



## Efficient Implementation of Symbolic Controllers for Cyber-Physical Systems

Mahmoud Khaled Mohamed Mahmoud

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik  
der Technischen Universität München zur Erlangung des akademischen Grades eines

**Doktor-Ingenieurs (Dr.-Ing.)**

genehmigten Dissertation.

**Vorsitzender:**

Prof. Dr.-Ing. Werner Hemmert

**Prüfende der Dissertation:**

1. Prof. Dr.-Ing. Martin Buss
2. Prof. Dr. Majid Zamani

Die Dissertation wurde am 13.04.2021 bei der Technischen Universität München  
eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am  
10.08.2021 angenommen.



This dissertation is dedicated to my wife *Mona* and my sons *Youssef* and *Younes*.



# Acknowledgments

This dissertation is the result of the years of my doctoral research studies in the Department of Electrical and Computer Engineering at the Technical University of Munich (TUM), Germany. In this short note, I would like to take the opportunity and acknowledge those people who supported me to make the completion of this dissertation possible.

First and foremost, I would like to express my sincere gratitude to my advisor Prof. Dr. Majid Zamani for his consistent support, encouragement, and generous advice throughout these years. I would like to thank him for introducing me to this interesting topic and guiding me through the whole research work.

Besides, I would also like to extend my sincere gratitude to Prof. Dr.-Ing./Univ. Tokio Martin Buss for welcoming me to his research group in the Chair of Automatic Control Engineering at the Technical University of Munich, Germany, since July 2019. I would greatly appreciate all his generous help, support, and consideration during this time.

I would like to thank Prof. Dr. Murat Arcaak for inviting me to his research group during 2019 at the University of California, Berkeley, USA, and also to the fruitful discussions with him during several meetings. My thanks also go to his former Ph.D. student, Dr. Eric S. Kim, for the worthwhile discussions.

My deep thanks also go to the German Academic Exchange Service (DAAD), Germany, and the Cultural Affairs and Missions Sector (CAM), Ministry of Higher Education, Egypt, for supporting my Ph.D. studies through the German Egyptian Long-term Research Program (GERLS), and for enabling such a remarkable interdisciplinary research experience.

I also thank all of my colleagues at HyConSys Lab ([www.hyconsys.com](http://www.hyconsys.com)), and the co-authors of the publications I took part in, for the fruitful discussions and enjoyable times we shared during the last five years.

Finally, I would like to thank my parents, my brothers, and my sister for their enduring support which brought me into the position of writing this thesis, and their sympathy and understanding when I had hard times.

I also gratefully acknowledge the support I received from Intel, Amazon, NVIDIA, and Xilinx. Many of the case studies reported in the thesis were run on hardware resources provided through grants from them in the form of donated hardware platforms and/or access to their remote computing platforms.

This work was also supported in part by the H2020 ERC Starting Grant AutoCPS (grant agreement No. 804639).

*M. Khaled*  
Munich, April 2021



# Abstract

Cyber-Physical Systems (CPSs) are complex systems resulting from an intricate interaction of digital computational devices with physical systems. In safety-critical CPS, a failure or malfunction may result in death/injury to people, loss/damage to equipment/property, or environmental harm. Safety-Critical Control Software (SCCS) are the main cores of safety-critical CPSs. They interface with multiple sensors, perform control and planning tasks, and command the actuators to interact with the physical environment.

SCCS are currently designed using approaches that may result into unsafe CPSs. Their design requirements are not defined in a formal way which leaves a chance for ambiguity and false specifications. The design/development phases of SCCS involve many human factors which may result in faulty and buggy software. Additionally, traditional testing phases leave many edge-cases undetected. Since any failure in any life-critical SCCS can potentially cause death or injury to human lives, ensuring their correctness is very important and new design approaches need to be investigated.

Symbolic control is a promising approach for designing automatically correct-by-construction SCCS. Given models of CPS and formally-described design requirements, symbolic control techniques design algorithmically certifiable controllers that enforce the design requirements on CPS. Unfortunately, symbolic control suffers from major issues that hinder its application in today's CPS: (1) the algorithms of symbolic control are computationally complex which makes it limited to small-sized systems, (2) its current implementations can only deal with simple design requirements, and finally (3) it has no standard approach for extracting and implementing the designed controllers, which leaves such critical task to ad-hoc techniques that may ruin the obtained correctness guarantees.

In this thesis, we propose solutions to address the identified issues of symbolic control and make it applicable to real-world CPS. We introduce data-parallel algorithms, which, along with variable computing resources, allow controlling the computational complexity of symbolic control. We then introduce an approach for handling specifications given as automata on infinite strings allowing symbolic control to support much more practical design requirements. We finally discuss the types of controllers resulting from the algorithms introduced in the thesis, and introduce automated formal deployments of them. The results of the thesis introduce a practical end-to-end framework for symbolic control in the sense that, given high-level design requirements and models of systems, certifiable deployments of controllers are automatically generated.



# Zusammenfassung

Cyber-Physical Systems (CPS) sind komplexe Systeme, die aus einer komplexen Interaktion digitaler Rechengерäte mit physischen Systemen resultieren. Bei sicherheitskritischen CPS kann ein Ausfall oder eine Fehlfunktion zum/zur Tod/Verletzung von Personen, zum/zur Verlust/Beschädigung von Geräten oder zu Umweltschäden führen. Sicherheitskritische Steuerungssoftware (SCCS) sind die Gehirne der sicherheitskritischer CPS. Sie verbinden mehrere Sensoren, führen Steuerungs- und Planungsaufgaben aus und befehlen den Aktuatoren, mit der physischen Umgebung zu interagieren.

SCCS werden derzeit unter Verwendung von Ansätzen entwickelt, die zu unsicheren CPS führen. Ihre Entwurfsanforderungen sind nicht formal definiert, was zu Unklarheiten und falschen Angaben führen kann. Die Entwurfs- / Entwicklungsphasen von SCCS beinhalten viele menschliche Faktoren, was zu fehlerhafter und fehlerhafter Software führt. Darüber hinaus lassen herkömmliche Testphasen viele Randfälle unentdeckt. Da ein Ausfall eines lebenskritischen SCCS möglicherweise zum Tod oder zur Verletzung von Menschen führen kann, ist es nicht tolerierbar, sicherzustellen, dass diese korrekt sind, und es müssen neue Entwurfsansätze untersucht werden.

Die symbolische Steuerung ist ein vielversprechender Ansatz für das Entwerfen von SCCS mit automatischer Konstruktionskorrektur. Angesichts von CPS-Modellen und formal beschriebenen Entwurfsanforderungen entwerfen symbolische Steuerungsansätze algorithmisch zertifizierbare Steuerungen, die die Entwurfsanforderungen an CPS durchsetzen. Die Modelle werden verwendet, um Abstraktionen mit endlichen Zuständen zu konstruieren, die wichtige Merkmale ursprünglicher Systeme erfassen. Unter Verwendung von Ansätzen aus der Informatik wie "Fixed-Point Operations" und "Two-player Games" werden formal korrekte Controller algorithmisch entworfen, um die Entwurfsanforderungen durchzusetzen. Die symbolische Steuerung weist drei Hauptprobleme auf, die verhindern, dass sie zum Entwerfen des SCCS des heutigen CPS verwendet wird: (1) Die Algorithmen der symbolischen Steuerung sind rechnerisch komplex, wodurch sie auf kleine Systeme beschränkt ist, (2) ihre aktuellen Implementierungen können befassen sich nur mit einfachen Spezifikationen, und (3) es gibt keinen einheitlichen Standardansatz zum Extrahieren der entworfenen Controller und zum Generieren ihrer Bereitstellungen.

In dieser Doktorarbeit befassen wir uns mit den drei identifizierten Problemen der symbolischen Kontrolle, um sie in realen Anwendungen verwendbar zu machen. Wir führen datenparallele Algorithmen ein, die zusammen mit variablen Rechenressourcen die Steuerung der Rechenkomplexität der symbolischen Steuerung ermöglichen. Wir haben auch einen Ansatz der symbolischen Steuerung eingeführt, der Spezifikationen verarbeiten kann, die als Automaten für unendliche Zeichenfolgen angegeben werden, sodass die symbolische Steuerung wesentlich praktischere Entwurfsanforderungen erfüllen kann. Wir diskutieren schließlich die Arten von Controllern, die sich aus den in der

## *Zusammenfassung*

Doktorarbeit vorgestellten Algorithmen ergeben, und führen formale Bereitstellungen für sie ein. Die Ergebnisse der Arbeit führen zusammen ein praktisches "End-to-End Framework" für die symbolische Steuerung in dem Sinne ein, dass angesichts der praktischen Anforderungen und der Modelle von Systemen zertifizierbare Bereitstellungen von Steuerungen automatisch generiert werden.

# List of Publications

## Patents

1. **M. Khaled** and M. Zamani. Distributed Automated Synthesis Of Correct-by-construction Controllers. Patent Nr. EP3633468A1. *European Patent Office (EPO)*. April 2020.
2. **M. Khaled** and M. Zamani, Distributed Automated Synthesis Of Correct-by-construction Controllers. Patent Nr. WO2020070206. *World Intellectual Property Organization (WIPO)*. April 2020.

## Journal Papers

1. **M. Khaled**, K. Zhang, and M. Zamani. Output-Feedback Symbolic Control. *IEEE Transactions on Automatic Control*. Submitted for publication.
2. **M. Khaled** and M. Zamani, Cloud-Ready Acceleration of Formal Method Techniques for Cyber-Physical Systems. *IEEE Design & Test*. Oct. 2020.
3. M. Zamani, M. Mazo Jr, **M. Khaled**, and A. Abate. Symbolic models for networked control systems. *IEEE Transactions on Control of Network Systems*. Dec. 2018.

## Conference Papers

1. **M. Khaled** and M. Zamani. OmegaThreads: Symbolic Control from  $\omega$ -regular Specifications. *24th ACM International Conference on Hybrid Systems: Computation and Control (HSCC 2021)*. (Accepted, to appear).
2. A. Abate, H. Blom, N. Cauchi, J. Delicaris, A. Hartmanns, **M. Khaled**, A. Lavaei, C. Pilch, A. Remke, S. Schupp, F. Shmarov, S. Soudjani, A. Vinod, B. Wooding, M. Zamani, P. Zuliani. ARCH-COMP20 Category Report: Stochastic Models. *7th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH20)*. Sep. 2020.
3. A. Lavei<sup>1</sup>, **M. Khaled**<sup>1</sup>, S. Soudjani, M. Zamani. AMYTISS: Parallelized Automated Controller Synthesis for Large-Scale Stochastic Systems. *32nd Conference on Computer Aided Verification (CAV 2020)*. July 2020. (Acceptance rate: 27%).
4. A. Devonport<sup>1</sup>, **M. Khaled**<sup>1</sup>, M. Arcak, M. Zamani. PIRK: Scalable Interval Reachability Analysis for High-Dimensional Nonlinear Systems. *32nd Conference on Computer Aided Verification (CAV 2020)*. July 2020. (Acceptance rate: 27%).
5. **M. Khaled**, E. S. Kim, M. Arcak, M. Zamani. Synthesis of Symbolic Controllers: A Parallelized and Sparsity-Aware Approach. *25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. April 2019. (Acceptance rate: 31%).

---

<sup>1</sup>Both authors have contributed equally.

## List of Publications

6. **M. Khaled** and M. Zamani. pFaces: An Acceleration Ecosystem for Symbolic Control. *22nd ACM International Conference on Hybrid Systems: Computation and Control (HSCC 2019)*. April 2019.
7. **M. Khaled**, M. Rungger, and M. Zamani. SENSE: Abstraction-Based Synthesis of Networked Control Systems. *1st International Workshop on Methods and Tools for Rigorous System Design (MeTRiD 2018)*. April 2018.
8. **M. Khaled**, M. Rungger, M. Zamani. Symbolic models of networked control systems: A feedback refinement relation approach. *The 54th Annual Allerton Conference on Communication, Control, and Computing*. Sept. 2016.

## Posters

1. **M. Khaled**, M. Zamani. Poster: OmegaThreads: Symbolic Control from  $\omega$ -regular Specifications. *24th ACM International Conference on Hybrid Systems: Computation and Control (HSCC 2021)*. (Accepted, to appear).
2. A. Lavaei<sup>1</sup>, **M. Khaled**<sup>1</sup>, S. Soudjani, M. Zamani. AMYTISS: A Parallelized Tool on Automated Controller Synthesis for Large-Scale Stochastic Systems. *23rd ACM International Conference on Hybrid Systems: Computation and Control (HSCC 2020)*. April 2020.
3. **M. Khaled**, E. Kim, M. Arcak, M. Zamani. Synthesis of Symbolic Controllers: A Parallelized and Sparsity-Aware Approach. *European Joint Conferences on Theory and Practice of Software (ETAPS 2019)*. April 2019.
4. E. Kim, **M. Khaled**, M. Zamani, Major, Arcak, Murat. Major Computational Breakthroughs in the Synthesis of Symbolic Controllers via Decomposed Algorithms. *21st ACM International Conference on Hybrid Systems: Computation and Control (HSCC 2018)* April 2018.
5. **M. Khaled**, M. Rungger, M. Zamani, Symbolic Models of Networked Control Systems. *NET-CPS 2016: International Symposium on Networked Cyber-Physical Systems*. September 2016.

---

<sup>1</sup>Both authors have contributed equally.

# Contents

<b>Acknowledgments</b>	<b>v</b>
<b>Abstract</b>	<b>vii</b>
<b>Zusammenfassung</b>	<b>ix</b>
<b>List of Publications</b>	<b>xi</b>
<b>Contents</b>	<b>xiii</b>
<b>List of Figures</b>	<b>xvii</b>
<b>Acronyms</b>	<b>xix</b>
<b>1 Introduction</b>	<b>3</b>
1.1 Promising Design Approach for SCCS . . . . .	6
1.2 Contributions . . . . .	9
1.3 Thesis Organization . . . . .	10
<b>2 Preliminaries</b>	<b>13</b>
2.1 Notations . . . . .	13
2.2 Mathematical Framework for Systems . . . . .	14
2.2.1 Composition of Systems . . . . .	15
2.3 Symbolic Models . . . . .	15
2.3.1 Control Systems . . . . .	16
2.3.2 Control Systems as Systems . . . . .	17
2.3.3 Symbolic Models of Control Systems . . . . .	17
2.4 Symbolic Controller Synthesis . . . . .	18
2.4.1 Behaviors and Specifications . . . . .	18
2.4.2 Synthesis and Refinement of Symbolic Controllers . . . . .	20
2.5 Limitations of Current Symbolic Control Techniques . . . . .	21
2.5.1 The Curse of Dimensionality . . . . .	22
2.5.2 Impractical Specifications . . . . .	23
2.5.3 Ad-Hoc Deployments . . . . .	24
2.6 Summary . . . . .	24
<b>3 A Framework for Designing Efficient Algorithms of Symbolic Control</b>	<b>25</b>
3.1 High Performance Computing (HPC) . . . . .	25

## CONTENTS

3.2	Ecosystem for Parallel Computing . . . . .	26
3.3	Hardware Configuration (HWC)-Level and Compute Node (CN)-Level Accelerations . . . . .	28
3.3.1	Internal Design Structure . . . . .	29
3.3.2	Resource Management and Kernel Tuning . . . . .	30
3.3.3	Managing Computation and Memory Resources . . . . .	30
3.3.4	Modules for Supporting Kernel Development . . . . .	31
3.3.5	Supporting Symbolic Control Approaches . . . . .	31
3.4	Workflow of Kernels . . . . .	32
3.5	A Cloud-Ready Installation . . . . .	32
3.6	Summary . . . . .	33
<b>4</b>	<b>Efficient Algorithms for Symbolic Control</b>	<b>35</b>
4.1	Existing Implementations of Symbolic Control . . . . .	35
4.2	Data-Parallel Algorithms for Symbolic Control . . . . .	36
4.2.1	Data-Parallel Construction of Symbolic Models . . . . .	36
4.2.2	Data-Parallel Synthesis of Symbolic Controllers . . . . .	38
4.2.3	Memory-Efficient Kernels for Data-Parallel Symbolic Control . . . . .	39
4.2.4	Implementation Details . . . . .	39
4.2.5	Controlling Time Complexity of Symbolic Control Applications . . . . .	43
4.3	Data-Parallel Sparsity-Aware Algorithms for Symbolic Control . . . . .	45
4.3.1	Sparsity of Discrete-Time Systems . . . . .	46
4.3.2	Sparsity-Aware Distributed Constructions of Abstractions . . . . .	48
4.3.3	Sparsity-Aware Distributed Synthesis of Symbolic Controllers . . . . .	51
4.3.4	Sparsity-Aware Data-Parallelism for Symbolic Controller Synthesis . . . . .	55
4.3.5	Case Study: Autonomous Vehicle Avoiding Crash on Highway . . . . .	56
4.4	Summary . . . . .	57
<b>5</b>	<b>Efficient Algorithms for the Computation of Reachable Sets</b>	<b>59</b>
5.1	Approximations of Reachable Sets . . . . .	59
5.2	Interval Reachability Analysis . . . . .	60
5.2.1	Contraction/Growth Bound (GB) Method . . . . .	62
5.2.2	Continuous-Time Mixed-Monotonicity (CTMM) Method . . . . .	62
5.2.3	Monte Carlo (MC) Method . . . . .	62
5.3	Data-Parallel Algorithms for Computing Interval Reachable Sets . . . . .	63
5.3.1	Data-Parallel Runge-Kutta Scheme . . . . .	64
5.3.2	Parallelizing Interval Reachability Methods . . . . .	65
5.4	Case Studies . . . . .	66
5.4.1	Multi-Link Road Traffic Model . . . . .	67
5.4.2	Quadrotor Swarm . . . . .	69
5.4.3	Quadrotor Swarm with Artificial Potential Field . . . . .	69
5.4.4	Heat Diffusion . . . . .	71
5.4.5	Overtaking Maneuver on Highway . . . . .	72
5.4.6	Performance on ARCH Benchmarks . . . . .	73

5.5	Summary . . . . .	73
<b>6</b>	<b>Efficient Algorithms for Stochastic Symbolic Control</b>	<b>75</b>
6.1	Discrete-Time Stochastic Control Systems (dt-SCS) . . . . .	76
6.2	Markov Decision Processes (MDPs) as Symbolic Models . . . . .	77
6.3	Parallel Construction of Finite MDPs . . . . .	79
6.3.1	Data-Parallel Threads for Computing Transitions . . . . .	80
6.3.2	Less Memory for Post States in the Transitions . . . . .	80
6.3.3	Less Memory for Handling Disturbances of Dynamics . . . . .	81
6.3.4	Data-Parallel Algorithm for Constructing MDPs . . . . .	82
6.4	Data-Parallel Synthesis of Symbolic Controllers . . . . .	84
6.4.1	On-the-Fly Construction of Transitions . . . . .	86
6.4.2	Supporting Multiplicative Noises and Practical Distributions . . . . .	86
6.5	Illustrative Examples . . . . .	88
6.5.1	Synthesis of a Safety Controller . . . . .	88
6.5.2	Synthesis of a Reach-Avoid Controller . . . . .	90
6.6	Benchmarking and Case Studies . . . . .	90
6.6.1	Controlling Computational Complexities of Stochastic Applications . . . . .	90
6.6.2	Room Temperature Network . . . . .	93
6.6.3	Road Traffic Network . . . . .	94
6.6.4	Autonomous Vehicle . . . . .	95
6.6.5	Benchmarking Against Most Recent State-of-the-art Tool . . . . .	96
6.7	Summary . . . . .	97
<b>7</b>	<b>Supporting Practical Design Requirements</b>	<b>99</b>
7.1	Specifications and Control Problems . . . . .	100
7.2	Specifications as Automata on Infinite Strings . . . . .	100
7.3	Construction of Parity Games and Symbolic Controller Synthesis . . . . .	101
7.4	Implementation Details . . . . .	103
7.4.1	Submitting Control Problems . . . . .	104
7.4.2	Synthesizing Symbolic Controllers . . . . .	106
7.4.3	Collecting Results and Simulations . . . . .	106
7.5	Examples . . . . .	109
7.5.1	Motion Planning for Autonomous Vehicles . . . . .	109
7.5.2	Pickup-Delivery Drone on Battery . . . . .	110
7.6	Summary . . . . .	112
<b>8</b>	<b>Standardized Implementations of Synthesized Symbolic Controllers</b>	<b>115</b>
8.1	Types of Symbolic Controllers . . . . .	115
8.2	Formal Implementations of Static Symbolic Controllers . . . . .	116
8.2.1	Look-Up-Tables (LUTs) . . . . .	116
8.2.1.1	Example Implementations: . . . . .	117
8.2.2	Binary Decision Diagram (BDD)-encoded Symbolic Controllers . . . . .	118
8.2.2.1	Example Implementations: . . . . .	119

## CONTENTS

8.2.3	Boolean Functions as Control Laws . . . . .	120
8.2.3.1	Example Hardware Implementation . . . . .	121
8.2.3.2	Example Software Implementation . . . . .	121
8.3	Formal Implementations of Dynamic Symbolic Controllers . . . . .	121
8.4	Summary . . . . .	122
<b>9</b>	<b>Conclusions and Future Works</b>	<b>123</b>
9.1	Conclusions . . . . .	123
9.2	Strengths, Weaknesses and Limitations . . . . .	125
9.3	Recommendations for Future Works . . . . .	126
9.3.1	Memory-efficient Data-parallelism for Symbolic Control . . . . .	127
9.3.2	User-Friendly Cloud-Application for Designing and Implementing Parallel Algorithms . . . . .	127
9.3.3	Supporting Continuous-time Stochastic Control Systems . . . . .	128
9.3.4	Supporting Output-based Control Systems . . . . .	129
9.3.5	Stochastic Parity Games and Counter-strategy Analysis . . . . .	129
	<b>Bibliography</b>	<b>131</b>

# List of Figures

1.1	Examples of safety-critical CPSs. Top left: a surgical robot, top right: an autonomous vehicle, bottom left: a railway signaling and control system, and bottom right: an insulin pump. Source: Wikipedia.com. License: Creative Commons (CC). . . . .	3
1.2	The cockpit of Tesla Model 3 car consists of a single 15.4-inch touch screen and a driving wheel. Source: piqsels.com. License: CC. . . . .	4
1.3	Uber’s self-driving car struck and killed a jaywalker, Tempe, Arizona, 2018. Source: ABC 15. . . . .	5
1.4	A waterfall model simplifying the development cycle of SCCS. . . . .	5
1.5	Controller synthesis for control systems based on symbolic models. . . . .	8
1.6	A dependency map for the chapters of the thesis. . . . .	11
2.1	Symbolic control as an abstraction-refinement approach. . . . .	21
3.1	An example Cloud-based deployment of pFaces. . . . .	27
3.2	Screenshot from the web-based interface of pFaces. . . . .	28
3.3	(a) Internal structure of pFaces. (b) General workflow inside pFaces. . . . .	29
4.1	Workflow of the symbolic control kernel inside pFaces. . . . .	41
4.2	(a) A truck with $N$ trailers. (b) real-time implementation of the kernel. . . . .	44
4.3	The sparsity graph of the vehicle example as introduced in [GKA17]. . . . .	46
4.4	An example task distributions for the data-parallel sparsity-aware abstraction. . . . .	49
4.5	Comparison between the serial and parallel algorithms for constructing abstractions of a traffic network model by varying the dimensions. . . . .	50
4.6	A visualization of one arbitrary fixed-point iteration of the sparsity-aware synthesis technique for a two-dimensional robot system. . . . .	53
4.7	The evolution of the fixed-point sets for the robot example by the end of fixed-point iterations 5 (left side) and 228 (right side). . . . .	55
4.8	An autonomous vehicle trying to avoid a sudden obstacle on the highway. . . . .	56
5.1	Interval approximation of the flow pipe for the Van der Pol oscillator. . . . .	61
5.2	Speed test results for kernel PIRK. . . . .	68
5.3	Interval flow pipe approximating the behavior of the BMW 320i car. . . . .	72
6.1	A 2-dimensional visualization of the cutting probability region. . . . .	80

LIST OF FIGURES

6.2	A 2-dimensional visualization of the cutting probability region after approximating the effect of $\hat{W}$ . . . . .	81
6.3	A visualization of transitions for one source state $x := (0, 0)$ and input $\nu := (0.7, 0.8)$ of the MDP approximating the robot example. . . . .	89
6.4	Repeated simulations of the closed-loop behavior of the robot under a safety controller. . . . .	90
6.5	Different simulations of the closed-loop behavior of the robot example under a reach-avoid controller. . . . .	91
6.6	Model of a road traffic network composed of 5 cells of 500 meters with 2 entries and 2 ways out. . . . .	94
7.1	A technique on symbolic control that supports $\omega$ -regular specifications. . .	101
7.2	Closed-loop simulation of the vehicle example captured once region <code>target2</code> (blue rectangle) is reached. . . . .	108
7.3	Simulation of the drone example after 63 seconds as the drone starts a charging task. . . . .	111
8.1	A LUT encoding controller $C_q$ for system $S_q$ . . . . .	116
8.2	An implementation of a LUT encoding controller $C_q$ . . . . .	117
8.3	A BDD encoding a controller $C_q$ for a symbolic model with $ X_q  = 16$ and $ U_q  = 4$ . . . . .	118
8.4	An implementation of a BDD encoding controller $C_q$ . . . . .	119
8.5	A Boolean circuit of $N_U$ Boolean functions encoding a controller $C_q$ . . .	120

# Acronyms

AI	Artificial Intelligence.
AMI	Amazon Machine Image.
API	Application Programming Interface.
ASR	Alternating (Bi-)Simulation Relation.
AWS	Amazon Web Services.
BDC	Binary Coded Decimal.
BDD	Binary Decision Diagram.
BFS	Breadth First Search.
CC	Creative Commons.
CN	Compute Node.
CoD	Curse of Dimensionality.
CoG	Center of Gravity.
CPS	Cyber-Physical System.
CPU	Central Processing Unit.
CU	Compute Unit.
DNF	Disjunctive Normal Form.
DNS	Domain Name System.
DPA	Deterministic Parity Automata.
dt-SCS	Discrete-time Stochastic Control System.
EC2	Elastic Computing.
FLTS	Finite Labeled Transition System.
FPGA	Field Programmable Gate Array.
FRR	Feedback Refinement Relation.
GBFP	Growth-Bound and Fixed-Point.
GPGPU	General Purpose GPU.
GPU	Graphics Processing Unit.
GR(1)	General Reactivity (1).
HDL	Hardware Description Language.
HPC	High-Performance Computing.

## Acronyms

HW	Hardware.
HWA	Hardware Accelerator.
HWC	Hardware Configuration.
IDE	Integrated Development Environment.
IoR	Internet-of-Robots.
IoT	Internet-of-Things.
LTI	Linear Time-Invariant.
LTL	Linear Temporal Logic.
LUT	Lookup Table.
MCAS	Maneuvering Characteristics Augmentation System.
MDP	Markov Decision Process.
MemGBFP	Memory-Efficient GBFP.
MPI	Message Passing Interface.
MTL	Metric Temporal Logic.
OARS	Over-Approximation of the Reachable Set.
ODE	Ordinary Differential Equation.
OS	Operating System.
PC	Personal Computer.
PDE	Partial Differential Equation.
PDF	Probability Density Function.
PE	Processing Element.
PRAM	Parallel Random Access Memory.
RAM	Random Access Memory.
RDS	Relational Database Service.
RESTful	Representational State Transfer.
RHS	Right-Hand Side.
SaaS	Software as a Service.
SCCS	Safety-Critical Control Software.



## *Acronyms*

# 1 Introduction

CPSs are complex systems resulting from an intricate interaction of digital computational devices with physical systems. Within CPSs, (embedded) control software monitors and adjusts several physical variables, e.g. temperature, velocity, pressure, density, and so on, through feedback loops where physical processes affect computation and vice versa [Maj16, Ray16]. Safety-critical CPS is a class of CPSs where a failure or malfunction may result in death/injury to people, loss/damage to equipment/property, or environmental harm. Examples of safety-critical CPSs are airplanes, autonomous vehicles, and surgical robots.



**Figure 1.1:** Examples of safety-critical CPSs. Top left: a surgical robot, top right: an autonomous vehicle, bottom left: a railway signaling and control system, and bottom right: an insulin pump. Source: Wikipedia.com. License: CC.

SCCS plays a significant role in safety-critical CPSs by interfacing with multiple sensors, performing control and planning tasks, and commanding the actuators to interact with the physical environment. A simple program with few lines of code and a complex software library with a million lines of code are both considered SCCS if they take part

## 1 Introduction



**Figure 1.2:** The cockpit of Tesla Model 3 car consists of a single 15.4-inch touch screen and a driving wheel. Source: piqsels.com. License: CC.

in safety-critical CPSs. Examples of SCCS are the embedded software in insulin pumps, the autopilot software in autonomous vehicles, or the fly-by-wire software in airplanes.

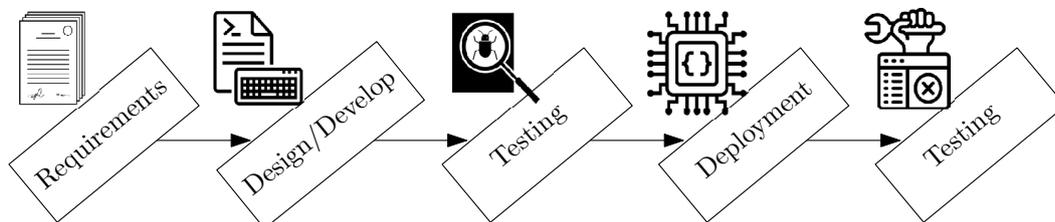
SCCS is becoming more and more ubiquitous in many application domains including automotive, aerospace, transportation systems, robotics, healthcare, etc. With the current growing trend in computational devices, the designers of safety-critical CPSs tend to use off-the-shelf computing devices to standardize system design and allow for software reusability and maintainability. Many hardware components are consequently replaced with software which eventually increases the complexity of SCCS. Consider for example the user interface of the Tesla Model 3 car as depicted in Fig. 1.2. The single 15.4-inch touch screen provides access to all the driving and infotainment functionalities of the car. Such a minimalistic user interface is only the tip of the iceberg, and it hides underneath a complex software design that communicates with hundreds of sensors, performs intensive image processing and Artificial Intelligence (AI) computations, handles automated driving/control tasks and provides this easy-to-use user interface. Although the user interfaces of today's SCCS are becoming more user-friendly and, in some cases, oversimplified, their internal designs are unfortunately becoming more complex and error-prone. This is anticipated because every application has an inherent amount of complexity that cannot be removed or hidden [Yab20, Tesler's Law].

As SCCS becomes more complex internally, there are increasingly higher possibilities of faults or failures, and consequently fatalities, injuries, or at least, less trust from the community. Since the current designs of SCCS are still based on ad-hoc solutions that try to make connections between classical techniques of control theory and (embedded) systems engineering, they are more vulnerable to software errors and bugs. Two re-



**Figure 1.3:** Uber’s self-driving car struck and killed a jaywalker, Tempe, Arizona, 2018. Source: ABC 15.

cent examples show how devastating the failures of SCCS can be: (1) in 2018, Uber’s self-driving car struck and killed a jaywalker as its autopilot software failed to stop the car before the crash [1518], and (2) in 2019, the faulty Maneuvering Characteristics Augmentation System (MCAS) in Boeing 737 Max caused the death of around 350 passengers in two consecutive crashes [oTotUSHoR19]. Ensuring the highest possible levels of correctness of SCCS should be the first requirement in any SCCS design process. This is unfortunately not the case with today’s SCCS.



**Figure 1.4:** A waterfall model simplifying the development cycle of SCCS.

Figure 1.4 shows a diagram of the development cycle of SCCS. This waterfall software model, although not exactly what is used nowadays in practice, has at least the main building blocks that we must pay attention to. It starts with a phase where the requirements are collected and analyzed. Here, humans collect, analyze, write and communicate the requirements. Humans do mistakes all the time. As a result, the requirements can be false or ambiguous. They may be also miscommunicated to other persons responsible for the next steps. Second step in the diagram is where the SCCS is actually designed and coded. A developer responsible for coding some features of the SCCS may make mistakes and write buggy code. Some of these bugs are logical ones that are hard to

## 1 Introduction

find using traditional static/dynamic testing techniques. Next step is testing. Testing of SCCS is also usually done by humans who can not cover all testing scenarios and consequently many edge-cases are left undetected. Even if testing is automated, it is usually based on scenarios that are defined by humans. Next step is deployment where the final product is integrated into the target CPS either as a software running on a computer or as a hardware circuit that implements the designed logic. Here, issues may happen due to the unplanned integrations of software and hardware. Notice that SCCS is originally intended to interface various sensors and actuators. This requires an additional final step of testing, which would also leave many edge-cases undetected.

In order to detect and eliminate design flaws and inevitable software bugs, a large portion of the design budget nowadays is consumed with testing and validation efforts which are often very costly. As an example, a recent report [Mat] showed that, in 2017, software failures affected 3.6 billion people, and caused \$1.7 trillion in financial losses and a cumulative total of 268 years of downtime.

Clearly, current design approaches of SCCS do not consider the correctness of the final product as their first goals. Ensuring the correctness of SCCS requires some radical changes in their design processes. The sources of human-related faults must be eliminated or minimized. New techniques for declaring the design requirements in a more precise, formal and unambiguous way, are required, so that they can be safely communicated. The design/development tasks need to be automated to reduce many of the human-related bugs in today's SCCS. Less testing is needed and this can be achieved through design/development steps that produce correct-by-construction software.

### 1.1 Promising Design Approach for SCCS

From control theory perspective, CPSs and their SCCS are considered hybrid systems. The computation units combining hardware and software are modeled with discrete dynamics, while physical components are modeled with continuous dynamics. Hybrid systems have state variables of real numbers, and consequently, they have an uncountable number of states. This makes it intractable to operate on them algorithmically in their original versions in order to automatically synthesize their controllers. Consequently, classical control theory techniques have been traditionally used to synthesize their controllers. They are however limited to basic design requirements such as stabilizability and transient response specifications. This is unfortunately not sufficient for today's CPS which usually require approaches that can handle complex high-level design specifications.

On the other hand, computer science literature is rich with approaches for automated synthesis of systems from high-level formal specifications (see [LMS20, PR89, and the references therein]) which are traditionally known as *reactive synthesis* techniques. Given design requirements in formal languages, e.g. Linear Temporal Logic (LTL) formulae [Pnu77], these approaches can automatically design systems that implement the given requirements. This is promising since (1) no human interaction is involved which minimizes human-related faults, and (2) the final products are certifiable in the sense that one

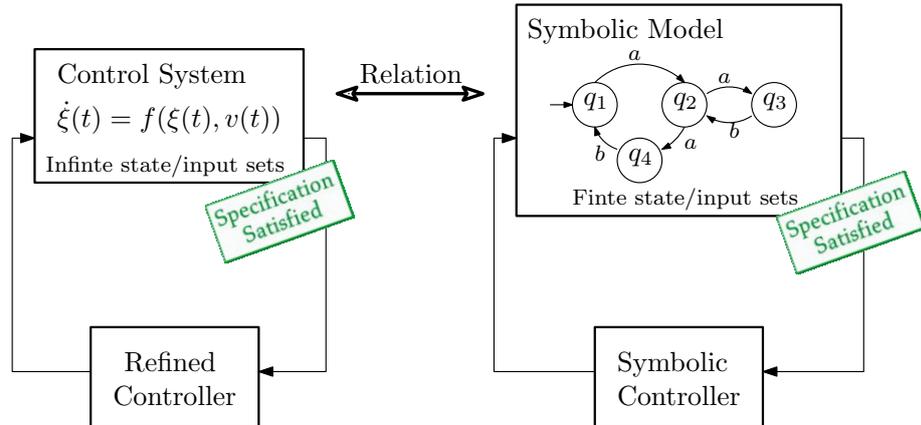
can guarantee that they must comply with the original design requirements, and hence, no post-development testing phases are necessary. Unfortunately, those approaches can not be used directly to design SCCS for CPSs. This is again because of the uncountable number of states of CPSs which makes it impossible to programmatically operate on them in their original forms. Moreover, these approaches accept specifications describing the systems to be designed, whereas here we need to provide requirements describing behaviors of systems to be enforced by the to-be-designed SCCS.

The need for utilizing reactive synthesis techniques to design correct-by-construction SCCS for CPSs resulted recently in several formal methods techniques for control systems such as *symbolic control* [Tab09, ZPMT12, CDD<sup>+</sup>13]. In symbolic control, systems are abstracted as finite-state models (a.k.a. symbolic models) that can be analyzed algorithmically to synthesize digital controllers (a.k.a. symbolic controllers) that enforce given high-level specifications on original systems. Symbolic control is promising for the design of SCCS since, unlike reactive synthesis techniques, hybrid systems can be handled. The designed symbolic controllers are also guaranteed to enforce the given design requirements on original systems, and post-development testing phases are consequently minimal.

In symbolic control, symbolic models replace the original (a.k.a. concrete) systems in the analysis and controller synthesis phases. They are finite-state abstractions of continuous-space hybrid systems in which each discrete state and input correspond to an aggregate of continuous states and inputs of the original system, respectively. In general, there exist two types of symbolic models: (1) sound symbolic models, whose behaviors (approximately) contain the behaviors of the concrete systems and (2) complete symbolic models, whose behaviors are (approximately) equivalent to the behaviors of concrete systems [Tab09]. The existence of a complete symbolic model results in a sufficient and necessary guarantee in the sense that, there exists a controller enforcing the desired specifications on the concrete system if and only if there exists a controller enforcing the same specifications on the symbolic model. On the other hand, sound symbolic models provide only sufficient guarantees in the sense that failing to find controllers for symbolic models does not imply the nonexistence of controllers for concrete systems.

Several studies have investigated the construction of symbolic models for various classes of control systems; linear systems [TP06], nonlinear systems [ZPMT12, PGT08a, RWR17, PGT08b, PT09, BH06, Tab08, ZMAL13, LLO15], mixed monotone systems [CA17], switched systems [ADLB14, GPT10], singularly perturbed hybrid affine systems [KG19], time-delay systems [PPBT10, PPB15], infinite dimensional systems [Gir14], networked control systems [ZMKA18, KRZ16, BPB19], and stochastic systems [ZA14, ZMM<sup>+</sup>14, ZMAL13]. Interested reader can also find more details about some of the results mentioned above in [BYG17, GP11].

Since symbolic models are finite, controller synthesis problems can be algorithmically solved over them by resorting to automata-theoretic approaches [MPS95, Tho95, BJP<sup>+</sup>12]. Here, symbolic models are viewed as labelled graphs that can be analyzed and control laws can be extracted from them. Given formally-defined specifications, techniques like search on graphs [Val02], fixed-point operations [PBW06], or two-player games [EGW02] can be applied to synthesize symbolic controllers that enforce the given



**Figure 1.5:** Controller synthesis for control systems based on symbolic models.

specifications. One can then refine the synthesized symbolic controllers back to the original systems based on some behavioral relations between original systems and their symbolic models. This includes for example (approximate) Alternating (Bi-)Simulation Relations (ASRs) [PT09] or Feedback Refinement Relations (FRRs) [RWR17]. The relations from original systems to their symbolic models ensure that controllers synthesized for the symbolic models can be refined with some suitable interfaces to work with the original systems. It is here also guaranteed that the refined controllers will enforce the given design requirements on the original systems. Figure 1.5 schematically depicts symbolic control.

Unfortunately, the construction of symbolic models and synthesis of their controllers, for large-scale CPSs, suffer from the so-called Curse of Dimensionality (CoD). Specifically, the computational complexity of the algorithms responsible for computing the models or synthesizing the controllers grow exponentially with respect to the dimensions of the state and input sets. Consequently, symbolic control becomes limited to small case studies with fewer symptoms of CoD. Several approaches have been proposed in the literature to overcome this scalability problem. Adaptive multi-resolution and multi-scale state-space discretization approaches have been proposed in [TI09, CGG11, HMMS18b]. A state-space discretization free approach was introduced in [LCGG13, ZAG15, Gir14] where symbolic states are given by input sequences. In [WRR17], the size of symbolic models were minimized using optimal discretization parameters. In [HMMS18a], a lazy version of multi-layered abstractions for nonlinear systems against safety and reachability specifications have been proposed. The authors in [GGM16] introduced lazy safety synthesis for incrementally stable switched systems using multi-scale symbolic models. In [GKA17], the authors proposed constructing symbolic models that utilize the sparsity of the dynamics in discrete-time systems. Unfortunately, all of these approaches do not

consider the inherent parallelism in the algorithms of symbolic control. If the algorithms of symbolic control are parallelized and, at the same time, distributed computing resources are considered, the computational complexity of symbolic control can be tuned to match the timing requirements of target applications.

Scalability is not the only challenge facing symbolic control and preventing its application to practical CPSs. The types of specifications or the class of systems supported by all current methodologies of symbolic control, and the tools that implement them, are limited. Tools like *Pessoa* [MDT10], *Tulip* [WTO<sup>+</sup>11], *CoSyMA* [MGG13] *SCOTS* [RZ16], and *ROCS* [LL18], which have been used in the past few years to construct symbolic models and design automatically their symbolic controllers, cannot handle complex dynamics and advanced practical specifications. *Tulip* handles only piece-wise affine control systems, and *CoSyMA* accepts only incrementally stable systems. *ROCS* and *CoSyMA* support only reachability and safety requirements, while *Tulip* supports only General Reactivity (1) (GR(1)) specifications [PR89], which is a fragment of LTL. Tools *Pessoa* and *SCOTS* requires the users to encode any extended requirements in  $\mu$ -calculus expressions before manually implementing them, which is very tedious and highly error-prone. In all mentioned tools, to extract the resulting symbolic controllers and deploy them, users have to manually define custom data structures and algorithms to implement the control laws. Notice that such ad-hoc approaches of controller extraction and deployment are not formally-verifiable. Consequently, they negatively affect the correctness guarantees obtained from the symbolic control techniques.

In a summary, symbolic control is a promising approach for constructing, automatically, formally-verified controllers for CPSs. It is however still limited in functionality and usability because of the following main three issues:

- (1) The construction of symbolic models and synthesis of symbolic controllers suffers from the CoD,
- (2) The class of specifications supported by most of existing methodologies of symbolic control, and all of existing tools, are limited and impractical, and
- (3) The ad-hoc techniques used to extract the designed controllers and deploy them weaken, and may ruin, the correctness guarantees obtained from symbolic control.

## 1.2 Contributions

In the previous section, we argued that SCCS is currently designed using approaches that may result into unsafe software. Moreover, we showed that today’s safety-critical CPSs require new approaches for designing bug-free, reliable and safe SCCS. We also introduced briefly symbolic control as a promising approach for designing automatically SCCS. Symbolic control is not ready for real-world CPSs due to the major issues we identified earlier. This thesis offers solutions to each of these issues so that symbolic control can be possibly applied to today’s safety-critical CPSs. The main contributions of this thesis are:

## 1 Introduction

- (1) We introduce a framework for designing and implementing efficient scalable parallel algorithms while utilizing all available computing platforms such as Central Processing Units (CPUs), Graphics Processing Units (GPUs), Hardware Accelerators (HWAs), and interconnections of them. It is used throughout the thesis to design efficient scalable algorithms for symbolic control.
- (2) We introduce novel data-parallel algorithms for:
  - the construction of symbolic models from different classes of control systems,
  - the construction of Markov Decision Process (MDPs) as abstractions of a class of stochastic control systems,
  - the synthesis of symbolic controllers using fixed-point operations, dynamic programming or solving two-player games, and
  - the computation of reachable sets for extremely high-dimensional systems.The algorithms scale with available computing resources, and can, using the introduced framework, utilize High-Performance Computing (HPC) platforms to mitigate the effects of the CoD.
- (3) We introduce an approach that allows symbolic control to utilize more complex and practical design requirements given as LTL formulae or as automata on infinite strings.
- (4) We introduce unified formal approaches for extracting the designed controllers and deploying them as software or hardware. As a result, the correctness guarantees obtained from symbolic control is maintained.

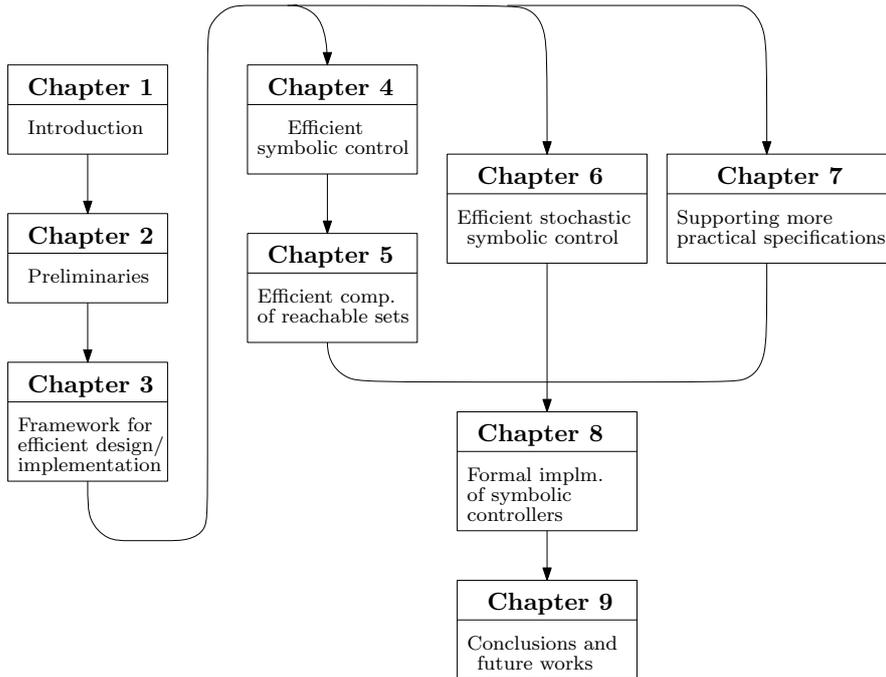
### 1.3 Thesis Organization

This thesis is divided into 9 chapters, the first of which is the current introduction. Figure 1.6 shows the chapters and their dependencies.

Chapter 2 introduces some preliminaries. First, symbolic control is introduced as a promising approach for designing, automatically, correct-by-construction controllers that can enforce high-level specifications on original systems. We introduce formally control systems, the construction of symbolic models and the synthesis of symbolic controllers. We show with examples that current techniques of symbolic control suffer from the major issues discussed earlier. We also show that these issues prevent symbolic control from being applicable to designing the SCCS of today's CPS.

In Chapters 3-8, we propose solutions to the identified issues of symbolic control. Chapter 3 introduces a framework used throughout the thesis for designing and implementing the introduced novel algorithms.

Chapter 4 introduces parallel scalable algorithms for constructing the symbolic models and synthesizing their symbolic controllers. We show that these algorithms, alongside with variable computing resources, allow controlling the computational complexities of symbolic control algorithms and mitigating the effects of the CoD.



**Figure 1.6:** A dependency map for the chapters of the thesis.

In Chapters 5 and 6, we introduce two additional enhancements to symbolic control. Chapter 5 introduces scalable algorithms for selected methodologies of reachability analysis that we use internally to construct the symbolic models. Chapter 6 introduces extensions to symbolic control to support stochastic control systems. Efficient parallel algorithms for automated controller synthesis of controllers for stochastic control systems are introduced.

Chapter 7 handles the issue of symbolic control related to the lack of support for practical formal specifications. The chapter introduces a technique that allows handling specifications given as LTL formulae or as Deterministic Parity Automata (DPA).

Chapter 8 discusses the types of symbolic controllers resulting from the algorithms introduced in the first 7 chapters, and introduces formal deployments for the designed symbolic controllers.

Chapter 9 provides a summary of the results introduced in all chapter and discusses possible future research directions.



## 2 Preliminaries

We start this chapter with some general notations. Then, in Section 2.2, we introduce the notion of systems as a general mathematical framework to describe original systems, symbolic models, symbolic controllers and their interconnections. In Section 2.3, the process of constructing symbolic models from original systems is formally introduced. Later in Section 2.4, we discuss the process of controller synthesis. Finally, Section 2.5 discusses the issues of symbolic control that we address in the next chapters.

### 2.1 Notations

Symbols  $\mathbb{N}$ ,  $\mathbb{Z}$ , and  $\mathbb{R}$  denote, respectively, the set of natural, integer, and real numbers. Additionally, symbols  $\mathbb{N}_+$ ,  $\mathbb{R}_+$ , and  $\mathbb{R}_{0+}$  denote, respectively, the set of positive natural, positive real, and nonnegative real numbers, as restricted versions of their original number sets.

The identity map on a set  $X$  is denoted by  $id_X$ . The relative complement of a set  $A$  in a set  $B$  is denoted by  $B \setminus A$ . For a set  $A$ , we denote by  $|A|$  the cardinality of the set and by  $2^A$  the set of all subsets of  $A$  including the empty set  $\emptyset$ . For sets  $A$  and  $B$ , we denote by  $A \times B$  the Cartesian product of  $A$  and  $B$ , and by  $A \setminus B$  the Pontryagin difference between the sets  $A$  and  $B$ . A *cover* of a set  $A$  is a set of subsets of  $A$  whose union equals  $A$ . A *partition* of a set  $A$  is a set of pairwise disjoint subsets of  $A$  whose union equals  $A$ .

We denote by  $A^*$  the set of all finite strings (a.k.a. sequences) obtained by concatenating elements in  $A$ , by  $A^\omega$  the set of all infinite strings obtained by concatenating elements in  $A$ , and by  $A^\infty$  the set of all finite and infinite strings obtained by concatenating elements in  $A$ . For any finite string  $s$ ,  $|s|$  denotes the length of the string,  $s_i$ ,  $i \in \{0, 1, \dots, |s| - 1\}$ , denotes the  $i$ -th element of  $s$ , and  $s[i, j]$ ,  $j \geq i$ , denotes the substring  $s_i s_{i+1} \dots s_j$ . Symbol  $\mathbf{e}$  denotes the empty string and  $|\mathbf{e}| = 0$ . We use the dot symbol  $\cdot$  to concatenate two strings.

