage aan LGT is een nieuw leeralgoritme dat reguliere leeralgoritmes voor eindigetoestandsautomaten uitbreidt met automatische verfijning van gepartitioneerde alfabetten. Het belangrijkste kenmerk van ons algoritme is een procedure die controleert of het gepartitioneerde alfabet of de geleerde hypothese-automaat moet worden verfijnd. De beslissing van de procedure is gebaseerd op het oplossen van non-determinisme dat kan ontstaan als de partitionering van het alfabet te grof is. We onderwerpen het nieuwe algoritme opnieuw aan de RERS-wedstrijd, om softwaretesters inzicht te geven in de prestaties van de leeralgoritmes die beschikbaar zijn in de LearnLib: het ADT-algoritme lijkt in het algemeen een goede keuze te zijn.

Deel 2: Symbolisch Model-Checken

De eerste bijdrage aan symbolisch model-checken maakt gebruik van de gescheiden lees-, schrijf- en kopieerafhankelijkheden tussen variabelen en expressies over deze variabelen. Door zorgvuldig te kijken welke variabelen gelezen, geschreven of gekopieerd worden in toewijzingen, kunnen we het aantal VOLGENDE-TOESTAND functieoproepen die nodig zijn om de toestandruimte van een formele systeemspecificatie te berekenen, verminderen. De nieuwe VOLGENDE-TOESTAND functie, een interface tussen back-end die de toestandruimte opslaat, en een taalfront-end, toont een gereduceerde rekentijd voor het analyseren van specificaties geschreven in de mCRL2 taal in het bijzonder. We tonen ook aan dat het analyseren van Petri-netten uit de softwarecompetitie genaamd ‘Model Checking Contest (MCC)’, met onze verbeteringen beter hanteerbaar wordt.

De tweede bijdrage aan symbolisch model-checken, gebaseerd op bandbreedte-reductie, berust op het symmetriseren van de matrix die de afhankelijkheidsrelatie tussen variabelen en toewijzingen in systeemspecificaties codeert. We bespreken verschillende manieren om een symmetrische afhankelijkheidsmatrix tot stand te brengen, evenals een methode om de matrix te de-symmetriseren. Deze laatste methode is nodig om voor zowel variabelen als toewijzingen afzonderlijk geoptimaliseerde ordes te verkrijgen. Decennia oude bandbreedte-reductiealgoritmes kunnen worden uitgevoerd op symmetrische afhankelijkheidsmatrices, inclusief Cuthill-McKee’s en Sloan’s algoritmes. Door te experimenteren met de MCC, constateren we dat het uitvoeren van Sloan’s algoritme meestal de beste resultaten oplevert; tenminste op hetzelfde niveau als het FORCE-algoritme, dat de huidige stand van de techniek is.

De verbeteringen zijn geïmplementeerd in de LTSmin model-checker en vrij be-

schikbaar gesteld in het publieke domein. Door de koppeling van LTSmin en de ProB model-checker laten we zien dat de specificaties, geschreven in de B-methode, nu efficiënter kunnen worden geverifieerd.

Titles in the IPA Dissertation Series since 2016

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming.* Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects.* Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence.* Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare.* Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance.* Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices.* Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization.* Faculty of

Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. *Run-time resource management for component-based systems.* Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. *Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified Proofs.* Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. *Modeling the Dynamics of Requirements Process Improvement.* Faculty of Mathematics and Computer Science, TU/e. 2016-12

F.M.J. van den Broek. *Mobile Communication Security.* Faculty of Science, Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning.* Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *Efficient Immutable Collections.* Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via*

Model Checking. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors.* Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT).* Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data.* Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities.* Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too.* Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutii. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod.* Faculty of

Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages.* Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences.* Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. *Active Model Learning for the Analysis of Network Protocols.* Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java Code.* Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic.* Faculty of Science,

Mathematics and Computer Science,
RU. 2018-06

A. Lele. *Response Modeling: Model Refinements for Timing Analysis of Runtime Scheduling in Real-time Streaming Systems.* Faculty of Mathematics and Computer Science, TU/e. 2018-07