Given a map  $R : A \rightarrow B$  and a set  $\mathcal{A} \subseteq A$ , we define

$$R(\mathcal{A}) := \bigcup_{a \in \mathcal{A}} \{R(a)\}.$$

Similarly, given a set-valued map  $Z : A \rightarrow 2^B$  and a set  $\mathcal{A} \subseteq A$ , we define

$$Z(\mathcal{A}) := \bigcup_{a \in \mathcal{A}} Z(a).$$

Consider a relation  $\mathcal{R} \subseteq A \times B$  where  $A$  and  $B$  are sets.  $\mathcal{R}$  is strict when  $\mathcal{R}(a) \neq \emptyset$  for every  $a \in A$ .  $\mathcal{R}$  naturally introduces a map  $\mathcal{R} : A \rightarrow 2^B$  such that  $\mathcal{R}(a) = \{b \in$

$B \mid (a, b) \in \mathcal{R}$ .  $\mathcal{R}$  also admits an inverse relation  $\mathcal{R}^{-1} := \{(b, a) \in B \times A \mid (a, b) \in \mathcal{R}\}$ . Given an element  $r := (a, b) \in \mathcal{R}$ ,  $\pi_A(r)$  denotes the natural projection of  $r$  on the set  $A$ , i.e.,  $\pi_A(r) = a$ . We sometimes abuse the notation and apply the projection map  $\pi_A$  to a string (resp., a set of strings) of elements of  $\mathcal{R}$ , which means applying it iteratively to all elements in the string (resp., all strings in the set). When  $\mathcal{R}$  is an equivalence relation on a set  $X$ , we denote by  $[x]$  the equivalence class of  $x \in X$  and by  $X/\mathcal{R}$  the set of all equivalence classes (a.k.a. quotient set).  $\pi_{\mathcal{R}} : X \rightarrow X/\mathcal{R}$  is a natural projection map taking a point  $x \in X$  to its equivalence class, i.e.,  $\pi_{\mathcal{R}}(x) = [x] \in X/\mathcal{R}$ . We say that an equivalence relation is finite when it has finitely many equivalence classes.

Set  $\mathbb{R}^n$  represents the  $n$ -dimensional Euclidean space of real number vectors with  $n$  elements. Given a vector  $v \in \mathbb{R}^n$ , we denote by  $v_i$ ,  $i \in \{0, 1, \dots, n-1\}$ , the  $i$ -th element (a.k.a. component) of  $v$  and by  $\|v\|$  the infinity norm of  $v$ . Given  $N$  vectors  $x_i \in \mathbb{R}^{n_i}$ ,  $n_i \in \mathbb{N}_+$ , and  $i \in \{1, \dots, N\}$ , we use  $x = [x_1; \dots; x_N]$  to denote the corresponding augmented vector of dimension  $\sum_i n_i$ . Given a matrix  $A$  in  $\mathbb{R}^{n \times m}$ ,  $A(:, b)$  denotes the all rows together with the  $b^{\text{th}}$  column of  $A$ , and  $A(b, :)$  the other way around.

Any  $n$ -dimensional hyper-rectangle (a.k.a. hyper interval) is characterized by two corner vectors  $x_{lb}, x_{ub} \in \mathbb{R}^n$ , and we denote the hyper-rectangle by

$$[x_{lb}, x_{ub}] := [x_{lb,1}, x_{ub,1}] \times [x_{lb,2}, x_{ub,2}] \times \dots \times [x_{lb,n}, x_{ub,n}].$$

## 2.2 Mathematical Framework for Systems

We introduce the notion of *systems* as a general mathematical framework used throughout this thesis to describe sampled original systems, symbolic models, symbolic controllers, and their interconnections. We use a similar definition for systems as in [Tab09].

**Definition 2.2.1** (System). *A system is a tuple*

$$S := (X, X_0, U, T, Y, H),$$

where  $X$  is a set of states,  $X_0 \subseteq X$  is a set of initial states,  $U$  is a set of inputs,  $T \subseteq X \times U \times X$  is a transition relation,  $Y$  is a set of outputs, and  $H : X \rightarrow Y$  is an output map.

All sets in tuple  $S$  are assumed to be non-empty. For any  $x \in X$  and  $u \in U$ , we denote by  $\text{Post}_u^S(x) := \{x' \in X \mid (x, u, x') \in T\}$  the set of  $u$ -successors of  $x$  in  $S$ . When  $S$  is known from the context, the set of  $u$ -successors of  $x$  is simply denoted by  $\text{Post}_u(x)$ . We sometimes abuse the notation and apply  $\text{Post}_u^S(\cdot)$  to a subset  $\tilde{X} \subseteq X$  which means applying it to all the elements of  $\tilde{X}$ . Specifically, given subset  $\tilde{X} \subseteq X$ , we have that

$$\text{Post}_u^S(\tilde{X}) := \bigcup_{x \in \tilde{X}} \{\text{Post}_u^S(x)\}.$$

The inputs admissible to a state  $x$  is denoted by  $U_S(x) := \{u \in U \mid \text{Post}_u(x) \neq \emptyset\}$ .

System  $S$  is said to be static if  $X$  is singleton; autonomous if  $U$  is singleton; state-based (a.k.a. simple system [RWR17]) when  $X = Y$ ,  $H = id_X$ , and all states are admissible

as initial ones, i.e.,  $X = X_0$ ; output-based when  $X \neq Y$ ; total when for any  $x \in X$  and any  $u \in U$  there exists at least one  $x' \in X$  such that  $x' \in \text{Post}_u(x)$ ; deterministic when for any  $x \in X$  and any  $u \in U$  we have  $|\text{Post}_u(x)| \leq 1$ ; and symbolic when  $X$  and  $U$  are both finite sets. For any  $\bar{x} \subseteq X_0$ , we denote by  $S^{(\bar{x})}$  the restricted version of  $S$  with  $X_0 = \bar{x}$ .

A *run* of system  $S$  is an infinite sequence  $r := x_0 u_0 x_1 u_1 \cdots x_{n-1} u_{n-1} x_n \cdots$  such that  $x_0 \in X_0$ , and for any  $i \geq 0$  we have  $(x_i, u_i, x_{i+1}) \in T$ . The *prefix* up to  $x_n$  of  $r$  is denoted by  $r(n)$  and its last element is  $\text{Last}(r(n)) := x_n$ . The set of all runs and the set of all  $n$ -length prefixes are denoted by  $\text{RUNS}(S)$  and  $\text{PREFS}^n(S)$ , respectively. A state  $x$  is said to be *reachable* iff there exists at least one *prefix*  $r(n) \in \text{PREFS}^n(S)$  such that  $\text{Last}(r(n)) = x$  for some  $n \in \mathbb{N}$ .

### 2.2.1 Composition of Systems

Systems can be composed together to construct new systems. Here, we define formally different types of compositions.

**Definition 2.2.2** (Serial Composition). *Let  $S_i := (X_i, X_{i,0}, U_i, T_i, Y_i, H_i)$ ,  $i \in \{1, 2\}$ , be two systems such that  $Y_1 \subseteq U_2$ . The serial (a.k.a. cascade) composition of  $S_1$  and  $S_2$ , denoted by  $S_2 \circ S_1$ , is a new system  $S_{12} := (X_1 \times X_2, X_{1,0} \times X_{2,0}, U_1, T_{12}, Y_2, H_{12})$ , where  $((x_1, x_2), u_1, (x'_1, x'_2)) \in T_{12}$  iff there exist two transitions  $(x_1, u_1, x'_1) \in T_1$  and  $(x_2, H_1(x_1), x'_2) \in T_2$ , and map  $H_{12}$  is defined as follows for any  $(x_1, x_2) \in X_1 \times X_2$ :  $H_{12}((x_1, x_2)) := H_2(x_2)$ .*

**Definition 2.2.3** (Feedback Composition). *Let  $S_i := (X_i, X_{i,0}, U_i, T_i, Y_i, H_i)$ ,  $i \in \{1, 2\}$ , be two systems such that  $Y_1 \subseteq U_2$ ,  $Y_2 \subseteq U_1$ , and the following holds:*

$$y_2 = H_2(x_2) \wedge y_1 = H_1(x_1) \wedge \text{Post}_{y_2}^{S_1}(x_1) = \emptyset \implies \text{Post}_{y_1}^{S_2}(x_2) = \emptyset.$$

*Then,  $S_1$  is said to be feedback-composable with  $S_2$  and the new composed system is  $S_{12} := (X_1 \times X_2, X_{1,0} \times X_{2,0}, \{0\}, T_{12}, Y_1 \times Y_2, H_{12})$ , where  $((x_1, x_2), 0, (x'_1, x'_2)) \in T_{12}$  iff there exist two transitions  $(x_1, H_2(x_2), x'_1) \in T_1$  and  $(x_2, H_1(x_1), x'_2) \in T_2$ , and the map  $H_{12}$  is defined as follows for any  $(x_1, x_2) \in X_1 \times X_2$ :*

$$H_{12}((x_1, x_2)) := (H_1(x_1), H_2(x_2)).$$

*The feedback composition between  $S_1$  and  $S_2$  is denoted by  $S_1 \times S_2$ .*

## 2.3 Symbolic Models

In symbolic control, a control system (e.g., a physical process described by a set of differential equations of state variables and input variables) is related to an abstraction (i.e., a system with finite state and input sets) via a formal relation. Throughout his thesis, control systems may be referred to as *original systems* or *concrete systems*, while their abstractions are referred to as *symbolic models* or *abstract systems*. A formal relation

## 2 Preliminaries

between a control system and its symbolic model ensures that the latter captures some required features from the original system. Symbolic models can be used to abstract several classes of control systems as discussed in Chapter 1. As symbolic models are finite, reactive synthesis techniques [PR89, Var95, BJP<sup>+</sup>12] are applicable to algorithmically synthesize controllers enforcing given high-level specifications. The designed controllers are usually referred to as *symbolic controllers*. In this section, we construct symbolic models as abstractions of control systems.

Hereinafter, we consider mainly state-based systems to represent original systems and their symbolic models. Hence, the representation of systems is reduced to

$$S := (X, U, T). \quad (2.3.1)$$

We now revise FRRs [RWR17] which we use later in this section to relate original systems with their symbolic models.

**Definition 2.3.1** (FRR). *Let  $S_i := (X_i, U_i, T_i)$ ,  $i \in \{1, 2\}$ , be two systems such that  $U_2 \subseteq U_1$ . A strict relation  $Q \subseteq X_1 \times X_2$  is an FRR from  $S_1$  to  $S_2$  if all of the followings hold for all  $(x_1, x_2) \in Q$ :*

- (i)  $U_{S_2}(x_2) \subseteq U_{S_1}(x_1)$ , and
- (ii)  $u \in U_{S_2}(x_2) \implies Q(\text{Post}_u^{S_1}(x_1)) \subseteq \text{Post}_u^{S_2}(x_2)$ .

When  $Q$  is an FRR from  $S_1$  to  $S_2$ , this is denoted by  $S_1 \preceq_Q S_2$ .

FRRs are introduced to resolve common shortcomings in ASRs and their approximate versions. As discussed in [RWR17], using ASRs results in controllers that require exact state information of concrete systems while only quantized state information is usually available. Additionally, the refined controllers contain symbolic models of original systems as building blocks inside them, which makes the implementation much more complex. On the other hand, controllers designed for systems related via FRRs require only quantized-state information. They can be feedback-composed with original systems through static quantizers, and they do not require the symbolic models as building blocks inside them. Such features simplify refining and implementing the synthesized symbolic controllers.

### 2.3.1 Control Systems

Now, we introduce the class of control systems considered in this thesis. Unless stated otherwise, we consider general (possibly nonlinear) continuous-time systems.

**Definition 2.3.2** (Control System). *A control system is a tuple  $\Sigma := (\mathcal{X}, \mathcal{U}, f)$ , where  $\mathcal{X} \subseteq \mathbb{R}^n$  is the state set;  $\mathcal{U} \subseteq \mathbb{R}^m$  is an input set;  $f : \mathcal{X} \times \mathcal{U} \rightarrow \mathcal{X}$  is a continuous map satisfying the following Lipschitz assumption: for each compact set  $X \subseteq \mathcal{X}$ , there exists a constant  $L \in \mathbb{R}_+$  such that*

$$\|f(x_1, u) - f(x_2, u)\| \leq L\|x_1 - x_2\|,$$

for all  $x_1, x_2 \in X$  and all  $u \in \mathcal{U}$ .

Let  $\mathcal{U}$  be the set of all functions of time from  $]a, b[ \subseteq \mathbb{R}$  to  $\mathcal{U}$  with  $a < 0$  and  $b > 0$ . We define a trajectory of  $\Sigma$  by the locally absolutely continuous curve  $\xi : ]a, b[ \rightarrow \mathcal{X}$  if there exists a  $v \in \mathcal{U}$  that satisfies  $\dot{\xi}(t) = f(\xi(t), v(t))$  at any  $t \in ]a, b[$ . We redefine  $\xi : [0, t] \rightarrow \mathcal{X}$  for trajectories over closed intervals with the understanding that there exists a trajectory  $\xi' : ]a, b[ \rightarrow \mathcal{X}$  for which  $\xi = \xi'|_{[0, t]}$  with  $a < 0$  and  $b > t$ .  $\xi_{xv}(t)$  denotes the state reached at time  $t$  under input  $v$  and with the initial condition  $\xi_{xv}(0)$ . Such a state is uniquely determined since the assumptions on  $f$  ensure the existence and uniqueness of its trajectories [Son99]. System  $\Sigma$  is said to be *forward complete* if every trajectory is defined on an interval of the form  $]a, \infty[$ . Hereinafter, we consider forward complete control systems.

### 2.3.2 Control Systems as Systems

Let  $\Sigma$  be a control system as defined in Definition 2.3.2. The sampled version of  $\Sigma$  (a.k.a. concrete system) is a system

$$S_\tau(\Sigma) := (X_\tau, U_\tau, T_\tau), \quad (2.3.2)$$

that encapsulates the information contained in  $\Sigma$  at sampling times  $k\tau$ , for all  $k \in \mathbb{N}$ , where  $X_\tau \subseteq \mathcal{X}$ ,  $U_\tau$  is the set of piece-wise constant curves of length  $\tau$  defined as follows:

$$U_\tau := \{v_\tau : [0, \tau[ \rightarrow \mathcal{U} \mid v_\tau(t) = v_\tau(0) \wedge t \in [0, \tau[ \},$$

and a transition  $(x_\tau, v_\tau, x'_\tau) \in T_\tau$  iff there exists a trajectory  $\xi : [0, \tau] \rightarrow \mathcal{X}$  in  $\Sigma$  such that  $\xi_{x_\tau v_\tau}(\tau) = x'_\tau$ . We sometimes use  $S_\tau$  to refer to the sampled-data system  $S_\tau(\Sigma)$ .

**Remark 2.3.3.** *System  $S_\tau$  is deterministic since any trajectory of the forward complete control system  $\Sigma$  is uniquely determined. Sets  $X_\tau$  and  $U_\tau$  are both uncountable, and hence,  $S_\tau$  is not symbolic. Since all trajectories of  $\Sigma$  are defined for all inputs and all states, we have  $U_{S_\tau}(x_\tau) = U_\tau$ , for all  $x_\tau \in X_\tau$ .*

In the next subsections, we construct symbolic models of sampled original systems, synthesize their symbolic controllers, and refine the designed symbolic controllers.

### 2.3.3 Symbolic Models of Control Systems

We utilize FRRs to construct symbolic models that approximate  $S_\tau$ . FRRs mainly ensure synchronized state-based evolutions between concrete systems and their symbolic models. Given a control system  $\Sigma$ , let  $S_\tau$  be its sampled-data representation, as defined in (2.3.2). A symbolic model of  $S_\tau$  is a system

$$S_q := (X_q, U_q, T_q), \quad (2.3.3)$$

where  $X_q := X_\tau / \bar{Q}$ ,  $\bar{Q}$  is a finite equivalence relation on  $X_\tau$ ,  $U_q$  is a finite subset of  $U_\tau$ , and  $(x_q, u_q, x'_q) \in T_q$  if there exist  $x \in x_q$  and  $x' \in x'_q$  such that  $(x, u_q, x') \in T_\tau$ .

The following theorem shows that the constructed symbolic model is related via an FRR with its original system.

**Theorem 2.3.4.** *Let  $S_\tau$  a control system as defined in (2.3.2). Let  $S_q$  the symbolic model of  $S_\tau$  as introduced in (2.3.3). Then, there exist an FRR from  $S_\tau$  to  $S_q$  such that  $S_\tau \preceq_Q S_q$*

*Proof.* Fix

$$Q := \{(x_\tau, x_q) \in X_\tau \times X_q \mid x_\tau \in x_q\},$$

where  $X_q$  is the states set of  $S_q$  as defined in (2.3.3).

We show that  $Q$  is an FRR from  $S_\tau$  to  $S_q$ . We show first that  $Q$  is strict. Consider any  $x_\tau \in X_\tau$ . We know that  $X_q$  is a partition of  $X_\tau$  since  $\bar{Q}$  is an equivalence relation. Consequently, there exist one  $x_q \in X_q$  such that  $x_\tau \in x_q$  and hence,  $(x_\tau, x_q) \in Q$ .

Condition (i) in Definition 2.3.1 is satisfied since by the definition of the input set  $U_q$  of  $S_q$  in (2.3.3), we have that  $U_{S_q}(x_q) \subseteq U_{S_\tau}(x_\tau)$  for any tuple  $(x_\tau, x_q) \in Q$ .

We now show that condition (ii) in Definition 2.3.1 is also satisfied. Consider one tuple  $(x_\tau, x_q) \in Q$ . Let  $u \in U_q$  be an arbitrary input. Let  $x' := \text{Post}_u^{S_\tau}(x_\tau)$  be the post state of  $S_\tau$  with  $u$  as an applied input. Notice that  $S_\tau$  is deterministic. We know from the definition of  $Q$  above that there exists  $x'_q := Q(x')$  for which  $x' \in x'_q$ . Let  $(x_\tau, u, x') \in T_\tau$  be the transition of  $S_\tau$  we just took with  $\text{Post}_u^{S_\tau}(x_\tau)$  and remember that  $x_\tau \in x_q$  and  $x' \in x'_q$ . We now conclude from the definition  $T_q$  that there exists a transition  $(x_q, u, x'_q) \in T_q$ , and hence,  $x'_q \in \text{Post}_u^{S_q}(x_q)$ . The last conclusion results in satisfying condition (ii) in Definition 2.3.1 and concludes the proof.  $\square$

## 2.4 Symbolic Controller Synthesis

Here, we discuss the second phase of symbolic control, that is, the synthesis of symbolic controllers using the constructed symbolic models. We first discuss the behaviors of systems and the considered specifications to be enforced on them. Then, we introduce control problems and controllers. Finally, we show how symbolic controllers are synthesized and refined.

### 2.4.1 Behaviors and Specifications

Let  $S$  be a system as defined in Definition 2.2.1. The behavior of  $S$  is a subset of the set of all (possibly infinite) prefixes of  $S$ , i.e.,

$$B(S) \subseteq \bigcup_{n \in \mathbb{N} \cup \{\infty\}} \text{PREFS}^n(S).$$

Unlike behaviors which describe possible runs of systems, specifications are mainly used to declare sets of runs that must conform with some given desired requirements.

**Definition 2.4.1** (Specification). *Let  $S$  be a system as defined in Definition 2.2.1. Let  $\Gamma_S := \pi_X(B(S))$  be the set of all state sequences of  $S$ . A specification  $\psi \subseteq \Gamma_S$  on  $S$  is a set of state sequences that must be enforced on  $S$ . System  $S$  satisfies  $\psi$  (denoted by  $S \models \psi$ ) iff  $\pi_X(B(S)) \subseteq \psi$ .*

Specifications can adopt formal requirements encoded as LTL formulae [Pnu77] or Automata on finite strings. Classical requirements like *invariance* (often referred to as *safety*) and *reachability* can be readily included. Given a safe set of observations  $F \subseteq X$ , we denote by  $\text{Safe}(F)$  the *safety specification*, and we define it as follows:

$$\text{Safe}(F) := \{x_0x_1x_2 \cdots \in \Gamma_S \mid \forall k \geq 0 (x_k \in F)\}.$$

The safety objective requires that the state of  $S$  always remains within subset  $F$ . Using LTL, such a safety specification is encoded as the formula  $\Box F$ . Similarly, for a target set of observations  $T \subseteq X$ , we denote by  $\text{Reach}(T)$  the *reachability specification*, and we define it as follows:

$$\text{Reach}(T) := \{x_0x_1x_2 \cdots \in \Gamma_S \mid \exists k \geq 0 (x_k \in T)\}.$$

The reachability objective requires that the state of  $S$  visits, at least once, some elements in  $T$ . Such a reachability specification is encoded as the LTL formula  $\Diamond T$ .

Specifications like *infinitely often* (a.k.a. *recurrence*) ( $\Box \Diamond G$ ) and *eventually forever* (a.k.a. *persistence*) ( $\Diamond \Box G$ ), for a set of observations  $G \subseteq X$ , can be defined similarly.

Now, we introduce the concept of control problems and the definition of controllers.

**Problem 2.4.2** (Control Problem). *Consider a system  $S$  as defined in Definition 2.2.1. Let  $\psi$  be a given specification on  $S$  following Definition 2.4.1. We denote by the tuple  $(S, \psi)$  the control problem of finding a system  $C$  such that  $C \times S \models \psi$ .*

System  $C$  that solves a control problem  $(S, \psi)$  (i.e.,  $C \times S \models \psi$ ) is called a controller. We define controllers formally next.

**Definition 2.4.3** (Controller). *Given a control problem  $(S, \psi)$  as defined in Problem 2.4.2, a controller solving  $(S, \psi)$  is a system*

$$C := (X_C, X_{C,0}, U_C, T_C, Y_C, H_C),$$

where  $U_C := X$  and  $Y_C := U$ . All of  $X_C$ ,  $X_{C,0}$ ,  $T_C$ , and  $H_C$  are constructed such that  $C \times S \models \psi$ .

The domain of controller  $C$  is the set of initial states of the controlled systems that can be controlled to solve the main control problem. We define it formally next.

**Definition 2.4.4** (Domain of Controller). *Consider a controller  $C$  solving a given control problem  $(S, \psi)$ , as defined in Definition 2.4.3. The domain of the controller is denoted by  $\mathcal{D}(C) \subseteq X_0$ , and it is defined as follows:*

$$\mathcal{D}(C) := \{x \in X_0 \mid C \times S^{\{\{x\}\}} \models \psi\}.$$

The introduced control problem follows a generic definition as it is defined based on Systems and their specifications. If system  $S$  used in Definition 2.4.2 is symbolic, the control problem is referred to as the symbolic control problem and its controller is referred to as its symbolic controller. We discuss this more precisely next.

### 2.4.2 Synthesis and Refinement of Symbolic Controllers

Consider a concrete system  $S_\tau$ . Let  $\psi_\tau$  be some given specification on  $S_\tau$ .  $S_\tau$  and  $\psi_\tau$  represent together a *concrete control problem*  $(S_\tau, \psi_\tau)$ . Since  $S_\tau$  is infinite, the concrete control problem is not directly solvable using algorithmic controller synthesis techniques. Instead, a *symbolic control problem*  $(S_q, \psi_q)$  is constructed from  $(S_\tau, \psi_\tau)$  and then used to synthesize a *symbolic controller*  $C_q$  solving  $(S_q, \psi_q)$ . Here,  $S_q$  is the *symbolic model* of  $S_\tau$ , as introduced in (2.3.3), and it is related to  $S_\tau$  via some FRR  $Q$ . Specification  $\psi_q$  is constructed by applying  $Q$  as a static map to all sequences of  $\psi_\tau$ . Formally, given  $\psi_\tau$ , we define  $\psi_q$  as follows:

$$\psi_q := \{\bar{s} \in \Gamma_{S_q} \mid \exists s \in \psi_\tau \forall i \in \{0, 1, \dots, |s| - 1\} (\bar{s}_i = Q(s_i))\}. \quad (2.4.1)$$

Now, since  $S_q$  is a finite-state system, controller  $C_q$  can be simply synthesized to solve the symbolic control problem  $(S_q, \psi_q)$  using techniques from computer science [Tho95, Tab09, EC82, GLPN93, PR89, Var95, BJP<sup>+</sup>12]. This includes fixed-point operations on  $S_q$ , searching on the graph constructed from the states of  $S_q$  as its vertices and the transition as its edges, and two-player games. Synthesis based on fixed-point operations is the state-of-the-art approach already implemented in almost all tools of symbolic control. Refer to the discussion in Chapter 1 for further details.

For the sake of demonstration, we introduce the process of synthesis of symbolic controllers using fixed-point operations. Given symbolic model  $S_q$ , we define the controllable predecessor map  $CPre^{T_q} : 2^{X_q \times U_q} \rightarrow 2^{X_q \times U_q}$  for  $Z \subseteq X_q \times U_q$  by:

$$CPre^{T_q}(Z) := \{(x_q, u_q) \in X_q \times U_q \mid \emptyset \neq T_q(x_q, u_q) \subseteq \pi_{X_q}(Z)\}, \quad (2.4.2)$$

where  $\pi_{X_q}(Z) := \{x_q \in X_q \mid \exists u_q \in U_q (u_q, x_q) \in Z\}$ , and  $T_q(x_q, u_q)$  is an interpretation of the transitions set  $T_q$  as a map  $T_q : X_q \times U_q \rightarrow 2^{X_q}$  that evaluates a set of post-states from a state-input pair.  $CPre^{T_q}(Z)$  is informally the set of non-blocking input/state pairs for which all successor states are in the projection of  $Z$  on  $X_q$ .

We consider for example reachability specifications given by the LTL formulae  $\psi = \diamond\phi$ , where  $\phi$  is a propositional formula over a set of atomic propositions  $AP \subseteq 2^{X_q \times U_q}$ .

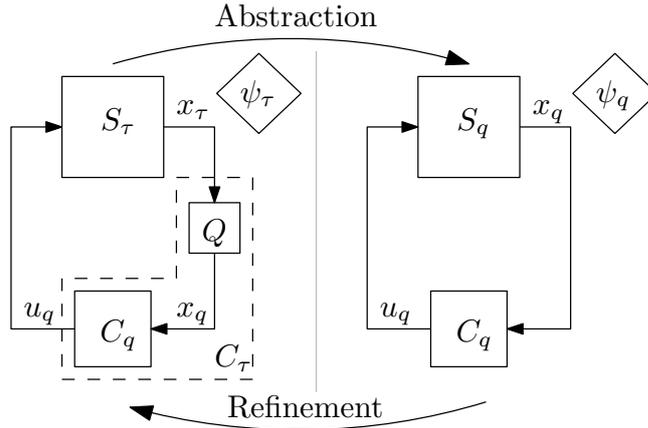
We first construct an initial winning set  $Z_\psi = \{(x_q, u_q) \in X_q \times U_q \mid L(x_q, u_q) \models \phi\}$ , where  $L : X_q \times U_q \rightarrow 2^{AP}$  is some labeling function.

To synthesize symbolic controllers for the selected reachability specifications, we utilize the monotone function

$$\underline{G}(Z) := CPre^{T_q}(Z) \cup Z_\psi \quad (2.4.3)$$

to iteratively solve the fixed-point equation  $Z_\infty = \mu Z. \underline{G}(Z)$  starting with  $Z_0 := \emptyset$ . Here, we adopt a notation from  $\mu$ -calculus with  $\mu$  as the minimal fixed-point operator and  $Z$  is the operated variable. Interested readers can find more details in [MPS95] and the references therein.

The synthesized controller is then a map  $C_q : X_w \rightarrow 2^{U_q}$ , where  $X_w \subseteq X_q$  represents a winning (a.k.a. controllable) set of states. Map  $C_q$  is defined by:  $C(x_q) := \{u_q \in U_q \mid (x_q, u_q) \in \mu^{j(x_q)} Z. \underline{G}(Z)\}$ , where  $j(x_q) := \inf\{i \in \mathbb{N} \mid x_q \in \pi_{X_q}(\mu^i Z. \underline{G}(Z))\}$ , and  $\mu^i Z. \underline{G}(Z)$  represents the value of  $i^{\text{th}}$  iteration of the minimal fixed-point computation.



**Figure 2.1:** Symbolic control as an abstraction-refinement approach.

Later in Chapter 7, we propose another approach for automated synthesis of  $C_q$  using two-player parity games, which allows handling much richer specifications.

Now, we explain the refinement process. We show that FRR preserve the behavioral inclusion from concrete systems to their symbolic models. More specifically, we show in the next result that the behavior of a refined concrete closed-loop  $C_\tau \times S_\tau$  is included in the behavior of its corresponding symbolic closed-loop  $C_q \times S_q$ .

**Theorem 2.4.5.** *Consider systems  $S_\tau$  and  $S_q$  as introduced in (2.3.2) and (2.3.3), respectively. Let  $Q \subseteq X_\tau \times X_q$  be an FRR such that  $S_\tau \preceq_Q S_q$ . Let  $C_q$  be a controller that solves  $(S_q, \psi_q)$ . Then,*

(i)  $(C_q \circ Q)$  is feedback-composable with  $S_\tau$ ; and

(ii)  $B((C_q \circ Q) \times S_\tau) \subseteq B(C_q \times S_q)$ ;

*Proof.* The proof is given in [RWR17]. □

Figure 2.1 provides an illustration for the synthesis and refinement of symbolic controllers for control systems. Systems are represented by rectangles, specifications by diamonds and interconnections by arrows. The refined controller is the symbolic controller serially composed after map  $Q$ , i.e.  $C_\tau = Q \circ C_q$  [RWR17].

## 2.5 Limitations of Current Symbolic Control Techniques

Symbolic control is a promising approach for automated synthesis of controllers for CPS. Yet, it is not used in practice due to some issues that limit it to small-sized control systems. Here, we discuss the limitations of symbolic control that hinder applying it to real-world applications.

### 2.5.1 The Curse of Dimensionality

---

**Algorithm 1:** A skeleton algorithm for constructing symbolic model  $S_q$ .

---

**Input:**  $S_\tau$ ,  $X_q$  and  $U_q$   
**Output:**  $S_q$

```

1  $T_q \leftarrow \emptyset$ ;
2 for all  $x_q \in X_q$  do
3   for all  $u_q \in U_q$  do
4      $X'_q \leftarrow \{x'_q \in X_q \mid \text{Post}_{u_q}^{S_\tau}(x_q) \cap x'_q \neq \emptyset\}$ ;
5     for all  $x'_q \in X'_q$  do
6        $T_q \leftarrow T_q \cup \{(x_q, u_q, x'_q)\}$ ;
7     end
8   end
9 end
10  $S_q \leftarrow (X_q, U_q, T_q)$ ;
```

---

Algorithm 1 shows a traditional implementation for constructing the symbolic model  $S_q$ . To show how the complexity of Algorithm 1 is sensitive to the dimensionality of  $\Sigma$ , we assume for the sake of simplicity that  $X_q$  is a partition of  $X_\tau$  constructed by a set of *hyper-cubes*. The following theorem shows the complexity of Algorithm 1.

**Theorem 2.5.1.** *The complexity of Algorithm 1 is  $O(N_X^{2n}|U_q|)$ , where  $N_X \in \mathbb{N}$ .*

*Proof.* Knowing that for any  $(x_q, u_q) \in X_q \times U_q$  the set of post states can be  $X_q$  (e.g.,  $f$  is a fast-exploding Ordinary Differential Equation (ODE)), the complexity of Algorithm 1 is seen initially as  $O(|X_q| \times |U_q| \times |X_q|)$ . Now, having hyper-cubes as elements of  $X_q$ , we conclude that each state variable (a.k.a. dimension) of  $X_\tau$  is quantized equally. Assume that the number of quantization steps for each variable of  $X_\tau$  is  $N_X$ . We then have that  $|X_q| = N_X^n$  where  $n$  is the number of state variables of  $\Sigma$ . Consequently, the complexity of Algorithm 1 is seen as  $O(N_X^{2n}|U_q|)$ .  $\square$

Theorem 2.5.1 presents formally what is known as the CoD. More precisely, the complexity of constructing symbolic model  $S_q$  is exponentially sensitive to number of state variables (a.k.a. dimensionality) of  $\Sigma$ .

The synthesis of symbolic controller is also sensitive to the dimensionality of  $\Sigma$ . To synthesize controller  $C_q$ , there exist however several techniques such as fixed-point operations and search on graphs. Each technique would require separate complexity analysis. Additionally, the complexity of each technique is sensitive to which specification  $\psi_\tau$  is being considered. For example, it is known that the complexity of controller synthesis for GR(1) specifications (a fragment of LTL) is  $O(wN_X^{2n})$  [MR15], where  $w$  is computed from the number of goals in the specification. Later in Chapter 4, we show one concrete example algorithm for automated synthesis of  $C_q$  using fixed-point operations. We discuss at that point the complexity of this specific implementation of controller synthesis.

**Table 2.1:** Examples of simple specifications  $\psi$ , their compact LTL representation, and their corresponding  $\mu$ -calculus equations.

$\psi$ as LTL	$\mu$ -calculus Formula	Notes
$\Box A$	$Z_\infty = \nu Z^\nu.(\text{CPre}(Z^\nu) \cap Z_\psi)$	Safety
$\Diamond A$	$Z_\infty = \mu Z^\mu.(\text{CPre}(Z^\mu) \cup Z_\psi)$	Reachability
$\Box \Diamond A$	$Z_\infty = \mu Z^\mu.(\text{CPre}(Z^\mu) \cap (\nu Z^\nu.(\text{CPre}(Z^\nu) \cup Z_\psi)))$	Persistence
$\Diamond \Box A$	$Z_\infty = \nu Z^\nu.(\text{CPre}(Z^\nu) \cup (\mu Z^\mu.(\text{CPre}(Z^\mu) \cap Z_\psi)))$	Recurrence

## 2.5.2 Impractical Specifications

The set of design requirements supported by current symbolic control techniques and the software tools implementing them are unfortunately limited and impractical. All existing tools of symbolic control [MDT10, WTO<sup>+</sup>11, MGG13, RZ16, LL18] rely mainly on computing the symbolic controllers by computing fixed-point operations on the graphs representing the symbolic models. The fixed-point operations are mainly deduced from  $\mu$ -calculus equations encapsulating the given high-level specifications.

Table 2.1 shows some common simple specifications  $\psi$  supported by these tools, their compact LTL representation and their corresponding  $\mu$ -calculus equations. See how a simple change to the specifications (e.g., from Safety to Persistence) results in a much complex and nested  $\mu$ -calculus expression. Nested  $\mu$ -calculus expressions result in practice in nested fixed-point operations. Unfortunately, it is impractical, complex, and tedious to represent high-level design requirements as  $\mu$ -calculus. Consequently, most of these tools support only the simplest of these specifications, namely, Safety and Reachability.

Only tools *Pessoa* and *SCOTS* provide general algorithms for the computation of minimal and maximal fixed-points allowing them to cover wider ranges of specifications. However, using them to design controllers for complex design requirements (e.g., those described with complex LTL formulae) is not practical due to the following two main reasons: (1) users have to analytically convert the LTL requirements as  $\mu$ -calculus expressions before implementing them manually in these tools, a task that is highly error-prone and provides no way to check its correctness, and (2) to extract the resulting dynamic controllers, users have to manually define custom data structures and algorithms to extract the controllers from subsets resulting from the computations of minimal and maximal fixed-points. Notice that such ad-hoc controller extraction and deployment are not formally-verifiable. Consequently, they negatively affect the correctness guarantees obtained from the symbolic control techniques.

A workaround to lift such burdens from the users is to implement an automatic translator from LTL to  $\mu$ -calculus. This is also not practical since the translation process is complex and the size of  $\mu$ -calculus expressions grows dramatically with respect to the size of the input LTL expression. Additionally, converting the LTL expressions to fixed-point operations may require dynamic code generation and run-time compilations which increase the complexity of the approach.

### 2.5.3 Ad-Hoc Deployments

The resulting symbolic controllers from the synthesis phase come in form of either static symbolic controllers or dynamic ones. All the tools of symbolic control produce such controllers in their raw forms. For example, tool **SCOTS** generates Binary Decision Diagrams (BDDs) encoding the synthesized static symbolic controllers. Users are then left to handle the actual deployments of the resulting controllers. Here, deployment means taking the resulting controller  $C_q$ , representing it as software or hardware, choosing suitable interfaces, and constructing the closed loop  $C_q \times \Sigma$ . The deployment process becomes more complex for the case of dynamic controllers. We discuss static and dynamic controllers and their representations in Chapter 8.

Such ad-hoc deployments implemented by the users do usually destroy the correctness-guarantees resulting from the symbolic control approach due to one or more of the following reasons:

- although the controllers are correct, their deployments may not be,
- the deployments may rely on other third-party libraries to interpret the raw representations of the controllers, which can be faulty or not verifiable, and
- the software and hardware components used to represent the various parts of the deployments may not be compatible and may fail during the runtime.

## 2.6 Summary

We first introduced some notations used in this thesis. We then introduced a general mathematical framework used throughout this thesis to describe concrete systems, symbolic models, symbolic controllers, and their interconnections.

Symbolic control is introduced as an approach for the algorithmic construction of formally-verified controllers for control systems. Control systems are approximated as symbolic models using FRRs and then digital controllers are algorithmically constructed for them to satisfy some given high-level specifications.

We showed in Theorem 2.4.5 that the controllers synthesized for the symbolic models can be refined to enforce the given specifications on the original systems.

Finally, we discussed the limitations of all current symbolic control techniques. We started with the CoD and showed that the algorithms used traditionally for constructing symbolic models suffer exponential time complexity with respect to the system's dimensionality. Then, we showed that the set of design requirements supported by current symbolic control techniques and the software tools implementing them are limited and impractical. We also showed that current ad-hoc deployments of symbolic controllers usually result in ruining the correctness guarantees obtained from the symbolic control approaches.

## 3 A Framework for Designing Efficient Algorithms of Symbolic Control

The correctness of control software in many safety-critical CPS applications such as autonomous vehicles and traffic networks is crucial. Having models of physical systems and high-level requirements, symbolic control (see Chapter 2) can be leveraged to provide, algorithmically, certifiable control software from given high-level specifications. However, the complexity of applying symbolic control to real-world applications is high and grows exponentially in the number of state variables of the physical systems. On the other hand, if distributed implementations are considered, HPC platforms, such as Many-core systems, clusters of computers, and Cloud-computing<sup>1</sup> platforms can be utilized to mitigate the effects of the CoD.

Since one of the main goals of this thesis is to provide solutions to the complexity problems of symbolic control, several scalable and parallelized algorithms will be introduced throughout the next chapters. Before we start introducing the algorithms, we present a framework that can host and facilitate their implementations.

In this chapter, we present an acceleration framework (we call it **pFaces** [KZ19]) that will be used throughout the thesis to design and implement several parallel algorithms for symbolic control. **pFaces** leverages computing resources, locally or in Cloud-computing platforms, to facilitate designing and implementing scalable parallel algorithms. We present the internal structure and workflow inside **pFaces**. We also introduce an installation of **pFaces** in Amazon Web Services (AWS) with a web-based interface to help deploy and run the designed parallel algorithms.

### 3.1 High Performance Computing (HPC)

Traditionally, compute platforms such as super-computers, which are able to solve complex computing problems, were expensive and inaccessible to many scientific communities. Motivated by the market (e.g., gaming and mining of crypto-currencies), compute-devices such as GPUs showed remarkable improvements in speed and usability. This introduced new paradigms of general-purpose computing like General Purpose GPU (GPGPU) to utilize such devices for scientific data-parallel tasks. One good example is how GPUs played a major role in crunching data collected by the LIGO observatories, in 2015, making the detection of gravitational waves possible [NVI]. Recently, Cloud-computing providers, like Amazon and Microsoft, made it possible for customers to build clusters combining CPUs, GPUs, and HWAs for general-purpose computing.

---

<sup>1</sup>Cloud-computing is a recent computing paradigm that provides public access to shared pools of configurable HPC resources along with high-level services to utilize them, all over the Internet.

Here, it is also worth mentioning that HPC platforms are mainly concerned with maximizing the performance of two main types of parallel computing: data-parallelism and task-parallelism. Data-parallelism is the simultaneous execution of the same function using multiple Processing Elements (PEs) across the elements of a dataset. Task-parallelism, on the other hand, is the simultaneous execution of different functions using multiple PEs across the same or different datasets.

Most of the current implementations of symbolic control techniques are unfortunately developed for serial execution (i.e., to run sequentially within one PE). They do not benefit from available computing resources containing large number of PEs, such as Many-core CPUs, GPUs, HWAs, clusters of computers, and Cloud-computing platforms. These unutilized resources can be leveraged for accelerating symbolic control techniques and to mitigate the effects of their state-explosion problem.

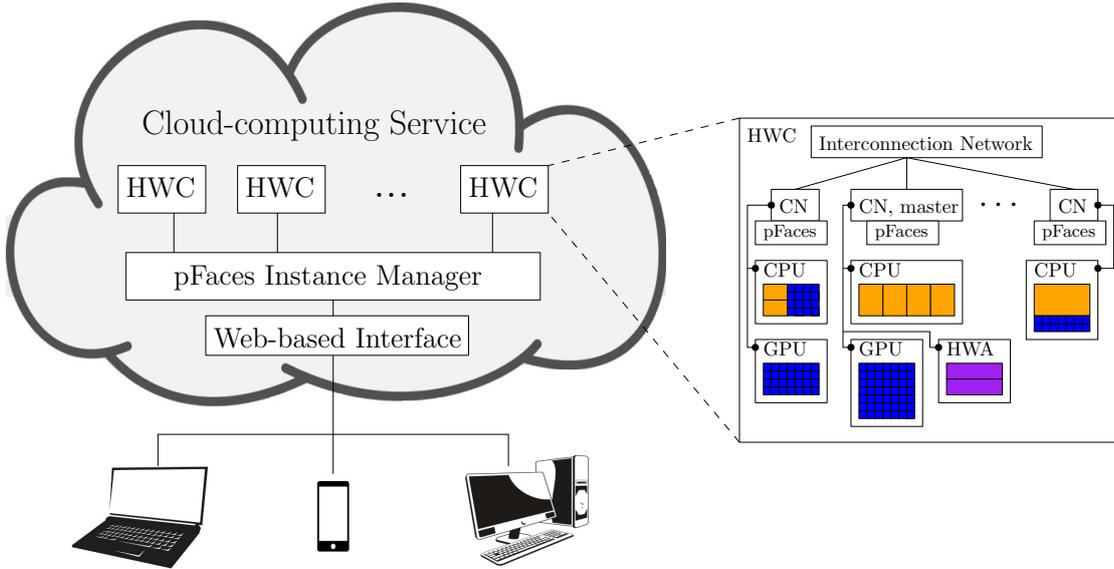
Tool **pFaces** [KZ19] is introduced as an acceleration ecosystem that facilitates utilizing HPC platforms for designing and implementing efficient parallel algorithms. It provides explicit support to design parallel algorithms for formal synthesis and verification techniques. Generally, **pFaces** serves as an interface between the user (e.g., an engineer developing parallel algorithms for CPS-based applications) and heterogeneous HPC platforms.

In the next subsections, we present the internal structure and the workflow of **pFaces**. Additionally, we illustrate how **pFaces** can be deployed in AWS, one of the commonly used Cloud-computing services that provide access to diverse computing platforms. The implementation includes a web-based interface that simplifies the development of parallel algorithms and allows running them remotely. In next chapters, we utilize **pFaces** to implement efficient algorithms for symbolic control.

## 3.2 Ecosystem for Parallel Computing

**pFaces** is designed to support heterogeneous computing platforms that are available locally or in Cloud-computing platforms. More specifically, **pFaces** targets Hardware Configurations (HWCs) similar to the ones depicted in Figure 3.1. An HWC is an HPC system generalized by the following top-bottom hierarchy:

- A network that has at least one Compute Node (CN). We sometimes refer to such network as compute cluster.
- A CN is a device that hosts at least one Compute Unit (CU). An example for a CN is a Personal Computer (PC), a Workstation, or machine instance in AWS. CNs have memories that are usually shared among all of their CUs. Each CN runs an operating system (e.g., Linux) that manages its CUs.
- A CU is a compute-device of particular type. For example, a GPU from a specific vendor is a CU. A Field Programmable Gate Array (FPGA) card is an additional example for CUs. At least one CU in each CN should be a CPU, and we call it the host-CPU. CUs have their own memory spaces, and they can access the memories of their hosting CNs.



**Figure 3.1:** An example Cloud-based deployment of pFaces.

- PEs represent the Hardware (HW) circuits doing computations (mathematical, logical and memory access operations). CPUs have a few number of PEs (a.k.a. cores), but they are able to do fast mathematical computations. GPUs have less powerful PEs, but they come in a very large number. For example, the NVIDIA GEFORCE RTX 3090 GPU has 10496 PEs. PEs of re-configurable HWAs like FPGAs have customizable HW circuits. Developers can construct the computation HW that perfectly match their needs.
- A *thread* is a sequence of compiled software codes that run inside a PE.

pFaces allows parallel algorithms to benefit from all available PEs in any heterogeneous HWC. Cloud-computing platforms provides access to HWCs that vary in computing power and pFaces can benefit from this to facilitate the design and implementation of efficient parallel algorithms for symbolic control techniques. In Figure 3.1, we also propose a Cloud-based deployment of pFaces. The system is initiated by users running terminal platforms (e.g., a cell phone with a web-browser). They access a web-based interface to select or configure HWCs that fit their needs. Then, they develop and run parallel algorithms using a browser-friendly Integrated Development Environment (IDE).

Figure 3.2 shows a screenshot of the web-based interface of pFaces. The figure shows the Terminal tab of the interface where users run three example commands: (1) `login`, a command to check-in in target HWCs, (2) `list -hwcs`, a command to show the attached HWCs, and (3) `list -devices`, a command to list available CNs/CUs in a specific HWC. The implementation presented in Figures 3.1 and 3.2 is deployed in AWS. It can however be deployed in any Cloud-computing platform or run locally.

### 3 A Framework for Designing Efficient Algorithms of Symbolic Control



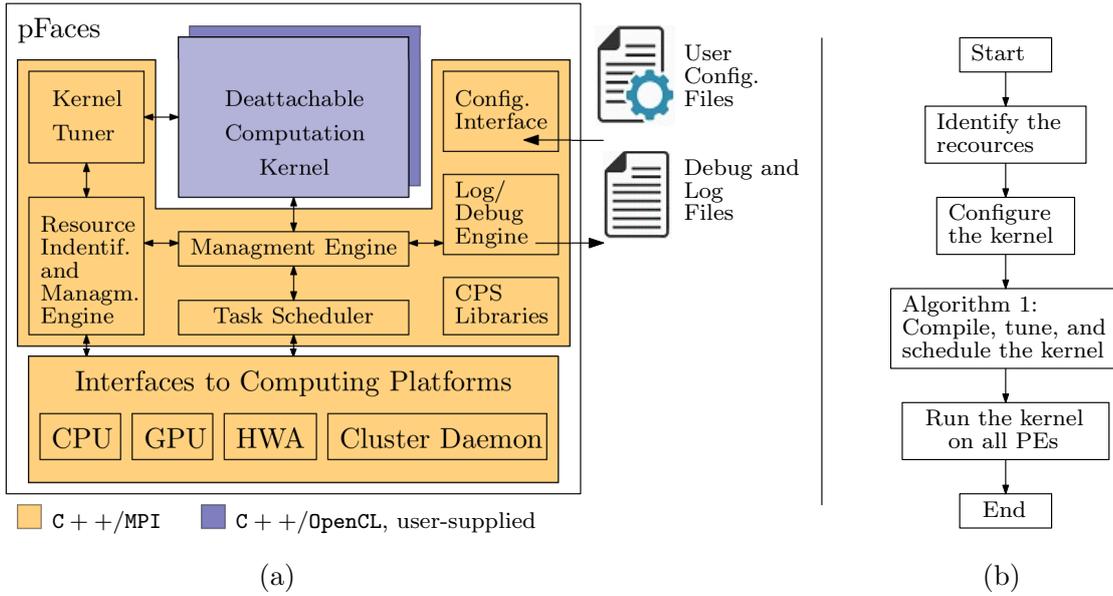
Figure 3.2: Screenshot from the web-based interface of pFaces.

Under the hood, an *instance manager* receives requests from different users and orchestrates the access to the HWCs. This includes launching, stopping, running programs inside, and collecting data from those HWCs. Finally, at the deepest-level inside this stack, instances of pFaces software are ready to accept and accelerate the provided parallel computing tasks, using all available PEs of the targeted HWC. In the next section, we discuss, with more details, the operation of pFaces within one HWC.

### 3.3 Hardware Configuration (HWC)-Level and Compute Node (CN)-Level Accelerations

We focus on one HWC that is deployed locally or in a Cloud-computing service. pFaces is assumed to be installed in all the CNs of the HWC and that one of the installations is designated as a master CN. When a job that spans all CNs in the HWC is requested, the master pFaces is responsible for distributing it to its CN and all other CN. This is achieved using a Message Passing Interface (MPI) protocol implemented by a library used by each pFaces installation within each CN. The master pFaces uses the library to communicate with other pFaces installations in order to coordinate the task scheduling on the HWC level. For the rest of this section, we discuss how pFaces works within one of the CNs, and how it identifies and utilizes the CUs attached to the targeted HWC.

### 3.3 Hardware Configuration (HWC)-Level and Compute Node (CN)-Level Accelerations



**Figure 3.3:** (a) Internal structure of pFaces. (b) General workflow inside pFaces.

#### 3.3.1 Internal Design Structure

Figure 3.3-(a) depicts the internal structure of pFaces. It consists of two main parts: (1) the exchangeable kernel, and (2) the management system. In order to allow for development of different parallel algorithms, pFaces implements the core computing parts as exchangeable kernels (shown in purple color). The user is responsible for supplying a kernel to pFaces, and the management system of pFaces is responsible for accelerating it.

A kernel is a compiled program that is executed by pFaces. Such modular design allows replacing the computation parts, when needed, with ones matching the problem under consideration. The kernels in pFaces are developed in two languages: C++ and OpenCL. OpenCL is a C-like standard for parallel computing. It is used to describe data-parallel tasks that should run on all the PEs belonging to a CN.

A kernel driver is a collection of serial instructions written in C++ that provides an execution plan that pFaces must follow to run the kernel. Each instruction of the plan belongs to one of three main categories: (1) serial code instructions, (2) parallel code instructions, and (3) management instructions. Serial code instructions request execution of codes that are written as C++ functions. Parallel code instructions request execution of codes written in OpenCL. Management instructions are used to reserve memory, to describe the code logic (i.e., what order code instructions should follow), to synchronize the parallel execution within all PEs of the CN, and to synchronize the parallel execution within all CNs.

**Remark 3.3.1.** *Using existing OpenCL codes in pFaces is readily doable. Users only need to develop the kernel driver (i.e., the execution plan) that guides pFaces to work with the data-parallel codes provided by the users.*

In comparison with pure OpenCL, the data-parallel codes in pFaces are dynamic. Users can use special keywords in the OpenCL code that are identified by pFaces and can be controlled from the kernel driver. This gives the user the ability to design more flexible codes. It also helps them change parameters in the OpenCL code after loading them from text config files (see Fig. 3.3(a)).

The management system (shown in light orange color) is responsible for accelerating the kernel, and managing the memory and the computing resources. It launches kernels and synchronizes their parallel execution within PEs in the CN. As depicted in Figure 3.3-(a), it consists of different modules, and we discuss them separately in the next subsections.

#### 3.3.2 Resource Management and Kernel Tuning

A module for *resource identification and management* identifies the CUs in the CN and their capabilities. This includes, for example, their computation power (e.g., processor frequency and number of PEs), their available memory spaces, and their supported communication interfaces. Such information is stored in a database for later use during task scheduling and execution.

A *kernel tuner* module executes parts of the kernel in order to identify near-accurate information about the computing power of available CUs. Such information, which is in the form of time measurements, helps other modules to distribute the tasks efficiently over all PEs. By this, pFaces makes sure that all PEs are fairly utilized during the run-time of the kernel.

#### 3.3.3 Managing Computation and Memory Resources

The *management engine* module uses the collected data from the modules in the previous subsection to decide how the kernels are scheduled. Since it is aware of their memory resources (as collected from the *resource identification and management* module) and their computation power (as collected from the *kernel tuner* module), it automatically, and efficiently, decides which balanced execution the kernels should undergo in the PEs. This is mainly achieved by assigning fair subsets of the input data to different PEs. In Subsection 3.4, we provide more details about the scheduling of tasks in pFaces.

Having a near-optimal task distribution, the management engine en-queues the kernel for actual execution using the *task scheduling* module. The *task scheduling* module manages the en-queued tasks for parallel execution. It reserves memory spaces on each CU, sends tasks for execution, collects results, and copies data between CUs when needed.

The *management engine* is also responsible for running the instructions supplied by the user in the kernel driver. Serial code instructions are dispatched for execution within the host-CPU. Parallel code instructions launch parallel codes in all available PEs as low-level threads. The number of low-level threads in each PE is assigned based on

### 3.3 Hardware Configuration (HWC)-Level and Compute Node (CN)-Level Accelerations

the decisions from the *task scheduling* module. To synchronize the execution among various low-level threads, **pFaces** offers various instructions for code synchronization. They range from instructions for low-level synchronization between PEs to instructions for balancing the execution among CNs over the network, using MPI messages.

#### 3.3.4 Modules for Supporting Kernel Development

As users develop their kernels, they can launch them with different configuration parameters. Such parameters are placed in configuration text files. The users can specify new configuration parameters to be included in the configuration files. New parameters can, for example, be used to set some variables in the developed kernel before launching it. A *configuration interface* module reads and parses these configuration files and apply them when requested to the kernel.

A *logging and debugging* module reports information collected from the management system and the kernel. This includes: (1) suggestions for improving the kernel instructions to make it more efficient; (2) a task distribution report showing how the kernel is distributed among the targeted CUs; (3) a detailed memory report about the reserved memory buffers; (4) a detailed compilation report for each CUs that allows refining the warnings/bugs; and (5) messages regarding the progress of the kernel during the parallel execution.

**pFaces** also generates several reports to assist the developers. A task allocation report shows how low-level threads are distributed among the CUs. A memory allocation report shows, per memory buffer, the allocated memory. In case multiple CUs are attached to the CN, **pFaces** constructs sub-buffers in the CUs and generates a sub-buffering report. Finally, upon the completion of execution, **pFaces** reports the time spent to execute parallel and serial codes, and the time spent to access the memory.

#### 3.3.5 Supporting Symbolic Control Approaches

**pFaces** provides additional helper libraries and modules to facilitate the development of kernels symbolic control techniques. This includes:

1. libraries for managing data structures used in those domains, such as raw-data, BDDs, bitmaps and sparse-matrices,
2. a library for parsing mathematical expressions,
3. a library for symbolic mathematics,
5. a library for code generation (C++ and VHDL/Verilog),
4. an interface with MATLAB/Simulink for accessing the generated files, and
5. a library for developing Representational State Transfer (RESTful) web services as web-based interfaces to the kernels.

### 3.4 Workflow of Kernels

Having PEs belonging to CUs of different types, **pFaces** relies on the following two assumptions to ensure that a heterogeneous parallelization is successful:

- (1) the kernels' parallel codes follows the **OpenCL** standard, and
- (2) each vendor (e.g., Intel for CPUs, NVIDIA for GPUs, and Intel-Altera for FPGAs) provides an OpenCL-compiler that produces binaries (resp. HW circuit in case of HWAs) from the kernels' parallel codes.

Now, using different compiled binaries of the same kernel, **pFaces** asks different CUs to execute the same kernel in parallel. The same applies to all neighboring CNs in the HWC. The master **pFaces** installation ensures that all CN share the same kernel and any required configuration files.

Users need then to represent the parallelizable parts of their algorithms as data-parallel codes following the OpenCL standard. Data is here assumed to be big chunks of one-, two-, or three-dimensional memory spaces. **pFaces** then schedules, in a computationally balanced manner, PEs to run the binaries on disjoint subsets of the data spaces. Users may aid **pFaces** in such scheduling, or it can be done completely automatic from the collected tuning data.

Figure 3.3-(b) depicts the general workflow inside **pFaces**. In the beginning, **pFaces** identifies the resources available CUs and memories in the current CN. Then, it reads any provided configuration file and re-configures the kernel based on it. Finally, **pFaces** compiles, tunes and schedules the kernel.

### 3.5 A Cloud-Ready Installation

Algorithm 2 gives more insights about the compilation of kernels and the tune-based scheduling. Function `ScheduleCUThreads` uses the collected tune data  $(t_1, t_2, \dots, t_D)$  and the stored resource-information to give a decision regarding the number of low-level threads  $\hat{N}_d$  that should be assigned to any CU with an index  $d \in D$ . Developers can override the function and assign fixed distribution of low-level threads. Finally, the kernel is sent to each CU for parallel execution.

The general Cloud-based architecture presented in Figure 3.1 is used to deploy **pFaces** on AWS. We provide more details about the implementation within such specific Cloud-computing service. An HWC is one or more (networked) instances deployed using the AWS-Elastic Computing (EC2) service (a service for hosting virtual private machines). **pFaces** is installed in all instances and it is managed through one Amazon Machine Image (AMI) configured with all required libraries. The service AWS-CloudFormation is used to template and deploy the HWC. The **pFaces instance manager** is developed in ASP.NET (a programming language for web application development) and deployed within one EC2-instance and it uses the AWS Application Programming Interface (API) to manage the HWCs. The *web-based interface* is developed on top of Eclipse Orion

---

**Algorithm 2:** Kernel compilation and tune-based task scheduling.

---

**Input:**  $K$ : the kernel under consideration,  $N$ : number of low-level threads,  $D$ : a list of indices of targeted CUs.  
**Parameters:**  $\tilde{N}$ : Constant number of threads for tuning.  
**Output:**  $\hat{N}_1, \hat{N}_2, \dots, \hat{N}_D$ : per-CU number of low-level threads.

```

1 for  $d \in D$  do
2   if  $K$  is not compiled for  $CU_d$  then
3     |  $b_d := \text{Compile}(CU_d, K)$ ; ▷ Per-CU compilation
4   else
5     |  $b_d := \text{LoadBinaries}(K)$ ; ▷ Load pre-compiled binaries
6   end
7   if  $K$  is not tuned for  $CU_d$  then
8     |  $t_d := \text{KernelTuner.Run}(b_d, \tilde{N})$ ; ▷ Run for fixed threads
9   else
10    |  $t_d := \text{LoadTuneData}(K)$ ; ▷ Load tune data
11  end
12 end
13 for  $d \in D$  do
14  |  $\hat{N}_d := \text{ScheduleCUThreads}(N, d, t_1, t_2, \dots, t_D)$ ;
15 end

```

---

and it is deployed as an independent web-server in one EC2-instance using the AWS-ElasticBeanstalk service. Databases for the *web-based interface* and *instance manager* are managed by Microsoft SQL Servers through the AWS-Relational Database Service (RDS). Finally, a Domain Name System (DNS)-lookup-table and a domain-name are managed through the AWS-Route53 service.

### 3.6 Summary

We presented **pFaces** as an acceleration framework that will be used throughout the thesis to design and implement several parallel algorithms for symbolic control. **pFaces** leverages computing resources, locally or in Cloud-computing platforms, to facilitate designing and implementing scalable parallel algorithms. It allows parallel algorithms to benefit from all available PEs in any heterogeneous HWC.

The internal structure of **pFaces** was introduced. It consists of two main parts: (1) the exchangeable kernel, and (2) the management system. A kernel is a compiled program that is executed by **pFaces**. The management system, on the other hand, is responsible for running the kernel, and managing its memory and computing resources.

The workflow in **pFaces** was also introduced. Users need to represent the parallelizable parts of their algorithms as data-parallel and serial codes as kernels acceptable by **pFaces**. **pFaces** then launches the kernel and synchronizes its parallel execution within all available PEs.

### *3 A Framework for Designing Efficient Algorithms of Symbolic Control*

An example Cloud-based installation of **pFaces** on Amazon AWS was also presented. An HWC here is one or more (networked) instances deployed using the AWS-EC2 service. **pFaces** was then installed in all instances and it is managed through one AMI configured with all required libraries.

## 4 Efficient Algorithms for Symbolic Control

Current state-of-the-art implementations of symbolic control are designed to run serially in one PE. This way of implementation interacts poorly with the symbolic control approach, whose complexity grows exponentially in the number of state variables in the original systems. The same problem is also observed when original systems are an interconnection of many subsystems, which is always the case in Internet-of-Things (IoT) and networked control systems [KRZ18]. Consequently, such implementations are limited to small dynamical systems where controllers are computed offline. In this chapter, we investigate efficient data-parallel algorithms and distributed data structures to address the computational complexity issue of symbolic control. Using `pFaces`, we design kernels that utilize HPC platforms to mitigate the effects of the state explosion problem that appears in the majority of symbolic control techniques.

### 4.1 Existing Implementations of Symbolic Control

Most of the existing techniques of symbolic control take a monolithic view of systems, where the entire system is modeled, abstracted, and then a controller is synthesized from the overall state-space. Tools like `Pessoa` [MDT10], `Tulip` [WTO<sup>+</sup>11], `CoSyMA` [MGG13], `SCOTS` [RZ16], `SENSE` [KRZ18], `ROCS` [LL18], and `MASCOTS` [HMMS18b] consider a monolithic view of systems where the original system is abstracted as a whole. Consequently, they severely suffer from the CoD. With respect to their implementations, they are all designed to run serially in one CPU. On the other hand, the process of constructing the abstractions is inherently parallelizable. The states of symbolic models can be constructed independently and in parallel. Also, the process of automated controller synthesis can be partially parallelized. This makes symbolic control in general parallelizable.

Surprisingly, most modern CPUs come equipped with multiple cores and internal GPUs that never get utilized by serial programs. For example, the Intel Core i5 6200U processor has two CPU cores and an internal GPU that can outperform similar single-core CPUs if a data-parallel task is well implemented. Now, having any of those existing tools run in this processor, might not utilize the second CPU core and will never utilize the internal GPU, which is a waste of resources. Parallelizing the phases of abstraction construction and controller synthesis, while considering HPC platforms as computing platforms, can help control the computational complexity of symbolic control.

In the next subsections, we redesign the algorithms of the symbolic control technique used in [MDT10, MGG13, RZ16, HMMS18b], implement them using `pFaces` and compare them with existing tools. To be implemented in `pFaces`, the two main phases in

symbolic control (i.e., constructing the symbolic model and synthesizing the controllers) must be parallelized.

## 4.2 Data-Parallel Algorithms for Symbolic Control

We consider symbolic model constructed based on the theory in [RWR17], which utilizes a growth-bound formula as an Over-Approximation of the Reachable Set (OARS). Algorithmic controller synthesis is done based on the technique presented in [RZ16], which uses fixed-point computations on the constructed symbolic models. We refer to this technique on symbolic control as the Growth-Bound and Fixed-Point (GBFP).

### 4.2.1 Data-Parallel Construction of Symbolic Models

As declared earlier in Chapter 2, we consider general nonlinear continuous-time systems as control systems  $\Sigma$ . Let us be more specific here and provide a definition for map  $f$  in  $\Sigma$ . Consider the following control system  $\Sigma$  given in the form of a differential equation:

$$\dot{x}(t) = f(x(t), u), \quad (4.2.1)$$

where  $x(t) \in X_\tau \subseteq \mathbb{R}^n$  is the state vector and  $u \in U_\tau \subseteq \mathbb{R}^m$  is the input vector. Consequently, the considered concrete system is  $S_\tau := (X_\tau, U_\tau, T_\tau)$ , where  $T_\tau$  is as defined in (2.3.2).

For the construction of  $S_q$ , we consider set  $X_q$  as a finite partition on  $X_\tau$  constructed by a set of hyper-rectangles of identical widths  $\eta \in \mathbb{R}_+^n$ , and set  $U_q$  as a finite subset of  $U_\tau$ . The symbolic model is then the finite-state system  $S_q := (X_q, U_q, T_q)$  as introduced in (2.3.3). In Theorem 2.3.4, we showed that there exist an FRR relating  $S_q$  to  $S_\tau$ . An FRR here is simply the equivalence relation used to construct the partition  $X_q$  of  $X_\tau$ . We denote such FRR by  $Q \subseteq X_\tau \times X_q$ .

We now discuss in details how the construction of  $S_q$  is implemented. A discussion of an algorithm that constructs  $X_q$  and  $U_q$  is omitted since they are simply constructed via the quantization of their corresponding spaces (i.e.,  $X_\tau$  and  $U_\tau$ ). For interested readers, an example implementation for the quantization algorithms are given in [RZ16]. We focus here on the construction of  $T_q$  since it is the part that suffers from the CoD, as discussed earlier in Section 2.5.1.

For the vector field of (4.2.1), a function  $\Omega^f : X_q \times U_q \rightarrow X_\tau^2$  characterizes the over-approximations of the reachable sets starting from symbolic state  $x_q \in X_q$  when an input  $u_q \in U_q$  is applied. For example, if the growth-bound map ( $\beta : \mathbb{R}^n \times U_\tau \rightarrow \mathbb{R}^n$ ) introduced in [RWR17] is used,  $\Omega^f$  can be defined as follows:  $\Omega^f(x_q, u_q) = (x_{lb}, x_{ub}) := (-r + \xi_{x_c, u_q}(\tau), r + \xi_{x_c, u_q}(\tau))$ , where  $r = \beta(\eta/2, u)$ , and  $x_c \in x_q$  denotes the center of  $x_q$ . An over approximation of the reachable sets can then be obtained by the map

$$O^f : X_q \times U_q \rightarrow 2^{X_q}, \quad (4.2.2)$$

defined by  $O^f(x_q, u_q) := \mathcal{Q} \circ \Omega^f(x_q, u_q)$ , where  $\mathcal{Q}$  is a quantization map defined by:

$$\mathcal{Q}(x_{lb}, x_{ub}) := \{x'_q \in X_q \mid x'_q \cap \llbracket x_{lb}, x_{ub} \rrbracket \neq \emptyset\}. \quad (4.2.3)$$

**Remark 4.2.1.** Map  $\mathcal{Q}$  is a cosmetic encapsulation of FRR  $Q$ . While  $Q$  accepts one element  $x \in X_\tau$  as input,  $\mathcal{Q}$  operates on a hyper-rectangle  $\tilde{X} \subseteq X_\tau$  defined by the lower and upper bound points  $x_{lb}$  and  $x_{ub}$ .  $\mathcal{Q}$  then returns all elements  $x'_q \in X_q$  that  $Q$  would return if it is applied iteratively to all points in  $\tilde{X}$ .

---

**Algorithm 3:** Serial algorithm for constructing abstractions.

---

**Input:**  $X_q, U_q, O^f$   
**Output:** A transition relation  $T_q \subseteq X_q \times U_q \times X_q$ .

```

1  $T_q \leftarrow \emptyset$ ;
2 for all  $x_q \in X_q$  do
3   for all  $u_q \in U_q$  do
4     for all  $x'_q \in O^f(x_q, u_q)$  do
5        $T_q \leftarrow T_q \cup \{(x_q, u_q, x'_q)\}$ ;
6     end
7   end
8 end
    
```

---

Algorithm 3 depicts the traditional algorithm for constructing finite abstractions of dynamical systems. It is a more detailed version of Algorithm 1. It constructs, serially,  $T_q \subseteq X_q \times U_q \times X_q$  by iterating over all elements of  $X_q \times U_q$ . For any  $(x_q, u_q) \in X_q \times U_q$ , the evaluation of  $O^f$  and  $T_q|_{(x_q, u_q)}$  is independent of any other elements of  $X_q \times U_q$ . Hence, such evaluation can be implemented completely in parallel. This is an ideal case of data-parallelism.

---

**Algorithm 4:** Proposed data-parallel algorithm for constructing discrete abstractions.

---

**Input:**  $X_q, U_q, \Omega^f$   
**Output:** A characteristic set  $K \subseteq X_q \times U_q \times X_\tau^2$ .

```

1  $K \leftarrow \emptyset$ ;
2 for all  $p \in \{1, 2, \dots, P\}$  do
3    $K_{loc}^p \leftarrow \emptyset$ ;
4 end
5 for all  $(x_q, u_q) \in X_q \times U_q$  in parallel with index  $i$  do
6    $p = I(i)$ ;
7    $(x_{lb}, x_{ub}) \leftarrow \Omega^f(x_q, u_q)$ ;
8    $K_{loc}^p \leftarrow K_{loc}^p \cup \{(x_q, u_q, (x_{lb}, x_{ub}))\}$ ;
9 end
10 for all  $p \in \{1, 2, \dots, P\}$  do
11    $K \leftarrow K \cup K_{loc}^p$ ;
12 end
    
```

---

Algorithm 4 is the proposed parallelization of Algorithm 3 to exploit its inherent data-parallelism. In Algorithm 4, each PE, annotated with an index  $p \in \{1, 2, \dots, P\}$ , where  $P$  is the number of available PEs, handles one  $(x_q, u_q) \in X_q \times U_q$ . Function  $I : \mathbb{N}_+ \setminus \{\infty\} \rightarrow \{1, 2, \dots, P\}$  maps a parallel job (i.e., an iteration of the **parallel for-all** statement) with index  $i$  to a PE with an index  $p = I(i)$ . The algorithm introduces the abstraction task as an ideal data-parallel task with no communication overhead among the processing elements. It is more suitable for CUs with massive number of PEs (e.g. GPUs and super computers).

**Remark 4.2.2.** *Having  $P > |X_q \times U_q|$  is a waste of computation power.*

In Algorithm 4, instead of storing symbolic transitions in  $T_q$ ,  $\Omega^f$  is used to construct a distributed container  $K := K_{loc}^1 \cup K_{loc}^2 \cup \dots \cup K_{loc}^P$ , where the subscript *loc* indicates that  $K_{loc}^p \subseteq X_q \times U_q \times X_\tau^2$  is stored in a local-memory of the PE with index  $p$ . Lines 10-12 in Algorithm 4 are optional and can be omitted if there is no interest to obtain a combined abstraction. We show later in Subsection 4.2.2 that the proposed algorithm for controller synthesis requires only the distributed containers  $K_{loc}^p$  to synthesize symbolic controllers for  $S_q$ .

**Remark 4.2.3.** *Using  $K$  and not  $T_q$  is much more efficient since  $|T_q|$  is sensitive to  $|O^f(x_q, u_q)|$ , while  $|K| = 2n|X_q \times U_q|$  is constant and consumes, practically, less memory. It is efficient when such operations are executed in PEs of GPUs or HWAs known for having limited memory. Later in Subsection 4.2.3, we show that the distributed container  $K_{loc}^p$  can be omitted and the abstraction is done on-the-fly.*

To investigate the complexity of Algorithm 4, we assume for the sake of simplicity that  $X_q$  is a partition of  $X_\tau$  constructed by a set of *hyper-cubes*. The following theorem shows the complexity of Algorithm 4.

**Theorem 4.2.4.** *The complexity of Algorithm 4 is  $O(\frac{N_X^n |U_q|}{P})$ , where  $N_X \in \mathbb{N}$  and  $P$  is a variable number of PEs.*

*Proof.* The proof follows directly the computation of parallel complexity for Parallel Random Access Memory (PRAM) models as introduced in [Jaj92, Chapter 1].  $\square$

**Remark 4.2.5.** *A variant of Algorithm 4, which is more suitable for CUs with small number of fast PEs (e.g., CPUs and FPGAs), can be easily provided by aggregating the computation of all  $(x_q, u_q)$  having the same  $x_q$  in one PE.*

## 4.2.2 Data-Parallel Synthesis of Symbolic Controllers

Let  $S_q := (X_q, U_q, T_q)$  be a symbolic model and recall the procedure of symbolic controller synthesis introduced in Section 2.4.2. Algorithm 5 shows a traditional serial implementation of the minimal fixed-point computation  $Z_\infty = \mu Z.G(Z)$  to synthesize a symbolic controller that enforces reachability specifications. Synthesizing controllers for invariance specifications is much similar and uses a maximal fixed-point computation. Interested readers can find more details in [RZ16].

---

**Algorithm 5:** Traditional serial algorithm to synthesize  $C_q$  enforcing the specification  $\psi := \diamond\phi$ .

---

**Input:** Initial winning domain  $Z_\psi \subset X_q \times U_q$  and  $T_q$   
**Output:** A controller  $C_q : X_w \rightarrow 2^{U_q}$ .

```

1  $Z_\infty \leftarrow \emptyset$  ;
2  $X_w \leftarrow \emptyset$  ;
3 do
4    $Z_0 \leftarrow Z_\infty$  ;
5    $Z_\infty \leftarrow CPre^{T_q}(Z_0) \cup Z_\psi$  ;
6    $D \leftarrow Z_\infty \setminus Z_0$  ;
7   foreach  $x_q \in \pi_{X_q}(D)$  with  $x_q \notin X_w$  do
8      $X_w \leftarrow X_w \cup \{x_q\}$  ;
9      $C_q(x_q) := \{u_q \in U_q \mid (x_q, u_q) \in D\}$  ;
10  end
11 while  $Z_\infty \neq Z_0$  ;
```

---

Algorithm 6 is the proposed parallelization of Algorithm 5. We assume using the same indexing map  $I(\cdot)$  from Algorithm 4. Line (11) corresponds to computing  $T(x_q, u_q)$  from the stored characteristic values  $(x_{lb}, x_{ub})$ . Since all PEs use  $Z_0$  when running lines (12) and (15), the synchronization among all PEs is required to ensure all PEs get the most updated version of  $Z_0$ . Such synchronization happens every iteration of the fixed-point by collecting all local versions  $Z_{loc}^p$  in line (21) and the update in line (4) before starting another parallel synthesis iteration of the **parallel for-loop** in line (9). Similar to the discussion in Remark 4.2.5, a variant of the algorithm, that is more suitable for CPUs and HWAs, is provided in the implementation of the GBFP kernel.

### 4.2.3 Memory-Efficient Kernels for Data-Parallel Symbolic Control

Modern CUs contain hundreds to thousands of PE. This motivates the concept of more-compute/less-memory where recomputing results between repeated iterations is favored over storing them. We apply this by eliminating the use of  $K_{loc}^p$  in line (11) of Algorithm 6 and computing it on the fly using the same way done in lines (7) and (8) of Algorithm 4. We refer to such modified kernel as Memory-Efficient GBFP (MemGBFP).

### 4.2.4 Implementation Details

Figure 4.1 shows, as a flow-chart, the overall execution of the GBFP kernel inside pFaces. The steps (blocks) in the diagram describes each part of the implementation. Blocks in orange correspond to software components of pFaces while those in purple correspond to parts of the GBFP kernel.

pFaces starts reading and applying the configuration files describing the control problem. A configuration file for the symbolic control kernel should follow a template text file. This includes configurations such as the dynamics of the physical systems in the form of

---

**Algorithm 6:** Proposed parallel algorithm to synthesize  $C_q$  enforcing the specification  $\psi := \diamond\phi$ .

---

**Input:** Initial winning domain  $Z_\psi \subset X_q \times U_q$  and  $T_q$   
**Output:** A controller  $C_q : X_w \rightarrow 2^{U_q}$ .

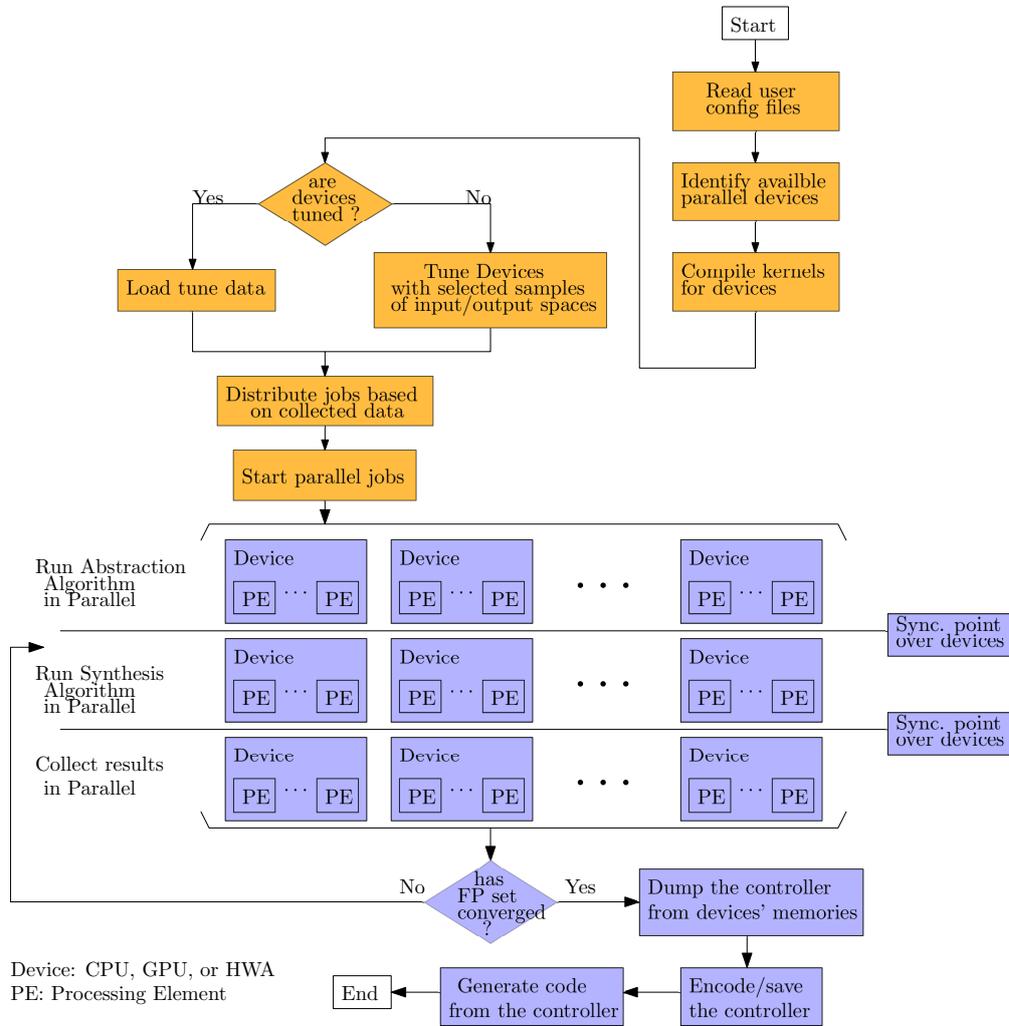
```

1  $Z_\infty \leftarrow \emptyset$ ;
2  $X_w \leftarrow \emptyset$ ;
3 do
4    $Z_0 \leftarrow Z_\infty$ ;
5   for all  $p \in \{1, 2, \dots, P\}$  do
6      $Z_{loc}^p \leftarrow \emptyset$ ;
7      $X_{w,loc}^p \leftarrow \emptyset$ ;
8   end
9   for all  $(x_q, u_q) \in X_q \times U_q$  in parallel with index  $i$  do
10     $p = I(i)$ ;
11     $Posts \leftarrow \mathcal{Q} \circ K_{loc}^p(x_q, u_q)$ ;
12    if  $Posts \subseteq Z_0 \cup Z_\psi$  then
13       $Z_{loc}^p \leftarrow Z_{loc}^p \cup \{(x_q, u_q)\}$ ;
14       $X_{w,loc}^p \leftarrow X_{w,loc}^p \cup \{\hat{x}\}$ ;
15      if  $x_q \notin \pi_{X_q}(Z_0)$  then
16         $C_q(x_q) \leftarrow C(x_q) \cup \{u_q\}$ ;
17      end
18    end
19  end
20  for all  $p \in \{1, 2, \dots, P\}$  do
21     $Z_\infty \leftarrow Z_\infty \cup Z_{loc}^p$ ;
22     $X_w \leftarrow X_w \cup X_{w,loc}^p$ ;
23  end
24 while  $Z_\infty \neq Z_0$ ;

```

---

## 4.2 Data-Parallel Algorithms for Symbolic Control



**Figure 4.1:** Workflow of the symbolic control kernel inside pFaces.

**Table 4.1:** Used HWCs for the case studies in this chapter.

Code	Details	# PEs
CPU1	Intel Core i5-6200U	2
CPU2	Intel Xeon E5-2630	10
CPU3	Intel Xeon E-2666	36
CPU4	Intel Xeon E5-1620 (3.6 GHz)	8
CPU5	Intel Xeon E5-2686 (2.3 GHz)	64
CPU6	Intel Xeon Platinum 8000 (3.6 GHz)	72
GPU1	NVIDIA Quadro P5000	2560
GPU2	NVIDIA Tesla V100	5120
GPU3	AMD Radeon Pro Vega 20	1280
FPGA1	Altera DE5-Net FPGA Board	2
FPGA2	Xilinx Kintex UltraScale FPGA KCU1500	2
MIX1	CPU1 and its internal GPU	24
MIX2	8 × GPU1 with NVLink interconnection	40960
CLS1	Two networked CNs: 32-core CPU and GPU2	5152

**Table 4.2:** Comparison between SCOTS and GBFP for the 2d-Robot example.

HWC	CPU1	CPU2	GPU1	GPU2	GPU3	FPGA1	MIX1	MIX2	CLS1
Time: SCOTS	4423	3949	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Time: pFaces	154	97	8.3	1.85	13.2	147	136	0.309	96.2
Speedup	25x	40x	475x	2134x	299x	26x	29x	12779x	41x

ODEs mixed with C code. The configurations also allow describing the requirements to be enforced on the system by the synthesized controllers.

Then, three main parallel code blocks are developed in the kernel: (1) the parallel algorithm for constructing the abstraction, (2) the parallel algorithm for synthesizing the controller, and (3) a parallel algorithm to collect the data (i.e., the controller). The controller synthesis algorithm is based on a fixed-point operation. This requires a repeated computation of the parallel algorithms till it converges to a fixed-point set-of-states. Some synchronization points are then required to ensure the integrity of data shared among the different CUs (and/or among different CNs in the HWC using MPI messages).

Once the fixed-point iterations settle, the kernel starts to dump the final controller which is distributed on many memory spaces in different CUs (and/or different CNs in the HWC). Controllers are given in the form of Lookup Tables (LUTs) and can be stored in different formats (e.g., BDDs or bitmaps). With one of the libraries shipped in pFaces, C++ and VHDL codes can be also automatically generated. The generated codes are ready for implementation in target (possibly embedded) devices that will control the original system.

In [KZ19], we presented a detailed benchmark for the GBFP kernel covering several case studies and using different HWCs. For the sake of demonstration, we present here a subset of this benchmark. We mainly show a comparison between the implemented

kernel in `pFaces` and tool `SCOTS` [RZ16], which is the current state-of-the-art tool for symbolic controller synthesis. Table 4.2 shows this comparison for an example of a 2-dimensional robot (introduced in [KRZ18, Section 4.2]) whose abstraction has 1364889 symbols. Table 4.1 shows the used HWCs and the total number of PEs in all of their CNs/CUs. HWCs with codes CPU, GPU, or FPGA refer to HWCs with a single CU within a single CN. Each GPU-HWC, or FPGA-HWC, has a host-CPU attached to its CN and it is used for serial code execution. However, since most of the contribution to the speedup comes from the parallel codes, we don't report the details of the host-CPU. A HWC with the code MIX refers to a HWC with a single CN that has multiple CUs with different classes. The HWC CLS1 is a cluster of two CNs. All reported times are in seconds. `SCOTS` can only work with the first two HWCs and when we compute the speedup `pFaces` achieves, we use the fastest result of `SCOTS` (3949 seconds). Speedup is calculated by dividing the time required by `SCOTS` by the time required by `pFaces`.

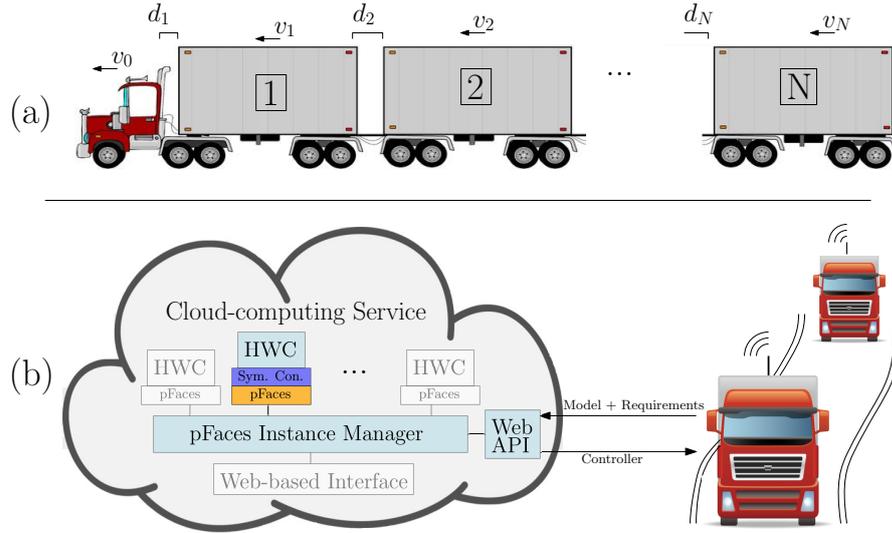
**Remark 4.2.6.** *The GBFP kernel requires intensive synchronization at each fixed-point convergence check. Such synchronization requires repeated sharing of large data blocks between different CNs/CUs. Although GPUs2 and CLS1 has a similar number of PEs, GPUs2 achieves much higher speedup. This is basically due to the time wasted in the synchronization between the two CNs of CLS1 which comes in the form of MPI messages over a slow Ethernet network.*

The results show how controlling the number of PEs can be utilized to reduce the computational complexities of symbolic control. In the next subsection, we leverage the kernel to mitigate the effects of the CoD and deploy it as an online Cloud-computing service.

### 4.2.5 Controlling Time Complexity of Symbolic Control Applications

We show how `pFaces` can be used to mitigate computation complexities resulting from the CoD in symbolic control algorithms. We consider a truck-with-a-trailer system as presented in [RMT13]. It is a three-dimensional system, where each dimension represents a state variable in the ODEs describing the system. More specifically, one state variable is dedicated to the velocity of the truck, another to the distance between the truck and the trailer, and the third for the velocity of the trailer. As depicted in Figure 4.2-(a), we generalize the example for  $N$  trailers. Remark that adding an extra trailer increases the dimension by two.  $v_0$  is the velocity of the truck and  $v_i$ , for  $i \in \{1, 2, 3 \dots N\}$ , is the velocity of trailer  $i$ , while  $d_i$ , for  $i \in \{1, 2, 3 \dots N\}$ , is the distance from trailer  $i$  and the trailer  $i - 1$ .

The requirement in this example is to reach some target speed while maintaining safe distances between the trailers. We require the controller to be computed in real-time with a deadline window of 1.0 second. This allows recomputing and deploying the controller within the deadline window if environment parameters or system parameters change. For different  $N$ , we are concerned with the number of transitions in the symbolic model as it affects directly the complexity of the parallel algorithms.



**Figure 4.2:** (a) A truck with  $N$  trailers. (b) real-time implementation of the kernel.

**Table 4.3:** Details and results for the truck-and-trailers example.

	$N = 1$	$N = 2$	$N = 3$	$N = 4$
Dimension of $\Sigma$	3	5	7	9
$ T_q $ ( $\times 10^6$ )	2.64	36.17	398.29	5520.4
Total memory (M.B.)	63	1414	379	5264
Memory-optimized kernel	No	No	Yes	Yes
HWC	CPU3	GPU2	MIX2	MIX2
Number of PEs	36	5120	40960	40960
Time to find $C_q$ (sec.)	0.42	0.98	0.96	46.6

For the original problem (i.e., one trailer), the existing state-of-the-art tool SCOTS [RZ16] solves the problem in 87 seconds using the HWC CPU3, which violates the real-time constraint. Using the same HWC, the implementation of GBFP in pFaces solves the same problem in 0.42 seconds (a speedup of around 207x). Clearly, such speedup is due to the ability of pFaces in utilizing all the 36 PEs of CPU3.

We upgrade the system and synthesize a controller for 2 trailers. In order to respect the real-time deadline, we update the HWC to GPU2. The problem is then solved in 0.98 seconds. Notice the increase in total-memory as reported in Table 4.3 when targeting more complex systems. We benefit from the feature in pFaces allowing different variations of the same kernel, and apply the variant kernel MemGBFP which saves more memory.

For 3 trailers, we use the HWC MIX2. Now, the problem is solved in 0.96 seconds. We report the time needed for a system with 4 trailers using the last HWC to illustrate that increasing the complexity of the problem without sufficient number of PEs uncovers the CoD. All details about this case study are given in Table 4.3.

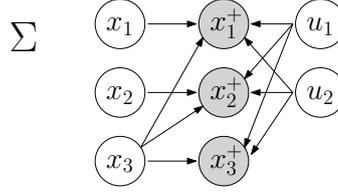
Now, having a kernel that respects the real-time constraint, we present a Cloud-based online real-time implementation of the truck-and-trailers system. Figure 4.2-(b) depicts the proposed implementation providing automated controller synthesis as a service for many trucks. The trucks are assumed to have some access to the internet. They submit requests for new controllers. The requests are packages containing the updated model and requirements to be enforced. Such packages are delivered to a web-based API (a.k.a. web service) developed with the RESTful interfaces library in `pFaces`. This communication is assumed to take less than 4 milliseconds so that the real-time constraint is still respected for  $N \in \{1, 2, 3\}$ . The packages are parsed and analyzed using the API. Then, the API utilizes the *instance manager* module to get access to one HWC with suitable resources that can respond in real-time to the request. Then, the computation is done in the HWC using `pFaces` running in the launched HWC. Once the controller is synthesized, it is delivered back to the truck that initiated the request. The truck is responsible for putting the controller into action. This implementation is deployed in Amazon’s AWS.

## 4.3 Data-Parallel Sparsity-Aware Algorithms for Symbolic Control

Traditional symbolic control algorithms (i.e., Algorithms 3 and 5) traverse the state space in a brute force way and suffer from an exponential runtime with respect to the number of state variables. The authors of [GKA17] noticed that a majority of continuous-space systems exhibit a coordinate structure, where the governing equation for each state variable is defined independently. When the equations of discrete-time systems depend only on few continuous variables, then they are said to be sparse. They proposed a modification to the traditional brute-force procedure to take advantage of such sparsity only in constructing abstractions. Unfortunately, their work suffers from the following drawbacks: (1) sparsity of dynamics is not leveraged to improve synthesis of symbolic controllers, which is, practically, more computationally complex than the construction of symbolic models (cf. the discussion in Section 2.5.1), and (2) the algorithms they introduced are designed for serial execution and consequently, they still suffer from the CoD.

In this section, we propose a data-parallel implementation of sparsity-aware algorithms that can utilize HPC platforms. In particular, we

- (1) introduce a data-parallel algorithm for constructing abstractions with a distributed data container. The algorithm utilizes sparsity and can run on HPC platforms. We implement it in `pFaces` and compare it with the results in [GKA17].
- (2) introduce a data-parallel algorithm that integrates sparsity of dynamical systems into the controller synthesis phase. This algorithm returns the same result as Algorithm 5, while exhibiting much lower running time.



**Figure 4.3:** The sparsity graph of the vehicle example as introduced in [GKA17].

### 4.3.1 Sparsity of Discrete-Time Systems

The concept of sparsity is currently limited to discrete time systems [GKA17]. If system  $\Sigma$  is given in form of an ODE as in (4.2.1), we consider its forward Euler approximation. More precisely, (4.2.1) is approximated by the following update equation (a.k.a. difference equation):

$$\Sigma : x^+ = x + \tau f(x, u), \quad (4.3.1)$$

where  $x^+ \in \mathbb{R}^n$  is the approximation of the system's state after  $\tau$  seconds starting from state  $x \in \mathbb{R}^n$  and applying constantly input  $u \in \mathbb{R}^m$ .  $\tau$  is the same sampling period used in (2.3.2) to embed  $\Sigma$  as system  $S_\tau$ .  $S_q := (X_q, U_q, T_q)$  should then be constructed as introduced earlier in Section 2.3.

**Remark 4.3.1.** Notice that when the approximation in (4.3.1) is used to construct the transition relation  $T_q$  of  $S_q$ , the approximation error (a.k.a. local truncation error) does not propagate since  $x^+$  is computed once for each state-input pair. Therefore, one can include this error (see [Cha18] for an upper bound of the local truncation error in Euler-approximated ODE) in the symbolic model which is manifested as extra nondeterminism in  $T_q$ . Consequently, a controller synthesized for  $S_q$  can still enforce the specification on the original continuous-time system in (4.2.1).

For a simpler presentation throughout the section, we abuse the notation and refer to the Right-Hand Side (RHS) of equation (4.3.1) as  $f(x, u)$  instead of  $x + \tau f(x, u)$ . Given a system  $S_\tau$ , an update-dependency graph is a directed graph of vertices representing input variables  $\{u_1, u_2, \dots, u_m\}$ , state variables  $\{x_1, x_2, \dots, x_n\}$ , and updated state variables  $\{x_1^+, x_2^+, \dots, x_n^+\}$ , and edges that connect input (resp. states) variables to the affected updated state variables based on map  $f$ . For example, Figure 4.3 depicts the update-dependency graph of the vehicle case study presented in [GKA17] with the update equation:

$$\begin{bmatrix} x_1^+ \\ x_2^+ \\ x_3^+ \end{bmatrix} = \begin{bmatrix} f_1(x_1, x_3, u_1, u_2) \\ f_2(x_2, x_3, u_1, u_2) \\ f_3(x_3, u_1, u_2) \end{bmatrix},$$

for some nonlinear functions  $f_1, f_2$ , and  $f_3$ . The state variable  $x_3$  affects all updated state variables  $x_1^+, x_2^+$ , and  $x_3^+$ . Hence, the graph has edges connecting  $x_3$  to  $x_1^+, x_2^+$ , and  $x_3^+$ , respectively. As update-dependency graphs become denser, sparsity of their corresponding symbolic models is reduced. The same graph applies to the symbolic model  $\Sigma_q$ .

### 4.3 Data-Parallel Sparsity-Aware Algorithms for Symbolic Control

During this section, we sometimes refer to  $X_q$ ,  $U_q$ , and  $T_q$  as monolithic state set, monolithic input set and monolithic transition relation, respectively. A generic projection map

$$P_i^f : A \rightarrow \pi^i(A)$$

is used to extract elements of the corresponding subsets affecting the updated state  $x_{q,i}^+$ . Note that  $A \subseteq X_q := X_{q,1} \times X_{q,2} \times \cdots \times X_{q,n}$  when we are interested in extracting subsets of the state set and  $A \subseteq U_q := U_{q,1} \times U_{q,2} \times \cdots \times U_{q,m}$  when we are interested in extracting subsets of the input set. When extracting subsets of the state set,  $\pi^i$  is the projection map  $\pi_{X_{q,k_1} \times X_{q,k_2} \times \cdots \times X_{q,k_K}}$ , where  $k_j \in \{1, 2, \dots, n\}$ ,  $j \in \{1, 2, \dots, K\}$ , and  $X_{q,k_1} \times X_{q,k_2} \times \cdots \times X_{q,k_K}$  is a subset of states affecting the updated state variable  $x_{q,i}^+$ . When extracting subsets of the input set,  $\pi^i$  is the projection map  $\pi_{U_{q,p_1} \times U_{q,p_2} \times \cdots \times U_{q,p_P}}$ , where  $p_i \in \{1, 2, \dots, m\}$ ,  $i \in \{1, 2, \dots, P\}$ , and  $U_{q,p_1} \times U_{q,p_2} \times \cdots \times U_{q,p_P}$  is a subset of inputs affecting the updated state variable  $x_{q,i}^+$ .

For example, assume that the monolithic state (resp. input) set of the system  $S_q$  in Figure 4.3 is given by  $X_q := X_{q,1} \times X_{q,2} \times X_{q,3}$  (resp.  $U_q := U_{q,1} \times U_{q,2}$ ) such that for any  $x_q := (x_{q,1}, x_{q,2}, x_{q,3}) \in X_q$  and  $u_q := (u_{q,1}, u_{q,2}) \in U_q$ , one has  $x_{q,1} \in X_{q,1}$ ,  $x_{q,2} \in X_{q,2}$ ,  $x_{q,3} \in X_{q,3}$ ,  $u_{q,1} \in U_{q,1}$ , and  $u_{q,2} \in U_{q,2}$ . Now, based on the dependency graph,  $P_1^f(x_q) := \pi_{X_{q,1} \times X_{q,3}}(x_q) = (x_{q,1}, x_{q,3})$  and  $P_1^f(u_q) := \pi_{U_{q,1} \times U_{q,2}}(u_q) = (u_{q,1}, u_{q,2})$ . We can also apply the map to subsets of  $X_q$  and  $U_q$ , e.g.,  $P_1^f(X_q) = X_{q,1} \times X_{q,3}$ , and  $P_1^f(U_q) = U_{q,1} \times U_{q,2}$ .

For a transition  $t := (x_q, u_q, x'_q) \in T_q$ , we define  $P_i^f(t) := (P_i^f(x_q), P_i^f(u_q), \pi_{X_{q,i}}(x'_q))$ , for any component  $i \in \{1, 2, \dots, n\}$ . Note that for  $t$ , the successor state  $x'_q$  is treated differently as it is related directly to the updated state variable  $x_{q,i}^+$ . We can apply the map to subsets of  $T_q$ , e.g., for the given update-dependency graph in Figure 4.3, one has  $P_1^f(T_q) = X_{q,1} \times X_{q,3} \times U_{q,1} \times U_{q,2} \times X_{q,1}$ .

A generic recovery map

$$D_i^f : P_i^f(A) \rightarrow 2^A$$

is also used to recover elements (resp. subsets) from the projected subsets back to their original monolithic sets. Similarly,  $A \subseteq X_q := X_{q,1} \times X_{q,2} \times \cdots \times X_{q,n}$  when we are interested in subsets of the state set and  $A \subseteq U_q := U_{q,1} \times U_{q,2} \times \cdots \times U_{q,m}$  when we are interested in subsets of the input set.

For the same example in Figure 4.3, let  $x_q := (x_{q,1}, x_{q,2}, x_{q,3}) \in X_q$  be a state. Now, define  $x_{q,p} := P_1^f(x_q) = (x_{q,1}, x_{q,3})$ . We then have  $D_1^f(x_{q,p}) := \{(x_{q,1}, x_{q,2}^*, x_{q,3}) \mid x_{q,2}^* \in X_{q,2}\}$ . For a transition element  $t := ((x_{q,1}, x_{q,2}, x_{q,3}), (u_{q,1}, u_{q,2}), (x'_{q,1}, x'_{q,2}, x'_{q,3})) \in T_q$  and its projection  $t_p := P_1^f(t) = ((x_{q,1}, x_{q,3}), (u_{q,1}, u_{q,2}), (x'_{q,1}))$ , the recovered transitions is the set  $D_1^f(t_p) = \{((x_{q,1}, x_{q,2}^*, x_{q,3}), (u_{q,1}, u_{q,2}), (x'_{q,1}, x_{q,2}^*, x'_{q,3})) \mid x_{q,2}^* \in X_{q,2}, x_{q,2}^* \in X_{q,2}, \text{ and } x'_{q,3} \in X_{q,3}\}$ .

**Remark 4.3.2.** Given a subset  $\tilde{X} \subseteq X_q$ , let  $[\tilde{X}] := D_1^f \circ P_1^f(\tilde{X})$ . Note that  $[\tilde{X}]$  is not necessarily equal to  $\tilde{X}$ . However, we have that  $\tilde{X} \subseteq [\tilde{X}]$ . Here,  $[\tilde{X}]$  over-approximates  $\tilde{X}$ .

Now, recall map  $O^f$  defined in (4.2.2). We assume it can be decomposed component-wise (i.e., per dimension  $i \in \{1, 2, \dots, n\}$ ) such that for any  $(x_q, u_q) \in X_q \times U_q$ ,  $O^f(x_q, u_q) = \bigcap_{i=1}^n D_i^f(O_i^f(P_i^f(x_q), P_i^f(u_q)))$ , where  $O_i^f : P_i^f(X_q) \times P_i^f(U_q) \rightarrow 2^{P_i^f(X_q)}$  is an over-approximation function restricted to component  $i \in \{1, 2, \dots, n\}$  of  $f$ . The same assumption applies to the underlying characterization function  $\Omega^f$ .

### 4.3.2 Sparsity-Aware Distributed Constructions of Abstractions

---

**Algorithm 7:** Serial sparsity-aware algorithm for constructing abstractions as introduced in [GKA17].

---

**Input:**  $X_q, U_q, O^f$   
**Output:** A transition relation  $T_q \subseteq X_q \times U_q \times X_q$ .

- 1  $T_q \leftarrow X_q \times U_q \times X_q$ ;
- 2 **for all**  $i \in \{1, 2, \dots, n\}$  **do**
- 3      $T_{q,i} \leftarrow \text{Algorithm-3}(P_i^f(X_q), P_i^f(U_q), O_i^f)$ ;
- 4      $T_q \leftarrow T_q \cap D_i^f(T_{q,i})$ ;
- 5 **end**

---

Earlier in this chapter, Algorithm 3 is presented as the traditional serial algorithm for constructing  $S_q$ . The drawback of the exhaustive iteration over  $X_q$  in Algorithm 3 is mitigated in Algorithm 7 by the technique introduced in [GKA17] which utilizes the sparsity of  $S_q$ .  $T_q$  is constructed by applying the Algorithm 3 to subsets of each component. Algorithm 7 presents a sparsity-aware serial algorithm for constructing  $S_q$ .