N. Bezirgiannis. *Abstract Behavioral Specification: unifying modeling and programming.* Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. *Trajectory Analysis: Bridging Algorithms and Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2018-09

E.J.J. Ruijters. *Zen and the art of railway maintenance: Analysis and optimization of maintenance via fault trees and statistical model checking.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-10

F. Yang. *A Theory of Executability: with a Focus on the Expressivity of Process Calculi.* Faculty of Mathematics and Computer Science, TU/e. 2018-11

L. Swartjes. *Model-based design of baggage handling systems.* Faculty of Mechanical Engineering, TU/e. 2018-12

T.A.E. Ophelders. *Continuous Similarity Measures for Curves and Surfaces.* Faculty of Mathematics and Computer Science, TU/e. 2018-13

M. Talebi. *Scalable Performance Analysis of Wireless Sensor Network.* Faculty of Mathematics and Computer Science, TU/e. 2018-14

R. Kumar. *Truth or Dare: Quantitative security analysis using attack trees.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-15

M.M. Beller. *An Empirical Evaluation of Feedback-Driven Software Development.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2018-16

M. Mehr. *Faster Algorithms for Geometric Clustering and Competitive Facility-Location Problems.* Faculty of Mathematics and Computer Science, TU/e. 2018-17

M. Alizadeh. *Auditing of User Behavior: Identification, Analysis and Understanding of Deviations.* Faculty of Mathematics and Computer Science, TU/e. 2018-18

P.A. Inostroza Valdera. *Structuring Languages as Object-Oriented Libraries.* Faculty of Science, UvA. 2018-19

M. Gerhold. *Choice and Chance - Model-Based Testing of Stochastic Behaviour.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-20

A. Serrano Mena. *Type Error Customization for Embedded Domain-*

Specific Languages. Faculty of Science, UU. 2018-21

S.M.J. de Putter. *Verification of Concurrent Systems in a Model-Driven Engineering Workflow.* Faculty of Mathematics and Computer Science, TU/e. 2019-01

S.M. Thaler. *Automation for Information Security using Machine Learning.* Faculty of Mathematics and Computer Science, TU/e. 2019-02

Ö. Babur. *Model Analytics and Management.* Faculty of Mathematics and Computer Science, TU/e. 2019-03

A. Afroozeh and A. Izmaylova. *Practical General Top-down Parsers.* Faculty of Science, UvA. 2019-04

S. Kisfaludi-Bak. *ETH-Tight Algorithms for Geometric Network Problems.* Faculty of Mathematics and Computer Science, TU/e. 2019-05

J. Moerman. *Nominal Techniques and Black Box Testing for Automata Learning.* Faculty of Science, Mathematics and Computer Science, RU. 2019-06

V. Bloemen. *Strong Connectivity and Shortest Paths for Checking Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-07

T.H.A. Castermans. *Algorithms for Visualization in Digital Humanities.* Faculty of Mathematics and Computer Science, TU/e. 2019-08

W.M. Sonke. *Algorithms for River Network Analysis.* Faculty of Mathematics and Computer Science, TU/e. 2019-09

J.J.G. Meijer. *Efficient Learning and Analysis of System Behavior.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-10

In this thesis, we present techniques for more efficient learning and analysis of system behavior.

The first part covers novel algorithms and tooling for testing systems based on active automata learning and Linear-time Temporal Logic (LTL) model checking, also called Learning-Based Testing (LBT). Next, we provide an improved learning algorithm that is able to deal with huge alphabets. These are commonly seen in large-scale industrial systems where input symbols contain data parameters.

In the second part we discuss improvements for analyzing formal system specifications. We start out by looking at separated read, write and copy dependencies for symbolic model checking to speed up the verification of these specifications. Then, we show that bandwidth reduction techniques, originally designed for sparse matrix solvers, are very capable at reducing the memory footprint of the specifications' symbolic state space. Implementations of the presented algorithms are subjected to case studies and rigorous experimentation with scientific software competitions.