If we assume a bounded number of elements in subsets of each component (i.e.,  $|P_i^f(X_q)|$  and  $|P_i^f(U_q)|$  from line 3 in Algorithm 7), we would expect a near-linear complexity of the algorithm. This is not clearly the case in [GKA17, Figure 3] as the authors decided to use BDDs to represent transition relation  $T_q$ .

Clearly, representing  $T_q$  as a single storage entity is a drawback in Algorithm 7. All component-wise transition sets  $T_{q,i}$  will eventually need to push their results into  $T_q$ . This hinders any attempt to parallelize it unless a lock-free data structure is used, which affects the performance dramatically.

On the other hand, Algorithm 4 introduces a technique for constructing  $S_q$  by using a distributed data container to maintain the transition set  $T_q$  without constructing it explicitly. Using a continuous over-approximation  $\Omega^f$  is favored as opposed to the discrete over-approximation  $O^f$  since it requires less memory in practice. The actual computation of transitions (i.e., using  $O^f$  to compute discrete successor states) is delayed to the synthesis phase and done on the fly. The parallel algorithm scales remarkably with respect to the number of PEs  $P$  since the task is parallelizable with no data dependency. However, it still handles the problem monolithically which means, for a fixed  $P$ , it will not probably scale as the system dimension  $n$  grows.

We then introduce Algorithm 8 which utilizes sparsity to construct  $S_q$  in parallel. It is a combination of Algorithm 4 and Algorithm 7. Function  $I : \mathbb{N}_+ \setminus \{\infty\} \times \mathbb{N}_+ \setminus \{\infty\} \rightarrow$

---

**Algorithm 8:** Proposed sparsity-aware data-parallel algorithm for constructing discrete abstractions.

---

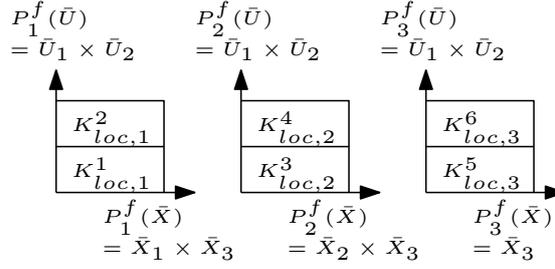
**Input:**  $X_q, U_q, \Omega^f$

**Output:** A list of characteristic sets:  $K := \bigcup_{p=1}^P \bigcup_{i=1}^n K_{loc,i}^p$ .

```

1 for all  $i \in \{1, 2, \dots, n\}$  do
2   for all  $p \in \{1, 2, \dots, P\}$  do
3      $K_{loc,i}^p \leftarrow \emptyset$ ;
4   end
5 end
6 for all  $i \in \{1, 2, \dots, n\}$  in parallel do
7   for all  $(x_q, u_q) \in P_i^f(X_q) \times P_i^f(U_q)$  in parallel with index  $i$  do
8      $p = I(i, j)$ ;
9      $(x_{lb}, x_{ub}) \leftarrow \Omega^f(x_q, u_q)$ ;
10     $K_{loc,i}^p \leftarrow K_{loc,i}^p \cup \{(x_q, u_q, (x_{lb}, x_{ub}))\}$ ;
11  end
12 end
    
```

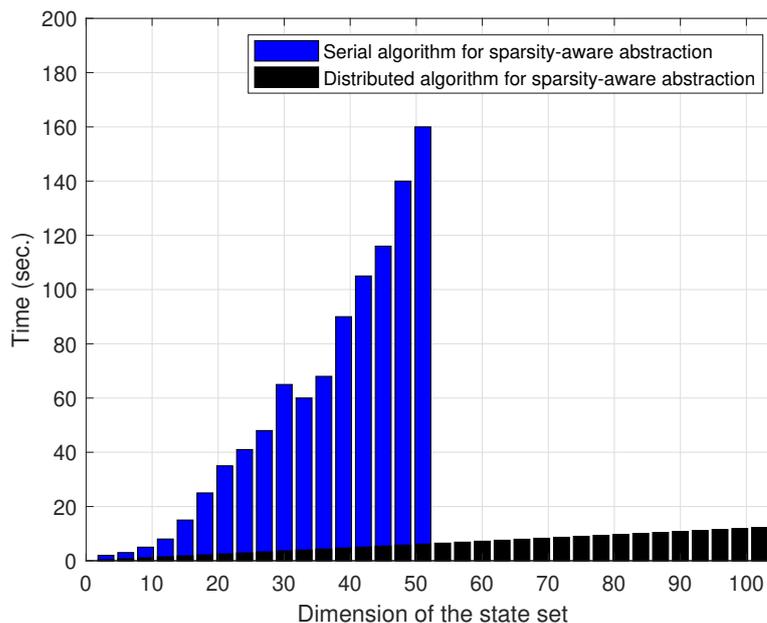
---



**Figure 4.4:** An example task distributions for the data-parallel sparsity-aware abstraction.

$\{1, 2, \dots, P\}$  maps a parallel job (i.e., lines 9 and 10 inside the inner **parallel for-all statement**), for a component  $i$  and a tuple  $(x_q, u_q)$  with index  $j$ , to a PE with an index  $p = I(i, j)$ .  $K_{loc,i}^p$  stores the characterizations of abstraction of  $i$ th component and is located in PE of index  $p$ . Collectively,  $K_{loc,1}^1, \dots, K_{loc,i}^p, \dots, K_{loc,n}^P$  constitute a distributed container that stores the abstraction of the system.

Figure 4.4 depicts an example of the job and task distributions for the example presented in Figure 4.3. Here, we use  $P = 6$  with a mapping  $I$  that distributes one partition element of one subset  $P_i^f(X_q) \times P_i^f(U_q)$  to one PE. We also assume that the used PEs have equal computation power. Consequently, we try to divide each subset  $P_i^f(X_q) \times P_i^f(U_q)$  into two equal partition elements such that we have, in total, 6 similar computation spaces. Inside each partition element, we indicate which distributed storage container  $K_{loc,i}^p$  is used.



**Figure 4.5:** Comparison between the serial and parallel algorithms for constructing abstractions of a traffic network model by varying the dimensions.

To assess the distributed algorithm in comparison with the serial one presented in [GKA17], we implement it in `pFaces`. We use the same traffic model presented in [GKA17, Subsection VI-B] and the same parameters. For this example, the authors of [GKA17] construct  $T_{q,i}$ , for each component  $i \in \{1, 2, \dots, n\}$ . They combine them incrementally in a BDD that represents  $T_q$ . A monolithic construction of  $T_q$  from  $T_{q,i}$  is required in [GKA17] since symbolic controller synthesis is done monolithically. On the other hand, using  $K_{loc,i}^p$  in our technique plays a major role in reducing the complexity of constructing higher dimensional abstractions. Later in Subsection 4.3.3, we utilize  $K_{loc,i}^p$  directly to synthesize symbolic controllers with no need to explicitly construct  $T_q$ .

Figure 4.5 depicts a comparison between the results reported in [GKA17, Figure 3] and the ones obtained from our implementation in `pFaces`. We use an Intel Core i5 CPU, which comes equipped with an internal GPU yielding around 24 PEs being utilized by `pFaces`. The implementation stores the distributed containers  $K_{loc,i}^p$  as raw-data inside the memories of their corresponding PEs. As expected, the distributed algorithm scales linearly and we are able to go beyond 100 dimensions in a few seconds, whereas Figure 3 in [GKA17] shows only abstractions up to a 51-dimensional traffic model because constructing the monolithic  $T_q$  begins to incur an exponential cost for higher dimensions.

**Remark 4.3.3.** *Algorithms 7 and 8 utilize the sparsity of  $\Sigma$  to reduce the space complexity of abstractions from  $O(|X_q \times U_q|)$  to  $O(\max_{i=1} |P_i^f(X_q) \times P_i^f(U_q)|)$ . However, Algorithm 7 iterates over the space serially. Algorithm 8, on the other hand, handles*

the computation over the space in parallel using variable  $P$  of PEs and hence reduces the complexity to  $O(\frac{\max_{i=1} |P_i^f(X_q) \times P_i^f(U_q)|}{P})$ .

### 4.3.3 Sparsity-Aware Distributed Synthesis of Symbolic Controllers

Recall the controllable predecessor map defined in (2.4.2). Now, we introduce a component-wise controllable predecessor map  $CPre^{T_{q,i}} : 2^{P_i^f(X_q) \times P_i^f(U_q)} \rightarrow 2^{P_i^f(X_q) \times P_i^f(U_q)}$ , for any component  $i \in \{1, 2, \dots, n\}$  and any  $\tilde{Z} := P_i^f(Z) := \pi_{P_i^f(X_q) \times P_i^f(U_q)}(Z)$ , as follows:

$$CPre^{T_{q,i}}(\tilde{Z}) := \{(x_q, u_q) \in P_i^f(X_q) \times P_i^f(U_q) \mid \emptyset \neq T_{q,i}(x_q, u_q) \subseteq \pi_{X_{q,i}}(\tilde{Z})\}. \quad (4.3.2)$$

**Proposition 4.3.4.** *The following inclusion holds for any  $i \in \{1, 2, \dots, n\}$  and any  $Z \subseteq X_q \times U_q$ :*

$$P_i^f(CPre^{T_q}(Z)) \subseteq CPre^{T_{q,i}}(P_i^f(Z)).$$

*Proof.* Consider an element  $z_p \in P_i^f(CPre^{T_q}(Z))$ . This implies that there exists  $z \in X_q \times U_q$  such that  $z \in CPre^{T_q}(Z)$  and  $z_p = P_i^f(z)$ . Consequently,  $T_{q,i}(z_p) \neq \emptyset$  since  $T_q(z) \neq \emptyset$ . Also, since  $z \in CPre^{T_q}(Z)$ , then  $T_q(z) \subseteq \pi_{X_q}(Z)$ . Now, recall how  $T_{q,i}$  is constructed as a component-wise set of transitions in line 2 in Algorithm 7. Then, we conclude that  $T_{q,i}(z_p) \subseteq \pi_{X_{q,i}}(P_i^f(Z))$ . By this, we already satisfy the requirements in (4.3.2) such that  $z_p = (x_q, u_q) \in CPre^{T_{q,i}}(Z)$ .  $\square$

To demonstrate the intended improvements to the synthesis phase of symbolic control, we focus on reachability specifications. A similar discussion can be pursued for specifications that can be represented with fixed-point operations. To synthesize symbolic controllers for the reachability specifications  $\psi := \diamond\phi$ , recall a monotone function  $\underline{G}(Z)$  defined in 2.4.3 and Algorithm 5 for traditional serial algorithm of symbolic controller synthesis for reachability specifications. In Algorithm 6, we already introduced a parallel implementation of Algorithm 5 that mitigates the complexity of the fixed-point computation. In that algorithm, for a set  $Z \subseteq X_q \times U_q$ , each iteration of  $\mu Z.\underline{G}(Z)$  is computed via parallel traversal in the complete space  $X_q \times U_q$ . Each PE is assigned a disjoint set of state-input pairs from  $X_q \times U_q$  and it declares whether, or not, each pair belongs to the next winning pairs (i.e.,  $\underline{G}(Z)$ ). Although Algorithm 6 scales well with respect to  $P$ , it still suffers from the CoD for a fixed  $P$ . In the next Subsection, we will present a modified version of the algorithm that utilizes sparsity to reduce the parallel search space at each iteration.

First, we introduce the component-wise monotone function:

$$\underline{G}_i(Z) := CPre^{T_{q,i}}(P_i^f(Z)) \cup P_i^f(Z_\psi), \quad (4.3.3)$$

for any  $i \in \{1, 2, \dots, n\}$  and any  $Z \subseteq X_q \times U_q$ . Now, an iteration in the sparsity-aware fixed-point can be summarized by the following three steps:

#### 4 Efficient Algorithms for Symbolic Control

- (1) Compute the component-wise sets  $\underline{G}_i(Z)$ . Note that  $\underline{G}_i(Z)$  lives in the set  $P_i^f(X_q) \times P_i^f(U_q)$ .
- (2) Recover a monolithic set  $\underline{G}_i(Z)$ , for each  $i \in \{1, 2, \dots, n\}$ , using the map  $D_i^f$  and intersect these sets. Formally, we denote this intersection by:

$$[\underline{G}(Z)] := \bigcap_{i=1}^n (D_i^f(\underline{G}_i(Z))). \quad (4.3.4)$$

Note that  $[\underline{G}(Z)]$  is an over-approximation of the monolithic set  $\underline{G}(Z)$ , which we prove in Theorem 4.3.5.

- (3) Now, based on the next theorem, there is no need for a parallel search in  $X_q \times U_q$  and the search can be done in  $[\underline{G}(Z)]$ . More accurately, the search for new elements in the next winning set can be done in  $[\underline{G}(Z)] \setminus Z$ .

**Theorem 4.3.5.** *Consider a symbolic model  $S_q := (X_q, U_q, T_q)$ . For any set  $Z \subseteq X_q \times U_q$ ,  $\underline{G}(Z) \subseteq [\underline{G}(Z)]$ .*

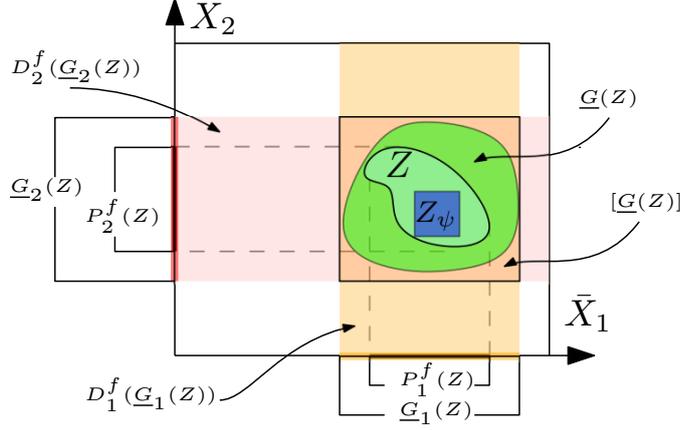
*Proof.* Consider any element  $z \in \underline{G}(Z)$ . This implies that  $z \in Z$ ,  $z \in Z_\psi$  or  $z \in CPre^{T_q}(Z)$ . We show that  $z \in [\underline{G}(Z)]$  for any of these cases.

Case 1 [ $z \in Z$ ]: By the definition of map  $P_i^f$ , we know that  $P_i^f(z) \in P_i^f(Z)$ . By the monotonicity of map  $\underline{G}_i$ ,  $P_i^f(Z) \subseteq \underline{G}_i(Z)$ . This implies that  $P_i^f(z) \in \underline{G}_i(Z)$ . Also, by the definition of map  $D_i^f$ , we know that  $z \in D_i^f(\underline{G}_i(Z))$ . The above argument holds for any component  $i \in \{1, 2, \dots, n\}$  which implies that  $z \in \bigcap_{i=1}^n (D_i^f(\underline{G}_i(Z))) = [\underline{G}(Z)]$ .

Case 2 [ $z \in Z_\psi$ ]: The same argument used for the previous case can be used for this one as well.

Case 3 [ $z \in CPre^{T_q}(Z)$ ]: We apply the map  $P_i^f$  to both sides of the inclusion. We then have  $P_i^f(z) \in P_i^f(CPre^{T_q}(Z))$ . Using Proposition 4.3.4, we know that  $P_i^f(CPre^{T_q}(Z)) \subseteq CPre^{T_q, i}(P_i^f(Z))$ . This implies that  $P_i^f(z) \in CPre^{T_q, i}(P_i^f(Z))$ . From (4.3.3) we obtain that  $P_i^f(z) \in \underline{G}_i(Z)$ , and consequently,  $z \in D_i^f(\underline{G}_i(Z))$ . The above argument holds for any component  $i \in \{1, 2, \dots, n\}$ . This, consequently, implies that  $z \in \bigcap_{i=1}^n (D_i^f(\underline{G}_i(Z))) = [\underline{G}(Z)]$ , which completes the proof. □

**Remark 4.3.6.** *The computation of the controllable predecessor is done component-wise in step (1) utilizing the sparsity of  $S_q$  and can be implemented in parallel. In step (3), a monolithic search is required. However, unlike the implementation in Algorithm 6, the search is performed only for a subset of  $X_q \times U_q$ , which is  $[\underline{G}(Z)] \setminus Z$ .*



**Figure 4.6:** A visualization of one arbitrary fixed-point iteration of the sparsity-aware synthesis technique for a two-dimensional robot system.

Note that dynamical systems enjoy some locality properties (i.e., starting from nearby states, successor states are also nearby) and an initial winning set will grow incrementally with each fixed-point iteration. This makes the set  $[\underline{G}(Z)] \setminus Z$  relatively small with respect to  $|X_q \times U_q|$ . We clarify this and the result in Theorem 4.3.5 with a small two-dimensional example.

Consider a robot described by the following ODE:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix},$$

where  $(x_1, x_2) \in X_q := X_{q,1} \times X_{q,2}$  is a state vector and  $(u_1, u_2) \in U_q := U_{q,1} \times U_{q,2}$  is an input vector.

The ODE is approximated with Euler's method as discussed earlier and the following difference equation is the resulting discrete-time dynamics:

$$\begin{bmatrix} x_1^+ \\ x_2^+ \end{bmatrix} = \begin{bmatrix} x_1 + \tau u_1 \\ x_2 + \tau u_2 \end{bmatrix},$$

Figure 4.6 shows a visualization of the sets related to this sparsity-aware technique for symbolic controller synthesis for one fixed-point iteration. Set  $Z_\psi$  is the initial winning-set (a.k.a. target-set for reachability specifications) constructed from a given specification (e.g., a region in  $X_q$  to be reached by the robot) and  $Z$  is the winning-set of the current fixed-point iteration. For simplicity, all sets are projected on  $X_q$  and the readers can think of  $U_q$  as an additional dimension perpendicular to the surface of this paper.

As depicted in Figure 4.6, the next winning-set  $\underline{G}(Z)$  is over-approximated by  $[\underline{G}(Z)]$ , as a result of Theorem 4.3.5. Algorithm 6 searches for  $\underline{G}(Z)$  in  $(X_{q,1} \times X_{q,2}) \times (U_{q,1} \times U_{q,2})$ . The modification we suggest is to search for  $\underline{G}(Z)$  in  $[\underline{G}(Z)] \setminus Z$  instead.

---

**Algorithm 9:** Proposed parallel sparsity-aware algorithm to synthesize  $\underline{C}$  enforcing specification  $\psi := \diamond\phi$ .

---

**Input:** Initial winning domain  $Z_\psi \subset X_q \times U_q$  and  $T_q$

**Output:** A controller  $\underline{C} : X_w \rightarrow 2^{U_q}$ .

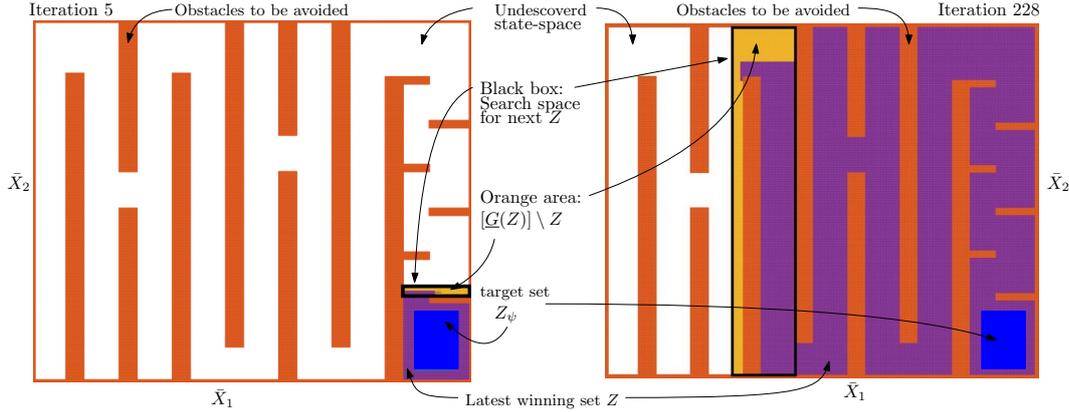
---

```

1  $Z_\infty \leftarrow \emptyset;$ 
2  $X_w \leftarrow \emptyset;$ 
3 do
4    $Z_0 \leftarrow Z_\infty;$ 
5   for all  $p \in \{1, 2, \dots, P\}$  do
6      $Z_{loc}^p \leftarrow \emptyset;$ 
7      $X_{w,loc}^p \leftarrow \emptyset;$ 
8   end
9    $[\underline{G}] \leftarrow X_q \times U_q;$ 
10  for all  $i \in \{1, 2, \dots, n\}$  do
11     $[\underline{G}] \leftarrow [\underline{G}] \cap D_i^f(G_i(Z_\infty));$ 
12  end
13  for all  $(x_q, u_q) \in [\underline{G}] \setminus Z_\infty$  in parallel with index  $i$  do
14     $p = I(i);$ 
15     $Posts \leftarrow Q \circ K_{loc}^p(x_q, u_q);$ 
16    if  $Posts \subseteq Z_0 \cup Z_\psi$  then
17       $Z_{loc}^p \leftarrow Z_{loc}^p \cup \{(x_q, u_q)\};$ 
18       $X_{w,loc}^p \leftarrow X_{w,loc}^p \cup \{x_q\};$ 
19      if  $x_q \notin \pi_{X_q}(Z_0)$  then
20         $\underline{C}(x_q) \leftarrow \underline{C}(x_q) \cup \{u_q\};$ 
21      end
22    end
23  end
24  for all  $p \in \{1, 2, \dots, P\}$  do
25     $Z_\infty \leftarrow Z_\infty \cup Z_{loc}^p;$ 
26     $X_w \leftarrow X_w \cup X_{w,loc}^p;$ 
27  end
28 while  $Z_\infty \neq Z_0;$ 

```

---



**Figure 4.7:** The evolution of the fixed-point sets for the robot example by the end of fixed-point iterations 5 (left side) and 228 (right side).

#### 4.3.4 Sparsity-Aware Data-Parallelism for Symbolic Controller Synthesis

We propose Algorithm 9 as a parallel algorithm for sparsity-aware controller synthesis. The main difference between it and Algorithm 6 are lines 9-12. They correspond to computing  $[G(Z)]$  at each iteration of the fixed-point computation. Line 13 is modified to do the parallel search inside  $[G(Z)] \setminus Z$  instead of  $X_q \times U_q$  in the original algorithm. The rest of the algorithm is similar to what we already discussed earlier in this chapter.

The algorithm, along with Algorithm 8, are implemented in **pFaces** as updated versions of the kernels GBFP and MemGBFP. We denote the sparsity-aware versions by GBFP-s and MemGBFP-s [KKAZ19]. We synthesize a reachability controller for the robot example presented earlier. Figure 4.7 shows an arena with obstacles depicted as red boxes. It depicts the result at the fixed point iterations 5 and 228. The blue box indicates the target set (i.e.,  $Z_\psi$ ). The region colored with purple indicates the current winning states. The orange region indicates  $[G(Z)] \setminus Z$ . The black box is the next search region which is a rectangular over approximation of the  $[G(Z)] \setminus Z$ . We over-approximate  $[G(Z)] \setminus Z$  with such rectangle as it is straightforward for PEs in **pFaces** to work with rectangular parallel jobs. The synthesis problem is solved in 322 fixed-point iterations. Unlike Algorithm 6 which searches for the next winning region inside  $X_q \times U_q$  at each iteration, the implementation of the proposed algorithm reduces the parallel search by an average of 87% when searching inside the black boxes in each iteration.

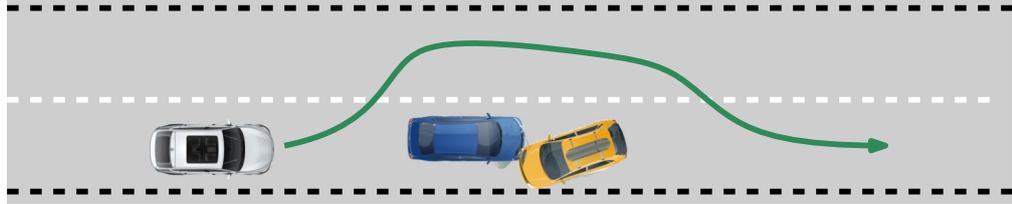


Figure 4.8: An autonomous vehicle trying to avoid a sudden obstacle on the highway.

### 4.3.5 Case Study: Autonomous Vehicle Avoiding Crash on Highway

We consider a vehicle described by the following 7-dimensional continuous-time single track model [Alt17]:

$$\begin{aligned}
 \dot{x}_1 &= x_4 \cos(x_5 + x_7), \\
 \dot{x}_2 &= x_4 \sin(x_5 + x_7), \\
 \dot{x}_3 &= u_1, \\
 \dot{x}_4 &= u_2, \\
 \dot{x}_5 &= x_6, \\
 \dot{x}_6 &= \frac{\mu m}{I_z(l_r + l_f)} (l_f C_{S,f}(gl_r - u_2 h_{cg})x_3 + (l_r C_{S,r}(gl_f + u_2 h_{cg}) - l_f C_{S,f}(gl_r - u_2 h_{cg}))x_7 - (l_f l_f C_{S,f}(gl_r - u_2 h_{cg}) + l_r^2 C_{S,r}(gl_f + u_2 h_{cg}))\frac{x_6}{x_4}), \\
 \dot{x}_7 &= \frac{\mu}{x_4(l_f + l_r)} (C_{S,f}(gl_r - u_2 h_{cg})x_3 - (C_{S,r}(gl_f + u_2 h_{cg}) + C_{S,f}(gl_r - u_2 h_{cg}))x_7 + (C_{S,r}(gl_f + u_2 h_{cg})l_r - C_{S,f}(gl_r - u_2 h_{cg})l_f)\frac{x_6}{x_4}) - x_6,
 \end{aligned}$$

where  $x_1$  and  $x_2$  are the position coordinates,  $x_3$  is the steering angle,  $x_4$  is the heading velocity,  $x_5$  is the yaw angle,  $x_6$  is the yaw rate, and  $x_7$  is the slip angle. Variables  $u_1$  and  $u_2$  are inputs, and they control the steering angle and heading velocity, respectively. Input and state variables are all members of  $\mathbb{R}$ . The model takes into account tire slip making it a good candidate for studies that consider planning of evasive maneuvers that are very close to the physical limits. We consider an update period  $\tau = 0.1$  seconds and the following parameters for a BMW 320i car:  $m = 1093$  [kg] as the total mass of the vehicle,  $\mu = 1.048$  as the friction coefficient,  $l_f = 1.156$  [m] as the distance from the front axle to Center of Gravity (CoG),  $l_r = 1.422$  [m] as the distance from the rear axle to CoG,  $h_{cg} = 0.574$  [m] as the height of CoG,  $I_z = 1791.0$  [kg m<sup>2</sup>] as the moment of inertia for entire mass around z axis,  $C_{S,f} = 20.89$  [1/rad] as the front cornering stiffness coefficient, and  $C_{S,r} = 19.89$  [1/rad] as the rear cornering stiffness coefficient.

We first construct a discrete-time approximation of the model as discussed earlier. Then, to construct a symbolic model  $S_q$ , we consider a bounded version of the state set  $X_\tau := [0, 84] \times [0, 6] \times [-0.18, 0.8] \times [12, 21] \times [-0.5, 0.5] \times [-0.8, 0.8] \times [-0.1, 0.1]$ , a state quantization vector  $\eta_X = (1.0, 1.0, 0.01, 3.0, 0.05, 0.1, 0.02)$ , a input set  $U_\tau := [-0.4, 0.4] \times [-4, 4]$ , and an input quantization vector  $\eta_U = (0.1, 0.5)$ .

We are interested in an autonomous operation of the vehicle on a highway. Consider a situation on two-lane highway when an accident happens suddenly on the same lane on which our vehicle is traveling. The vehicle's controller should find a safe maneuver to avoid the crash with the next-appearing obstacle. Figure 4.8 depicts such a situation.

**Table 4.4:** Results obtained after running the experiments EX<sub>1</sub> and EX<sub>2</sub>.

EX <sub>1</sub> (Memory = 22.1 G.B.) $ X_q \times U_q  = 23.8 \times 10^9$				EX <sub>2</sub> (Memory = 49.2 G.B.) $ X_q \times U_q  = 52.9 \times 10^9$			
HW	Time MemGBFP	Time MemGBFP-s	Speedup	HW	Time MemGBFP	Time MemGBFP-s	Speedup
CPU5	2.1 hours	0.5 hours	4.2x	CPU4	$\geq 24$ hours	8.7 hours	$\geq 2.7x$
CPU6	1.9 hours	0.4 hours	4.7x	CPU5	8.1 hours	3.2 hours	2.5x

We over-approximate the obstacle with the hyper-box  $[28, 50] \times [0, 3] \times [-0.18, 0.8] \times [12, 21] \times [-0.5, 0.5] \times [-0.8, 0.8] \times [-0.1, 0.1]$ .

We run the implementation on different HWCs. We run two different experiments. For the first one (denoted by EX<sub>1</sub>), the goal is to only avoid the crash with the obstacle. We use a smaller version of the original state set  $X_\tau := [0, 50] \times [0, 6] \times [-0.18, 0.8] \times [11, 19] \times [-0.5, 0.5] \times [-0.8, 0.8] \times [-0.1, 0.1]$ . The second one (denoted by EX<sub>2</sub>) targets the full-sized highway window (84 meters), and the goal is to avoid colliding with the obstacle and get back to the right lane. Table 4.4 reports the obtained results. The reported times are for constructing finite abstractions of the vehicle and synthesizing symbolic controllers. Note that our results outperform easily the initial kernels GBFP and MemGBFP which themselves outperform serial implementations with speedups up to 30000x as reported in [KZ19]. The speedup in EX<sub>1</sub> is higher as the obstacle consumes a relatively bigger volume in the state space. This makes  $[G(Z)] \setminus Z$  smaller and, hence, faster for our implementation.

## 4.4 Summary

Most of the existing techniques on symbolic control take a monolithic view of systems, where the entire system is modeled, abstracted, and then a controller is synthesized from the overall state-space. On the other hand, the process of constructing finite abstractions is inherently parallelizable. In this chapter, traditional serial algorithms on symbolic control were redesigned as data-parallel algorithms that scale with the number of PEs. This, with the availability of variable number of PEs, allows controlling the computational complexity of symbolic control.

We first introduced a data-parallel algorithms for the construction of symbolic models and the synthesis of their symbolic controllers. Algorithm 4 was proposed as parallelization of Algorithm 3 to construct symbolic models of control systems. The algorithm introduces the abstraction task as an ideal data-parallel task with no communication overhead among the processing elements. As shown in Theorem 4.2.4, the parallel algorithm allows controlling the complexity of the serial one by allowing it to run on multiple parallel PEs. Algorithm 6 was also the proposed parallelization of Algorithm 5 to synthesize symbolic controllers using parallel fixed-point operations.

We also introduced an approach that utilizes sparsity of dynamical systems for both the construction of finite abstractions and the synthesis of their symbolic controllers.

#### 4 Efficient Algorithms for Symbolic Control

First, a parallel sparsity-aware algorithm was introduced for the constructions of abstractions. It is a combination of Algorithm 4 and the traditional serial algorithm for sparse construction of abstractions. Similar to the serial one, it constructs in parallel lower-dimensional abstractions using the information extracted from the sparsity graph of the system. It however allows running the algorithm on parallel PEs which makes the algorithm highly scalable. Then, we introduced a sparsity-aware algorithm for the distributed synthesis of symbolic controllers. The computation of the controllable predecessor is done at lower-dimensional versions of the state space with the help of the sparsity information. Unlike the serial implementation and the parallel non-sparse implementation, the search for the predecessor set in the last implementation is only performed inside smaller subsets of the  $X_q \times U_q$ .

All the designed parallel algorithms were implemented as kernels on top of `pFaces` and the results showed remarkable reductions in computation time. We showed the effectiveness of the introduced approaches using several examples including a 7-dimensional model of a BMW 320i car.

# 5 Efficient Algorithms for the Computation of Reachable Sets

Reachable sets characterize the sets of states a control system can reach in a given time range, starting from a certain initial set and subjected to certain inputs. They play an important role in several formal methods approaches including the verification of behaviors of control systems and the synthesis of their controllers.

In Chapter 4, the computation of reachable sets is considered implicitly in Algorithms 3, 4, 7, 8. We used  $\text{map } \Omega^f$ , as defined in Section 4.2, to represent the reachable states starting from a hyper-rectangular set  $x_q \subseteq X_\tau$ , which is an essential step for constructing symbolic model  $S_q$ . Although Algorithm 4 runs in parallel over the symbolic states in  $X_q$ , the computation of  $\Omega^f$  is done serially in every parallel thread. As the dimension of  $\Sigma$  grow, computing  $\Omega^f$  becomes very complex.

In this chapter, we provide efficient parallel algorithms for the computation of reachable sets for extremely high-dimensional systems. We show that considering data-parallel algorithms and implementing them in HPC platforms can lead to a significant reduction in the computation time, and allow for handling dynamical systems with billions of state variables. The parallelized algorithms are implemented as a kernel on top of `pFaces`. We denote this kernel by `PIRK` [DKAZ20].

## 5.1 Approximations of Reachable Sets

Computing the exact reachable set is generally not possible. For example, even for the case of discrete-time Linear Time-Invariant (LTI) systems it is not known whether the exact reachable set is computable in many important applications [FOP<sup>+</sup>19]. Therefore, most practical methods resort to computing over-approximations or under-approximations of the reachable sets, depending on the desired guarantee. Computing these approximations to a high degree of accuracy is still a computationally intensive task, particularly for high-dimensional systems.

Many software tools have been developed to address the various challenges of approximating reachable sets. Each of these tools uses different methods and leverages different system assumptions to achieve different goals related to computing reachable sets. For example, `CORA` [Alt15] and `SpaceEx` [FLGD<sup>+</sup>11] tools are designed to compute reachable sets of high accuracy for very general classes of nonlinear systems, including hybrid ones. Some reachability analysis methods rely on specific features of dynamical systems, such as linearity of the dynamics or sparsity in the interconnection structure [BTJ19]. This allows computing the reachable sets in shorter time or for relatively high-dimensional systems. However, it limits the approach to smaller classes of applications, less practical

specifications, or requires the use of less accurate (e.g., linearized) models. Some examples of toolboxes that make these kinds of assumptions are the `EllipsoidalToolbox` [KV06], which assumes that system is linear (though it is allowed to be time-varying and subject to disturbance), and `SAP0` [Dre17], which assumes polynomial dynamics. Other methods attack the computational complexity problem by computing reachable set approximations from a limited class of set representations. Most tools use some combination of these two approximations. For example, the `EllipsoidalToolbox` computes ellipsoid over-approximations, and `SAP0` produces approximations in the form of polytopes.

An example of limiting the set of allowed over-approximations are *interval reachability* methods, in which reachable sets are approximated by Cartesian products of intervals. Interval reachability methods allow for computing the reachable sets of very general non-linear and high-dimensional systems in a short amount of time. They also pose mild constraints on the systems under consideration, usually only requiring some kind of boundedness constraint instead of a specific form for the system dynamics. Many reachability tools that are designed to scale well with state dimension focus on interval reachability methods: these include `Flow*` [CÁS13], `CAPD` [Gro19], `C2E2` [DMVP15], `VNODE – LP` [Ned11], `DynIbex` [SC16], and `TIRA` [MDA19].

For the rest of this chapter, we consider interval reachability methods for computing reach sets for continuous-time nonlinear systems. Such choice coincides with the structure of the state set in symbolic model  $S_q$  introduced in the previous chapter which is a partition on  $X_\tau$  constructed by a set of hyper-rectangles. In the next subsection, we introduce different interval reachability analysis methods. We consider a *simulation-based* approach for interval reachability analysis [HM12, JP09, MA14], which compute rigorous approximations to reachable sets by integrating one or more systems of ODEs. By focusing on simulation-based methods, the problem of parallelizing the methods is reduced to the problem of parallelizing ODE integration, a problem which is well-studied. Since a particularly novel parallelization scheme is not required, we could focus our work on contributing a high-quality data-parallel implementation. Also choosing a relatively simple data-parallelization scheme allows providing scalable large-scale parallelization that can handle systems of extremely high dimensions.

## 5.2 Interval Reachability Analysis

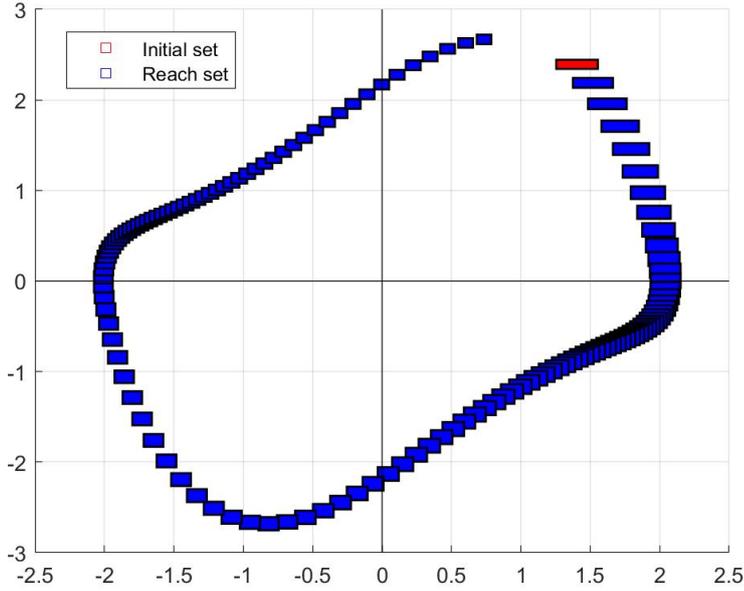
Recall the definition of control system  $\Sigma$  and its right-hand  $f$  as introduced in (4.2.1). For a simpler analysis, we redefine  $f$  and expose the time variable  $t$  as input:

$$\dot{x}(t) = f(t, x, u).$$

Let  $\mathcal{X}_0 \subseteq X_\tau$  be a set of initial states,  $[t_0, t_1]$  be a time interval, and  $\mathcal{U}$  be a set of time-varying inputs defined over  $[t_0, t_1]$ . Let  $\Phi(t; t_0, x_0, u)$  denote the state of the system, at time  $t$ , of the trajectory beginning at time  $t_0$  at initial state  $x_0$  under input  $u \in \mathcal{U}$ .

The finite-time forward reachable set is defined as

$$R_{[t_0, t_1]} = \{\Phi(t_1; t_0, x, u) \mid x \in \mathcal{X}_0 \wedge u \in \mathcal{U}\}.$$



**Figure 5.1:** Interval approximation of the flow pipe for the Van der Pol oscillator.

Computing an exact reachable set is typically an impossible or at best intractable task, so we instead aim to compute an approximation  $\hat{R}_{[t_0, t_1]}$  to the reachable set, either an over-approximation (so that  $R_{[t_0, t_1]} \subset \hat{R}_{[t_0, t_1]}$ ) or an under-approximation (so that  $\hat{R}_{[t_0, t_1]} \subset R_{[t_0, t_1]}$ ). An example of several finite-time reachable sets is shown in Figure 5.1. The example shows the solution to an interval reachability problem for the Van der Pol oscillator, a nonlinear system with two states. The red rectangle is the interval initial set. The blue rectangles are interval reachable sets for several final times  $t_1$ .

For the problem of interval reachability analysis, there are a few more constraints on the problem structure. An *interval set* is a set of the form of a hyper-rectangle  $\llbracket \underline{a}, \bar{a} \rrbracket$ . Vectors  $\underline{a}$  and  $\bar{a}$  are, respectively, the lower and upper bounds of the interval set. An interval set can alternatively be described by its center  $a^* := \frac{1}{2}(\bar{a} + \underline{a})$  and half-width  $[a] := \frac{1}{2}(\bar{a} - \underline{a})$ .

In interval reachability analysis, the initial set must be an interval, and inputs values restricted to an interval set, i.e.  $u(t) \in \llbracket \underline{u}, \bar{u} \rrbracket$ , and the reachable set approximation must also be an interval. Furthermore, certain methods for computing interval reachable sets require further restrictions on the system dynamics, such as the state and input Jacobian matrices being bounded or sign-stable. We discuss the following three simulation-based methods for computing interval reachable sets: (1) the contraction/growth bound method, (2) the mixed monotonicity method, and (3) a Monte Carlo based method. They allow for different trade-offs between conservatism and computation speed, as well as different forms of problem data. Further details on the requirements and limitations of these methods are presented in [MDA19].

### 5.2.1 Contraction/Growth Bound (GB) Method

The method computes the reachable set using component-wise contraction properties of the system [RWR17, TK09, FKJM16]. It may be applied to input-affine systems of the form  $\dot{x} = f(t, x) + u$ . The growth and contraction properties of each component of the system are first characterized by a *contraction matrix*  $C$ . Matrix  $C$  is a component-wise generalization of the matrix measure of the Jacobian [MA14, AM18], satisfying  $C_{ii} \geq J_{x,ii}(t, x, u^*)$  for diagonal elements, and  $C_{ij} \geq |J_{x,ij}(t, x, u^*)|$  for off-diagonal elements. The method constructs a reachable set over-approximation as interval by separately establishing its *center* and *half-width*. The center is found by simulating the trajectory of the center of the initial set, that is as  $\Phi(t_1; t_0, x^*, u^*)$ . The half width is found by integrating the *growth dynamics*  $\dot{r} = g(r, u) = Cr + [u]$ , where  $r$  denotes the half-width, over the time range  $[t_0, t_1]$  with initial condition  $r(t_0) = [x]$ .

### 5.2.2 Continuous-Time Mixed-Monotonicity (CTMM) Method

The method computes the reachable set by separating the increasing and decreasing portions of the system dynamics in an auxiliary system called the *embedding system* whose state dimension is twice that of the original system [CAB17, GH94]. The embedding system is constructed using a *decomposition function*  $d(t, x, u, \hat{x}, \hat{u})$ , which encodes the increasing and decreasing parts of the system dynamics and satisfies  $d(t, x, u, x, u) = f(t, x, u)$ . The embedding system dynamics are then defined as

$$\begin{bmatrix} \dot{x} \\ \dot{\hat{x}} \end{bmatrix} = h(x, u, \hat{x}, \hat{u}) = \begin{bmatrix} d(t, x, u, \hat{x}, \hat{u}) \\ d(t, \hat{x}, \hat{u}, x, u) \end{bmatrix}. \quad (5.2.1)$$

For states having  $x = \hat{x}$ , (5.2.1) becomes two copies of the original system dynamics by the properties of the decomposition function: this is the sense in which the embedding system embeds the original system dynamics.

The evaluation of a single trajectory of the embedding system can be used to find a reachable set over-approximation for the original system. Specifically, consider the solution of (5.2.1) over the interval  $[t_0, t_1]$  with the initial condition  $[x_0 \ \hat{x}_0]^T = [\underline{x} \ \bar{x}]^T$ , and subject to the constant inputs  $u = \underline{u}$ ,  $\hat{u} = \bar{u}$ . Then, the integrated final state  $[x_1 \ \hat{x}_1]^T$  serves as an over-approximation of the reachable set of the original system dynamics, that is  $R_{[t_0, t_1]} \subset \llbracket x_1, \hat{x}_1 \rrbracket$ .

### 5.2.3 Monte Carlo (MC) Method

The method computes a probabilistic approximation to the reachable set by evaluating the trajectories of a finite number of *sample points* in the initial set and input set, and selecting the smallest interval that contains the final points of the trajectories. Unlike the other two methods, the Monte Carlo method is restricted to constant-valued inputs, i.e. inputs of the form  $u(t) = u$ , where  $u \in \llbracket \underline{u}, \bar{u} \rrbracket$ . Formally, we take probability distributions whose supports are the initial set  $\llbracket \underline{x}, \bar{x} \rrbracket$  and input value set  $\llbracket \underline{u}, \bar{u} \rrbracket$  respectively (for example, a uniform distribution over these sets), and pick  $m$  pairs  $(x_0^{(i)}, u^{(i)})$ ,  $i \in$

$\{1, 2, \dots, m\}$ ) of independent and identically distributed (iid) samples from the two distributions. Each initial state  $x_0^{(i)}$  is integrated over  $[t_0, t_1]$  with its paired input  $u^{(i)}$  to yield a final state  $x_1^{(i)}$ . The interval reachable set is then approximated by the element-wise minimum and maximum of the  $x_1^{(i)}$ .

This approximation satisfies a probabilistic guarantee of correctness, provided that enough sample states are chosen [DA20]. Let  $[[\underline{R}, \overline{R}]]$  be the approximated reachable set,  $\epsilon, \delta \in (0, 1)$ , and  $m \geq (\frac{2n}{\epsilon}) \log(\frac{2n}{\delta})$ . Then, with probability  $1 - \delta$ , the approximation  $[[\underline{R}, \overline{R}]]$  satisfies  $P(R_{[t_0, t_1]} \setminus [[\underline{R}, \overline{R}]]) \leq \epsilon$ , where  $P(A)$  denotes the probability that a sampled initial state will yield a final state in the set  $A$ .

### 5.3 Data-Parallel Algorithms for Computing Interval Reachable Sets

Although most reachability methods are presented as serial algorithms, many of them have some inherent parallelism that can be exploited. One example of a tool that exploits parallelism is XSpeed [RGD<sup>+</sup>15], which implements a parallelized version of a support function-based reachability method. However, this parallel method is limited to linear systems, and in some cases only linear systems with invertible dynamics. Further, the parallelization is not suitable for massively parallel hardware: only some of the work (sampling of the support functions) is offloaded to the parallel device, so only a relatively small number of parallel processing elements may be employed. Here, on the other hand, we provide parallel algorithms of the three interval reachability methods introduced earlier. This allows handling general nonlinear systems. As we implement the algorithms later in pFaces, the algorithms can run on massively parallel hardware and consequently, can target extremely high-dimensional systems.

The bulk of the computational work in each of the methods introduced earlier is spent in ODE integration. Hence, the most effective approach by which to parallelize the three methods is to design a parallel ODE integration method. There are several available methods for parallelizing the task of ODE integration. Several popular methods for parallel ODE integration are parallel extensions of Runge-Kutta integration methods, which are the most popular serial methods for ODE integration. For these methods, the parallelization may be done in essentially two different ways [SI16]. First, they may be parallelized *across space*, in which case the computations associated to each state variable will be done in parallel. This allows for as many parallel computation elements as there are state variables. Second, they may be parallelized *across time*, in which case the time interval of integration is split up into several parts, and the solution on each sub-interval is computed in parallel. In contrast to parallelization across space, parallelization across time can be scaled to an arbitrary degree, as any number of sub-intervals may be used. However, the iterative solution by shooting introduces additional computations. To avoid this additional overhead, we introduce parallelizations across space. Since we target computing reachable sets for extremely high-dimensional systems, parallelization across space allows for a sufficient degree of parallelization in most cases.

## 5.3.1 Data-Parallel Runge-Kutta Scheme

---

**Algorithm 10:** State-parallelized fourth-order Runge-Kutta scheme for ODE integration.

---

**Input:** Initial state  $x_0$ ; input  $u$ ; and initial and final times  $t_0, t_1$ .

**Parameters:** State dimension  $n$ ; time step size  $h$  and ODE RHS  $f(t, x, u)$ .

**Output:** Final state  $x_1$  as the solution to the ODE at time  $t_1$ .

```

1 for all  $i \in \{1, \dots, n\}$  in parallel do
2    $k_{0,i} = k_{1,i} = k_{2,i} = k_{3,i} = \text{tmp}_i = 0$ ;
3    $x_{1,i} = x_{0,i}$ ;
4 end
5 for  $t \in \{t_0, t_0 + h, \dots, t_1 - h, t_1\}$  do
6   for all  $i \in \{1, \dots, n\}$  in parallel do
7      $k_{0,i} = f_i(t, x_1, u)$ ;  $\text{tmp}_i = x_1 + \frac{h}{2}k_{0,i}$ ;
8   end
9   for all  $i \in \{1, \dots, n\}$  in parallel do
10     $k_{1,i} = f_i(t + \frac{h}{2}, \text{tmp}, u)$ ;  $\text{tmp}_i = x_1 + \frac{h}{2}k_{1,i}$ ;
11  end
12  for all  $i \in \{1, \dots, n\}$  in parallel do
13     $k_{2,i} = f_i(t + \frac{h}{2}, \text{tmp}, u)$ ;  $\text{tmp}_i = x_1 + \frac{h}{2}k_{2,i}$ ;
14  end
15  for all  $i \in \{1, \dots, n\}$  in parallel do
16     $k_{3,i} = f_i(t + h, \text{tmp}, u)$ ;  $\text{tmp}_i = x_1 + \frac{h}{2}k_{2,i}$ ;
17     $x_{1,i} = x_{1,i} + \frac{1}{6}(k_{0,i} + 2k_{1,i} + 2k_{2,i} + k_{3,i})$ ;
18  end
19 end

```

---

Algorithm 10 shows a data-parallelization across space of the  $n$ -dimensional fourth-order Runge-Kutta scheme for ODE integration. It starts with a parallel initialization (steps 1-4) of the internal variables ( $k_0$ ,  $k_1$ ,  $k_2$ ,  $k_3$ , and  $\text{tmp}$ ) and the final state  $x_1$ . Then, for each quantized time step  $t$ , as demonstrated by the **for** loop in step 5, four parallel sections (starting with steps 6, 9, 12 and 15) are executed on all available PEs. Each parallel section is responsible for computing intermediate component-wise values of the integration and storing them in the memory spaces controlled by the corresponding internal variables. By the end of the last parallel section, step 18, the component-wise values of the solution to the ODE are computed for the current time step  $t$ .

In a parallelization across space, the separation of integration results in less synchronization overhead between the threads executing the parallel code. All that is required is to make sure that the threads are synchronized after each of the parallel sections, which is already achieved in the algorithm using separate parallel **for** loops. This data-parallel approach fits massively-parallel CUs, such as GPUs and HWAs.

### 5.3 Data-Parallel Algorithms for Computing Interval Reachable Sets

For a system with  $n$  dimensions and assuming all PEs have uniform access time to the memory space, Algorithm 10 scales linearly as the number of PEs (denoted by  $P$ ) increases. In a computer with a single PE (i.e.,  $P = 1$ ), the algorithm reduces to the original traditional serial algorithm. Let  $T$  be the time needed to run algorithm on such single-PE computer. Then, if a parallel computer has  $P$  PEs of same type, where  $P \leq n$ , users may expect a speedup in time of up to ideally  $P$  times. Here, each PE will be responsible for computing  $n/P$  components of the state vector.

**Remark 5.3.1.** For fixed initial and final times,  $t_0$  and  $t_1$ , and variable number of PEs  $P$ , the time complexity of Algorithm 10 is  $O(\frac{n}{P})$ . Using  $P > n$  as a variable number of PEs is a waste of computation resources since one or more PEs will be left idle.

#### 5.3.2 Parallelizing Interval Reachability Methods

---

**Algorithm 11:** Parallel contraction/growth bound method.

---

**Input:** System  $\Sigma$ , interval initial set  $\mathcal{X}_0$ , interval input set  $\mathcal{U}$ , initial time  $t_0$ , and final time  $t_1$ .  
**Output:** Reachable set  $\hat{R}_{[t_0, t_1]}$ .

- 1 **for all**  $i \in \{1, \dots, n\}$  **in parallel do**
- 2     Set  $c_{0,i}$  as the center of the interval  $\mathcal{X}_{0,i}$ ;
- 3     Set  $r_{0,i}$  as the radius of the interval  $\mathcal{X}_{0,i}$ ;
- 4 **end**
- 5 Set  $c_1$  as the result of running Algorithm 10 with  $c_0$  as initial state, the center of  $\mathcal{U}$  as input, and  $f$  as RHS function;
- 6 Set  $r_1$  as the result of running Algorithm 10 with  $r_0$  as initial state, the half-width of  $\mathcal{U}$  as input, and  $g$  as RHS function;
- 7 **for all**  $i \in \{1, \dots, n\}$  **in parallel do**
- 8     Set  $\hat{R}_{[t_0, t_1], i} = [c_{1,i} - r_{1,i}, c_{1,i} + r_{1,i}]$ ;
- 9 **end**

---

Algorithm 11 presents the parallel version of the contraction/growth bound method. It uses Algorithm 10 twice. First, it is used to compute the solution of the system's ODE  $f$  for the center of the initial set  $\mathcal{X}_0$ . Then, it is used to compute the growth/contraction of the initial set  $\mathcal{X}_0$  by solving the ODE  $g$  of the growth dynamics. Finally, the reachable set is constructed from the computed center and radius values.

**Remark 5.3.2.** Since Algorithm 11 uses Algorithm 10 directly, its time complexity is also  $O(\frac{n}{P})$ , for fixed  $t_0$  and  $t_1$ , and variable  $P$ .

Algorithm 12 presents the parallel version of the method based on the mixed monotonicity. The parallelized implementation of the mixed-monotonicity method uses only one call to Algorithm 10 to integrate the embedding system. First, the upper and lower bound vectors of the initial sets are extracted in parallel. Then, Algorithm 10 is called

---

**Algorithm 12:** Parallel mixed monotonicity method.

---

**Input:** System  $\Sigma$ , interval initial set  $\mathcal{X}_0$ , decomposition function  $d$  as defined in (5.2.1), interval input set  $\mathcal{U}$ , initial time  $t_0$ , and final time  $t_1$ .

**Output:** Reachable set  $\hat{R}_{[t_0, t_1]}$ .

```

1 for all  $i \in \{1, \dots, n\}$  in parallel do
2   | Set  $x_{0,i}$  as the lower bound of the interval  $\mathcal{X}_{0,i}$ ;
3   | Set  $\hat{x}_{0,i}$  as the upper bound of the interval  $\mathcal{X}_{0,i}$ ;
4   | Set  $u_i$  as the lower bound of the interval  $\mathcal{U}_i$ ;
5   | Set  $\hat{u}_i$  as the upper bound of the interval  $\mathcal{U}_i$ ;
6 end
7 Run Algorithm 10 with  $[x_0 \hat{x}_0]^T$  as initial state,  $[u \hat{u}]^T$  as input, and  $d$  as RHS
  function;
8 Store the result of Algorithm 10 in  $[x_1 \hat{x}_1]^T$ ;
9 for all  $i \in \{1, \dots, n\}$  in parallel do
10  | Set  $\hat{R}_{[t_0, t_1], i} = [x_{1,i}, \hat{x}_{1,i}]$ ;
11 end

```

---

once but with an augmented vector constructed by the upper and lower bounds, duplicated ODE RHS and inputs. This is similar to running Algorithm 10 twice.

**Remark 5.3.3.** *Since Algorithm 12 uses Algorithm 10 directly, its time complexity is also  $O(\frac{n}{P})$ , for fixed  $t_0$  and  $t_1$ , and variable  $P$ . However, since Algorithm 12 is run on a system of dimension  $2n$ , it requires twice as much memory as Algorithm 11.*

Since Algorithm 12 requires twice as much memory as Algorithm 11, it is limited to systems with half the dimensions for the same HWC.

Algorithm 13 presents the parallelized implementation of the Monte Carlo method. Algorithm 13 uses Algorithm 10  $m$  times, for  $m$  sampled initial states, for which each time Algorithm 10 is run on an input vectors of dimension  $n$ . The implementation uses two levels of parallelization. The first level is a set of parallel threads over the samples used for simulations. Then, within each thread, another parallel set of threads are launched, as a result of calling Algorithm 10. This is realized as one parallel job of  $m \times n$  threads. Consequently, the Monte Carlo method has a complexity of  $O(\frac{mn}{p})$ . Since only the element-wise minima and maxima of the sampled states need to be stored, this method only requires as much memory as the growth bound method.

## 5.4 Case Studies

The algorithms introduced in the previous section are implemented in tool PIRK as a kernel on top of pFaces in a process similar to the one discussed in Chapter 4. We present some case studies to demonstrate that the introduced algorithms can be used to compute interval forward reachable sets for high-dimensional nonlinear systems. In each of the case studies to follow, we report the time kernel PIRK takes to compute reachable

---

**Algorithm 13:** Parallel Monte Carlo simulation method.

---

**Input:** System  $\Sigma$ , interval initial set  $\mathcal{X}_0$ , interval input set  $\mathcal{U}$ , initial time  $t_0$ , final time  $t_1$ , and probabilistic guarantee parameters  $\epsilon$  and  $\delta$ .

**Output:** Reachable set  $\hat{R}_{[t_0, t_1]}$ .

- 1 Set number of samples  $m = \lceil \frac{2n}{\epsilon} \log(\frac{2n}{\delta}) \rceil$ ;
- 2 **for all**  $i \in \{1, \dots, m\}$  **in parallel do**
- 3     Sample  $x_0^{(i)}$  uniformly from  $\mathcal{X}_0$ ;
- 4     Sample  $u^{(i)}$  uniformly from  $\mathcal{U}$ ;
- 5     Run Algorithm 10 with  $x_0^{(i)}$  as initial state,  $u^{(i)}$  as input, and  $f$  as RHS function;
- 6     Store the result of Algorithm 10 in  $x_1^{(i)}$ ;
- 7 **end**
- 8 **for all**  $j \in \{1, \dots, n\}$  **in parallel do**
- 9     Set  $\hat{R}_{[t_0, t_1], j} = [\min_{i \in \{1 \dots m\}} x_{1, j}^{(i)}, \max_{i \in \{1 \dots m\}} x_{1, j}^{(i)}]$
- 10 **end**

---

sets for systems of varying dimension using all three of its methods on a variety of parallel computing platforms. We perform some of the same tests using the serial tool TIRA, to measure the speedup gained by PIRK's ability to use massively parallel hardware.

We set a time limit of 1 hour for all the targeted case studies, and report the maximum dimensions that could be reached under this limit. We use four AWS machines for the computations with PIRK:

- (1) `m4.10xlarge` which has a CPU with 40 PEs,
- (2) `c5.24xlarge` which a CPU with 96 PEs,
- (3) `g3.4xlarge` which has a GPU with 2048 PEs, and
- (4) `p3.2xlarge` which has a GPU with 5120 PEs.

For the computations with TIRA, we used a machine with a 3.6 GHz Intel i7 CPU.

The two most common sources of extremely high-dimensional systems are discretizations of continuum models (e.g. discretized Partial Differential Equations (PDEs)) and swarms of independent agents. Three of the following case studies come from these sources. Such systems naturally have sparse dynamics, which PIRK uses to speed up the ODE integration and reduce the memory usage. However, sparsity is not a requirement for PIRK to be effective, and we have used systems with dense dynamics for test cases as well.

#### 5.4.1 Multi-Link Road Traffic Model

We consider the road traffic analysis problem reported in [CA18], a proposed benchmark for formal controller synthesis. We are interested in the density of cars along a single one-

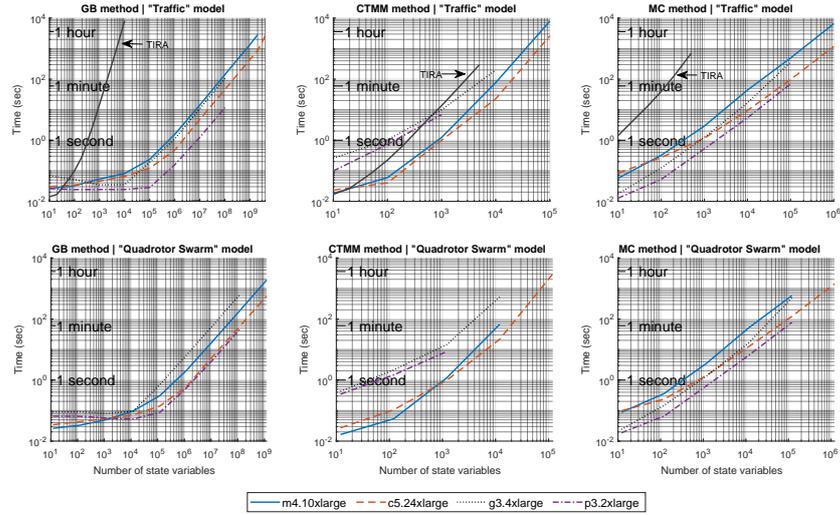


Figure 5.2: Speed test results for kernel PIRK.

way lane. The lane is divided into  $n$  segments, and the density of cars in each segment is a state variable. The continuous-time dynamics for the segment densities are derived from a spatially discretized version of the Cell Transmission Model [CA15], which have been also used in conjunction with abstraction-based formal control synthesis [CAB17].

For the single-road case described here, the dynamics are

$$\begin{aligned} \dot{x}_0 &= \frac{1}{T} (\beta \min(c, vx_1, w(\bar{x} - x_2)/\beta)) \\ \dot{x}_i &= \frac{1}{T} (\beta \min(c, vx_{i-1}, w(\bar{x} - x_i)/\beta) - \min(c, vx_i, w(\bar{x} - x_{i+1})/\beta)) \\ \dot{x}_{n-1} &= \frac{1}{T} (\beta \min(c, vx_{i-1}, w(\bar{x} - x_i)/\beta) - \min(c, vx_i)/\beta) \end{aligned}$$

where  $x_i$  represents the density of traffic in the  $i$ th discretized segment.  $T$ ,  $\beta$ ,  $v$ ,  $c$  and  $w$  are some constants. Since a system of this form can easily be defined for any natural number  $n$ , this system is a good candidate for testing the effectiveness of the parallelized reachability methods as a function of state dimension. This is a nonlinear system with sparse coupling between the state variables, in the sense that the dynamics for each state variable  $x_i$  depends only on itself and its neighbors  $x_{i-1}$  and  $x_{i+1}$ . This means that the system's Jacobian matrix is sparse, which PIRK can use to reduce the used memory.

The results of the speed test are shown in the first row of Figure 5.2. The machines `m4.10xlarge` and `c5.24xlarge` reach up to 2 billion and 4 billion dimensions, respectively, using the growth/contraction method, in 47.3 minutes and 44.7 minutes, respectively. Due to memory limitations of the GPUs, the machines `g3.4xlarge` and `p3.2xlarge` both reach up to 400 million in 106 seconds and 11 seconds, respectively.

The relative improvement of PIRK's computation time over TIRA's is significantly larger for the growth bound method than for the other two. This difference stems from a

difference in how each tool computes the half-width of the reachable set from the radius dynamics. TIRA solves the radius dynamics by computing the full matrix exponential using MATLAB’s `expm`, whereas TIRA directly integrates the dynamics using parallel Runge-Kutta. This caveat applies to the next case study as well.

### 5.4.2 Quadrotor Swarm

The second test system is a swarm of  $K$  identical quadrotors. The dynamics for an individual quadrotor is given as follows

$$\begin{aligned}\ddot{p}_n &= \frac{F}{m}(-\cos(\phi)\sin(\theta)\cos(\psi) - \sin(\phi)\sin(\psi)) \\ \ddot{p}_e &= \frac{F}{m}(-\cos(\phi)\sin(\theta)\sin(\psi) + \sin(\phi)\cos(\psi)) \\ \ddot{h} &= g - \frac{F}{m}\cos(\phi)\cos(\theta) \\ \ddot{\phi} &= \frac{1}{J_x}\tau_\phi \\ \ddot{\theta} &= \frac{1}{J_y}\tau_\theta \\ \ddot{\psi} &= \frac{1}{J_z}\tau_\psi\end{aligned}$$

where  $p_n$  and  $p_e$  denote the y-axis (“north”) and x-axis (“east”) position of the quadrotor,  $h$  its height, and  $\phi$ ,  $\theta$ , and  $\psi$  its pitch, roll, and yaw angles respectively. The system dynamics of each quadrotor model are derived in a similar way to the model used in the ARCH competition [IAC<sup>+</sup>18], with the added simplification of a small angle approximation in the angular dynamics and the neglect of Coriolis force terms. A derivation of both models is available in [Bea08]. Similar to the n-link traffic model in the previous subsection, this system is convenient for scaling: system consisting of one quadrotor can be expressed with 12 states, so the state dimension of the swarm system is  $n = 12K$ .

The results of the speed test are shown in figure 5.2 (second row). The machines `m4.10xlarge` and `c5.24xlarge` reach up to 1.8 billion dimensions and 3.6 billion dimensions, respectively, (using the growth/contraction method) in 48 minutes and 32 minutes, respectively. The machines `g3.4xlarge` and `p3.2xlarge` both reach up to 120 million dimensions in 10.6 minutes and 46 seconds, respectively.

### 5.4.3 Quadrotor Swarm with Artificial Potential Field

The third test system is a modification of the quadrotor swarm system which adds interactions between the quadrotors. In addition to the base quadrotor dynamics described in Section 5.4.2, this model augments each quadrotor with an artificial potential field to guide the quadrotors to the origin while avoiding their collisions.

This controller applies a force to the quadrotors that seeks to minimize a *artificial potential*  $U$  that depends on the position of all of the quadrotors. The potential applied

## 5 Efficient Algorithms for the Computation of Reachable Sets

to each quadrotor is intended to repel it from the other quadrotors, and to attract it towards the origin. Each quadrotor will be subject to three potential fields, which separately consider  $p_n$ ,  $p_e$ , and  $h$ . This allows us to ensure that the forces resulting from the potential fields are contained in an interval set.

The potential fields for the  $i^{\text{th}}$  quadrotor are

$$\begin{aligned} U_{i,p_n} &= F_r e^{-\min_{j \neq i} |p_n^{(i)} - p_n^{(j)}|} + F_a |p_n^{(i)}| \\ U_{i,p_e} &= F_r e^{-\min_{j \neq i} |p_e^{(i)} - p_e^{(j)}|} + F_a |p_e^{(i)}| \\ U_{i,h} &= F_r e^{-\min_{j \neq i} |h^{(i)} - h^{(j)}|} + F_a |e^{(i)}|, \end{aligned}$$

where the  $(i)$  and  $(j)$  superscripts denote state variables belonging to the  $i^{\text{th}}$  and  $j^{\text{th}}$  quadrotors respectively. The magnitude of this potential field depends on the distance from the origin and the position of the nearest quadrotor in the  $p_n$ ,  $p_e$ , and  $h$  directions, so the control action arising from this potential field will tend to guide a quadrotor away from whichever other quadrotor is nearest to it while also guiding it towards the origin.

The forces applied to the system in each direction are just the negative derivatives of each direction's potential field. Therefore, the dynamics for quadrotor  $i$  are

$$\begin{aligned} \ddot{p}_n &= \frac{F}{m} (-\cos(\phi) \sin(\theta) \cos(\psi) - \sin(\phi) \sin(\psi)) + F_{i,p_n} \\ \ddot{p}_e &= \frac{F}{m} (-\cos(\phi) \sin(\theta) \sin(\psi) + \sin(\phi) \cos(\psi)) + F_{i,p_e} \\ \ddot{h} &= g - \frac{F}{m} \cos(\phi) \cos(\theta) + F_{i,h} \\ \ddot{\phi} &= \frac{1}{J_x} \tau_\phi \\ \ddot{\theta} &= \frac{1}{J_y} \tau_\theta \\ \ddot{\psi} &= \frac{1}{J_z} \tau_\psi, \end{aligned}$$

where  $F_{i,p_n}$ ,  $F_{i,p_e}$ , and  $F_{i,h}$  are the forces induced by the artificial potential fields. These have the form

$$\begin{aligned} F_{i,p_n} &= -\frac{\partial U_{i,p_n}}{\partial p_n^{(i)}} = F_r \operatorname{sgn}(p_n^{(i)} - p_n^{(j^*)}) e^{-|p_n^{(i)} - p_n^{(j^*)}|} - F_a \operatorname{sgn}(p_n^{(i)}) \\ F_{i,p_e} &= -\frac{\partial U_{i,p_e}}{\partial p_e^{(i)}} = F_r \operatorname{sgn}(p_e^{(i)} - p_e^{(j^*)}) e^{-|p_e^{(i)} - p_e^{(j^*)}|} - F_a \operatorname{sgn}(p_e^{(i)}) \\ F_{i,h} &= -\frac{\partial U_{i,h}}{\partial h^{(i)}} = F_r \operatorname{sgn}(h^{(i)} - h^{(j^*)}) e^{-|h^{(i)} - h^{(j^*)}|} - F_a \operatorname{sgn}(h^{(i)}), \end{aligned}$$

where  $j^* = \operatorname{argmin}_{j \neq i} |x_i - x_j|$  in the  $F_{i,x}$  equation, and is defined analogously for the other two. Each force is bounded by the interval  $[-(F_r + F_a), F_r + F_a]$ . The second derivatives of the potential functions, which appear in the system Jacobian, are bounded

**Table 5.1:** Results for running PIRK to compute the reach set of the quadrotors swarm with artificial potential field.  $N/M$  means that the machine does not have enough memory to compute the reachable set.

Method	No. of States	Memory (MB)	Time (seconds)			
			m4.10xlarge	c5.24xlarge	g3.4xlarge	p3.2xlarge
GB	1200	2.8	$\leq 1.0$	$\leq 1.0$	4.3	$\leq 1.0$
GB	12000	275.3	$\leq 1.0$	$\leq 1.0$	47.1	$\leq 1.0$
GB	120000	27,473.1	69.6	68.3	N/M	N/M
MC	1200	45.7	1.0	$\leq 1.0$	2.0	$\leq 1.0$
MC	12000	457.5	56.8	23.7	233.1	40.6
MC	120000	4577.6	$\geq 2h$	3091.8	N/M	5081.0

by the interval  $[-F_r, F_r]$  almost everywhere. Due to the interaction of the state variables in the  $F_{i,p_n}$ ,  $F_{i,p_e}$ , and  $F_{i,h}$  terms, this system has a dense Jacobian. In particular, at least 25% of the Jacobian elements will be nonzero for any number of quadrotors.

Table 5.1 shows the times of running PIRK using his system within the four machines `m4.10xlarge`, `c5.24xlarge`, `g3.4xlarge` and `p3.2xlarge` in AWS. Due to the high density of this example, the required memory grows very fast. Here, we focus on the growth bound and the Monte-Carlo methods since they require less memory. For the Monte-Carlo method, we fix the number of samples to 1000 samples.

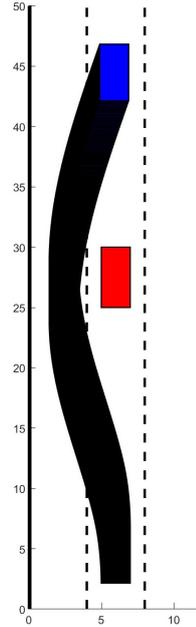
PIRK managed to compute the reach sets of systems up to 12,000 state variables (i.e., 1,000 quadrotors). Up to 120 states, all machines solve the problems in less than one second. Some of the machines lack the required memory to solve the problems with large memory requirement (e.g., a 27.7 GB memory is required to compute the reach set of the system with 12,000 state variables using the growth bound method).

#### 5.4.4 Heat Diffusion

The fourth test system is a model for the diffusion of heat in a 3-dimensional cube. The model is based on a benchmark used in [BTJ19] to test a method for numerical verification of affine systems. The solid is a cube of unit length, which is insulated on all sides but one: the non-insulated side can exchange heat with the outside environment, which is assumed to be at zero temperature. A portion of the cube is heated at an initial time, and the heat diffuses through the rest of the cube as time progresses, until all the heat has left through the non-insulated side. The dynamics for the transfer of heat through the cube are governed by the *heat equation*, a classical second-order linear partial differential equation:

$$\frac{\partial u}{\partial t} = \alpha \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right),$$

where  $u(t; x, y, z)$  denotes the temperature of the cube at coordinates  $x$ ,  $y$ , and  $z$  at time  $t$ , and  $\alpha$  is the diffusivity of the material of the cube. A model of the form  $\dot{x} = f(t, x, u)$



**Figure 5.3:** Interval flow pipe approximating the behavior of the BMW 320i car.

which approximates the heat transfer through the cube according to the heat equation can be obtained by discretizing the cube into an  $\ell \times \ell \times \ell$  grid, yielding a system with  $\ell^3$  states. The temperature at each grid point is taken as a state variable. Each spatial partial derivative is replaced with its first-order discrete approximation based on the discretized state variables, accounting for boundary conditions. Since the heat equation is a linear PDE, the discretized system is linear.

We take a fixed state dimension of  $n = 10^9$  by fixing  $\ell = 1000$ . This example uses a finer time-sampling constant compared to other examples. Integration takes place over  $[t_0, t_1] = [0, 20]$  with time step size  $h = 0.02$ , leading to 1000 integration steps. PIRK solves the problem on `m4.10xlarge` in 472 minutes, and in 350.2 minutes on `c5.24xlarge`.

#### 5.4.5 Overtaking Maneuver on Highway

This and the remaining case studies focus on models of practical importance which have relatively low state dimension. Although the introduced algorithms are designed to perform well on high-dimensional systems, they are also effective at quickly computing reachable sets for lower-dimensional systems, for applications that need many reachable sets.

We consider the BMW's 7-dimensional continuous-time model introduced in Chapter 4. We verify the safety of a lane change maneuver in a highway. We consider a step-size of 0.005 seconds between the reach sets in a time window of 6.5 seconds. This results in computing 1300 reach sets of the system.

**Table 5.2:** Results from running PIRK (growth bound method) to compute the reach sets for the examples reported in the ARCH-2018 competition.

Benchmark model	PIRK	CORA	CORA/SX	C2E2	Flow*	Isabelle	SymReach
Van der Pol (2 states)	0.13	2.3	0.6	38.5	1.5	1.5	17.14
Laub-Loomis (7 states)	0.04	0.82	0.85	0.12	4.5	10	1.93
Quadrotor (12 states)	0.01	5.2	1.5	-	5.9	30	2.96

We are interested in verifying the autonomous operation of the vehicle. We fix an input that performs a maneuver to overtake a car in the middle lane of a 3-lane highway. We ran PIRK with the growth-bound method and extracted the reach pipe to verify the maneuver is applied successfully. Within an Intel i7 processor with 8GB of Random Access Memory (RAM), PIRK managed to compute the reach pipe in 0.25 seconds. The reach pipe is shown in Fig. 5.3.

#### 5.4.6 Performance on ARCH Benchmarks

In order to compare PIRK’s performance to existing toolset, we tested PIRK’s growth bound implementation on three systems from the ARCH-COMP’18 category report for systems with nonlinear dynamics [IAC<sup>+</sup>18]. This report contains benchmark data from several popular reachability analysis tools (C2E2, CORA, Flow\*, Isabelle, SpaceEx, and SymReach) on several nonlinear reachability problems with state dimensions between 2 and 12.

Table 5.2 compares the computation times for PIRK on the three systems are to those in the report. The results are compared to the times reported by other tools in [IAC<sup>+</sup>18]. All times are in seconds. PIRK ran on an i9 CPU, while the others ran on i7 and i5 (see [IAC<sup>+</sup>18] for more hardware details).

PIRK solves each of the benchmark problems faster than the other tools. Both of the used i7 and i9 processors have multi processors (from 6 cores to 8 cores). The advantage of PIRK is its ability to utilize all available cores.

## 5.5 Summary

Reachable sets play an important role in several formal methods approaches including the verification of behaviors of control systems and the synthesis of their controllers. In previous chapters, the computation of reachable sets is considered implicitly during the construction of abstractions. In this chapter, we provided efficient parallel algorithms for the computation of reachable sets for extremely high-dimensional systems. Using simple parallelizations of interval reachability analysis techniques, we were able to compute reachable sets for nonlinear systems faster and at higher dimensions than all existing tools.

Interval reachability was selected as candidate technique for the construction of reachable sets as it is already used in many symbolic control approaches. Certain methods

for computing interval reachable sets were selected. More precisely, we discussed the following three simulation-based methods for computing interval reachable sets: (1) the contraction/growth bound method, (2) the mixed monotonicity method, and (3) a Monte Carlo based method. They allow for different trade-offs between conservatism and computation speed, as well as different forms of problem data. We showed that the bulk of the computational work in all methods of computing the reach sets is spent in ODE integration. Hence, we concluded that the most effective approach by which to parallelize these methods is to design a parallel ODE integration method.

Algorithm 10 introduced a data-parallelization across space of the  $n$ -dimensional fourth-order Runge-Kutta scheme for ODE integration. For a system with  $n$  dimensions and assuming all PEs have uniform access time to the memory space, Algorithm 10 scales linearly as the number of PEs increases. Algorithm 11 presented the parallel version of the contraction/growth bound method, Algorithm 12 presented the parallel version of the method based on the mixed monotonicity method, and Algorithm 13 presented the parallelized implementation of the Monte Carlo method. All algorithms allow controlling the computational complexity by introducing more PEs. The algorithms introduced in this chapter are implemented in tool PIRK as a kernel on top of pFaces.

We also presented some case studies to demonstrate that the introduced algorithms can be used to compute interval forward reachable sets for high-dimensional nonlinear systems. The performance increase shown by all case studies comes from pFaces's ability to use massively parallel hardware such as GPUs, CPUs, and clusters, to accelerate the parallelized simulation-based methods. Of the three interval reachability methods implemented, the growth bound method was the most efficient.

## 6 Efficient Algorithms for Stochastic Symbolic Control

Stochastic systems are an important modeling framework to describe many safety-critical CPSs such as power grids and traffic networks. For such complex systems, automating the synthesis of controllers that achieve some high-level specifications is inherently very challenging. Symbolic models can be employed as replacements of original systems in the controller synthesis procedure. More precisely, one can abstract a given original stochastic system by a finite MDP, and then performs the analysis and synthesis over the MDP using algorithmic techniques from computer science [BK08]. The results are then carried back to the original system, while providing a guaranteed error bound.

There exist few tools for synthesizing controllers of stochastic control systems. To the best of our knowledge, all available tools for automatic synthesis of formally-correct controllers use finite MDPs as abstractions that capture the stochasticity in original systems. Unfortunately, constructing finite MDPs for dynamical systems suffers from the CoD. Tool FAUST [SGA15] generates MDPs from uncountable-state discrete-time stochastic processes, and performs verification and synthesis for safety and reachability specifications. However, it is originally implemented in MATLAB and suffers severely from the CoD. Tool Stochy [CDA19] can analyze discrete-time stochastic hybrid systems, construct interval-MDPs, and perform verification and synthesis tasks. Although it is implemented in C++, it also suffers the CoD because of its serial implementation. In practice, both tools are limited to small-sized systems (up to 2 dimensions).

In this chapter, we propose scalable parallel algorithms and efficient distributed data structures for constructing finite MDPs of large-scale stochastic systems and automating the computation of their correct-by-construction controllers. The proposed algorithms handle stochastic systems with noises and bounded disturbances. We implement the algorithms as a kernel on top of pFaces and compare the results with existing similar tools. The implemented kernel is referred to as AMYTISS [LKSZ20]. We then apply the implementation to some real-world applications including a traffic network and an autonomous vehicle.

The implementations discussed in this chapter differs from all available tools in two main directions. First, we consider parallel algorithms and distributed data structures and target HPC platforms to reduce the effects of the CoD. Additionally, we target stochastic systems with bounded disturbances which, e.g., in case of interconnected systems, can be used to model the effects of other subsystems (internal inputs) as bounded disturbances. The mentioned tools, unlike this work, can only handle disturbance-free systems.

## 6.1 Discrete-Time Stochastic Control Systems (dt-SCS)

We recall some concepts needed throughout this chapter. A topological space  $S$  is a Borel space if it is homeomorphic to a Borel subset of a Polish space (i.e., a separable and completely metrizable space). The Euclidean spaces  $\mathbb{R}^n$ , its Borel subsets endowed with a subspace topology, and hybrid spaces are examples of Borel spaces. A Borel sigma-algebra is denoted by  $\mathcal{B}(S)$ , and any Borel space  $S$  is assumed to be endowed with it. A map  $f : S \rightarrow Y$  is measurable if it is Borel measurable.

A probability space is a tuple  $(\Omega, \mathcal{F}_\Omega, \mathbb{P}_\Omega)$ , where  $\Omega$  is the sample space,  $\mathcal{F}_\Omega$  is a sigma-algebra on  $\Omega$ , which comprises subsets of  $\Omega$  as events, and  $\mathbb{P}_\Omega$  is a probability measure that assigns probabilities to events. A random variable  $\mathcal{X}$  is a measurable function  $\mathcal{X} : \mathcal{F}_\Omega \rightarrow \mathcal{F}_\mathcal{X}$  inducing a probability measure  $Prob\{A\} := \mathbb{P}_\Omega\{\mathcal{X}^{-1}(A)\}$  for any  $A \in \mathcal{F}_\mathcal{X}$ . Here, for simplicity, we directly present the probability measure on the space of  $\mathcal{X}$  without explicitly mentioning the underlying probability space.

Now, we formally introduce Discrete-time Stochastic Control Systems (dt-SCSs).

**Definition 6.1.1.** *A dt-SCS is a tuple*

$$\Sigma := (X, U, W, \varsigma, f, Y, h), \quad (6.1.1)$$

where  $X \subseteq \mathbb{R}^n$  is a Borel space representing the state set and  $(X, \mathcal{B}(X))$  is its measurable space,  $U \subseteq \mathbb{R}^m$  is a Borel space representing the input set,  $W \subseteq \mathbb{R}^p$  is a Borel space representing a disturbance set,  $\varsigma$  is a sequence of independent and identically distributed (i.i.d.) random variables from a sample space  $\Omega$  to a measurable set  $V_\varsigma$  and it is defined as follows:

$$\varsigma := \{\varsigma(k) : \Omega \rightarrow V_\varsigma, k \in \mathbb{N}\},$$

$f : X \times U \times W \times V_\varsigma \rightarrow X$  is a measurable function characterizing the state evolution of the system,  $Y \subseteq \mathbb{R}^q$  is a Borel space as an output set, and  $h : X \rightarrow Y$  is a measurable function that maps a state  $x \in X$  to its output  $y = h(x)$ .

The state evolution of  $\Sigma$ , for a given initial state  $x(0) \in X$ , an input sequence  $\nu(\cdot) : \mathbb{N} \rightarrow U$ , and a disturbance sequence  $w(\cdot) : \mathbb{N} \rightarrow W$ , is characterized by the difference equations

$$x(k+1) = f(x(k), \nu(k), w(k)) + \Upsilon(k), \quad k \in \mathbb{N}, \quad (6.1.2)$$

where  $\Upsilon(k) := \varsigma(k)$  with  $\mathcal{V}_\varsigma := \mathbb{R}^n$  for the case of the additive noise, and  $\Upsilon(k) := \varsigma(k)x(k)$  with  $\mathcal{V}_\varsigma$  equals to the set of diagonal matrices of the dimension  $n$  for the case of the multiplicative noise [WTS05]. We keep the notation  $\Sigma$  to indicate both cases and use respectively  $\Sigma_a$  and  $\Sigma_m$  when discussing these cases individually.

**Remark 6.1.2.** *Although we assume discrete-time systems, an extension to support continuous-time system is possible in the same way introduced in Subsection 4.3.1. More precisely, continuous-time systems as given in (4.2.1) are approximated to discrete-time systems using forward Euler method and the local truncation error of the approximation is considered as disturbance by inflating set  $W$ .*

We should mention that the parallel algorithms we plan to introduce in this chapter are generally independent of the noise distribution. Only for an easier presentation, we present the algorithms and case studies based on normal distributions knowing that they can be extended to support other practical distributions including uniform, exponential, and beta.

**Definition 6.1.3.** *For the dt-SCS  $\Sigma$  in (6.1.1), a Markov policy is a sequence  $\rho := (\rho_0, \rho_1, \rho_2, \dots)$  of universally measurable stochastic kernels  $\rho_n$  [BS96], each defined on the input space  $U$  given  $X \times W$  and such that for all  $(x_n, w_n) \in X \times W$ ,  $\rho_n(U \mid (x_n, w_n)) = 1$ . The class of all such Markov policies is denoted by  $\Pi_M$ .*

We are interested in Markov policies to control the dt-SCS system in (6.1.1). One may be also interested in analyzing dt-SCSs without disturbances. In this case, the tuple (6.1.1) reduces to

$$\Sigma := (X, U, \varsigma, f, Y, h), \quad (6.1.3)$$

with  $f : X \times U \times V_\varsigma \rightarrow X$ , and the evolution equation (6.1.2) can be re-written as follows:

$$x(k+1) = f(x(k), \nu(k), \varsigma(k)), \quad k \in \mathbb{N}.$$

In the next section, we formally define MDPs and discuss how to build finite ones from given dt-SCSs.

## 6.2 Markov Decision Processes (MDPs) as Symbolic Models

A dt-SCS  $\Sigma$  is equivalently represented by an MDP [HSA17]:

$$\Sigma = (X, U, W, T_x, Y, h),$$

where the map  $T_x : \mathcal{B}(X) \times X \times U \times W \rightarrow [0, 1]$  is a conditional stochastic kernel that assigns to any  $x \in X$ ,  $\nu \in U$ , and  $w \in W$ , a probability measure  $T_x(\cdot \mid x, \nu, w)$  on the measurable space  $(X, \mathcal{B}(X))$  so that for any set  $\mathcal{A} \in \mathcal{B}(X)$ , we have

$$\mathbb{P}(x(k+1) \in \mathcal{A} \mid x(k), \nu(k), w(k)) := \int_{\mathcal{A}} T_x(d\bar{x} \mid x(k), \nu(k), w(k)).$$

For given input  $\nu(\cdot)$ , and disturbance  $w(\cdot)$ , the stochastic kernel  $T_x$  captures the evolution of the state of  $\Sigma$  and can be uniquely determined by the pair  $(\varsigma, f)$  from (6.1.1). In other words,  $T_x$  contains the information of function  $f$  and the distribution of noise  $\varsigma(\cdot)$  in the dynamical representation.

The alternative representation as MDP is utilized in [SAM15] to approximate a dt-SCS  $\Sigma$  with a finite MDP  $\hat{\Sigma}$ . Algorithm 14, which is adopted from [SAM15], presents such approximation. The algorithm first constructs a finite partition of state set  $X$ , input set  $U$ , and disturbance set  $W$ . Then, representative points  $\bar{x}_i \in X_i$ ,  $\bar{\nu}_i \in U_i$  and  $\bar{w}_i \in W_i$  are selected as abstract states, inputs and disturbances. Transition probabilities in the finite MDP  $\hat{\Sigma}$  are also computed according to (6.2.1). The output map  $\hat{h}$  is the same as  $h$  with their domain restricted to finite state set  $\hat{X}$  (Step 7), and the output set  $\hat{Y}$  is just image of  $\hat{X}$  under  $h$  (Step 6).

**Algorithm 14:** Abstraction of dt-SCS  $\Sigma$  by a finite MDP  $\widehat{\Sigma}$ **Input:** dt-SCS  $\Sigma = (X, U, W, T_x, Y, h)$ .**Output:** Finite MDP  $\widehat{\Sigma} = (\widehat{X}, \widehat{U}, \widehat{W}, \widehat{T}_x, \widehat{Y}, \widehat{h})$ .

- 1 Select finite partitions of  $X, U$ , and  $W$  s.t.  $X = \cup_{i=1}^{n_x} X_i$ ,  $U = \cup_{j=1}^{n_u} U_j$ ,  
 $W = \cup_{k=1}^{n_w} W_k$ ;
- 2 For each  $X_i, U_j$ , and  $W_k$ , select unique representative points  $\bar{x}_i \in X_i$ ,  $\bar{u}_j \in U_j$ ,  
 $\bar{w}_k \in W_k$ ;
- 3 Define  $\widehat{X} = \{\bar{x}_i, i = 1, \dots, n_x\}$  as a finite state set,  $\widehat{U} = \{\bar{u}_j, j = 1, \dots, n_u\}$  as a  
finite input set, and  $\widehat{W} = \{\bar{w}_k, k = 1, \dots, n_w\}$  as a finite disturbance set;
- 4 Define the map  $\Xi : X \rightarrow 2^{\widehat{X}}$  that assigns to any  $x \in X$ , the corresponding  
partition element it belongs to, i.e.,  $\Xi(x) = X_i$  if  $x \in X_i$ ;
- 5 For all  $x, x' \in \widehat{X}, \nu \in \widehat{U}, w \in \widehat{W}$ , compute the discrete transition probability  
matrix  $\widehat{T}_x$  as follows:

$$\widehat{T}_x(x' | x, \nu, w) := T_x(\Xi(x') | x, \nu, w). \quad (6.2.1)$$

- 6 Define  $\widehat{Y} = h(\widehat{X})$  as an output set;
- 7 Define  $\widehat{h} = h |_{\widehat{X}}$  as an output map;

**Remark 6.2.1.** Since  $\widehat{X}, \widehat{U}$  and  $\widehat{W}$  are finite sets,  $\widehat{T}_x$  is a static map. It can be represented with a matrix and we will refer to it hereinafter as the transition probability matrix.

Given a dt-SCS  $\Sigma = (X, U, W, \varsigma, f, Y, h)$ , the finite MDP  $\widehat{\Sigma}$  constructed in Algorithm 14, can be represented as an finite dt-SCS:

$$\widehat{\Sigma} := (\widehat{X}, \widehat{U}, \widehat{W}, \varsigma, \widehat{f}, \widehat{Y}, \widehat{h}),$$

where  $\widehat{f} : \widehat{X} \times \widehat{U} \times \widehat{W} \times V_\varsigma \rightarrow \widehat{X}$  is defined as

$$\widehat{f}(\hat{x}, \hat{u}, \hat{w}, \varsigma) := \Pi_x(f(\hat{x}, \hat{u}, \hat{w}, \varsigma)),$$

and  $\Pi_x : X \rightarrow \widehat{X}$  is a map that assigns to any  $x \in X$ , the representative point  $\bar{x} \in \widehat{X}$  of the corresponding partition set containing  $x$ . Map  $\Pi_x$  satisfies the inequality

$$\|\Pi_x(x) - x\| \leq \delta, \quad \forall x \in X,$$

where  $\delta := \sup\{\|x - x'\| \mid x, x' \in X_i, i = 1, 2, \dots, n_x\}$  is a state discretization parameter. An initial state of  $\widehat{\Sigma}$  is also selected according to  $\hat{x}_0 := \Pi_x(x_0)$  with  $x_0$  being an initial state of  $\Sigma$ .

For a given specification  $\varphi$  and accuracy level  $\epsilon$ , the discretization parameter  $\delta$  can be selected a priori such that

$$|\mathbb{P}(\Sigma \models \varphi) - \mathbb{P}(\widehat{\Sigma} \models \varphi)| \leq \epsilon,$$

where  $\epsilon$  depends on the horizon of formula  $\varphi$ , the Lipschitz constant of the stochastic kernel, and  $\delta$  (cf. [SAM15, Theorem 9]).

In the next sections, we propose novel data-parallel algorithms for the construction of finite MDPs and the synthesis of their controllers.

### 6.3 Parallel Construction of Finite MDPs

In this section, we propose a technique to efficiently compute the transition probability matrix  $\hat{T}_x$  of the finite MDP  $\hat{\Sigma}$ , which is essential for any controller synthesis procedure, as we discuss later in Section 6.4.

---

**Algorithm 15:** Serial algorithm for computing  $\hat{T}_x$

---

**Input:**  $\hat{X}, \hat{U}, \hat{W}$ , and a noise covariance matrix  $\Sigma \in \mathbb{R}^{n \times n}$ .

**Output:** Transition probability matrix  $\hat{T}_x$  with dimension of  $(n_x \times n_\nu \times n_w, n_x)$ .

```

1 for all  $\bar{x}_i \in \hat{X}$ , s.t.  $i \in \{1, \dots, n_x\}$ , do
2   for all  $\bar{v}_j \in \hat{U}$ , s.t.  $j \in \{1, \dots, n_\nu\}$ , do
3     for all  $\bar{w}_k \in \hat{W}$ , s.t.  $k \in \{1, \dots, n_w\}$ , do
4       Compute mean  $\mu = f(\bar{x}_i, \bar{v}_j, \bar{w}_k, 0)$ ;
5       Construct a Probability Density Function (PDF) as follows:
           PDF( $x \mid \mu, \Sigma$ ) =  $\frac{1}{((2\pi)^n \det(\Sigma))^{\frac{1}{2}}} \exp[-\frac{1}{2}(x - \mu)^T \Sigma^{-1}(x - \mu)]$ ,
           where  $\pi = 3.1415$ ;
6       for all  $\bar{x}'_l \in \hat{X}$ , s.t.  $l \in \{1, \dots, n_x\}$ , do
7         Set  $\hat{T}_x(\bar{x}'_l \mid \bar{x}_i, \bar{v}_j, \bar{w}_k) := \int_{\Xi(x')} \text{PDF}(dx \mid \mu, \Sigma)$ ;
8       end
9     end
10  end
11 end
```

---

Algorithm 15 presents the traditional serial algorithm for computing  $\hat{T}_x$ . It is a refinement of Step 5 in Algorithm 14, which represents the most computationally demanding part in Algorithm 14.

**Remark 6.3.1.** *If there are no disturbances in the given dynamics as discussed in (6.1.3), one can still employ Algorithm 15 to compute transition probability matrix but without step 3.*

In the following subsections, we address improvements of Algorithm 15. Each of the subsections targets one inefficient aspect of Algorithm 15 and discusses how to improve it. Then, in subsection 6.3.4, we combine all the improvements and introduce a parallel

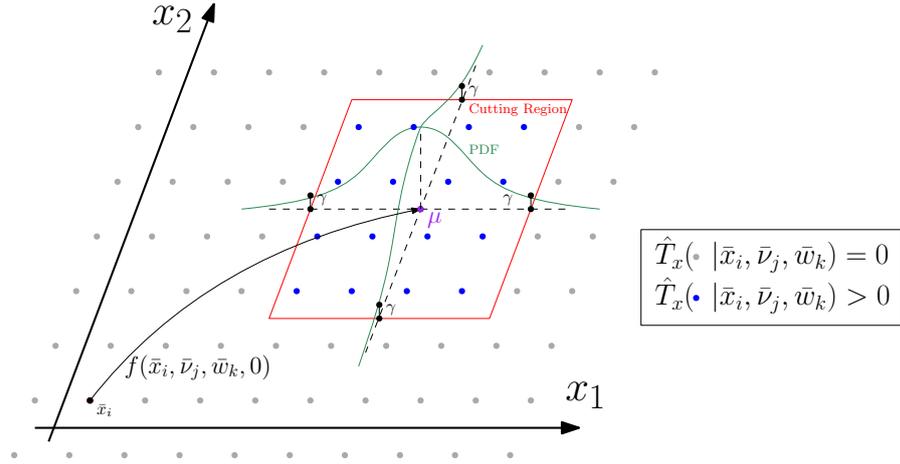


Figure 6.1: A 2-dimensional visualization of the cutting probability region.

algorithm for constructing  $\hat{T}_x$ . Note that, for now, PDFs follow Gaussian distributions and the improvements are tuned to this. Extending the ideas in the next subsections to different distributions is discussed later.

### 6.3.1 Data-Parallel Threads for Computing Transitions

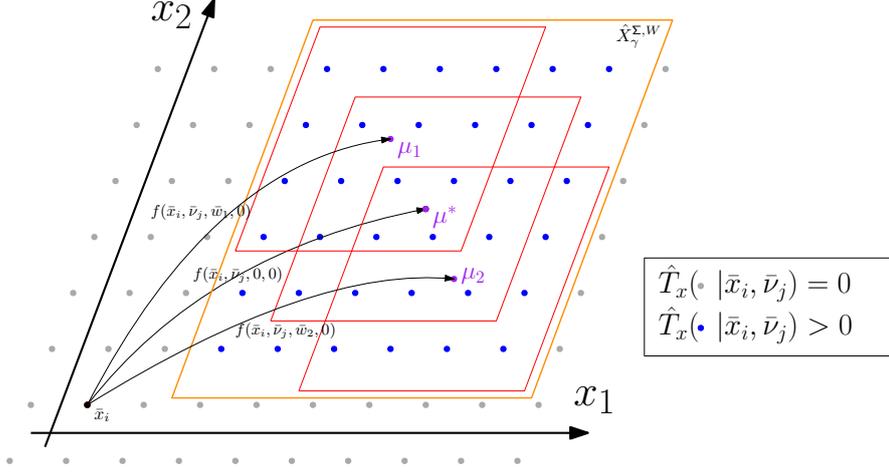
The inner steps inside the nested for-loops 1, 2, and 3 in Algorithm 15 are computationally independent. More specifically, the computations of  $\mu$ ,  $PDF(x | \mu, \Sigma)$ , and  $\hat{T}_x$  all do not share data from one inner-loop to another. This is clearly an embarrassingly data-parallel section of the algorithm. **pFaces** can be used to launch necessary number of parallel threads on the employed HWC to control the computation time of the algorithm. Each thread will eventually compute and store, independently, its corresponding values within  $\hat{T}_x$ .

### 6.3.2 Less Memory for Post States in the Transitions

$\hat{T}_x$  is a matrix with a dimension of  $(n_x \times n_\nu \times n_w, n_x)$ . The number of its columns is  $n_x$  as we need to compute and store the probability for each reachable partition element  $\Xi(x'_i)$ , corresponding to the representing post state  $x'_i$ .

For simplicity, we now focus on the computation done for a tuple  $(\bar{x}_i, \bar{v}_j, \bar{w}_k)$ . In many applications, when the PDFs are decaying fast, only those partition elements near  $\mu$  have relatively high probability values for being reached, starting from  $\bar{x}_i$  and applying an input  $\bar{v}_j$ .

We set a cutting probability threshold  $\gamma \in [0, 1]$  to control how much information for the partition elements around  $\mu$  is stored. For a given mean value  $\mu$ , a covariance matrix  $\Sigma$  and a cutting probability threshold  $\gamma$ ,  $x \in X$  is called a cutting point of the PDF if  $\gamma = PDF(x | \mu, \Sigma)$ . Since PDFs here are symmetric, we have cutting points that form a hyper-rectangle in  $X$ , which we call the cutting region and denote it by  $\hat{X}_\gamma^\Sigma$ . This is visualized in Fig. 6.1 for a 2-dimensional system. The boundary of the



**Figure 6.2:** A 2-dimensional visualization of the cutting probability region after approximating the effect of  $\hat{W}$ .

cutting probability region is shown in red. The cutting region encloses representative post states (blue dots) that have non-zero probabilities in  $\hat{T}_x$ . Other representative post states outside the cutting region are considered to have zero probabilities in  $\hat{T}_x$ .

For a tuple  $(\bar{x}_i, \bar{\nu}_j, \bar{w}_k)$ ,  $\hat{X}_\gamma^\Sigma$  is the set of representative points with probabilities of being reached greater than  $\gamma$ . Formally,

$$\hat{X}_\gamma^\Sigma := \{\bar{x} \in \hat{X} \mid \mathbb{P}(x(k+1) \in \Xi(\bar{x}) \mid x(k) = x_i, \nu(k) = \nu_j, w(k) = w_k) \geq \gamma\}.$$

Any partition element  $\Xi(x'_l)$  with  $x'_l$  outside the cutting region is considered to have a zero probability of being reached. Such approximation allows controlling the sparsity of the columns of  $\hat{T}_x$ . The closer the value of  $\gamma$  to zero, the more accurate  $\hat{T}_x$  in representing the transitions of  $\hat{\Sigma}$ . On the other hand, the closer the value of  $\gamma$  to one, fewer values need to be stored as columns in  $\hat{T}_x$ . The number of probabilities to be stored for each tuple  $(\bar{x}_i, \bar{\nu}_j, \bar{w}_k)$  is  $|\hat{X}_\gamma^\Sigma|$ . Figure 6.1 also visualizes how the proposed  $\gamma$  allows controlling the required memory for storing the transitions in  $\hat{T}_x$ .

Note that since  $\Sigma$  is fixed prior to running the algorithm, number of columns needed for a fixed  $\gamma$  can be identified before launching the computation. We can then accurately allocate a uniform fixed number of memory locations for any tuple  $(\bar{x}_i, \bar{\nu}_j, \bar{w}_k)$  in  $\hat{T}_x$ . Hence, there is no need for a dynamic sparse matrix data structure and  $\hat{T}_x$  is now a matrix with a dimension of  $(n_x \times n_\nu \times n_w, |\hat{X}_\gamma^\Sigma|)$ .

### 6.3.3 Less Memory for Handling Disturbances of Dynamics

Fix an initial state  $x_i$  and an input  $\nu_j$ . This corresponds to an analysis of Algorithm 15 starting from step 3. The disturbance value  $w_k$  now affects the location of the mean post state  $\mu$  inside the state set  $X$ . Accordingly, one post state  $\bar{x}'_l$  (step 6 in Algorithm 15), and its enclosing partition element  $\Xi(\bar{x}'_l)$ , may be reached with different probabilities for different values of  $\bar{w}_k$ . This is mainly because the origin of the PDF (i.e.,  $\mu$ ) is shifted

according to the value of  $\bar{w}_k$  resulting into different values of the integration in step 6. This issue is visually depicted in Fig. 6.2. It shows a 2-dimensional visualization of the cutting probability region  $(n_x \times n_\nu \times n_w, |\hat{X}_\gamma^\Sigma|)$  after approximating the effect of  $\hat{W} = \{\bar{w}_1, \bar{w}_2\}$ . Red regions are for cutting probability corresponding to specific disturbance values. The orange region is  $(n_x \times n_\nu \times n_w, |\hat{X}_\gamma^\Sigma|)$ . PDFs are not visualized to make the figure simpler.

All the computed probabilities are stored in  $\hat{T}_x$ . Later when  $\hat{T}_x$  is used to synthesize controllers for safety and reachability requirements using dynamic programming, we only need the minimum value of  $\hat{T}_x$  with respect to all  $\bar{w}_k \in \hat{W}$ , for a fixed  $(\bar{x}_i, \bar{\nu}_j)$ . Then, instead of storing probability values for each different  $\bar{w}_k \in \hat{W}$  inside  $\hat{T}_x$ , we can compute the minimum values and store them. We also store the maximum values in case the user is interested to see how the disturbance affects the probability of reaching a specific post state. Now, when storing minimum and maximum values only,  $\hat{T}_x$  becomes a matrix with a dimension of  $(n_x \times n_\nu \times 2, |\hat{X}_\gamma^\Sigma|)$ .

Note that due to this shifting phenomenon, the probability threshold  $\gamma$  is also shifted. We then need to include more post states around  $\mu$  such that, for each reachable post state, the minimum probability value, with respect to different values of  $\bar{w}_i$ , is still greater than  $\gamma$ . We denote such set of considered post states by  $\hat{X}_\gamma^{\Sigma, W}$ . Set  $\hat{X}_\gamma^{\Sigma, W}$  is simply an inflation of  $\hat{X}_\gamma^\Sigma$  with  $\|W\| := \sup\{\|w\| \mid w \in W\}$ . Finally,  $\hat{T}_x$  becomes a matrix with a dimension of  $(n_x \times n_\nu \times 2, |\hat{X}_\gamma^{\Sigma, W}|)$ .

**Remark 6.3.2.** *Constructing  $\hat{X}_\gamma^\Sigma$  and  $\hat{X}_\gamma^{\Sigma, W}$  is practically simple. We start by solving the equation  $\text{PDF}(x^* \mid 0, \Sigma) = \gamma$  for  $x^* \in \mathbb{R}_+^n$  to compute the cutting points. Since the PDF is symmetric, we have that  $\hat{X}_\gamma^\Sigma$  is enclosed by the hyper-rectangle  $[\mu - x^*, \mu + x^*]$  and  $\hat{X}_\gamma^{\Sigma, W}$  is enclosed by the hyper-rectangle  $[\mu^* - x^* - \|W\|, \mu^* + x^* + \|W\|]$ , where  $\mu^* = \mu \mid_{w=0}$ . Finally, we have that*

$$\hat{X}_\gamma^\Sigma := \{\bar{x} \in \hat{X} \mid \bar{x} \in [\mu - x^*, \mu + x^*]\},$$

and

$$\hat{X}_\gamma^{\Sigma, W} := \{\bar{x} \in \hat{X} \mid \bar{x} \in [\mu^* - x^* - \|W\|, \mu^* + x^* + \|W\|]\}.$$

Note that the location of  $\hat{X}_\gamma^\Sigma$  (not its cardinality) varies based on  $\bar{x}$ ,  $\bar{\nu}$ , and the location of  $\hat{w}$ , while  $\hat{X}_\gamma^{\Sigma, W}$  (not its cardinality) varies only based on  $\bar{x}$  and  $\bar{\nu}$ . A simple visualization of the relation between both sets is depicted in Fig. 6.2.

### 6.3.4 Data-Parallel Algorithm for Constructing MDPs

Algorithm 14 is a utility high-level algorithm to put all necessary sets of  $\hat{\Sigma}$  together and no need to parallelize it. We present next a parallel algorithm to efficiently construct and store  $\hat{T}_x$  as a successor to Algorithm 15. We employ the discussed enhancements from subsections 6.3.1, 6.3.2, and 6.3.3 within the proposed algorithm.

Algorithm 16 is the proposed data-parallel version of Algorithm 15. We leave the for-loop in Algorithm 16 step 8 unparallelized to avoid any race conditions reading and

---

**Algorithm 16:** Proposed parallel algorithm for computing  $\hat{T}_x$

---

**Input:**  $\hat{X}, \hat{U}, \hat{W}, \gamma$ , and a noise covariance matrix  $\Sigma \in \mathbb{R}^{n \times n}$ .

**Output:** Transition probability matrix  $\hat{T}_x$  with dimension of  $(n_x \times n_u \times 2, |\hat{X}_{\gamma}^{\Sigma, W}|)$ .

```

1 for all  $(\bar{x}, \bar{v}) \in \hat{X} \times \hat{U}$  in parallel do
2   Set  $\mu^* = f(\bar{x}, \hat{\mu}, 0, 0)$ ;
3   Construct  $\hat{X}_{\gamma}^{\Sigma, W}$  as described in Remark 6.3.2;
4   for all  $x^* \in \hat{X}_{\gamma}^{\Sigma, W}$  do
5     Set  $\hat{T}_x(x^* | \bar{x}, \bar{v}, min) = 1.0$ ;
6     Set  $\hat{T}_x(x^* | \bar{x}, \bar{v}, max) = 0.0$ ;
7   end
8   for all  $\bar{w} \in \hat{W}$  do
9     Set  $\mu = f(\bar{x}, \hat{\mu}, \bar{w}, 0)$ ;
10    Construct  $\hat{X}_{\gamma}^{\Sigma}$  as described in Remark 6.3.2;
11    for all  $x^* \in \hat{X}_{\gamma}^{\Sigma}$  do
12      Set  $p := \int_{\Xi(x^*)} \text{PDF}(dx | \mu, \Sigma)$ ;
13      if  $p < \hat{T}_x(x^* | \bar{x}, \bar{v}, min)$  then
14        Set  $\hat{T}_x(x^* | \bar{x}, \bar{v}, min) = p$ ;
15      end
16      if  $p > \hat{T}_x(x^* | \bar{x}, \bar{v}, max)$  then
17        Set  $\hat{T}_x(x^* | \bar{x}, \bar{v}, max) = p$ ;
18      end
19    end
20  end
21 end

```

---

assigning values to  $\hat{T}_x(x^* | \bar{x}, \bar{\nu}, \cdot)$  in steps 14 and 17. A race condition is possible since, when  $\hat{W} \neq \emptyset$ , two overlapping  $\hat{X}_\gamma^\Sigma$  in two different threads may need to access the same memory location  $T_x(x^* | \bar{x}, \bar{\nu}, \cdot)$  at the same time.

**Theorem 6.3.3.** *The computational complexity of Algorithm 15 is  $O(|\hat{X}|^2|\hat{U}||\hat{W}|)$ , while the computational complexity of Algorithm 16 is  $O(|\hat{X}_\gamma^\Sigma| \frac{|\hat{X}||\hat{U}|}{P} \max\{|\hat{W}|, |\hat{X}_\gamma^{\Sigma, W}|\})$ , given that its main parallel for-loop is run on  $P$  PEs and  $P$  is variable.*

*Proof.* We hold the proof of the complexity of Algorithm 15 to be self-evident. The proof of the complexity of Algorithm 16 follows directly from the improvements in Subsections 6.3.1, 6.3.2, and 6.3.3, and the computation of parallel complexity for PRAM models as introduced in [Jaj92, Chapter 1].  $\square$

## 6.4 Data-Parallel Synthesis of Symbolic Controllers

---

**Algorithm 17:** Serial algorithm for controller synthesis satisfying safety specifications

---

**Input:** Transition probability matrix  $\hat{T}_x$  and bounded time horizon  $T_d$ .

**Output:** Optimal safety probability  $V_s$  at time step  $T_d = 1$ , and optimal policy  $\nu^*$  corresponding to optimal safety probability.

```

1 Set value function  $V_s := ones(n_x, T_d + 1)$ ;
2 for all  $k = T_d : -1 : 1$  backward in time do
3   Set  $V_{int} = \hat{T}_x V_s(:, k + 1)$ ;
4   Reshape  $V_{int}$  to a matrix  $\bar{V}_{int}$  of dimension  $(n_x \times n_\nu, n_w)$ ;
5   Minimize  $\bar{V}_{int}$  with respect to disturbance set  $\hat{W}$  as  $V_{min}$  of dimension
    $(n_x \times n_\nu, 1)$ ;
6   Reshape  $V_{min}$  to a matrix  $\bar{V}_{min}$  of dimension  $(n_x, n_\nu)$ ;
7   Maximize  $\bar{V}_{min}$  with respect to input set  $\hat{U}$  as  $V_{max}$  of dimension  $(n_x, 1)$ ;
8   Update  $V_s(:, k) := V_{max}$ ;
9 end
    
```

---

In this section, we employ dynamic programming to synthesize controllers for constructed finite MDPs satisfying safety and reachability properties [Esm14, SA13]. We first present traditional serial algorithm for controller synthesis satisfying safety specifications in Algorithm 17. Note that if there are no disturbances in the given dynamics, Steps 5, 6 of Algorithm 17, and 7, 8 of Algorithm 18 should not be taken into account.

Algorithms 17 and 18 are both doing the repetitive matrix multiplication in each loop that corresponds to different time instance of the bounded time  $T_d$ . Although we cannot parallelize the for-loop in Algorithm 17, step 2 and Algorithm 18, step 4 due to data dependency, we can still parallelize the contents of each loop by simply considering standard parallel algorithms for the matrix multiplication. Additional essential update

---

**Algorithm 18:** Serial algorithm for controller synthesis satisfying reachability (or reach-avoid in case an avoid set  $\mathcal{A}$  is provided) specifications

---

**Input:**  $\hat{X}, \hat{U}, \hat{W}$ , target-set  $\mathcal{T}$ , avoid set  $\mathcal{A}$ , and bounded time horizon  $T_d$ .  
**Output:** Optimal reachability probability  $V_r$  at time step  $T_d = 1$ , and optimal policy  $\nu^*$  corresponding to optimal reachability probability.

- 1 Compute transition probability matrix  $\hat{T}_{0x}$  from  $\hat{X} \setminus (\mathcal{T} \cup \mathcal{A})$  to  $\mathcal{T}$ ;
- 2 Compute transition probability matrix  $\hat{T}_{1x}$  in  $\hat{X} \setminus (\mathcal{T} \cup \mathcal{A})$ ;
- 3 Set value function  $V_r := \text{zeros}(n_x, T_d + 1)$ ;
- 4 **for all**  $k = T_d : -1 : 1$  *backward in time* **do**
- 5     Set  $V_{int} = \hat{T}_{0x} + \hat{T}_{1x}V_r(:, k + 1)$ ;
- 6     Reshape  $V_{int}$  to a matrix  $\bar{V}_{int}$  of dimension  $(n_x \times n_\nu, n_w)$ ;
- 7     Minimize  $\bar{V}_{int}$  with respect to disturbance set  $\hat{W}$  as  $V_{min}$ ;
- 8     Reshape  $V_{min}$  to a matrix  $\bar{V}_{min}$  of dimension  $(n_x, n_\nu)$ ;
- 9     Maximize  $\bar{V}_{min}$  with respect to input set  $\hat{U}$  as  $V_{max}$  of dimension  $(n_x, 1)$ ;
- 10    Update  $V_r(:, k) := V_{max}$ ;
- 11 **end**

---



---

**Algorithm 19:** Proposed parallel algorithm for controller synthesis satisfying safety specifications

---

**Input:** Transition probability matrix  $\hat{T}_x$  and bounded time horizon  $T_d$ .  
**Output:** Optimal safety probability  $V_s$  at time step  $T_d = 1$ , and optimal policy  $\nu$  corresponding to optimal safety probability.

- 1 Set, in parallel,  $V_s := \text{ones}(n_x, T_d + 1)$ ;
- 2 **for all**  $k = T_d : -1 : 1$  *backward in time* **do**
- 3     **for all**  $(\bar{x}, \bar{\nu}) \in \hat{X} \times \hat{U}$  **in parallel** **do**
- 4         Set  $\mu^* = f(\bar{x}, \hat{\mu}, 0)$ ;
- 5         Construct  $\hat{X}_\gamma^{\Sigma, W}$  as described in Remark 6.3.2;
- 6         Set  $\tilde{V}_{int}(\bar{x}, \bar{\nu}) := \sum_{x^* \in \hat{X}_\gamma^{\Sigma, W}} V_s(x^*, k + 1) * T_x(x^* | \bar{x}, \bar{\nu}, min)$ ;
- 7     **end**
- 8     **for all**  $\bar{x} \in \hat{X}$  **in parallel** **do**
- 9         Set  $V_s(\bar{x}, k) := \max_{\bar{\nu} \in \hat{U}} \{\tilde{V}_{int}(\bar{x}, \bar{\nu})\}$ ;
- 10        Set  $\nu(\bar{x}, k) := \operatorname{argmax}_{\bar{\nu} \in \hat{U}} \{\tilde{V}_{int}(\bar{x}, \bar{\nu})\}$ ;
- 11     **end**
- 12 **end**

---

is to use the pre-stored minimum values in  $T_x(x^* | \bar{x}, \bar{\nu}, \min)$  rather than minimizing the values with respect to  $\hat{W}$  (Algorithm 17, step 5 and Algorithm 18, step 7).

Algorithm 19 is a parallelization of Algorithm 17. Step 3 in Algorithm 19 is the parallel implementation of the matrix multiplication in Algorithm 17, step 3. Step 8 in Algorithm 19 selects and stores the inputs  $\bar{\nu}$  that maximizes the probabilities of enforcing the safety specification.

A significant reduction in the computation of the intermediate matrix  $V_{int}$  is also introduced in Algorithm 19. In Algorithm 17, step 3, the computation of  $V_{int}$  requires a matrix multiplication between  $P$  (dimension of  $(n_x \times n_\nu \times n_w, n_x)$ ) and  $V(\cdot, \cdot)$  (dimension of  $(n_x, 1)$ ). On the other hand, in the parallel version in Algorithm 19, the corresponding computation is done in parallel for  $\tilde{V}_{int}$  for which each element, i.e.,  $\tilde{V}_{int}(\bar{x}, \bar{\nu})$ , requires only  $|\hat{X}_\gamma^{\Sigma, W}|$  scalar multiplications. Here, we clearly utilize the techniques discussed in Subsections 6.3.2 and 6.3.3 to consider only those post states in the cutting region  $\hat{X}_\gamma^{\Sigma, W}$ . Remember that other post states outside  $\hat{X}_\gamma^{\Sigma, W}$  are considered to have probability zero which means we can avoid their scalar multiplications.

Algorithm 20 extends Algorithm 19 to support safety, reachability and reach-avoid specifications.

**Theorem 6.4.1.** *The computational complexity of Algorithm 17 and Algorithm 18 is  $O(T_d |\hat{X}|^2 |\hat{U}| |\hat{W}|)$ . The combinational complexity of Algorithm 19 is  $O(T_d \frac{|\hat{X}| |\hat{U}|}{P} |\hat{X}_\gamma^{\Sigma, W}|)$  and the computational complexity of Algorithm 20 is  $O(T_d \frac{|\hat{X}| |\hat{U}|}{P} |\hat{W}| |\hat{X}_\gamma^{\Sigma, W}|)$ , where  $P$  is the variable number of PEs used to run the parallel for-loop in the algorithm.*

*Proof.* The proof of the computational complexity of Algorithms 17 and 18 can be directly shown based on the fact that  $|\hat{T}| = |\hat{X}|^2 |\hat{U}| |\hat{W}|$ . The proof of the complexity of Algorithms 19 and 20 follows directly from the improvements in this Subsections and the computation of parallel complexity for PRAM models as introduced in [Jaj92, Chapter 1].  $\square$

### 6.4.1 On-the-Fly Construction of Transitions

We consider another technique that further reduces the required memory. We construct  $\hat{T}_x$  on-the-fly and refer to the modified version of Algorithm 20 as the OFA version. In the OFA version, we skip computing and storing the MDP  $\hat{T}_x$  and the matrix  $\hat{T}_{0x}$  (i.e., Steps 1 and 6). We instead compute the required entries of  $\hat{T}_x$  and  $\hat{T}_{0x}$  on-the-fly as they are needed (i.e., Steps 14 and 16). This significantly reduces the required memory for  $\hat{T}_x$  and  $\hat{T}_{0x}$  but at the cost of repeated computation of their entries in each time-step from 1 to  $T_d$ . This gives the user an additional control over the trade-off between the computation time and memory.

### 6.4.2 Supporting Multiplicative Noises and Practical Distributions

The introduced algorithms can be extended to support multiplicative noises and practical distributions such as uniform, exponential, and beta distributions. The technique

---

**Algorithm 20:** Proposed *parallel* algorithm for controller synthesis satisfying safety, reachability and reach-avoid specifications

---

**Input:**  $\hat{X}, \hat{U}, \hat{W}$ , bounded time horizon  $T_d$ ,  $specs \in \{Safety, Reachability, Reach-Avoid\}$ , target-set  $\mathcal{T}$  (in case  $specs = Reachability, Reach-Avoid$ ), and avoid set  $\mathcal{A}$  (in case  $specs = Reach-Avoid$ ).

**Output:** Optimal satisfaction probability  $V_v$  at time step  $T_d = 1$ , and optimal policy  $\nu^*$  corresponding to the optimal satisfaction probability.

- 1 Compute  $\hat{T}_x$  in parallel as presented in Algorithm 16;
- 2 **if**  $specs == Safety$  **then**
- 3 | Set value function  $V_v := ones(n_x, T_d + 1)$ ;
- 4 **end**
- 5 **else**
- 6 | Compute a transition probability matrix  $\hat{T}_{0x}$  from  $\hat{X} \setminus (\mathcal{T} \cup \mathcal{A})$  to  $\mathcal{T}$ ;
- 7 | Set  $\hat{T}_x$  to zero for any post-state in  $(\mathcal{T} \cup \mathcal{A})$ ;
- 8 | Set value function  $V_v := zeros(n_x, T_d + 1)$ ;
- 9 **end**
- 10 **for all**  $k = T_d : -1 : 1$  (*backward in time*) **do**
- 11 | **for all**  $(\bar{x}, \bar{v}) \in \hat{X} \times \hat{U}$  **in parallel do**
- 12 | | **for all**  $\bar{w} \in \hat{W}$  **do**
- 13 | | | Construct  $\hat{X}_\gamma^\Sigma$  as discussed in Subsection 6.3.2;
- 14 | | | Set  $V_{in}(\bar{x}, \bar{v}, \bar{w}) := \sum_{x^* \in \hat{X}_\gamma^\Sigma} V_v(x^*, k + 1) T_x(x^* | \bar{x}, \bar{v}, \bar{w})$ ;
- 15 | | | **if**  $specs == Reach-Avoid$  and  $\bar{x} \notin (\mathcal{T} \cup \mathcal{A})$  **then**
- 16 | | | | Set  $V_{in}(\bar{x}, \bar{v}, \bar{w}) := V_{in}(\bar{x}, \bar{v}, \bar{w}) + T_{0x}(\bar{x}, \bar{v}, \bar{w})$ ;
- 17 | | | **end**
- 18 | | **end**
- 19 | **end**
- 20 | **for all**  $\bar{x} \in \hat{X}$  **in parallel do**
- 21 | | Set  $V_v(\bar{x}, k) := \max_{\bar{v} \in \hat{U}} \{ \min_{\bar{w} \in \hat{W}} \{ V_{in}(\bar{x}, \bar{v}, \bar{w}) \} \}$ ;
- 22 | | Set  $\nu^*(\bar{x}, k) := \operatorname{argmax}_{\bar{v} \in \hat{U}} \{ \min_{\bar{w} \in \hat{W}} \{ V_{in}(\bar{x}, \bar{v}, \bar{w}) \} \}$ ;
- 23 | **end**
- 24 **end**

---

introduced in Subsection 6.3.2 for reducing the memory usage can be tuned for other distributions based on the support of their PDFs. In the implementation of **AMyTISS**, users can specify their desired PDFs and hyper-rectangles enclosing their supports so that **AMyTISS** can include them in the parallel computation of  $\hat{T}_x$ .

Kernel **AMyTISS** also supports multiplicative noises as introduced in (6.1.2). Currently, the memory reduction technique from Subsection 6.3.2 is disabled for systems with multiplicative noises. This means users should expect larger memory requirements for systems with multiplicative noises. However, users can still benefit from the proposed OFA version to compensate for the increase in memory requirement. We plan to include this feature for multiplicative noises in a future update of **AMyTISS**. Only for a better demonstration, the previous sections were presented assuming additive noise and Gaussian normal PDF.

## 6.5 Illustrative Examples

We introduce a simple 2-dimensional example to illustrate the proposed algorithms. Consider a robot described by the following ODE:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} \nu_1(k)\cos(\nu_2(k)) \\ \nu_2(k)\sin(\nu_2(k)) \end{bmatrix},$$

where  $(x_1, x_2) \in X := [-10, 10]^2$  is a state vector and  $(\nu_1, \nu_2) \in U := [-1, 1]^2$  is an input vector. As we consider discrete-time systems, we approximate the system using forward Euler method and include disturbances and noise so that the system is a dt-SCS. The following is the dt-SCS of the 2d robot:

$$\begin{bmatrix} x_1(k+1) \\ x_2(k+1) \end{bmatrix} = \begin{bmatrix} x_1(k) + \tau\nu_1(k)\cos(\nu_2(k)) + w(k) + \varsigma_1(k) \\ x_2(k) + \tau\nu_2(k)\sin(\nu_2(k)) + w(k) + \varsigma_2(k) \end{bmatrix}, \quad (6.5.1)$$

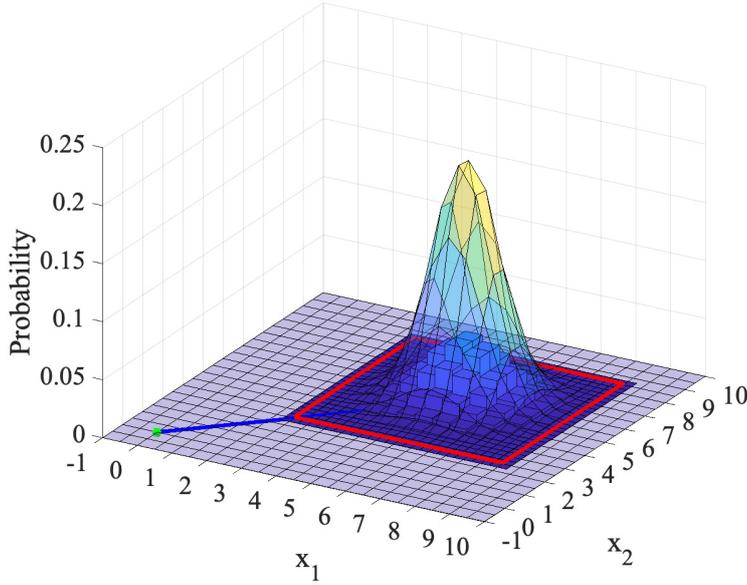
where  $w \in W := [-1, 1]$  is a disturbance set that includes the upper bound of the local truncation error from the forward Euler approximation,  $(\varsigma_1, \varsigma_2)$  is a noise following a Gaussian distribution with the covariance matrix  $\Sigma := \text{diag}(0.75, 0.75)^3$ , and  $\tau := 10$  is the sampling period.

To construct a PDF approximating the system, we consider a state quantization parameter of  $(0.5, 0.5)$ , an input quantization parameter of  $(0.1, 0.1)$ , a disturbance quantization parameter of  $0.2$ , and a cutting probability threshold  $\gamma$  of  $0.001$ . Using such quantization parameters, the number of state-input pairs  $|\hat{X} \times \hat{U}|$  in  $\hat{\Sigma}$  is 203401. We use  $|\hat{X} \times \hat{U}|$  as an indicator of the size of the system.

### 6.5.1 Synthesis of a Safety Controller

We synthesize a controller for the robot system in to keep the state of the robot inside  $X$  within 8 time steps. The synthesized controller should enforce the safety specification in

<sup>3</sup> $\text{diag}(d)$  builds an  $n \times n$  diagonal matrix from a supplied  $n$ -dimensional vector  $d$ .

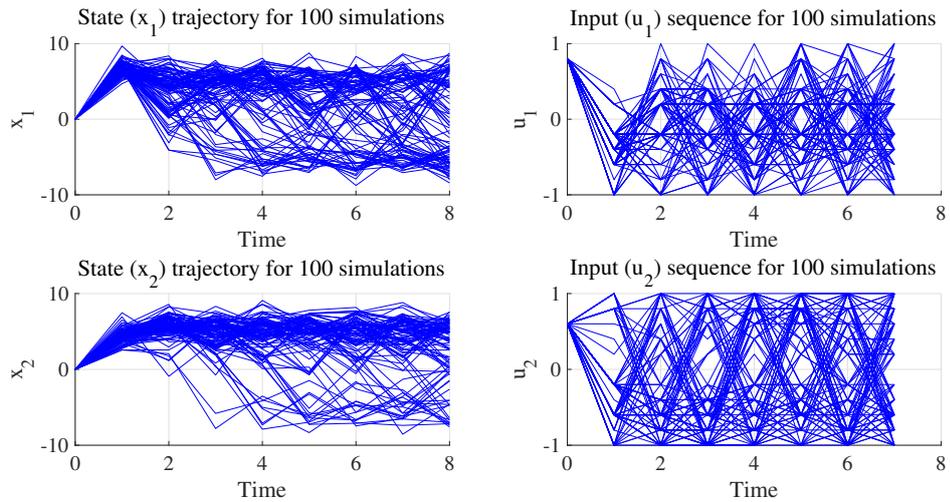


**Figure 6.3:** A visualization of transitions for one source state  $x := (0, 0)$  and input  $\nu := (0.7, 0.8)$  of the MDP approximating the robot example.

the presence of the disturbance and noise. We launch **AMYTESS** to construct the MDP of the robot and synthesize a safety controller for it. We visualize some transitions of the constructed PDF and show them in Figure 6.3. It shows a visualization of transitions for one source state  $x := (0, 0)$  and input  $\nu = (0.7, 0.8)$  of the MDP approximating the robot example. The green point is the source state, the transparent bell-like shape is the PDF and the red rectangle is the cutting region. Probabilities of reaching the partition elements inside the cutting regions are shown as bars below the PDF.

We simulate the closed-loop behavior of the robot with the synthesized controller. The simulation is done with random choices of  $\bar{w} \in \hat{W}$  and random values for the noise according to the given covariance matrix. At each time step, the simulation queries the strategy from the output file and applies it to the system. We repeat the simulation 100 times.

Figure 6.4 shows the closed-loop simulation results. It shows the 100 different simulations of the closed-loop behavior of the robot under a safety controller synthesized for maintaining the robot inside  $X$ . At left, we show the state trajectory of the system at each time step. At right, we show the applied input at each time step. For the sake of readability, the input plots are shown as piece-wise linear signal. Note that the input is always fixed at the time step  $k = 0$ . This is because we store only one input, which is the one maximizing the probability of satisfaction. After the time step  $k = 0$  and due to the noise/disturbance, the system lands in different states which requires applying different inputs to satisfy the specification.



**Figure 6.4:** Repeated simulations of the closed-loop behavior of the robot under a safety controller.

### 6.5.2 Synthesis of a Reach-Avoid Controller

We synthesize a controller for the robot system to reach the set  $[5, 7]^2$  while avoiding the set  $[-2, 2]^2$  within 16 time steps. Again, we launch **AMYTESS** to construct an MDP of the robot system and synthesize a reach-avoid controller for it. The **MATLAB** interface is used to simulate the closed-loop. We run the closed loop and generate 9 different simulations from 9 different initial states.

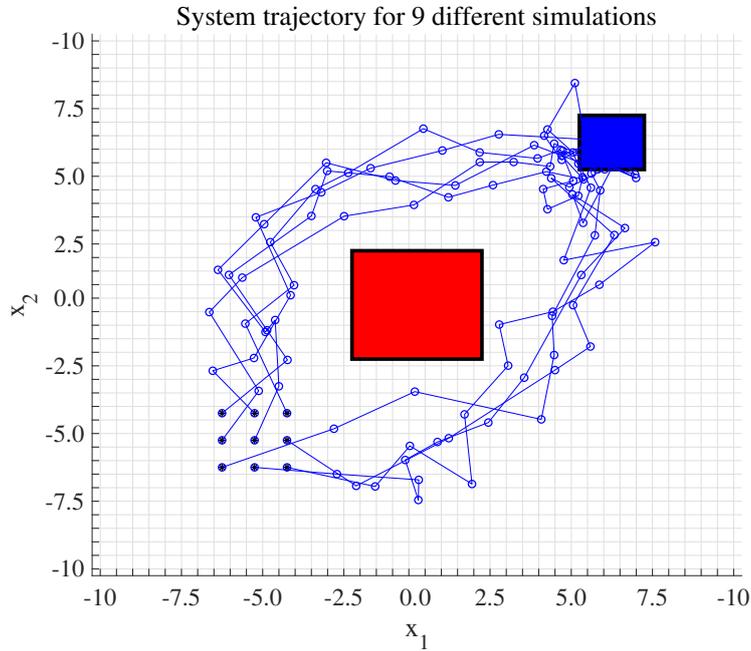
Figure 6.5 shows the closed-loop simulation results. It shows the 9 different simulations of the closed-loop behavior of the robot example under a synthesized controller enforcing the robot to reach a target-set while avoiding an avoid-set. The 9 dots at the left bottom correspond to 9 initial states for 9 different simulation runs. The red rectangle is the avoid-set of states. The blue rectangle is the target-set of states.

## 6.6 Benchmarking and Case Studies

### 6.6.1 Controlling Computational Complexities of Stochastic Applications

Using the introduced algorithms as implemented in **AMYTESS**, one can utilize the computing power in modern HPC systems and Cloud-computing platforms to control the computational complexities of stochastic symbolic control. We fix the robot example as a system in hand and show how the computation time of solving the control problem is controlled using different computing platforms.

Table 6.1 lists the HWCs used to benchmark the introduced algorithms. The devices range from local devices in laptops and desktop computers to advanced compute-devices in AWS.



**Figure 6.5:** Different simulations of the closed-loop behavior of the robot example under a reach-avoid controller.

**Table 6.1:** HWCs used for benchmarking the parallel algorithms of stochastic symbolic control.

<b>Id</b>	<b>Description</b>	<b>PEs</b>
<b>CPU<sub>1</sub></b>	PC: Intel Xeon E5-1620	8
<b>CPU<sub>2</sub></b>	AWS instance c5.18xlarge: Intel Xeon Platinum 8000	72
<b>GPU<sub>1</sub></b>	Macbook Pro 15 laptop: AMD Radeon Pro Vega 20	1280
<b>GPU<sub>2</sub></b>	AWS instance p3.2xlarge: NVIDIA Tesla V100	5120

## 6 Efficient Algorithms for Stochastic Symbolic Control

**Table 6.2:** Comparison between AMYTISS, FAUST and StocHy based on their native features for several (physical) case studies. CSB refers to the continuous-space benchmark provided in [CDA19]. † refers to cases when we run AMYTISS with the OFA algorithm. N/M refers to the situation when there is not enough memory to run the case study. N/S refers to the lack of native support for nonlinear systems. (Kx) refers to a 1000-times speedup. The presented speedup is the maximum speedup value across all reported devices. The reported times are in seconds, unless other units are denoted.

Problem/Spec.	$ \hat{X} \times \hat{U} $	$T_d$	AMYTISS (time)				FAUST Time	StocHy Time	Speedup vs.	
			CPU <sub>1</sub>	CPU <sub>2</sub>	GPU <sub>1</sub>	GPU <sub>2</sub>			FAUST	StocHy
2-d StocHy CSB/Safety	4	6	≤ 1.0	≤ 1.0	≤ 1.0	0.0001	0.002	0.015	<b>20 x</b>	<b>150 x</b>
3-d StocHy CSB/Safety	8	6	≤ 1.0	≤ 1.0	≤ 1.0	0.0001	0.002	0.08	<b>20 x</b>	<b>800 x</b>
4-d StocHy CSB/Safety	16	6	≤ 1.0	≤ 1.0	≤ 1.0	0.0002	0.01	0.17	<b>50 x</b>	<b>850 Kx</b>
5-d StocHy CSB/Safety	32	6	≤ 1.0	≤ 1.0	≤ 1.0	0.0003	0.01	0.54	<b>33 x</b>	<b>1.8 Kx</b>
6-d StocHy CSB/Safety	64	6	≤ 1.0	≤ 1.0	≤ 1.0	0.0006	1.2	2.17	<b>2.0 Kx</b>	<b>3.6 Kx</b>
7-d StocHy CSB/Safety	128	6	≤ 1.0	≤ 1.0	≤ 1.0	0.0012	6	9.57	<b>5 Kx</b>	<b>7.9 Kx</b>
8-d StocHy CSB/Safety	256	6	≤ 1.0	≤ 1.0	≤ 1.0	0.0026	37	40.5	<b>14.2 Kx</b>	<b>15.6 Kx</b>
9-d StocHy CSB/Safety	512	6	≤ 1.0	≤ 1.0	≤ 1.0	0.0057	501	171.6	<b>87.8 Kx</b>	<b>30.1 Kx</b>
10-d StocHy CSB/Safety	1024	6	≤ 1.0	≤ 1.0	≤ 1.0	0.0122	N/M	385.5	N/A	<b>32 Kx</b>
11-d StocHy CSB/Safety	2048	6	1.0912	≤ 1.0	≤ 1.0	0.0284	N/M	1708.2	N/A	<b>60 Kx</b>
12-d StocHy CSB/Safety	4096	6	4.3029	≤ 1.0	≤ 1.0	0.0624	N/M	11216	N/A	<b>179 Kx</b>
13-d StocHy CSB/Safety	8192	6	18.681	1.8515	≤ 1.0	0.1277	N/M	≥ 24h	N/A	≥ <b>676 Kx</b>
14-d StocHy CSB /Safety	16384	6	81.647	7.9987	6.1632	0.2739	N/M	≥ 24h	N/A	≥ <b>320 Kx</b>
2-d Robot†/Safety	203401	8	8.5299	0.7572	≤ 1.0	0.0154	N/A	N/A	N/A	N/A
2-d Robot/R.Avoid	741321	16	48.593	4.5127	3.4353	0.3083	N/S	N/S	N/A	N/A
2-d Robot†/R.Avoid	741321	16	132.10	11.745	3.6264	0.1301	N/A	N/A	N/A	N/A
3-d Room Temp./Safety	7776	8	0.1072	0.0120	≤ 1.0	0.0018	1247	N/M	<b>692 Kx</b>	N/A
3-d Room Temp.†/Safety	7776	8	0.5701	0.0627	≤ 1.0	0.0028	N/A	N/A	N/A	N/A
5-d Room Temp./Safety	279936	8	200.00	19.376	N/M	1.8663	3248	N/M	<b>1740 x</b>	N/A
5-d Room Temp.†/Safety	279936	8	716.84	63.758	22.334	0.5639	N/A	N/A	N/A	N/A
3-d Road Traffic/Safety	2125764	16	29.200	3.0508	10.234	1.2895	N/M	N/M	N/A	N/A
3-d Road Traffic†/Safety	2125764	16	160.45	13.632	11.657	0.3062	N/A	N/A	N/A	N/A
5-d Road Traffic/Safety	68841472	7	N/M	38.635	N/M	4.3935	N/M	N/M	N/A	N/A
5-d Road Traffic†/Safety	68841472	7	1148.5	95.767	36.487	0.7397	N/A	N/A	N/A	N/A
3-d Vehicle/R.Avoid	1528065	32	2.5h	871.89	271.41	10.235	N/S	N/S	N/A	N/A
3-d Vehicle†/R.Avoid	1528065	32	2.8h	879.78	613.55	107.68	N/A	N/A	N/A	N/A
7-d BMW 320i/R.Avoid	3937500	32	N/M	21.5h	N/M	825.62	N/S	N/S	N/A	N/A
7-d BMW 320i†/R.Avoid	3937500	32	≥ 24h	≥ 24h	≥ 24h	1251.7	N/A	N/A	N/A	N/A

Table 6.2 shows the benchmarking results running **AMyTISS** with these HWCs for several case studies and comparing against **FAUST**, and **StochHy**. We employ a machine with Windows operating system (Intel i7@3.6GHz CPU and 16 GB of RAM) for **FAUST**, and **StochHy**. It should be mentioned that **FAUST** predefines a minimum number of representative points based on the desired abstraction error, and accordingly the computation time and memory usage reported in Table 6.2 are based on the minimum number of representative points. In addition, to have a fair comparison, we run all the case studies with additive noises since neither **FAUST** nor **StochHy** supports multiplicative noises.

For each HWC, we show the time in seconds to solve the problem. Clearly, employing HWCs with more PEs reduces the time to solve the problem. This is a strong indication for the scalability of the proposed algorithms. This also becomes very useful in real-time applications, where users can control the computation time of their problems by adding more resources. Since, **AMyTISS** is the only tool that can utilize the reported HWCs, we do not compare it with other similar tools.

To show the applicability of our results to large-scale stochastic systems, we apply our proposed techniques to several physical case studies. First, we synthesize a controller for 3- and 5-dimensional room temperature networks to keep temperature of rooms in a comfort zone. Then we synthesize a controller for road traffic networks with 3 and 5 dimensions to keep the density of the traffic below some level. We then consider 3- and 7-dimensional stochastic versions of the nonlinear model of the autonomous vehicle used in the previous two Chapters, and synthesize reach-avoid controllers to automatically park the vehicles. For each case study, we compare the results against **FAUST** and **StochHy** and report the technical details in Table 6.2.

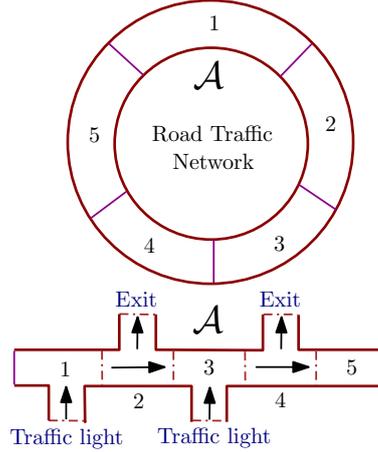
### 6.6.2 Room Temperature Network

We first apply our results to the temperature regulation of 5 rooms each equipped with a heater and connected on a circle. The model of this case study is borrowed from [LSZ18]. The evolution of temperatures  $T_{x_i}$  can be described by individual rooms as

$$\Sigma_{\mathbf{a}_i} : \begin{cases} T_{x_i}(k+1) = a_{ii}T_{x_i}(k) + \gamma T_h \nu_i(k) + \eta w_i(k) + \beta T_{ei} + 0.01 \varsigma_i(k), i \in \{1, 3\}, \\ T_{x_i}(k+1) = b_{ii}T_{x_i}(k) + \eta w_i(k) + \beta T_{ei} + 0.01 \varsigma_i(k), i \in \{2, 4, 5\}, \\ y_i(k) = T_{x_i}(k), \end{cases}$$

where  $a_{ii} = (1 - 2\eta - \beta - \gamma \nu_i(k))$ ,  $b_{ii} = (1 - 2\eta - \beta)$ , and  $w_i(k) = T_{x_{i-1}}(k) + T_{x_{i+1}}(k)$  (with  $T_{x_0} = T_{x_n}$  and  $T_{x_{n+1}} = T_{x_1}$ ). Parameters  $\eta = 0.3$ ,  $\beta = 0.022$ , and  $\gamma = 0.05$  are conduction factors, respectively, between rooms  $i \pm 1$  and the room  $i$ , between the external environment and the room  $i$ , and between the heater and the room  $i$ . Moreover,  $T_{ei} = -1^\circ C$ ,  $T_h = 50^\circ C$  are outside and heater temperatures, and  $T_i(k)$  and  $\nu_i(k)$  are taking values in sets  $[19, 21]$  and  $[0, 1]$ , respectively,  $\forall i \in \{1, \dots, n\}$ .

We synthesize a controller for  $\Sigma_{\mathbf{a}}$  via the abstraction  $\widehat{\Sigma}_{\mathbf{a}}$  such that the controller maintains the temperature of any room in the safe set  $[19, 21]$  for at least 8 time steps. We also apply our algorithms to a smaller version of this case study (3-dimensional system) with the results also reported in Table 6.2.



**Figure 6.6:** Model of a road traffic network composed of 5 cells of 500 meters with 2 entries and 2 ways out.

### 6.6.3 Road Traffic Network

Consider a road traffic network divided in 5 cells of 500 meters with 2 entries and 2 ways out, as schematically depicted in Figure 6.6. The model of this case study is borrowed from [LCGG13] by including the stochasticity in the model as the additive noise.

The two entries are controlled by traffic lights, denoted by  $\nu_1$  and  $\nu_3$ , that enable (green light) or not (red light) the vehicles to pass. In this model, the length of a cell is in kilometers [km] and the flow speed of vehicles is 100 kilometers per hour [km/h]. Moreover, during the sampling time interval  $\tau = 6.48$  seconds, it is assumed that 6 vehicles pass the entry controlled by the light  $\nu_1$ , 8 vehicles pass the entry controlled by the light  $\nu_3$ , and one quarter of vehicles that leave cells 1 and 3 goes out on the first exit (its ratio denoted by  $q$ ). We want to observe the density of the traffic  $x_i$ , given in vehicles per cell, for each cell  $i$  of the road. The model of cells is described by:

$$\begin{aligned} x_1(k+1) &= \left(1 - \frac{\tau v_1}{L_1}\right)x_1(k) + \frac{\tau v_5}{L_5}w_1(k) + 6\nu_1(k) + 0.7\varsigma_1(k), \\ x_i(k+1) &= \left(1 - \frac{\tau v_i}{L_i} - q\right)x_i(k) + \frac{\tau v_{i-1}}{L_{i-1}}w_i(k) + 0.7\varsigma_i(k), \quad i \in \{2, 4\}, \\ x_3(k+1) &= \left(1 - \frac{\tau v_3}{L_3}\right)x_3(k) + \frac{\tau v_2}{L_2}w_3(k) + 8\nu_3(k) + 0.7\varsigma_3(k), \\ x_5(k+1) &= \left(1 - \frac{\tau v_5}{L_5}\right)x_5(k) + \frac{\tau v_4}{L_4}w_5(k) + 0.7\varsigma_5(k), \end{aligned}$$

where  $w_i(k) = x_{i-1}(k)$  (with  $x_0 = x_5$ ). We are interested first in constructing the finite MDP of the given 5-dimensional system and then synthesizing policies keeping the density of the traffic lower than 10 vehicles per cell.

We have  $X := [0, 10]^5$  with a quantization parameter of  $(0.37, 0.37, 0.37, 0.37, 0.37)$ ,  $U := [0, 1]^2$  with a quantization parameter of  $(1, 1)$ , a noise covariance matrix  $\Sigma := \text{diag}(0.7, 0.7, 0.7, 0.7, 0.7)$ , and a cutting probability level  $\gamma$  of  $2e - 2$ . Our algorithms

are applied to the same case study but with 3 dimensions for the sake of benchmarking. The results for both cases are reported in Table 6.2.

#### 6.6.4 Autonomous Vehicle

Here, to show the applicability of the introduced approach to nonlinear models, we consider a vehicle described by the following hybrid 7-dimensional nonlinear single-track model introduced earlier in Chapter 4. The model is modified by including the stochasticity inside the dynamics as additive noise:

For  $|x_4(k)| < 0.1$ :

$$\begin{aligned} x_i(k+1) &= x_i(k) + \tau a_i(k) + R_{i\zeta_i}(k), \quad i \in \{1, \dots, 7\} \setminus \{3, 4\}, \\ x_3(k+1) &= x_3(k) + \tau \text{Sat}_1(u_1) + 0.2\zeta_3(k), \\ x_4(k+1) &= x_4(k) + \tau \text{Sat}_2(u_2) + 0.1\zeta_4(k), \end{aligned}$$

and for  $|x_4(k)| \geq 0.1$ :

$$\begin{aligned} x_i(k+1) &= x_i(k) + \tau b_i(k) + R_{i\zeta_i}(k), \quad i \in \{1, \dots, 7\} \setminus \{3, 4\}, \\ x_3(k+1) &= x_3(k) + \tau \text{Sat}_1(u_1) + 0.2\zeta_3(k), \\ x_4(k+1) &= x_4(k) + \tau \text{Sat}_2(u_2) + 0.1\zeta_4(k), \end{aligned}$$

where,

$$\begin{aligned} R_1 &= R_2 = 0.25, \quad R_5 = R_6 = R_7 = 0.2, \quad a_1 = x_4 \cos(x_5(k)), \quad a_2 = x_4 \sin(x_5(k)), \\ a_5 &= \frac{x_4}{l_{wb}} \tan(x_3(k)), \quad a_6 = \frac{u_2(k)}{l_{wb}} \tan(x_3(k)) + \frac{x_4}{l_{wb} \cos^2(x_3(k))} u_1(k), \quad a_7 = 0, \\ b_1 &= x_4(k) \cos(x_5(k) + x_7(k)), \quad b_2 = x_4(k) \sin(x_5(k) + x_7(k)), \quad b_5 = x_6(k), \\ b_6 &= \frac{\bar{\mu} m}{I_z (l_r + l_f)} (l_f C_{S,f} (gl_r - u_2(k) h_{cg}) x_3(k) + (l_r C_{S,r} (gl_f + u_2(k) h_{cg}) \\ &\quad - l_f C_{S,f} (gl_r - u_2(k) h_{cg})) x_7(k) - (l_f^2 C_{S,f} (gl_r - u_2(k) h_{cg}) \\ &\quad + l_r^2 C_{S,r} (gl_f + u_2(k) h_{cg})) \frac{x_6(k)}{x_4(k)}), \\ b_7 &= \frac{\bar{\mu} f}{x_4(k) (l_r + l_f)} (C_{S,f} (gl_r - u_2(k) h_{cg}) x_3(k) + (C_{S,r} (gl_f + u_2(k) h_{cg}) \\ &\quad + C_{S,f} (gl_r - u_2(k) h_{cg})) x_7(k) - (l_f C_{S,f} (gl_r - u_2(k) h_{cg}) \\ &\quad - l_r C_{S,r} (gl_f + u_2(k) h_{cg})) \frac{x_6(k)}{x_4(k)}) - x_6(k). \end{aligned}$$

We consider an update period  $\tau = 0.1$  seconds and the following parameters for the BMW 320i car:  $l_{wb} = 2.5789$  as the wheelbase,  $m = 1093.3$  [kg] as the total mass of the vehicle,  $\bar{\mu} = 1.0489$  as the friction coefficient,  $l_f = 1.156$  [m] as the distance from the front axle to the CoG,  $l_r = 1.422$  [m] as the distance from the rear axle to CoG,  $h_{cg} = 0.6137$  [m] as the height of CoG,  $I_z = 1791.6$  [kg m<sup>2</sup>] as the moment of inertia for

**Table 6.3:** Comparison between **StochHy** and **AMyTISS** for a continuous-space system with dimensions up to 12. The reported system is autonomous and, hence,  $\hat{U}$  is singleton.  $|\hat{X}|$  refers to the size of the system.

Dimension	2	3	4	5	6	7	8	9	10	11	12
$ \hat{X} $	4	8	16	32	64	128	265	512	1024	2048	4096
Time (s) - <b>StochHy</b>	0.015	0.08	0.17	0.54	2.17	9.57	40.5	171.6	385.5	1708.2	11216
Time (s) - <b>AMyTISS</b>	0.02	0.92	0.20	0.47	1.02	1.95	3.52	6.32	10.72	17.12	29.95

entire mass around  $z$  axis,  $C_{S,f} = 20.89$  [1/rad] as the front cornering stiffness coefficient, and  $C_{S,r} = 20.89$  [1/rad] as the rear cornering stiffness coefficient.

To construct a finite MDP  $\hat{\Sigma}_a$ , we consider a bounded version of the state set  $X := [-10.0, 10.0] \times [-10.0, 10.0] \times [-0.40, 0.40] \times [-2, 2] \times [-0.3, 0.3] \times [-0.4, 0.4] \times [-0.04, 0.04]$ , a state discretization vector  $[4.0; 4.0; 0.2; 1.0; 0.1; 0.2; 0.02]$ , an input set  $U := [-0.4, 0.4] \times [-4, 4]$ , and an input discretization vector  $[0.2; 2.0]$ .

We are interested in an autonomous operation of the vehicle. The vehicle should park itself automatically in the parking lot located in the projected set  $[-1.5, 0.0] \times [0.0, 1.5]$  within 32 time steps. The vehicle should avoid hitting a barrier represented by the set  $[-1.5, 0.0] \times [-0.5, 0.0]$ .

We also apply our algorithms to a 3-dimensional autonomous vehicle [RWR17, Section IX-A] for the sake of benchmarking. The results for both cases are reported in Table 6.2.

### 6.6.5 Benchmarking Against Most Recent State-of-the-art Tool

We benchmark our results against the ones provided by **StochHy** [CDA19]. We employ the same case study as in [CDA19, Case study 3] which starts from 2-dimensional to 12-dimensional continuous-space systems with the same parameters.

To have a fair comparison, we utilize a machine with the same configuration as the one employed in [CDA19] (a laptop having an Intel Core i7 – 8550U CPU at 1.80GHz with 8 GB of RAM). We build a finite MDP for the given model and compare our computation time with the results provided by **StochHy**.

Table 6.3 shows the comparison between **StochHy** and **AMyTISS**. **StochHy** suffers significantly from the CoD as seen from its exponentially growing computation time. **AMyTISS**, on the other hand, outperforms **StochHy** and can handle bigger systems using the same hardware. This comparison shows speedups up to maximum 375 times for the 12-dimensional system. Note that we only reported up to 12-dimensions but **AMyTISS** can readily go beyond this limit for this example. For instance, **AMyTISS** managed to handle the 20-dimensional version of this system in 1572 seconds using an NVIDIA Tesla V100 GPU in Amazon AWS.

Readers are highly advised to pay attention to the size of the system  $|\hat{X} \times \hat{U}|$  (or  $|\hat{X}|$  when  $\hat{U}$  is singleton), not to its dimension. Actually, here, the 12-dimensional system, which has a size of 4096 state-input pairs is much smaller than the 2-dimensional illustrative example we introduced in Section 6.5, which has a size of 203401 state-input pairs. The current example has a small size due to the very coarse quantization parameters and the tight bounds used to quantize  $X$ .

As seen in Table 6.2, **AMyTISS** outperforms **FAUST** and **StocHy** in all the case studies (maximum speedups up to 692000 times). Moreover, **AMyTISS** is the only tool that can utilize all available computing resources. The OFA version of **AMyTISS** reduces dramatically the required memory, while still solves the problems in a reasonable time. **FAUST** and **StocHy** fail to solve many of the problems since they lack the native support for nonlinear systems, they require large amounts of memory, or they do not finish computing within 24 hours.

## 6.7 Summary

Stochastic systems are an important modeling framework to describe many safety-critical CPSs such as power grids and traffic networks. For such complex systems, automating the synthesis of controllers that achieve some high-level specifications is inherently very challenging. In this chapter, we introduced efficient data-parallel algorithms to automatically design controllers for stochastic control systems.

dt-SCSs were chosen as modeling framework for stochastic control systems. dt-SCSs are usually abstracted using finite MDPs. We then propose a technique to efficiently compute the transition probability matrices of the finite MDPs. We presented a parallel algorithm (Algorithm 16) to efficiently construct and store the MDPs as a successor to a traditional serial algorithm. It employs several enhancements from Subsections 6.3.1, 6.3.2, and 6.3.3.

We then employed dynamic programming to synthesize controllers for constructed finite MDPs satisfying safety and reachability properties. Algorithm 20 was also introduced as a unified algorithm for the synthesis of symbolic controllers. It supports safety, reachability and reach-avoid specifications.

The algorithms presented in this chapter were implemented on top of **pFaces** as a kernel that constructs finite MDPs and synthesizes symbolic controllers satisfying given high-level specifications. Kernel **AMyTISS** can utilize HPC platforms and Cloud-computing services to reduce the effects of the CoD. As illustrated with several case studies, **AMyTISS** significantly outperforms all available tools with respect to the computation time.



## 7 Supporting Practical Design Requirements

Computer science literature is rich with approaches for automated synthesis of systems from high-level formal specifications (see [LMS20, PR89, BK08, and the references therein]) which are traditionally known as *reactive synthesis* techniques. Software tools such as STRIX [MSL18], Acacia+ [BBF<sup>+</sup>12], and SLUGS [ER16] provide various implementations of these techniques. Table 7.1 shows a comparison between state-of-art tools of reactive synthesis and symbolic control tools. Unlike the tools developed for symbolic control, the tools of reactive synthesis can handle much richer specifications, including requirements given in LTL or as automata on infinite strings. Unfortunately, they can not be used directly to design controller of control systems. This is mainly because control systems have an uncountable number of states which makes it impossible to programmatically operate on them in their original forms using these tools. Moreover, these tools accept specifications describing the systems to be designed (i.e., the controllers in our case), whereas here we need to provide requirements describing control systems' behaviors to be enforced by the synthesized controllers.

In the previous chapters, we discussed several improvements to the scalability and efficiency of symbolic control. In all the techniques (and algorithms) we introduced, we considered simple high-level specifications, namely, safety and reachability specifications. The goal was to focus on the improvements of the techniques, while knowing that extensions to them can be addressed later to include more practical specifications. In this chapter, we provide a technique on symbolic control that extends the class of supported specifications. More specifically, instead of considering only reachability and safety specifications, we consider  $\omega$ -regular specifications given as DPAs or as LTL formulae. The proposed extension is implemented on top of pFaces as a kernel that we refer to as OmegaThreads [KZ21].

**Table 7.1:** A comparison between state-of-art tools of reactive synthesis and symbolic control

Aspect	Discrete Synthesis Tools Strix <sup>†</sup> SLUGS <sup>‡</sup>	Symbolic Control Tools SCOTS <sup>§</sup> Pessoa <sup>¶</sup>
Systems	Systems	Finite-state systems
Specs.	LTL <sup>†</sup> , GR(1) <sup>‡</sup>	general nonlinear <sup>§</sup> linear <sup>¶</sup>
Algorithms	Parallel and serial	safety/reachability
Platforms	Parallel and serial	Serial
Output	CPUs (or limited GPUs)	CPUs
	Machines/Auto-gen.	Ad-hoc

## 7.1 Specifications and Control Problems

Recall the definition of  $S_\tau$  as a general continuous-time nonlinear control system which is introduced in (2.3.2) as an embedding of the sampled version of the control system  $\Sigma$ . Assume that  $x_0 \in X_0$  is the initial state of  $S_\tau$ , where  $X_0 \subseteq X_\tau$  is a set of initial states. When a sequence of control inputs  $\tilde{u} := \tilde{u}(0)\tilde{u}(1)\cdots$ , where  $\tilde{u}(i) \in U_\tau$  and  $i \in \mathbb{N}$ , is applied to  $S_\tau$  such that  $u(t) = \tilde{u}(i)$  and  $t \in [i\tau, (i+1)\tau[$  for every  $i$ , system  $S_\tau$  generates a run captured by the sequence of  $\tau$ -sampled states  $\tilde{x} := \tilde{x}(0)\tilde{x}(1)\cdots\tilde{x}(k)\cdots$ , where  $\tilde{x}(k) = \xi_{(\tilde{x}(k-1), \tilde{u}(k-1))}(\tau)$  for every  $k \in \mathbb{N}_+$ , and  $\tilde{x}(0) = x_0$ .

Now, consider one control system  $S_\tau$  and its initial set of states  $X_0$ . We provide a definition of  $\omega$ -regular specifications.

**Definition 7.1.1** ( $\omega$ -regular Specification). *Let  $\mathcal{P}_X$  and  $\mathcal{P}_U$  be sets of atomic propositions labeling subsets in  $X_\tau$  and  $U_\tau$ , respectively, such that  $\mathcal{P}_X \cap \mathcal{P}_U = \emptyset$ . Also, let  $\mathcal{L}_X : X_\tau \rightarrow 2^{\mathcal{P}_X}$  and  $\mathcal{L}_U : U_\tau \rightarrow 2^{\mathcal{P}_U}$  be labeling functions. An  $\omega$ -regular specification  $\psi$  for  $S_\tau$  is a set of  $\omega$ -words on  $\mathcal{P} := 2^{\mathcal{P}_X \cup \mathcal{P}_U}$  (i.e.,  $\psi \subseteq \mathcal{P}^\omega$ ).*

Given a sequence  $\tilde{u}$  of control inputs,  $S_\tau$  is said to satisfy  $\psi$  (denoted as usual by  $S_\tau \models \psi$ ) if the following holds:

$$\forall x_0 \in X_0. \exists \alpha \in \psi. \forall i \in \mathbb{N}. \mathcal{L}_X(\tilde{x}(i)) \cup \mathcal{L}_U(\tilde{u}(i)) \subseteq \alpha(i).$$

The controller synthesis problem (denoted by the tuple  $(S_\tau, \psi)$ ) is to find a function  $\delta : X_\tau^* \rightarrow U_\tau$  such that  $S_\tau \models \psi$  when  $\tilde{u}(i) = \delta(\tilde{x}(0)\tilde{x}(1)\cdots\tilde{x}(i))$  for all  $i \geq 0$ .

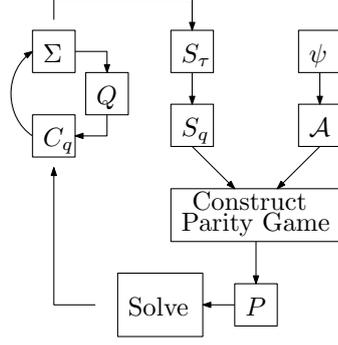
## 7.2 Specifications as Automata on Infinite Strings

Consider symbolic model  $S_q$  of  $S_\tau$  constructed as discussed in Section 4.2. We then construct a two-player parity game (a turn-based game played on a finite graph) [BCJ18] via simultaneous exploration of both  $S_q$  and a parity Automaton  $\mathcal{A}$  whose language is  $\psi$ . Finally, we solve the game and extract a winning strategy in the form of a Mealy machine  $C_q$  encapsulating function  $\delta : X_\tau^* \rightarrow U_\tau$  such that, when it is refined with FRR  $Q$  as a static map and attached in a closed-loop fashion to  $S_\tau$ , we have that  $S_\tau \models \psi$ . We discuss each of these steps next with brief details and refer the reader to their corresponding literature for more details.

Notice that set  $X_q$  is a finite partition of set  $X_\tau$  constructed by a set of hyper-rectangles of identical widths  $\eta_X \in \mathbb{R}_+^n$ . Hereinafter, each hyper-rectangle in  $X_q$  is represented with its center. The representative of the lower-left (resp. upper-right) hyper-rectangle in  $X_q$  is denoted by  $x_{\text{first}}$  (resp.  $x_{\text{last}}$ ). Similarly, set  $U_q$  is a finite set of centers of hyper-rectangles of identical widths  $\eta_U \in \mathbb{R}_+^m$  forming a partition of  $U_\tau$ . The first (resp. last) element in  $U_q$  is denoted by  $u_{\text{first}}$  (resp.  $u_{\text{last}}$ ).

Now, we formally introduce DPAs.

**Definition 7.2.1** (Deterministic Parity Automaton (DPA)). *A DPA is a tuple  $\mathcal{A} := (Q, q_0, \Gamma, T_{\mathcal{A}}, \mathcal{X}, d, p)$ , where  $Q$  is a set of states,  $q_0$  is an initial state,  $\Gamma$  is alphabet,*



**Figure 7.1:** A technique on symbolic control that supports  $\omega$ -regular specifications.

$T_A : Q \times \Gamma \rightarrow Q$  is a transition map,  $\mathcal{X} : Q \times \Gamma \rightarrow \{0, 1, \dots, d\}$  is a transition coloring map,  $d \geq 1$  is a maximal color, and  $p \in \{0, 1\}$  is the parity defining which runs of  $\mathcal{A}$  are accepted.

Run  $\mu := q_0\gamma_0q_1\gamma_1\gamma_0 \dots \in (Q \times \Gamma)^\omega$  of a DPA is  $\mathcal{A}$  is accepting iff the highest value that occurs infinitely often in the sequence  $c(\mu_0\gamma_0)c(\mu_1\gamma_1)c(\mu_2\gamma_2) \dots$  is even/odd (depending on  $p$ ). The language of DPA  $\mathcal{A}$  is the set of all its accepting runs. More details about DPAs can be found in [BCJ18].

We assume that specification  $\psi$  is given as either a DPA  $\mathcal{A}$  whose language is  $\psi$ , or as an LTL formula. In case an LTL formula is given, we construct a DPA from it as discussed in [EKS18]. LTL formulae make describing the requirements of systems easier, formal, compact, and unambiguous. In both cases, we end up having a DPA  $\mathcal{A}$ .

## 7.3 Construction of Parity Games and Symbolic Controller Synthesis

We introduce parity games.

**Definition 7.3.1** (Parity Game). *A parity game is a two-player turn-based game defined by a tuple  $P = (V_S, V_C, V_0, E_S, E_C, \mathcal{K})$ , where  $V_C \subseteq (X_q \cup d_x) \times (U_q \cup d_u) \times Q$  is the set of nodes from which Player 1 can play,  $d_x$  and  $d_u$  are dummy symbols,  $V_C \subseteq \mathbb{N}$  is the set of nodes from which Player 2 can play,  $V_0 := (d_1, d_2, q_0)$  is an initial node,  $E_S \subseteq V_S \times V_C \times X_q$  are  $X_q$ -labelled edges that can be played by Player 1,  $E_C \subseteq V_C \times V_S \times U_q$  are  $U_q$ -labelled edges that can be played by Player 2, and  $\mathcal{K} : E_C \rightarrow \{0, 1, \dots, d\}$  is an edge coloring map for edges played by  $C_q$ . All the edges played by Player 1 are colored with a neutral color (see [LMS20, Section 3] for more details).*

Figure 7.1 depicts the proposed technique on symbolic control that supports  $\omega$ -regular specifications. Having a symbolic model  $S_q$  and a DPA  $\mathcal{A}$ , we construct a parity game  $P$  by exploring simultaneously  $S_q$  (starting at each state of some initial finite set  $X_{q,0} \subseteq X_0$ ) and  $\mathcal{A}$  (starting at  $q_0$ ). In the constructed game, Player 1 is  $S_q$  and Player 2 is a controller  $C_q$  that is defined later as Mealy machine.

**Algorithm 21:** Construction of the parity game  $P$ .

---

**Input:**  $S_q, \mathcal{A}$   
**Output:**  $P = (V_S, V_C, V_0, E_S, E_C, \mathcal{K})$ .

- 1  $V_0 := (d_1, d_2, q_0); V_{new} := \{V_0\};$
- 2  $V_S = \{V_0\}; I_C := 0;$
- 3 **while**  $V_{new} \neq \emptyset$  **do**
- 4      $v := (x, u, q) := \text{Pop}(V_{new});$
- 5     **if**  $x == d_x$  **and**  $u == d_u$  **then**
- 6          $X_{posts} := X_{q,0};$
- 7     **else**
- 8          $X_{posts} := T_q(x, u);$
- 9     **end**
- 10    **for all**  $x_{post} \in X_{posts}$  **do**
- 11          $V_C := V_C \cup I_C;$
- 12          $E_S := E_S \cup (v, I_C, x_{post});$
- 13         **for all**  $u_{apply} \in U_q$  **do**
- 14              $w := \mathcal{L}_X(x_{post}) \cup \mathcal{L}_U(u_{apply});$
- 15              $q_{new} := T_{\mathcal{A}}(q, w);$
- 16              $color := \mathcal{X}(q, label);$
- 17              $n_S := (x_{post}, u_{apply}, q_{new});$
- 18             **if**  $n_S \notin V_S$  **then**
- 19                  $V_S := V_S \cup n_S;$
- 20                  $V_{new} := V_{new} \cup n_S;$
- 21             **end**
- 22              $E_C := E_C \cup (I_C, n_S, u_{apply});$
- 23              $\mathcal{K} := \mathcal{K} \cup ((I_C, n_S, u_{apply}), color);$
- 24         **end**
- 25          $I_C := I_C + 1;$
- 26     **end**
- 27 **end**

---

The construction of  $P$  is presented in Algorithm 21.  $V_{new}$  is a queue of newly discovered nodes in  $V_S$ . Operation  $\text{Pop}(V_{new})$  refers to the extraction (i.e., read and remove) of the first element in  $V_{new}$ .  $I_C$  represents a new node created for player  $C_q$ .  $n_S$  represents a new node created for player  $S_q$ . The construction ends when there are no more new nodes in  $V_{new}$ .

Game  $P$  can then be passed to an off-the-shelf parity game solver. We use the open source implementation of the solver provided in tool **STRIX** which is based on the algorithm in [Lut08]. If  $C_q$  has a winning strategy, the solver marks one or more edges of  $C_q$ , for each edge of  $S_q$ , as winning counter-play edges.

For a simpler presentation, we assume the winning strategy is given as a set of winning rounds (one turn from  $S_q$  followed by one turn from  $C_q$ )  $\widetilde{W} \subseteq E_S \times E_C$  in the sense that  $C_q$  wins by applying  $u_q$  that labels edge  $e_c \in E_C$  when  $S_q$  chooses state  $x_q$  that labels edge  $e_s \in E_S$  if  $(e_s, e_c) \in \widetilde{W}$ . The controller is then extracted as a Mealy machine

$$C_q := (Q_C, q_{C,0}, \delta_C, \lambda_C), \quad (7.3.1)$$

where  $Q_C := \{\underline{v}_S \in V_S \mid \exists (v_S, v_C, x) \in E_S ((v_S, v_C, x), e_C) \in \widetilde{W}\}$  is a set of states,  $q_{C,0} := V_0$  is an initial state,  $\delta_C := \{(\underline{v}_S, \underline{x}_q, \underline{v}'_S) \in Q_C \times X_q \times Q_C \mid ((v_S, v_C, \underline{x}_q), (v_C, v'_S, u_q)) \in \widetilde{W}\}$  is a transition relation, and

$$\begin{aligned} \lambda_C := \{ & (\underline{v}_S, \underline{x}_q, \underline{U}) \in Q_C \times X_q \times 2^{U_q} \mid \exists \widehat{W} \subseteq \widetilde{W} \\ & (\forall ((\tilde{v}_S, v_C, \hat{x}), e_C) \in \widehat{W} (\tilde{v}_S = \underline{v}_S \wedge \hat{x} = \underline{x}_q)) \wedge \\ & (\underline{U} \equiv \{u_q \in U_q \mid (e_S, (v_C, v'_S, u_q)) \in \widehat{W}\})\} \end{aligned}$$

is an output map.

## 7.4 Implementation Details

Similar to all kernels introduced in the previous chapters, **OmegaThreads** is implemented on top of **pFaces**. **pFaces** takes care of running **OmegaThreads** on all available compute platforms, which allows the users to control the trade-off between computation time and available resources.

**OmegaThreads** requires a configuration file (.cfg file) describing one control problem  $(S_\tau, \psi)$ . If  $\psi$  is given in the form of a text file describing a DPA, we load the DPA file to be used later. If  $\psi$  is given in the form of an LTL formula, we use library **OWL** [KMS18] to construct a DPA whose language is  $\psi$ .

**OmegaThreads** constructs the symbolic model  $S_q$  in parallel by implementing Algorithm 6 from Chapter 4. Afterwards, it constructs maps  $\mathcal{L}_X$  (resp.  $\mathcal{L}_U$ ) by running, in parallel, threads for each representative point in  $X_q$  (resp.  $U_q$ ). Then, it constructs  $P$  by running the implementation of Algorithm 21 with  $S_q$  and  $\mathcal{A}$  as inputs.

We use the parity game solver in **STRIX** which combines serial and parallel parts. It is rewritten to work with **pFaces**.  $P$  is passed to the parity game solver to find a winning strategy. Once a winning strategy is found, we extract the controller  $C_q$  as a Mealy

machine and write it to a text file. In `OmegaThreads`, we implemented a Python interface to read the controller  $C_q$  and a 2D simulator to simulate the closed-loop constructed from  $\Sigma$ ,  $C_q$  and FRR  $Q$  as the interface between them.

Kernel `OmegaThreads` combines the best of all the techniques introduced in the previous chapters (excluding Chapter 6, which deals with stochastic systems). It is developed for with simplicity if usage in mind. More precisely, three simple steps get users started with `OmegaThreads`: (1) submitting control problem  $(S_\tau, \psi)$ , (2) running `OmegaThreads` to synthesize controller  $C_q$ , and (3) simulating the closed-loop. We consider motion planning for an autonomous vehicle as a running example. Complete details about this example are presented later in Subsection 7.5.1.

### 7.4.1 Submitting Control Problems

As discussed earlier in this chapter, the synthesis of controller  $C_q$  requires two inputs: (1) a description of the considered control system  $S_\tau$ , and (2) a specification  $\psi$  that  $C_q$  should enforce on  $S_\tau$ . In `OmegaThreads`, all user inputs are given in configuration (`.cfg`) text files such that each file describes one control problem  $(S_\tau, \psi)$ . They contain scopes (i.e., titled sections) containing sub-scopes or related configuration contents, which are lists of (`key="value"`) pairs ending with semicolons. The contents of a scope is enclosed by curly brackets. The keys and sub-scopes in any scope have unique names. Such hierarchical structure assigns by construction a unique address for each configuration value (e.g., in the form of `scope.subscope.key` in case of two-level nesting). The configuration file of the autonomous vehicle example in Subsection 7.5.1 is listed below:

```

1 project_name="vehicle";
2 system {
3   states {
4     dimension="3";
5     first_symbol="0.0,0.0,-1.6";
6     last_symbol="5.0,5.0,1.6";
7     quantizers="0.2,0.2,0.2";
8     initial_set="[2.35,2.45]x[0.55,0.65]x[-0.05,0.05]";
9     subsets {
10      names="target1, _target2, _obstacles";
11      mapping_target1="[3.7,5.2]x[-0.2,1.3]x[-2,2]";
12      mapping_target2="[3.7,5.2]x[3.7,5.2]x[-2,2]";
13      mapping_obstacles="[1.5,5.5]x[1.3,3.7]x[-2,2]";
14    }
15  }
16  controls {
17    dimension="2";
18    first_symbol="-2.0,-0.8";
19    last_symbol="2.0,0.8";
20    quantizers="0.4,0.2";
21    subsets {
22      names="lowspeed";
23      mapping_lowspeed="[-1.7,1.7]x[-1.0,1.0]";
24    }

```

```

25     }
26     dynamics {
27         code_file="vehicle.cl";
28     }
29 }
30 specifications {
31     ltl_formula="GF(target1) && GF(target2) && G(!obstacles) && G(lowspeed)";
32     write_dpa="false";
33 }
34 implementation {
35     type="mealy_machine";
36     generate_controller="true";
37 }
38 simulation {
39     ...
40 }

```

`project_name` gives a name for the control problem to be used later for saving any output files. Scopes `system` and `specifications` define  $S_\tau$  and  $\psi$ , respectively. Scope `implementation` declares how the synthesized controller should be generated.

Scope `system` contains sub-scopes `states` and `controls` describing, respectively, how states set  $X_q$  and inputs set  $U_q$  are constructed. Sub-scope `states` (resp. `controls`) must have the keys: `dimension` for dimension  $n$  (resp.  $m$ ), `first_symbol` for  $x_{first}$  (resp.  $u_{first}$ ), `last_symbol` for  $x_{last}$  (resp.  $u_{last}$ ), and `quantizers` for  $\eta_X$  (resp.  $\eta_U$ ). Note that vector-valued configurations are given as comma-separated lists of floating point numbers. Sub-scope `states` must also declare `initial_set` representing  $X_0 \subset X_\tau$  as a hyper-rectangle. Hyper-rectangles are given as Cartesian products of closed intervals.

Sub-scope `states` (resp. `controls`) defines atomic propositions  $\mathcal{P}_X$  (resp.  $\mathcal{P}_U$ ) in sub-scope `subsets` using key `names` to be used later for defining  $\psi$ . For each atomic proposition in `states.subsets.names` (resp. `controls.subsets.states`), one mapping value must be supplied. A mapping value is defined with a key that begins with `mapping_` followed by the atomic proposition. Mapping values are used by `OmegaThreads` to construct the labeling maps  $\mathcal{L}_X$  and  $\mathcal{L}_U$ . For example, in the above configuration file, atomic proposition `target1` in  $\mathcal{P}_X$  is identified by its preimage  $L_X^{-1}(\text{target1})$  defined in element `system.states.subsets.mapping_target1` as a hyper-rectangle  $[3.7, 5.2] \times [-0.2, 1.3] \times [-2, 2]$ . A mapping value can also be given as a union of hyper-rectangles using letter `U` as a separator.

Sub-scope `dynamics` defines the dynamics of  $S_\tau$  that is used to construct  $T_q$ . It should contain key `code_file` providing a path to a file that specifies the dynamics. The path should be relative to the configuration file and the file should declare at least one function written in C-language with the following function prototype:

```

1 void model_post(
2     concrete_t* post_x_lb, concrete_t* post_x_ub,
3     const concrete_t* x, const concrete_t* u);

```

where `concrete_t` is a floating point precision type shared between C++ and OpenCL languages. Function outputs `post_x_lb` and `post_x_ub` should be respectively assigned

by the user's function to the lower-left vector and upper-right vector defining together a hyper-rectangle that represents an OARS of  $S_\tau$  starting at the hyper-rectangle in  $X_q$  whose representative is  $\mathbf{x}$  and applying constant control input  $\mathbf{u}$  for one sampling period. Users can benefit from the parallel ODE solvers implemented in `pFaces` to evaluate function `model_post` for continuous-time and hybrid systems, as we show later in Section 7.5.

Scope `specifications` should have key `ltl_formula` whose value is an LTL formula describing  $\psi$ , or key `dpa_file` whose value specifies a file of a DPA representing  $\psi$ . In the value of `ltl_formula`, users can use atomic propositions defined in `states.names` and `controls.names`, logical operators (& for logical AND, | for logical OR, ! for logical NOT, and XOR for logical XOR), and modal operators (X for next, U for until, G for always, F for eventually, R for release, W for weak until, and M for strong release). Key `write_dpa` declares whether the DPA constructed from the LTL formula should be written as a file or not.

Scope `implementation` defines how the synthesized symbolic controller is implemented. Currently, `OmegaThreads` supports generating dynamic controllers as Mealy machines. Hence, the value of key `type` is set to `mealy_machine`. Key `generate_controller` declares whether the controller should be generated or not.

Scope `simulation` defines configurations needed by the simulator to simulate and visualize the closed-loop. The list of configuration items in scope `simulation` are omitted as we discuss them later in Subsection 7.4.3.

The file is saved as `vehicle.cfg` inside directory `./examples/vehicle3d/` relative to the installation root directory.

## 7.4.2 Synthesizing Symbolic Controllers

Having designed configuration files for control problems, users should run `OmegaThreads` to synthesize controllers solving them. Since `OmegaThreads` is a kernel built on top of `pFaces`, we launch it by passing it to `pFaces` as follows (Linux Shell syntax is used for demonstration):

```
1 $ pfaces -CG -d 1 -k omega -cfg vehicle.cfg
```

where (\$) is the command prompt, (`pfaces`) calls `pFaces`, (`-CG -d 1`) asks `pFaces` to run kernels in the first device of all available CPU and GPU devices, (`-k omega`) tells `pFaces` to load `OmegaThreads` as a kernel, (`-cfg vehicle.cfg`) asks `pFaces` to hand the configuration file `vehicle.cfg` to `OmegaThreads`. The command is assumed to be run in directory `./examples/vehicle3d/`. Users can run `OmegaThreads` on other devices by changing the device index.

## 7.4.3 Collecting Results and Simulations

If `OmegaThreads` is successful in solving the symbolic control problem, it generates a file (`project_name.mdf`) containing the controller as a Mealy machine, where `project_name` is the value of key `project_name`. For the current example, the resulting controller

file should be `vehicle.mdf`. The controller file is a text file that describes the states, transitions and outputs of controller  $C_q$ .

`OmegaThreads` provides a `Python` interface to access the generated controller. Users can load the controller as follows (assuming the following code runs from a `Python` file next to `vehicle.mdf`):

```
1 from OmegaInterface import Controller
2 C = Controller('vehicle.mdf')
```

which constructs object `C` encapsulating controller  $C_q$ . Now, if  $\mathbf{x}$  is the current state of system  $S_\tau$ , users can use object `C` as follows:

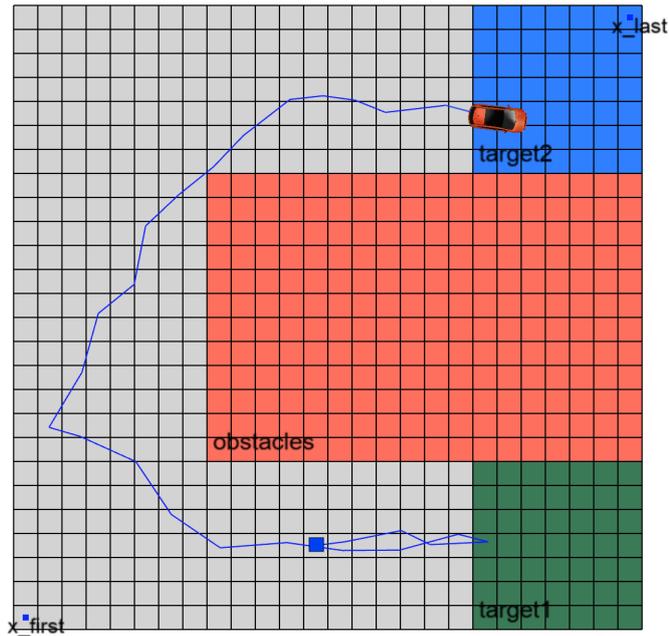
```
1 control_inputs = C.get_control_actions(x)
```

to updates the internal state of the Mealy machine and extract the list of admissible control inputs.

Users may also simulate the closed-loop using a 2D simulator. It is developed in `Python` and it requires some related configurations which need to be supplied in the same configuration file used to synthesize the controller. The following is scope `simulation` used in the configuration file:

```
1 simulation {
2     window_width = "600";
3     window_height = "600";
4     widow_title = "Autonomous_Vehicle";
5     initial_state = "center";
6     controller_file = "vehicle.mdf";
7     system_image = "vehicle.png";
8     system_image_scale = "0.035";
9     step_time = "0.3";
10    use_ode = "true";
11    visualize_3rdDim = "true";
12    skip_APs = "lowspeed";
13 }
```

where keys `window_width`, `window_height`, and `widow_title` configure, respectively, the width of the simulation window in pixels, its height in pixels, and its title. Key `initial_state` tells the simulator how to choose initial state  $x_0$ . Its value can be either `center`, `random` or a specific initial state  $\mathbf{x} \in X_{q,0}$ . For value `center`, the simulator will choose the center point in  $X_0$  as initial state. For value `random`, the simulator will choose a random point in  $X_0$  as initial state. Key `controller_file` should point to the file containing the synthesized controller. Key `system_image` should point to an image file of the system to be used for visualization. Key `system_image_scale` is a multiplier for the size of system's image. Key `step_time` specifies the simulation step time (i.e., sampling period  $\tau$ ). Key `use_ode` asks the simulator to use (or not use) an ODE solver when simulating the system which should be consistent with the implementation of function `model_post`. Setting key `visualize_3rdDim` to `true` will ask the simulator to simulate



**Figure 7.2:** Closed-loop simulation of the vehicle example captured once region `target2` (blue rectangle) is reached.

the third dimension of the system if  $n \geq 3$ . The third dimension is always simulated as a rotation angle of the system's image. Key `skip_APs` provides a list of atomic propositions that should not be visualized. Now, the simulation can be started using a simple Python script as follows:

```

1 from Omega2dSimulator import Omega2dSimulator
2 Omega2dSimulator(
3     model_post,      # system dynamics function
4     "vehicle.cfg"    # the config file
5 ).start()

```

where `model_post` is a model dynamics function implemented in Python. Users must supply an implementation of the system dynamics identical to the one in the file pointed to by key `code_file`.

Running the simulator will internally do the following: (1) read the provided configuration file, (2) create a controller object from the provided controller file, (3) draw the states set and its subsets on a 2D space using different colors, (3) start an infinite time closed-loop simulation between  $S_\tau$  and  $C_q$ . Figure 7.2 shows the simulator after few seconds from launching it.

## 7.5 Examples

We consider some examples to demonstrate the capabilities of `OmegaThreads`. All examples are run on a MacBook Pro (2018) laptop with 2.9 GHz Intel Core i9 processor and 32Gb RAM.

### 7.5.1 Motion Planning for Autonomous Vehicles

We consider an autonomous vehicle described by the following bicycle dynamics [RZ16]:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} = \begin{bmatrix} u_1 \frac{\cos(a+x_2)}{\cos(a)} \\ u_1 \frac{\sin(a+x_2)}{\cos(a)} \\ u_1 \tan(u_2) \end{bmatrix}, \quad (7.5.1)$$

where  $x_1$  and  $x_2$  are the position coordinates,  $x_3$  is the orientation of the vehicle, and  $a := \arctan(\tan(u_2/2))$ . Control inputs  $u_1$  and  $u_2$  are respectively the velocity and steering angle. The objective is given as an LTL formula

$$\psi := \square \diamond \text{target1} \wedge \square \diamond \text{target2} \wedge \square ! \text{obstacles} \wedge \square \text{low\_speed},$$

where `target1`, `target2`, `obstacles`, and `low_speed` are atomic propositions defined in the configuration file introduced in Subsection 7.4.1. The objective requires that the vehicle infinitely-often visits the subsets labelled by `target1` and `target2`, always avoids the subset labeled by `obstacles`, and always uses low velocity values labeled by `low_speed`.

To implement function (`model_post`), we first introduce map  $\beta : \mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{R}^n$ :

$$\beta\left(\begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix}, \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}\right) = \begin{bmatrix} r_1 + \tau C r_3 \\ r_2 + \tau C r_3 \\ r_3 \end{bmatrix}, \quad (7.5.2)$$

where  $C := |u_1 \sqrt{\frac{\tan^2(u_2)}{4.0} + 1}|$  and  $\tau := 0.3$ . Map  $\beta$  is used to compute an OARS starting at any hyper-rectangle in  $X_q$  by providing a bound on the growth of the solutions of (7.5.1).

Now, we provide the implementation of the `model_post` function as follows:

```

1 #include "rk4ode.cl"
2 void model_post(
3     concrete_t* post_x_lb, concrete_t* post_x_ub,
4     const concrete_t* x, const concrete_t* u){
5
6     // some required vars
7     concrete_t Q[ssDim] = {ssQnt};
8     concrete_t xx[ssDim];
9     concrete_t rr[ssDim];
10    concrete_t r[ssDim];
11
12    // initialization

```

## 7 Supporting Practical Design Requirements

```
13     for (unsigned int i = 0; i < ssDim; i++)
14         r[i] = Q[i] / 2.0f;
15
16     // solve the \gls{ode} and calculate the growth bound
17     rk4OdeSolver(xx, x, u, 'x');
18     radius_dynamics(rr, r, u);
19
20     // compute the OARS
21     for (unsigned int i = 0; i < 3; i++) {
22         post_x_lb[i] = xx[i] - rr[i];
23         post_x_ub[i] = xx[i] + rr[i];
24     }
25 }
```

File (`rk4ode.c1`) is provided by `pFaces` and it contains a Runge-Kutta ODE solver as a function `rk4OdeSolver` that is initialized with the ODE defined in (7.5.1). `radius_dynamics` is a function encapsulating map  $\beta$  defined in (7.5.2). Variable `xx` receives the solution of the system's ODE after one sampling period using input variable `x` as initial condition and input variable `u` as control input. Variable `r` is initialized to  $\eta_X/2$ . Variable `rr` receives the output of the growth bound map  $\beta$ .

We automate the steps needed for running `OmegaThreads` and the simulator using two script files and use them as follows:

```
1 $ sh solve.sh
2 $ python3 simulate.py
```

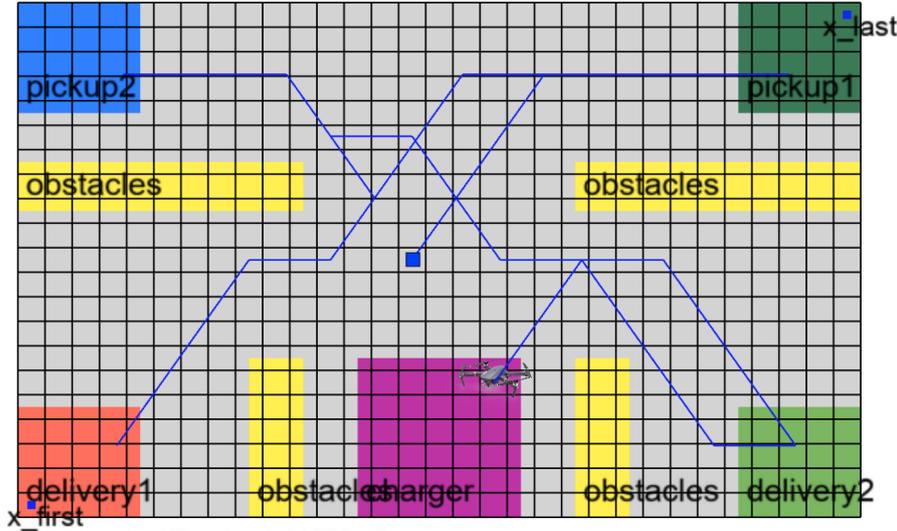
This should solve the vehicle motion planning control problem and simulate the closed-loop as depicted in Fig. 7.2.

DPA  $\mathcal{A}$  is constructed in less than one second and it has 3 states, i.e.  $|Q| = 3$ . Symbolic model  $S_q$  is constructed in less than a second, where  $|X_q| = 11492$  and  $|U_q| = 81$ . Parity game  $P$  is constructed in 5 minutes, where  $|V_S| = 1451521$ ,  $|E_S| = 10456606$ ,  $|V_C| = 18441$ , and  $|E_C| = 1493640$ . `OmegaThreads` wins  $P$  in 20 seconds. The example is implement in tool `SCOTS` for comparison. It solves the problem in 4.8 minutes. A special program is developed to extract the controller from the computed subsets after the computation of fixed points. Developing such a program costs around 1 hour.

### 7.5.2 Pickup-Delivery Drone on Battery

Now, we consider another example that is more complex in terms of system dynamics and requirements. We design a controller for a drone operating on a battery, described by the following nonlinear impulsive system:

$$\Sigma : \begin{cases} \dot{x}(t) = f(x(t), u) & t \in \mathbb{R}_{\geq 0} \setminus \Omega \\ x(t) = g(x(-t), u) & t \in \Omega \end{cases}, \quad (7.5.3)$$



**Figure 7.3:** Simulation of the drone example after 63 seconds as the drone starts a charging task.

where  $\Omega := \{k\tau\}_{k \in \mathbb{N}}$ ,  $\tau := 0.1$  is the sampling period,

$$f\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}\right) = \begin{bmatrix} u_1 \cos(u_2) \\ u_1 \sin(u_2) \\ 0 \end{bmatrix}, \quad g\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}\right) = \begin{bmatrix} x_1 \\ x_2 \\ B(x) \end{bmatrix},$$

$$B\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{cases} \min\{99, x_3 + 20\}, & \text{if } (x_1, x_2) \in \text{Charger}, \\ \max\{0, x_3 - 1\}, & \text{otherwise,} \end{cases}$$

$x_1$  and  $x_2$  are the 2D position coordinates of the drone,  $x_3$  is the state of charge of the battery, and  $\text{Charger} \subset \mathbb{R}^2$  is a selected area where a charger for the drone's battery is available. The control inputs  $u_1$  and  $u_2$  are respectively the velocity and heading angle. The control objective is given as the LTL formula

$$\begin{aligned} \psi := & \square(\neg \text{Obstacles}) \wedge \square\Diamond(\text{Pickup1}) \wedge \square\Diamond(\text{Pickup2}) \\ & \wedge \square(\text{Pickup1} \implies (\neg \text{Pickup2} \wedge \neg \text{Delivery2} \text{ U } \text{Delivery1})) \\ & \wedge \square(\text{Pickup2} \implies (\neg \text{Pickup1} \wedge \neg \text{Delivery1} \text{ U } \text{Delivery2})) \\ & \wedge \square(\neg \text{low\_battery}) \wedge \square\Diamond(\text{full\_battery}), \end{aligned}$$

which, in a simple language, asks that

- the drone performs infinitely-often two different pickup-and-delivery tasks,
- when a task starts, it should be completed without starting another task,
- some obstacles must be always avoided, and

- the battery must not reach a low state and from time to time, it should be fully charged.

We fix  $X := [0, 15] \times [0, 10] \times [0, 99]$ ,  $U := [-1, 1] \times [-1, 1]$ , and  $\text{Charger} := [6, 9] \times [-0.5, 3] \times [-0.5, 100.5]$ . The atomic propositions are defined as follows: `Pickup1` labels subset  $[13, 15.5] \times [8, 10.5] \times [-0.5, 100.5]$ , `Pickup2` labels subset  $[-0.5, 2] \times [8, 10.5] \times [-0.5, 100.5]$ , `Delivery1` labels subset  $[-0.5, 2] \times [-0.5, 2] \times [-0.5, 100.5]$ , `Delivery2` labels subset  $[13, 15.5] \times [-0.5, 2] \times [-0.5, 100.5]$ , `Obstacles` labels subset  $[10.0, 15.5] \times [6.0, 7.0] \times [-0.5, 100.5] \cup [-0.5, 5.0] \times [6.0, 7.0] \times [-0.5, 100.5] \cup [3.0, 4.0] \times [-0.5, 3.0] \times [-0.5, 100.5] \cup [11.0, 12.0] \times [-0.5, 3.0] \times [-0.5, 100.5]$ , `low_battery` labels subset  $[-0.5, 15.5] \times [-0.5, 10.5] \times [-0.5, 9.5]$ , and `full_battery` labels subset  $[-0.5, 15.5] \times [-0.5, 10.5] \times [98.5, 99.5]$ .

The two equations defined in (7.5.3) are implemented in C-function `model_post`. We utilize `pFaces`'s ODE solver for the ODE in (7.5.3). The difference equation in (7.5.3) is implemented directly inside function `model_post` where one time step is  $\tau$ . The configuration values and launch scripts are developed for this example and provided to `OmegaThreads` as explained in the previous subsection. Figure 7.3 shows the resulting simulation window.

DPA  $\mathcal{A}$  is constructed in less than a second, where  $|Q| = 10$ . Symbolic model  $S_q$  is constructed in less than a second where  $|X_q| = 65100$  and  $|U_q| = 9$ . Parity game  $P$  is constructed in 71 seconds, where  $|V_S| = 3372940$ ,  $|E_S| = 13270952$ ,  $|V_C| = 378550$ , and  $|E_C| = 3406941$ . `OmegaThreads` wins  $P$  in 58 seconds. A comparison with similar tools is not possible for this case study since all the existing symbolic control tools can not handle this control problem (see the discussion in Chapter 1 for further details).

## 7.6 Summary

In all the techniques (and algorithms) we introduced in the previous chapters, we considered simple high-level specifications, namely, safety and reachability specifications. The goal was to focus on the improvements of the techniques, while knowing that extensions to them can be addressed later to include more practical specifications. In this chapter, we provided a technique on symbolic control that extends the class of supported specifications, namely,  $\omega$ -regular specifications.

Specifications are given as either DPAs, or as LTL formulae. In case of LTL formulae, DPAs can be readily constructed from them. LTL formulae make describing the requirements of systems easier, formal, compact, and unambiguous. A parity game is then constructed from the specification's DPA and the symbolic model. Such construction is presented in Algorithm 21. We then used a standard parity game solver which combines serial and parallel parts. If succeeded, the solver provided a winning strategy to satisfy the specification. We then extract the controller from the winning strategy as a Mealy machine.

The technique introduced in this chapter is implemented in kernel `OmegaThreads` on top of `pFaces`. Inputs to `OmegaThreads` are given in a simple language. It constructs parity games from abstractions of control systems and DPAs describing high-level objec-

tives. After solving the parity games, it generates dynamic symbolic controllers as Mealy machines. The outputs of `OmegaThreads` are simulated visually using a 2D simulator



## 8 Standardized Implementations of Synthesized Symbolic Controllers

Symbolic controllers are attached to original systems in a closed-loop fashion to receive quantized states, and issue correspondingly suitable control inputs that steer the original systems ensuring the satisfaction of the design requirements. This process happens within a hard real-time control window (a.k.a. control cycle). Respecting the timing limits of control cycles is crucial to the correct operation of symbolic controllers. The implementations of symbolic controllers should then be highly predictable with respect to both time and memory requirements. Unfortunately, available state-of-the-art tools for symbolic controller synthesis (e.g., SCOTS) do not provide a formal approach to implementing the designed controllers. The implementations of the controllers are usually done using ad-hoc techniques (e.g., manually-coded error-prone implementations). Consequently, the final product may lose any correctness guarantee obtained from the controller synthesis tool.

In this chapter, we focus on implementing the symbolic controllers resulting from any symbolic controller synthesis technique introduced in the previous chapters. Hereinafter, the term “*implementation of symbolic controllers*” refers to the process of providing a software or hardware product that encapsulates a symbolic controller and can be directly placed in closed-loop with the original system. We discuss the types of symbolic controllers resulting from the algorithms and software tools introduced in the previous chapters. We also present formal implementations for both static and dynamic controllers.

### 8.1 Types of Symbolic Controllers

We discuss briefly the different types of symbolic controllers generated from the kernels introduced in the previous chapters. More precisely, we discuss the types of controllers resulting from kernels `GBFP`, `AMYTISS`, and `OmegaThreads`. One can generally classify the synthesized symbolic controllers resulting from these kernels into two types: static and dynamic symbolic controllers.

Recall the symbolic model  $S_q$  defined in (2.3.3). A static symbolic controller for  $S_q$ , as the name suggests, is a map  $C_q : X_q \rightarrow 2^{U_q}$  that provides a set of admissible control inputs for a given state of the symbolic model regardless of the wall-clock time and the history of previous states. Kernels `GBFP` and `AMYTISS` generate static symbolic controllers.

A dynamic symbolic controller has, on the other hand, an internal state (memory) that gets changed in response to external events (e.g., after receiving a specific sequence

$x \in X_q$	$C(x)$
$x_0$	$\{u_2, u_5, u_7\}$
$x_1$	$\emptyset$
$x_2$	$U_q$
$x_3$	$\{u_1, u_2, u_4, u_8\}$
$\dots$	$\dots$
$x_{ X_q -1}$	$C(x_{ X_q -1})$

**Figure 8.1:** A LUT encoding controller  $C_q$  for system  $S_q$ .

of states from the system or after some predefined time) and consequently, can provide different control inputs for the same received state of the symbolic model, if needed. A dynamic symbolic controller can be seen as map on the history of states  $C_q : X_q^* \rightarrow 2^{U_q}$ . However, since this requires infinite memory, it is more practical to maintain the control laws in dynamic controllers as systems  $C_q$  following the definition in 2.2.1. Kernel **OmegaThreads** generates dynamic symbolic controllers as Mealy machines (see Chapter 7 for further details).

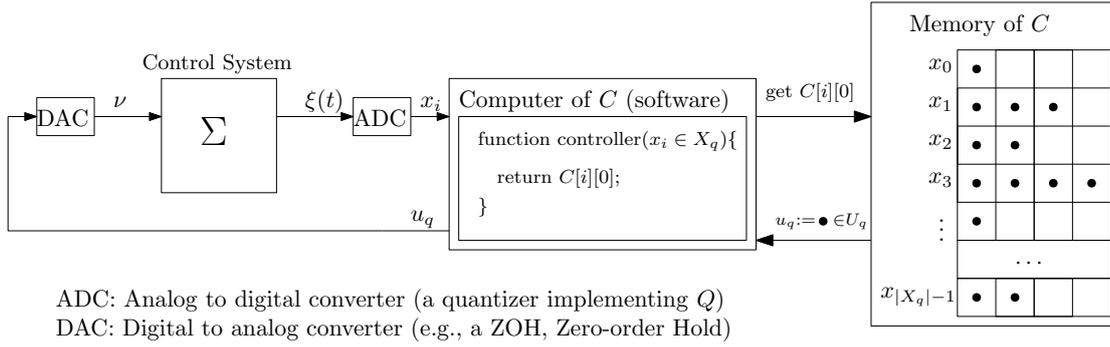
Initial design requirements help design engineers decide which tool to use and what type of controllers they would expect. Simple specifications like safety and reachability can be enforced using simple static controllers. Complex specifications (e.g., LTL-based specifications) usually require dynamic symbolic controllers to enforce them.

In the next subsections, we discuss several implementations of symbolic controllers. We address formally implementations of the two types of symbolic controllers by studying the space and time complexities of each proposed implementation. We assume the controllers are synthesized for a symbolic model  $S_q$  with  $|X_q| \in \mathbb{N}_+$  states and it accepts control inputs from a set of  $|U_q| \in \mathbb{N}_+$  possible inputs. It is also worth mentioning that, after the controller synthesis process, the set of control inputs  $U_q := \{u_0, u_1, \dots, u_{|U_q|-1}\}$  is known to  $C_q$ , and for each state  $x \in X_q := \{x_0, x_1, \dots, x_{|X_q|-1}\}$ ,  $U_{S_q}(x)$  is also known to  $C_q$ .

## 8.2 Formal Implementations of Static Symbolic Controllers

### 8.2.1 Look-Up-Tables (LUTs)

A LUT for a static symbolic controller is a table that has one entry corresponding to each state  $x_q \in X_q$  of the symbolic model. Within each entry, there is one or more admissible control inputs  $\{u \in U_q \mid u \in C_q(x_q)\}$ . Once a state  $x_q$  is received by the



**Figure 8.2:** An implementation of a LUT encoding controller  $C_q$ .

symbolic controller during the closed-loop, the controller picks one of the control inputs corresponding to the received state and send it to the original system. Figure 8.1 shows an example of a LUT. The inputs admissible to each state are selected randomly for the sake of demonstration.

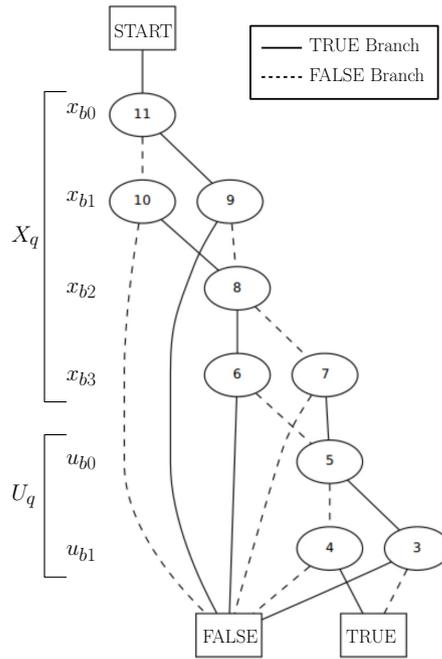
Such simple implementation is highly predictable in both space and time. In practice, the states representing of the symbolic model are encoded as integer values from 0 to  $|X_q| - 1$ . This means that the LUT does not need to store explicitly those states. Consequently, the table becomes an array of length  $|X_q|$  where the input admissible to a state  $x_i$  are stored in the array element of index  $i$ . Clearly, the space requirement for a LUT of a symbolic controller is  $O(|X_q| \times |U_q|)$ , since any element in the array can, in the worst case, contain  $U_q$ . In an actual implementation of  $C_q$ , the array would then be a 2d array.

Now, since the states are encoded as integers, acquiring the inputs for a state  $x_i$  reduces to accessing the entry at index  $i$  which is implemented in one computation step if the LUT is stored in a RAM and the width of the table is fixed. Hence, the time complexity for a LUT implementation is  $O(1)$ . Unfortunately, a LUT is not always practical since usually symbolic models are extremely large ( $|X_q|$  equals to millions or billions) which results in very large LUTs.

### 8.2.1.1 Example Implementations:

Figure 8.2 shows an example implementation of a LUT implementing a static controller  $C_q$ . The implementation consists of the following two parts:

- Memory: a table stored in RAM containing possible admissible inputs  $\{u \in U_q \mid u \in C_q(x_i)\}$  (represented with bold dot in Fig. 8.2) for each system state  $x_i \in X_q$ .
- Computer: a thin layer of software (direct address translation) to directly access the memory entry of a given state  $x_i$ . This simple implementation gets the first  $u$  from all those admissible control inputs at  $x_i$ .



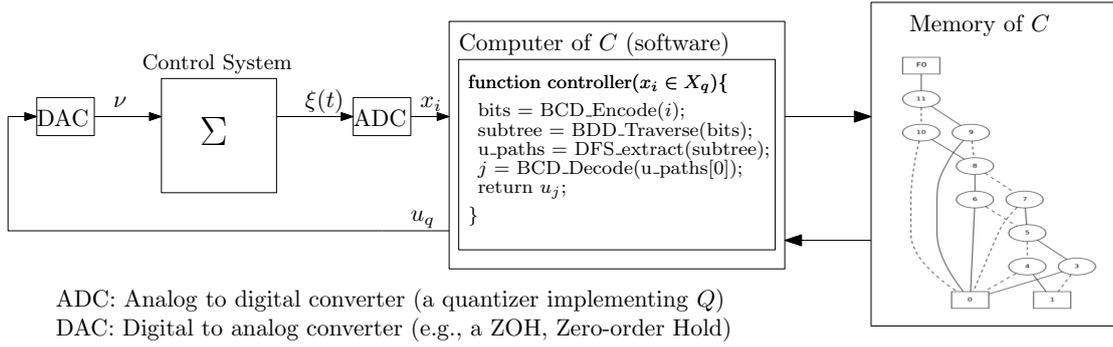
**Figure 8.3:** A BDD encoding a controller  $C_q$  for a symbolic model with  $|X_q| = 16$  and  $|U_q| = 4$ .

### 8.2.2 Binary Decision Diagram (BDD)-encoded Symbolic Controllers

One practical solution to store large control laws is to represent them with BDDs. Consider a control law represented by a set of state-input pairs  $\{(x_q, u_q) \in X_q \times U_q \mid u_q \in C_q(x_q)\}$ . In a BDD that represents the control law, the  $|X_q|$  states expected by the controller are encoded (e.g., Binary Coded Decimal (BCD) coding) with  $N_X := \log_2(|X_q|)$  binary variables. A binary variable for the symbolic states is denoted by  $x_{bi}$ ,  $i \in \{0, 1, \dots, N_X\}$ . Similarly, the  $|U_q|$  possible control inputs are encoded with  $N_U := \log_2(|U_q|)$  binary variables. A binary variable for the inputs is denoted by  $u_{bj}$ ,  $j \in \{0, 1, \dots, N_U\}$ .

Then a binary tree of  $N_X + N_U$  levels is constructed. Each level has some nodes and the nodes connect only to next levels. The tree starts with a single node and at each level it branches to a branch representing value 0 (FALSE branch) and a branch representing value 1 (TRUE branch). At the last level of the tree, one would expect  $2^{(N_X+N_U)}$  final branches representing the different  $|X_q| \times |U_q|$  possibilities of state-input pairs. At the bottom of the such binary tree, we have two special nodes to collect all the branches; a TRUE node and a FALSE node. Those paths along the tree that encode state-input pairs from the control law are connected to the TRUE terminal node.

Several reductions are then made to the diagram to reduce its size. This includes removing all nodes bound directly to the FALSE terminal node, merging equivalent leaves, or merging isomorphic nodes. This results in a reduced-order BDD which is practically



**Figure 8.4:** An implementation of a BDD encoding controller  $C_q$ .

known to be very compact for symbolic controllers since usually few control inputs are assigned to each state as a result of the natural sparsity and locality of original systems.

Figure 8.3 depicts an example reduced-order BDD encoding a controller  $C_q$  for a symbolic model with  $|X_q| = 16$  and  $|U_q| = 4$ . The branches leading to the **FALSE** terminal node are kept for the sake of demonstration. Let us traverse the rightmost path leading to the **TRUE** terminal node. Traversing this path leads to the sequence  $(x_{b_0}x_{b_1}x_{b_2}x_{b_3}) = (1001)$ , which corresponds to BDC code 1001 referring to state  $x_9$ . Now we know this branch representing all stat-input pairs of the control law having  $x_9$  as a state. We continue traverse the path to the end which leads to the sequence  $(u_{b_0}u_{b_1}) = (10)$ , which corresponds to BDC 01 referring to input  $u_1$ . Now, we know that the control law has the pair  $(x_9, u_1)$  meaning that  $u_1$  is admissible when the symbolic controller receives  $x_9$ .

The space complexity of BDDs depends on what reductions are made and how the binary variables are ordered, which usually includes many preprocessing heuristics to reach a small-sized BDD. This is however done offline and as the BDD is constructed, no further modifications to its structure are needed. Generally, the required number of nodes in the BDD is  $O((N_X + N_U) \times 2^M)$  where  $M$  is function of the cross-sectional size of the BDD after fixing the reduction operations and variable ordering.

The time complexity for accessing the elements of the BDD for one state  $x_q \in X_q$  is  $O(N_X + 2^{N_U})$  which reflects the fact that one must navigate the tree for  $N_X$  levels encoding the received state and then extract the possible control actions along the rest of levels of the tree (the subtree whose head starts after the  $N_X$ ).

### 8.2.2.1 Example Implementations:

Figure 8.4 shows an example implementation based on a BDD encoding a static controller  $C_q$ . The implementation consists of the following two parts:

- Memory: a data structure encoding the BDD graph.
- Computer: a Software to traverse the path encoding the received state  $x_i \in X_q$  for  $N_X$  levels and then explore the rest of the tree (e.g., using a Breadth First Search (BFS) algorithm) to collect the admissible control inputs  $\{u \in U_q \mid u \in C_q(x_i)\}$ .

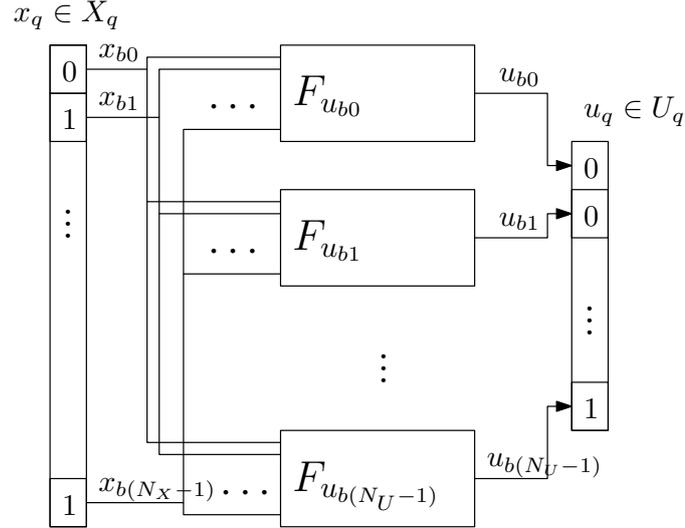


Figure 8.5: A Boolean circuit of  $N_U$  Boolean functions encoding a controller  $C_q$ .

### 8.2.3 Boolean Functions as Control Laws

Boolean functions are functions that accept boolean variables, apply AND, OR, NOT, or XOR operations to them, and finally generate either TRUE (1) or FALSE (0) as outputs. If a control law is to be encoded as Boolean functions, the states and inputs of the symbolic model are first encoded (e.g., BDC) as  $N_X$  and  $N_U$  binary variables, respectively. A binary variable for a symbolic state is denoted by  $x_{bi}$ ,  $i \in \{0, 1, \dots, N_X\}$ , and a binary variable for an input is denoted by  $u_{bj}$ ,  $j \in \{0, 1, \dots, N_U\}$ . The control law must then be determined in the sense that there exists only one control input  $u$  for each state  $x_q$ . More formally, in a determinized control law, the following holds for each  $x_q \in X_q$ :

$$|\{u \in U_q \mid u \in C_q(x_q)\}| \leq 1.$$

A determinized control law can still enforce the original specifications. It is however more conservative in the sense that there are no possible optimizations that can be done on the behaviors of original systems by choosing from different admissible control inputs. Finally, one constructs for each binary variable  $u_{bi}$  one Boolean function  $F^{u_{bi}}(x_{b0}, x_{b1}, \dots, x_{b(N_X-1)})$ . Function  $F^{u_{bi}}$  encodes the part of control law corresponding to control input  $u_{bi}$ . Each Boolean function can be simply implemented in Hardware as a sequence of logic gates (e.g., AND, OR and NOT gates) or as software that emulate the gates using logic operations. Figure 8.5 shows an example for a Boolean circuit of  $N_U$  Boolean functions encoding a controller  $C_q$ .

During runtime of the controller implementation, the BDC encoded value of  $x_i$  (i.e.,  $x_{b(N_X-1)} \dots x_{b1} x_{b0}$ ) is possibly supplied to all functions  $F^{u_{bi}}$ ,  $i \in \{0, 1, \dots, N_U - 1\}$ . Each function returns 1 or 0. The values are collected and concatenated to represent a BDC-encoded value of the index of  $u_j$  the controller should return.

### 8.3 Formal Implementations of Dynamic Symbolic Controllers

The space complexity of one Boolean function  $F_{x_u}$  in a naive non-reduced Sum-of-Product (a.k.a. Disjunctive Normal Form (DNF)) implementation is  $O(N_X \times 2^{N_X})$  gates. To implement the control law, we need to have  $N_U$  parallel circuits (or one combinational circuit with  $N_U$  outputs), where each output correspond to one bit  $x_u$  of the control input.

Time complexity of the circuit depends mainly on the implementation. If the Boolean functions are implemented as combinational logic circuits (e.g., in an FPGA), the time complexity is  $O(1)$ , assuming the implementation has the compute-capacity to propagate the input signals to the terminals of the circuit within one clock cycle. If the Boolean functions are emulated in software, each Boolean function is of a time complexity of  $O(N_X \times 2^{N_X})$  logic computation steps (i.e., evaluating AND, OR, or NOT operations) using the same naive non-reduced DNF. An advantage of using Boolean functions is that they can be translated automatically (and still formally-correct) from the representation generated by the controller synthesis tool to a Hardware Description Language (HDL) (e.g., Verilog or VHDL).

#### 8.2.3.1 Example Hardware Implementation

A symbolic controller can be represented as a multi-output combinational circuit (each output corresponds to one Boolean function  $F_{u_{bi}}$ ) as depicted by Fig. 8.5. Then, it is attached as a controller to the closed loop system.

#### 8.2.3.2 Example Software Implementation

A symbolic controller can be represented as a software emulating the computation of Boolean functions. The functions can be computed in parallel.

## 8.3 Formal Implementations of Dynamic Symbolic Controllers

Dynamic symbolic controllers are inherently Finite Labeled Transition System (FLTS) with outputs. They have internal states, accept inputs (i.e., the quantized states of original systems), change their internal states accordingly, and generate outputs (i.e., the control inputs). One formal implementation of dynamic symbolic controllers is to use Machines (e.g., Mealy or Moore machines). We consider Mealy machines as concrete examples since they are generated from kernel `OmegaThreads`.

To implement the control law as a Mealy machine, we represent it as a transition along with one memory register to represent its current state. Each row in the table correspond to one state from  $M$  different machine states. Within each row  $i$ , the transitions originated from machine state  $m_i$  are stored. Each transition contains the following information: (1) one expected input state (a state  $x_q$  of the symbolic model), (2) next state  $m'$  of the machine, which is considered if this transition is taken, and (3) a set of possible control inputs  $u$  the controller can generate if this transition is taken. The machine starts at some initial state  $m_0$  and waits for a system state  $x_q$ . Then, once  $x_q$  is received, the machine picks the one transition, that has  $x_q$  as expected input state,

from those transitions in the row of state  $m_0$ , updates the current machine state with the next state in the selected transition, and applies one of the control inputs  $u$ . This process is repeated infinitely, assuming the controlled system is not blocking (i.e., will always generate some state  $x_q$ ).

The space complexity of the Mealy machine is clearly  $O(M|X_q||U_q|)$ . The time complexity for extracting the control inputs is  $O(|X_q|)$  assuming the transition table has a fixed width.

## 8.4 Summary

The implementations of symbolic controllers for safety-critical CPS should be highly predictable with respect to both time and memory requirements. Available state-of-the-art tools for symbolic controller synthesis do not provide a formal approach to implementing the designed controllers. Consequently, the implementations of the controllers are usually done using ad-hoc techniques, which result in final products (i.e., SCCS) that do not have any correctness guarantees obtained from the symbolic controller synthesis tools.

We discussed the types of symbolic controllers resulting from the algorithms and software tools introduced in the previous chapters. We also introduced formal implementations for both static and dynamic controllers.

Formal implementations of static symbolic controllers are first introduced. All implementations were introduced along with their time and space complexity. LUTs offer great access time to the stored controllers. They suffer however from their large sizes. BDD-based implementations solve the problem of controller size. They require however more complex implementations. Boolean functions offer the most efficient implementation with respect to computation time since the controllers are implemented as hardware circuits. They require however determinizing the symbolic controllers.

Formal implementations of dynamic symbolic controllers were also discussed. We considered Mealy machines as concrete examples since they are generated from the kernel `OmegaThreads`. To implement the control law as a Mealy machine, we represented it as a transition along with one memory register to represent its current state.

The introduced implementations along with the techniques of symbolic control from previous chapters form together an end-to-end framework for symbolic control.

## 9 Conclusions and Future Works

In this chapter, we review the results introduced in all previous chapters. We also discuss and suggest future works to extend these results.

### 9.1 Conclusions

In this thesis, we argued in Chapter 1 that SCCS are currently designed using approaches that may result into unsafe products. Unfortunately, in current development cycles of SCCS, the design requirements are not defined in a formal way which may result in false and/or ambiguous requirements. The software design/development phases of SCCS involve many human factors which results in faulty and buggy software. Testing phases cannot cover all possible test scenarios and hence, many edge-cases are left undetected, which should be unacceptable in any safety-critical product. Since any failure in any life-critical SCCS can potentially cause death or injury to humans, ensuring the correctness of SCCS is very important and new design approaches need to be investigated.

In Chapter 2, we introduced symbolic control as a promising approach for designing, automatically, correct-by-construction SCCS. Given models of systems and formally-described design requirements, symbolic control techniques design algorithmically certifiable controllers that can enforce the design requirements on the original systems. The models are used to construct finite-state abstractions that capture important features of original models. Using approaches from computer science like search on graphs and fixed-point operations, formally-correct controllers are designed algorithmically to enforce the design requirements. Unfortunately, symbolic control is not applicable to today's real-world CPSs due to some major issues (see Section 2.5 for details):

- (1) Symbolic control suffers from the CoD problem in all of its phases: the construction of abstractions in form of symbolic models and the synthesis of controllers. This makes it limited to small-sized systems.
- (2) Existing tools of symbolic control can only deal with simple specifications like reachability and safety. This is mainly because handling complex specifications require complex algorithms of controller synthesis that make symbolic control highly impractical, even for simple case studies.
- (3) Symbolic control lacks a standard and unified approach for extracting the designed controllers and generating their deployments. This leaves such important task to ad-hoc techniques that ruin the formality and correctness-guarantees obtained from symbolic control.

## 9 Conclusions and Future Works

In Chapters 3-8, we introduced solutions to the problems of symbolic control to make it applicable to designing real-world SCCS.

In Chapter 3, we took a detour before starting to address the issues in symbolic control. The detour was necessary to lay a ground for the algorithms and techniques that we introduced in the next chapters, which all required a unified framework for their design and implementations. We introduced **pFaces**, a framework for designing and implementing parallel algorithms for symbolic control. **pFaces** leverages all available computing resources, locally or in Cloud-computing platforms, to facilitate designing and implementing certifiable control software. Our plan was to parallelize the algorithms of symbolic control so that they scale with available computing resources, which in return, give us the ability to control their computational complexity. As a result, symbolic control would have a change against the CoD problem, be able to support wider ranges of specifications, and have a unified approach for implementing their final products; the symbolic controllers.

In Chapter 4, we addressed the first issue in symbolic control. Two main concepts were utilized to provide enhancements to symbolic control algorithms:

- data-parallel algorithms, along with variable computing resources, allow controlling the time complexity of data-parallel tasks in PRAM machines, and
- sparsity and locality of control systems lead to less computational effort for constructing their symbolic models and synthesizing their controllers.

Traditional symbolic control techniques, which suffered significantly from the CoD, were redesigned as data-parallel algorithms that scale with number of available PEs. We introduced algorithms to construct symbolic models and synthesize their symbolic controllers in parallel (see Section 4.2). Then, the algorithms were enhanced to utilize the sparsity of dynamics in original systems (see Section 4.3). With several examples throughout the chapter, we showed that parallelized sparsity-aware symbolic control over-performs traditional implementations, even using the same serial computing platform. Implemented on top of **pFaces**, the algorithms introduced in Chapter 4 can utilize all computing platforms available locally or in Cloud-computing platforms.

In Chapters 5 and 6, we introduced two additional enhancements to symbolic control. Chapter 5 introduced a parallelizations of one reachability analysis approach that we use internally to construct the symbolic models (see map  $\Omega^f$  in Section 4.2). Using such additional parallelization, we were able to compute reachable sets for nonlinear systems faster and at higher dimensions. Note that map  $\Omega^f$  can be used independently in other approaches of formal methods in control like, for example, using reachability analysis for the behavioral verification of dynamical and control systems. Chapter 6 introduced enhancements to traditional techniques of stochastic symbolic control. Efficient data-parallel algorithms for automated controller synthesis of symbolic controllers for stochastic control systems were introduced. More specifically, one data-parallel algorithm was introduced to construct MDPs as symbolic models of dt-SCSs. Additionally, a data-parallel algorithm for automated synthesis of controllers based on the constructed MDPs was also introduced. The algorithms from both enhancements were implemented

on top of **pFaces**. Using several examples, we showed how superior the implementations are compared to all existing software tools, and that they can handle very complex case studies.

Chapter 7 handled the second issue in symbolic control, that is, the lack of support for practical classes of formal specifications, which all existing tools of symbolic control suffer from. The chapter introduced a technique for symbolic control that can handle  $\omega$ -regular specifications. It constructs parity games from symbolic models and DPAs describing high-level objectives. After solving the parity games, dynamic symbolic controllers are generated as Mealy machines. With some examples (e.g., see the example of a drone operating on battery in Section 7.5), we showed that this enhancement along with the ones in previous chapters allows symbolic control to handle complex practical case studies. The ability to generate controllers as Mealy machines provide a base for unifying the deployment phases of dynamic symbolic controllers, a task that was previously done using ad-hoc techniques and has negatively affected the formality and correctness-guarantees obtained from symbolic control.

Chapter 8 discussed the types of symbolic controllers resulting from the algorithms introduced in the previous chapters. It also introduced formal deployments for both static and dynamic controllers. Several deployments schemes were discussed like for example, LUTs, BDDs, and Boolean functions. Each scheme was discussed formally with respect to the time and space complexities of its deployment. The ideas introduced in Chapter 8 along with those enhancements from previous chapters form together an end-to-end framework for symbolic control.

## 9.2 Strengths, Weaknesses and Limitations

We discussed and addressed the issues of symbolic control hindering its applications to modern CPSs. By providing scalable algorithms for symbolic control in Chapters 3-6, we allow controlling the time complexity of symbolic control. Given that the number of PEs used to solve the problem is a control variable, the parallel algorithms introduced in these chapters can be used to reduce the complexity of the problem. Hence, symbolic control is now more applicable to more complex case studies. The work in Chapter 7 allows symbolic control to handle complex high-level specifications and produce standardized symbolic controllers as Mealy machines. With the introduction of formal implementations of symbolic controllers in Chapter 8, we presented a complete end-to-end framework that automates the design cycle of SCCS (see the initial design cycle of SCCS in Fig. 1.4). Requirements are now formally provided. The design/development phase is now automated. The deployments are no longer ad-hoc, but rather standardized and automatically generated.

As there is no work without weaknesses or limitations, we address here the limitations of the techniques introduced throughout the previous chapters. A general limitation of this work is the lack of comparison with traditional (classical) control system approaches. Although we revealed earlier in Chapter 1 the limitations of symbolic control compared to classical control approaches (see Section 2.5), we did not provide such a comparison

for the results obtained after resolving such limitations. The main justification for such limitation is that, this work mainly focuses on accelerating symbolic control as an approach that promises to provide formally-correct SCCS for safety-critical CPS. Hence, all the results are compared to traditional implementations of symbolic control. In the following, we discuss limitations for some approaches introduced in the thesis.

- **The parallel algorithms introduced in Chapter 4 and Chapter 6:** the algorithms handle the memory used to store and operate on the symbolic models as raw memory. For complex and high-dimensional systems, the required memory space becomes very large and may hinder the implementation of the algorithms on embedded and hardware devices with limited memory spaces. The algorithms are also limited to safety, reachability, and reach-avoid specifications.
- **The parallel algorithms for controller synthesis in Chapter 4 and Chapter 6:** although the algorithms serve as data-parallel algorithms, they suffer some computational bottlenecks due to the synchronization between the parallel threads by the end of each fixed-point iteration. The effects of such synchronization is minimal when the PEs are within one CU. Having many PEs within a heterogenous HWC will probably result in noticeable overhead since the synchronization will happen using the shared memories or communication networks between the CUs/CNs.
- **The sparsity-aware approach introduced in chapter 4:** the approach requires representing the original system as a discrete-time system so that a sparsity graph can be concluded. Discretizing continuous-time systems introduces additional approximation error that makes the solution (i.e., the synthesized controllers) more conservative and may result in failure to find a solution to the symbolic control problem.
- **The parallel algorithms for computing the OARS in Chapter 5:** as the parallelization of the ODE solver is done over the space of the differential equations, the approach becomes more useful for very high-dimensional systems. Applying the approaches of this chapter to small-sized systems and using HPC systems with thousands of PEs would be a waste of computational resources.
- **The technique introduced in Chapter 7:** the algorithm used to construct the parity game is a serial algorithm and needs to be parallelized. This is a bottleneck in the approach. Nevertheless, the abstraction construction and solving parity games are both done in parallel.

### 9.3 Recommendations for Future Works

The first recommendation to push the works in this thesis forward is to handle the limitations introduced in the previous subsection. We also propose some interesting subjects that could be considered as future research lines and improvements to the results introduced in this thesis.

### 9.3.1 Memory-efficient Data-parallelism for Symbolic Control

The data-parallel algorithms introduced in Chapters 4-6, although allow controlling the time complexity of symbolic control algorithms, they have the drawback of significant memory requirements. Traditional implementation of construction of symbolic models (as introduced in Algorithm 3) usually relies on efficient data structures that reduce the large sizes of symbolic models. For example, tools like **SCOTS** and **Pessoa** use BDDs to store the constructed symbolic models, which results in reductions of space requirements. Unfortunately, using such data structures within the introduced parallel algorithms is not practical. It will reduce the algorithm's ability to scale with available computing resources. The main reason is that, BDDs for example, are centralized data structure in the sense that adding or removing elements from the set represented by a BDD is a centralized operation, which will introduce a significant synchronization overhead when multiple threads try to access the data structure simultaneously.

In this context, we need to find memory-efficient data structures that have minimal multi-threaded data-access overhead. An ideal data structure, that parallel symbolic control would definitely benefit from, is a distributed wait-free lock-free data structure. For a data structure to be considered as lock-free, multiple threads must be able to concurrently and safely access the data structure. The threads don't have to be able to do the same operations. A lock-free data structure might then allow one thread to add a transition of the symbolic model and another thread to read the details of another transition. It should not allow two threads try to write to the same transition location at the same time. A wait-free data structure on the other hand is a lock-free data structure with the additional property that every thread accessing the data structure can complete its operation within a bounded number of steps, regardless of the behavior of other threads.

A future research line here would then focus on reviewing all applicable distributed wait/lock-free data structures and discussing their advantages and disadvantages. Then, candidate data structures are implemented as enhancement to the introduced data-parallel algorithms and their results are compared. A trade-off should eventually be made between the communication overhead resulting from introducing the data structures and the savings in memory they can offer.

### 9.3.2 User-Friendly Cloud-Application for Designing and Implementing Parallel Algorithms

The introduced acceleration ecosystem **pFaces**, as presented in Chapter 2, allows its users to run parallel algorithms on all available computing platforms locally or on Cloud-computing platforms. Currently, the followings are required by the users in order to use **pFaces** with computing resources in a Cloud-computing platform such as AWS:

- (1) create an account with the Cloud-computing provider and activate it with some payment method (e.g., a credit card),
- (2) choose what machines on the Cloud to launch,

## 9 Conclusions and Future Works

- (3) launch the machines and install the required Operating System (OS)/software on them,
- (4) configure the interconnection between multiple machines in case a computing cluster should be established,
- (5) install **pFaces** on all instances,
- (6) install **pFaces – Agent**, a software that orchestrates the work between different **pFaces** installations and provides remote access to each of the machines in the cluster, and
- (7) run the web-interface of **pFaces** and connect to the computing cluster.

This process is needed whether users need to design data-parallel algorithms or run kernels of **pFaces**. Clearly, this is a burden for a typical user or a developer who only wants to test the capabilities of **pFaces**.

A future work in this direction would focus on providing a user-friendly web-service to facilitate working with **pFaces**. Like any Software as a Service (SaaS) framework, the expected features of the web-service are:

- Users should be able to pick the computing resources in well-explained and easy-to-use way and don't care about the technical steps needed to make them ready for work,
- The service should automate the process of launching the machines on the Cloud, installing the required software, and configuring the interconnection between them,
- Users should have access to a professional web-based IDE that allows them to write their parallel algorithms and test them with one push-button on any selected HWC, and
- Users should be able to collect profiling information from the remotely-deployed machines and possibly visualize the collected profiling data.

### 9.3.3 Supporting Continuous-time Stochastic Control Systems

The enhancements introduced in Chapter 6 allow constructing abstractions of dt-SCSs. To consider continuous-time systems, we proposed discretizing them using Euler method (see Remark 6.1.2). This however introduces additional error that we include in the abstraction as a disturbance. If this error is big enough or the system itself has additional disturbances, finding a controller to enforce the given specifications may not be possible due to the high nondeterminism in the transitions of the abstraction. A future research line here is to investigate different mathematical tools for handling continuous-time systems.

### 9.3.4 Supporting Output-based Control Systems

All the techniques introduced in this thesis consider only state-based control systems, i.e., those control systems with full-state information (see Definition 2.2.1). Control systems are more practically modeled as systems with partial-state or output information. We refer to these particular types of control systems as output-based systems.

Future work in this direction should first consider extending the results in Chapter 2 to provide mathematical tools for constructing symbolic models of output-based systems. More precisely, new relations between symbolic models and output-based systems need to be considered. Different methodologies for constructing the symbolic models and synthesizing their controllers should be also investigated. Finally, using the ideas introduced in Chapters 4-7, new data-parallel algorithms for the construction of enhanced symbolic models and the synthesis of their controllers need to be developed.

### 9.3.5 Stochastic Parity Games and Counter-strategy Analysis

The enhancements introduced in Chapter 7 allow handling complex specification by representing the requirement formally as DPAs. The DPAs are used along with symbolic models to construct deterministic parity games. We only introduced the construction of parity games from those symbolic models abstracting nonstochastic continuous-time systems. Notice that in Chapter 6, we introduced MDPs as abstractions of dt-SCS, but we were only able to use them to synthesize controllers enforcing simple specifications like reachability and safety.

A future work would consider constructing MDPs as abstractions of stochastic control systems and use them, combined with the DPAs representing the specifications, to construct stochastic parity games. Here, one would also consider researching which types of strategies (e.g., almost-sure winning or optimal strategy) are convenient and practically implementable. The enhancement would result into new, possibly data-parallel, algorithms that will be amended to the ones in kernel `OmegaThreads`. Another future work would consider the generation of counter strategies and use them to automatically identify possible anomalies in model descriptions or to pinpoint any false requirements.



# Bibliography

- [1518] ABC 15. Abc 15: Self-driving uber car hits, kills pedestrian in tempe. <https://bit.ly/3zByn5i>, 2018.
- [ADLB14] E. Aydin Gol, X. Ding, M. Lazar, and C. Belta. Finite bisimulations for switched linear systems. *IEEE Transactions on Automatic Control*, 59(12):3122–3134, 2014.
- [Alt15] M. Althoff. An introduction to CORA 2015. In *Proceedings of the Workshop on Applied Verification for Continuous and Hybrid Systems*, 2015.
- [Alt17] M. Althoff. Commonroad: Vehicle models. *Technische Universität München, Garching*, pages 1–25, 2017.
- [AM18] M. Arcak and J. Maidens. Simulation-based reachability analysis for nonlinear systems using componentwise contraction properties. In *Principles of Modeling*, pages 61–76. Springer, 2018.
- [BBF<sup>+</sup>12] A. Bohy, V. Bruyère, E. Filiot, N. Jin, and J. Raskin. Acacia+, a tool for ltl synthesis. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, pages 652–657, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [BCJ18] R. Bloem, K. Chatterjee, and B. Jobstmann. *Graph Games and Reactive Synthesis*, pages 921–962. Springer International Publishing, Cham, 2018.
- [Bea08] R. Beard. Quadrotor dynamics and control rev 0.1. Technical report, Brigham Young University, 2008.
- [BH06] C. Belta and L. C. G. J. M. Habets. Controlling a class of nonlinear systems on rectangles. *IEEE Transactions on Automatic Control*, 51(11):1749–1759, 2006.
- [BJP<sup>+</sup>12] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa’ar. Synthesis of reactive(1) designs. *Journal of Computer and System Sciences*, 78(3):911 – 938, 2012. In Commemoration of Amir Pnueli.
- [BK08] C. Baier and J. P. Katoen. *Principles of model checking*. The MIT Press, April 2008.

## BIBLIOGRAPHY

- [BPB19] A. Borri, G. Pola, and M. D. D. Benedetto. Design of symbolic controllers for networked control systems. *IEEE Trans. Autom. Control*, 64(3):1034–1046, 2019.
- [BS96] D. P. Bertsekas and S. E. Shreve. *Stochastic Optimal Control: The Discrete-Time Case*. Athena Scientific, 1996.
- [BTJ19] S. Bak, H.-D. Tran, and T. T. Johnson. Numerical verification of affine systems with up to a billion dimensions. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 23–32. ACM, 2019.
- [BYG17] C. Belta, B. Yordanov, and E. Gol. *Formal Methods for Discrete-Time Dynamical Systems*. Springer, 2017.
- [CA15] S. Coogan and M. Arcak. A compartmental model for traffic networks and its dynamical behavior. *IEEE Transactions on Automatic Control*, 60(10):2698–2703, 2015.
- [CA17] S. Coogan and M. Arcak. Finite abstraction of mixed monotone systems with discrete and continuous inputs. *Nonlinear Analysis: Hybrid Systems*, 23:254 – 271, 2017.
- [CA18] S. Coogan and M. Arcak. A benchmark problem in transportation networks. *arXiv preprint arXiv:1803.00367*, 2018.
- [CAB17] S. Coogan, M. Arcak, and C. Belta. Formal methods for control of traffic flow: Automated control synthesis from finite-state transition models. *IEEE Control Systems Magazine*, 37(2):109–128, 2017.
- [CÁS13] X. Chen, E. Ábrahám, and S. Sankaranarayanan. Flow\*: An analyzer for non-linear hybrid systems. In *International Conference on Computer Aided Verification*, pages 258–263. Springer, 2013.
- [CDA19] N. Cauchi, K. Degiorgio, and A. Abate. StocHy: Automated verification and synthesis of stochastic processes, to appear. In *TACAS’19*, Lecture Notes in Computer Science. Springer, 2019.
- [CDD<sup>+</sup>13] A. Champion, R. Delmas, M. Dierkes, P. Garoche, R. Jobredeaux, and P. Roux. Formal methods for the analysis of critical control systems models: Combining non-linear and linear analyses. In C. Pecheur and M. Dierkes, editors, *Formal Methods for Industrial Critical Systems*, pages 1–16, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- [CGG11] J. Camara, A. Girard, and G. Gössler. Safety controller synthesis for switched systems using multi-scale symbolic models. In *Proceedings of the 50th IEEE Conference on Decision and Control and European Control Conference*, pages 520–525, 2011.

- [Cha18] S. Chapra. *Applied Numerical Methods with MATLAB for Engineers and Scientists*. Mc Graw Hill, 2018.
- [DA20] A. Devonport and M. Arcak. Data-driven reachable set computation using adaptive gaussian process classification and monte carlo methods. In *2020 American Control Conference (ACC)*, pages 2629–2634, 2020.
- [DKAZ20] A. Devonport, M. Khaled, M. Arcak, and M. Zamani. Pirk: Scalable interval reachability analysis for high-dimensional nonlinear systems. In *Proc. 32nd International Conference on Computer Aided Verification (CAV)*, LNCS. Springer, 2020.
- [DMVP15] P. S. Duggirala, S. Mitra, M. Viswanathan, and M. Potok. C2e2: A verification tool for stateflow models. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 68–82. Springer, 2015.
- [Dre17] T. Dreossi. SAPO: Reachability computation and parameter synthesis of polynomial dynamical systems. In *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control*, pages 29–34. ACM, 2017.
- [EC82] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241 – 266, 1982.
- [EGW02] W. Thomas E. Graedel and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*. Springer Heidelberg, 2002.
- [EKS18] J. Esparza, J. Křetínský, and S. Sickert. One theorem to rule them all: A unified translation of ltl into  $\omega$ -automata. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '18*, page 384–393, New York, NY, USA, 2018. Association for Computing Machinery.
- [ER16] R. Ehlers and V. Raman. Slugs: Extensible gr(1) synthesis. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 333–339, Cham, 2016. Springer International Publishing.
- [Esm14] S. Esmail Zadeh Soudjani. *Formal Abstractions for Automated Verification and Synthesis of Stochastic Systems*. PhD thesis, Technische Universiteit Delft, The Netherlands, 2014.
- [FKJM16] C. Fan, J. Kapinski, X. Jin, and S. Mitra. Locally optimal reach set over-approximation for nonlinear systems. In *2016 International Conference on Embedded Software (EMSOFT)*, pages 1–10. IEEE, 2016.

## BIBLIOGRAPHY

- [FLGD<sup>+</sup>11] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. Spaceex: Scalable verification of hybrid systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV)*, LNCS. Springer, 2011.
- [FOP<sup>+</sup>19] N. Fijalkow, J. Ouaknine, A. Pouly, J. Sousa-Pinto, and J. Worrell. On the decidability of reachability in linear time-invariant systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 77–86. ACM, 2019.
- [GGM16] A. Girard, G. Gössler, and S. Mouelhi. Safety controller synthesis for incrementally stable switched systems using multiscale symbolic models. *IEEE Transactions on Automatic Control*, 61(6):1537–1549, 2016.
- [GH94] J.-L. Gouzé and K. Haderer. Monotone flows and order intervals. *Non-linear World*, 1:23–34, 1994.
- [Gir14] A. Girard. Approximately bisimilar abstractions of incrementally stable finite or infinite dimensional systems. In *53rd IEEE Conference on Decision and Control*, pages 824–829, Dec 2014.
- [GKA17] F. Gruber, E. S. Kim, and M. Arcak. Sparsity-aware finite abstraction. In *Proceedings of 56th IEEE Annual Conference on Decision and Control (CDC)*, pages 2366–2371, USA, Dec 2017. IEEE.
- [GLPN93] G. Gallo, G. Longo, S. Pallottino, and S. Nguyen. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42(2):177 – 201, 1993.
- [GP11] A. Girard and G.J. Pappas. Approximate bisimulation: A bridge between computer science and control theory. *European Journal of Control*, 17(5):568 – 578, 2011.
- [GPT10] A. Girard, G. Pola, and P. Tabuada. Approximately bisimilar symbolic models for incrementally stable switched systems. *IEEE Transactions on Automatic Control*, 55(1):116–126, 2010.
- [Gro19] The CAPD Group. Computer assisted proofs in dynamics group, a c++ package for rigorous numerics. <http://capd.ii.uj.edu.pl/>, 2019.
- [HM12] Z. Huang and S. Mitra. Computing bounded reach sets from sampled simulation traces. In *Proceedings of the 15th ACM international conference on Hybrid Systems: Computation and Control*, pages 291–294. ACM, 2012.
- [HMMS18a] K. Hsu, R. Majumdar, K. Mallik, and A. Schmuck. Lazy abstraction-based control for safety specifications. In *2018 IEEE Conference on Decision and Control (CDC)*, pages 4902–4907, 2018.

- [HMMS18b] K. Hsu, R. Majumdar, K. Mallik, and A. K. Schmuck. Multi-layered abstraction-based controller synthesis for continuous-time systems. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (Part of CPS Week)*, HSCC '18, pages 120–129, New York, NY, USA, 2018. ACM.
- [HSA17] S. Haesaert, S. Soudjani, and A. Abate. Verification of general Markov decision processes by approximate similarity relations and policy refinement. *SIAM Journal on Control and Optimization*, 55(4):2333–2367, 2017.
- [IAC+18] F. Immler, M. Althoff, X. Chen, C. Fan, G. Frehse, N. Kochdumper, Y. Li, S. Mitra, M. S. Tomar, and M. Zamani. Arch-comp18 category report: Continuous and hybrid systems with nonlinear dynamics. In *Proceedings of the 5th International Workshop on Applied Verification for Continuous and Hybrid Systems*, 2018.
- [Jaj92] J. Jaja. *An Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.
- [JP09] A. A. Julius and G. J. Pappas. Trajectory based verification using local finite-time invariance. In *International Workshop on Hybrid Systems: Computation and Control*, pages 223–236. Springer, 2009.
- [KG19] Z. Kader and A. Girard. Symbolic models for incrementally stable singularly perturbed hybrid affine systems. In *2019 American Control Conference (ACC)*, pages 3002–3007, 2019.
- [KKAZ19] M. Khaled, E. S. Kim, M. Arcak, and M. Zamani. Synthesis of symbolic controllers: A parallelized and sparsity-aware approach. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 265–281, Cham, 2019. Springer International Publishing.
- [KMS18] J. Kretínský, T. Meggendorfer, and S. Sickert. Owl: A library for  $\omega$ -words, automata, and ltl. In *16th International on Automated Technology for Verification and Analysis, ATVA 2018*, October 7-10 2018.
- [KRZ16] M. Khaled, M. Rungger, and M. Zamani. Symbolic models of networked control systems: A feedback refinement relation approach. In *54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 187–193, Sept 2016.
- [KRZ18] M. Khaled, M. Rungger, and M. Zamani. SENSE: Abstraction-based synthesis of networked control systems. In *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, 272, pages 65–78, 111 Cooper Street, Waterloo, Australia, June 2018. Open Publishing Association (OPA).

## BIBLIOGRAPHY

- [KV06] A. A. Kurzhanskiy and P. Varaiya. Ellipsoidal toolbox (et). In *Proceedings of the 45th IEEE Conference on Decision and Control*, pages 1498–1503. IEEE, 2006.
- [KZ19] M. Khaled and M. Zamani. pfaces: An acceleration ecosystem for symbolic control. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '19, New York, NY, USA, 2019. ACM.
- [KZ21] M. Khaled and M. Zamani. Omegathreads: Symbolic controller design for  $\omega$ -regular objectives. In *Proceedings of the 24th ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '21. ACM, 2021.
- [LCGG13] E.l Le Corronc, A. Girard, and G. Goessler. Mode sequences as symbolic states in abstractions of incrementally stable switched systems. In *Proceedings of the 52th IEEE Conference on Decision and Control*, pages 3225–3230, 2013.
- [LKSZ20] A. Lavaei, M. Khaled, S. Soudjani, and M. Zamani. Amytiss: Parallelized automated controller synthesis for large-scale stochastic systems. In *Proc. 32nd International Conference on Computer Aided Verification (CAV)*, LNCS. Springer, 2020.
- [LL18] Y. Li and J. Liu. Rocs: A robustly complete control synthesis tool for nonlinear dynamical systems. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (Part of CPS Week)*, HSCC '18, pages 130–135, New York, NY, USA, 2018. ACM.
- [LLO15] Y. Li, J. Liu, and N. Ozay. Computing finite abstractions with robustness margins via local reachable set over-approximation. In *the 5th IFAC Conference on Analysis and Design of Hybrid Systems*, pages 1 – 6, 2015.
- [LMS20] M. Luttenberger, P. J. Meyer, and S. Sickert. Practical synthesis of reactive systems from ltl specifications via parity games. *Acta Informatica*, 57(1):3–36, 2020.
- [LSZ18] A. Lavaei, S. Soudjani, and M. Zamani. From dissipativity theory to compositional construction of finite Markov decision processes. In *Proceedings of the 21st ACM International Conference on Hybrid Systems: Computation and Control*, pages 21–30, 2018.
- [Lut08] M. Luttenberger. Strategy iteration using non-deterministic strategies for solving parity games. Technical report, Technische Universität München, Institut für Informatik, April 2008.

- [MA14] J. Maidens and M. Arcak. Reachability analysis of nonlinear systems using matrix measures. *IEEE Transactions on Automatic Control*, 60(1):265–270, 2014.
- [Maj16] R. Majumdar. Robots at the edge of the cloud. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 3–13, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
- [Mat] S. Matteson. Report: Software failure caused \$1.7 trillion in financial losses in 2017. <https://tek.io/3giVDyK>. Accessed: 2021-02-01.
- [MDA19] P.-J. Meyer, A. Devonport, and M. Arcak. Tira: Toolbox for interval reachability analysis. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '19, page 224–229, New York, NY, USA, 2019. Association for Computing Machinery.
- [MDT10] M. Mazo, A. Davitian, and P. Tabuada. Pessoa: A tool for embedded controller synthesis. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 566–569, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- [MGG13] S. Mouelhi, A. Girard, and G. Gössler. Cosyma: A tool for controller synthesis using multi-scale abstractions. In *Proceedings of 16th International Conference on Hybrid Systems: Computation and Control*, HSCC '13, pages 83–88, New York, NY, USA, 2013. ACM.
- [MPS95] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In E. W. Mayr and C. Puech, editors, *12th Annual Symposium on Theoretical Aspects of Computer Science (STACS 95)*, pages 229–242, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
- [MR15] S. Maoz and J. O. Ringert. Gr(1) synthesis for ltl specification patterns. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, page 96–106, New York, NY, USA, 2015. Association for Computing Machinery.
- [MSL18] P. J. Meyer, S. Sickert, and M. Luttenberger. Strix: Explicit reactive synthesis strikes back! In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 578–586, Cham, 2018. Springer International Publishing.
- [Ned11] N. S. Nedialkov. Implementing a rigorous ode solver through literate programming. In *Modeling, Design, and Simulation of Systems with Uncertainties*, pages 3–19. Springer, 2011.

## BIBLIOGRAPHY

- [NVI] NVIDIA. Gpu-accelerated black-hole simulations. <https://bit.ly/3tv0bEe>. Accessed: 2021-02-01.
- [oTotUSHoR19] Committee on Transportation and Infrastructure of the U. S. House of Representatives. Hearing: The boeing 737 max: Examining the design, development, and marketing of the aircraft. Technical report, The U. S. House of Representatives, October 30 2019.
- [PBW06] J. van Benthem P. Blackburn and F. Wolter, editors. *The Handbook of Modal Logic*. Elsevier, 2006.
- [PGT08a] G. Pola, A. Girard, and P. Tabuada. Approximately bisimilar symbolic models for nonlinear control systems. *Automatica*, 44(10):2508 – 2516, 2008.
- [PGT08b] G. Pola, A. Girard, and P. Tabuada. Approximately bisimilar symbolic models for nonlinear control systems. *Automatica*, 44(08):2508 – 2516, 2008.
- [Pnu77] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, 1977.
- [PPB15] G. Pola, P. Pepe, and M.D. Di Benedetto. Symbolic models for time-varying time-delay systems via alternating approximate bisimulation. *Int. J. of Robust and Nonlinear Control*, 25(14):2328–2347, 2015.
- [PPBT10] G. Pola, P. Pepe, M. D. [Di Benedetto], and P. Tabuada. Symbolic models for nonlinear time-delay systems using approximate bisimulations. *Systems & Control Letters*, 59(6):365 – 373, 2010.
- [PR89] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proceedings of the 16th International Colloquium on Automata, Languages and Programming, ICALP '89*, pages 652–671, London, UK, 1989. Springer-Verlag.
- [PT09] G. Pola and P. Tabuada. Symbolic models for nonlinear control systems: Alternating approximate bisimulations. *SIAM Journal on Control and Optimization*, 48(2):719–733, 2009.
- [Ray16] P. P. Ray. Internet of robotic things: Concept, technologies, and challenges. *IEEE Access*, 4:9489–9500, 2016.
- [RGD<sup>+</sup>15] R. Ray, A. Gurung, B. Das, E. Bartocci, S. Bogomolov, and R. Grosu. Xspeed: Accelerating reachability analysis on multi-core processors. In N. Piterman, editor, *Hardware and Software: Verification and Testing*, pages 3–18, Cham, 2015. Springer International Publishing.

- [RMT13] M. Rungger, M. Mazo, Jr., and P. Tabuada. Specification-guided controller synthesis for linear systems and safe linear-time temporal logic. In *16th International Conference on Hybrid Systems: Computation and Control*, HSCC '13, pages 333–342, New York, NY, USA, 2013. ACM.
- [RWR17] G. Reißig, A. Weber, and M. Rungger. Feedback refinement relations for the synthesis of symbolic controllers. *IEEE Transactions on Automatic Control*, 62(4):1781–1796, April 2017.
- [RZ16] M. Rungger and M. Zamani. Scots: A tool for the synthesis of symbolic controllers. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*, HSCC '16, pages 99–104, New York, NY, USA, 2016. ACM.
- [SA13] S. Soudjani and A. Abate. Adaptive and sequential gridding procedures for the abstraction and verification of stochastic processes. *SIAM Journal on Applied Dynamical Systems*, 12(2):921–956, 2013.
- [SAM15] S. Soudjani, A. Abate, and R. Majumdar. Dynamic Bayesian networks as formal abstractions of structured stochastic processes. In *Proceedings of the 26th International Conference on Concurrency Theory*, pages 1–14, 2015.
- [SC16] J. A. D. Sandretto and A. Chapoutot. Validated explicit and implicit runge-kutta methods. *Archive Ouverte, HAL*, 2016.
- [SGA15] S. Soudjani, C. Gevaerts, and A. Abate. FAUST<sup>2</sup>: Formal abstractions of uncountable-state stochastic processes. In *TACAS'15*, volume 9035 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2015.
- [SI16] S. I. Solodushkin and I. F. Iumanova. Parallel numerical methods for ordinary differential equations: a survey. In *CEUR Workshop Proceedings*, volume 1729, pages 1–10. CEUR-WS, 2016.
- [Son99] E. D. Sontag. *Mathematical control theory: Deterministic finite dimensional systems*, volume 6 of *Texts in Applied Mathematics*. Springer-Verlag, New York, 2 edition, 1999.
- [Tab08] P. Tabuada. An approximate simulation approach to symbolic control. *IEEE Transactions on Automatic Control*, 53(6):1406–1418, 2008.
- [Tab09] P. Tabuada. *Verification and control of hybrid systems: A symbolic approach*. Springer, USA, 2009.
- [Tho95] W. Thomas. On the synthesis of strategies in infinite games. In Ernst W. Mayr and Claude Puech, editors, *12th Annual Symposium on Theoretical Aspects of Computer Science (STACS 95)*, pages 1–13. Springer Berlin Heidelberg, 1995.

## BIBLIOGRAPHY

- [TI09] Y. Tazaki and J. Imura. Discrete-state abstractions of nonlinear systems using multi-resolution quantizer. In *Proceedings of the International Conference on Hybrid Systems: Computation and Control*, volume 3, pages 351–365, 2009.
- [TK09] P. Zgliczyński T. Kapela. A lohner-type algorithm for control systems and ordinary differential inclusions. *Discrete & Continuous Dynamical Systems - B*, 11(2):365–385, 2009.
- [TP06] P. Tabuada and G. J. Pappas. Linear time logic control of discrete-time linear systems. *IEEE Transactions on Automatic Control*, 51(12):1862–1877, 2006.
- [Val02] G. Valiente. *Algorithms on Trees and Graphs*. Springer, 2002.
- [Var95] M. Y. Vardi. *An automata-theoretic approach to fair realizability and synthesis*, pages 267–278. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.
- [WRR17] A. Weber, M. Rungger, and G. Reissig. Optimized state space grids for abstractions. *IEEE Transactions on Automatic Control*, 62(11):5816–5821, 2017.
- [WTO<sup>+</sup>11] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, and R. M. Murray. Tulip: A software toolbox for receding horizon temporal logic planning. In *14th International Conference on Hybrid Systems: Computation and Control*, HSCC '11, pages 313–314, New York, NY, USA, 2011. ACM.
- [WTS05] Weiwei Li, E. Todorov, and R. E. Skelton. Estimation and control of systems with multiplicative noise via linear matrix inequalities. In *Proceedings of the 2005, American Control Conference, 2005.*, pages 1811–1816 vol. 3, 2005.
- [Yab20] J. Yablonski. *Laws of UX: Using Psychology to Design Better Products & Services*. O'Reilly UK Ltd., Farnham, United Kingdom, 2020.
- [ZA14] M. Zamani and A. Abate. Approximately bisimilar symbolic models for randomly switched stochastic systems. *Systems and Control Letters*, 69:38 – 46, 2014.
- [ZAG15] M. Zamani, A. Abate, and A. Girard. Symbolic models for stochastic switched systems: A discretization and a discretization-free approach. *Automatica*, 55:183 – 196, 2015.
- [ZMAL13] M. Zamani, P. Mohajerin Esfahani, A. Abate, and J. Lygeros. Symbolic models for stochastic control systems without stability assumptions. In *2013 European Control Conference (ECC)*, pages 4257–4262, 2013.

## BIBLIOGRAPHY

- [ZMKA18] M. Zamani, M. Mazo, M. Khaled, and A. Abate. Symbolic abstractions of networked control systems. *IEEE Transactions on Control of Network Systems*, 5(4):1622–1634, 2018.
- [ZMM<sup>+</sup>14] M. Zamani, P. Mohajerin Esfahani, R. Majumdar, A. Abate, and J. Lygeros. Symbolic control of stochastic systems via approximately bisimilar finite abstractions. *IEEE Transactions on Automatic Control*, 59(12):3135–3150, 2014.
- [ZPMT12] M. Zamani, G. Pola, M. Mazo Jr., and P. Tabuada. Symbolic models for nonlinear control systems without stability assumptions. *IEEE Transactions on Automatic Control*, 57(7):1804–1809, July 2012.