Test and Evaluation of Autonomous Systems: Reactive Test Synthesis and Task-Relevant Evaluation of Perception

Thesis by

Apurva Srinivas Badithela

In Partial Fulfillment of the Requirements for the
Degree of
Doctor of Philosophy

CALIFORNIA INSTITUTE OF TECHNOLOGY
Pasadena, California

2024
Defended 21st May, 2024

ACKNOWLEDGEMENTS

I have the deepest respect and gratitude for my advisor, Richard M. Murray. Thank you, for your mentorship, encouragement, and support over the past six years. From you, I have learned the importance of being patient, persistent, and independent, and to think deeply about research directions. Most of all, to be optimistic through the highs and lows, and to handle challenges with courage and preparation. I admire the integrity, magnanimity, and time management that you bring to each of your roles as a professor, a leader, and as an advisor. I have much to learn from you, and you will always inspire me to be a better researcher and colleague. Thank you so very much for giving me the opportunity to learn from you. Finally, I appreciate your efforts to make CDS a welcoming and inclusive environment for all.

I am very grateful to Professor Tichakorn (Nok) Wongpiromsarn for her mentorship over the last several years. Thank you, Nok, for giving me the time and the chance to work with you and learn from you. I am inspired by our research discussions, and by your positive attitude and enthusiasm! I enjoy our research discussions very much, and always leave our meetings feeling more excited. I admire how you always come up with the simplest possible solution to every problem. Thank you for being patient, and for giving me this opportunity.

I would like to thank the members of my committee — Professors Joel W. Burdick, Aaron D. Ames, and K. Mani Chandy. Joel, thank you for taking the time to meet with me and give me feedback on my research. I have always felt welcome to approach you with questions, and I am grateful for our research discussions. I look forward to opportunities to chat about research in the coming years.

Thank you, Dr. Ames, for your valuable feedback over the years on communicating and presenting my research, including your recommendation to use hardware to showcase the full impact of my work. I have found your feedback incredibly useful, especially in writing and conveying my ideas in a more impactful way.

Thank you, Mani, for taking the time to carefully read through my work and for the technically in-depth discussions. I have found your perspectives valuable, and your questions push me to communicate my work in a better way.

I am also very grateful to my undergraduate research advisor, Professor Peter J. Seiler, for taking me under his wing and introducing me to control theory. Pete, I still remember walking into your office and asking if there was any work in your lab that I could help out with. Over the years, your advice has always helped me find my way. Without your support and encouragement, I would have not had these opportunities.

I would also like to thank other wonderful mentors who created such amazing undergraduate research opportunities for me, and encouraged me to apply to graduate school. Especially, I would like to thank Professors Neera Jain and Ufuk Topcu, and Dr. Austin Nash and Dr. Ivan Papusha. I would like to thank Dr. Eric Wolff and Dr. Tung Phan-Minh for a wonderful summer intership at Motional during my PhD.

I would like to thank my colleagues and collaborators, who have shaped this PhD experience. I would like to thank Josefine Graebener for her friendship and collaboration over the years. I fondly remember the countless hours we spent brainstorming at the whiteboard, and the sense of excitement and satisfaction after our understanding of the problem improved vastly after such whiteboard discussions. Her determination, effectiveness, and technical skills have inspired me greatly. I have also had the opportunity to collaborate with Inigo Incer, Wyatt Ubellacker, Denizalp Goktas, and Prithvi Akella. I would also like to thank Emily Fourney, Scott Livingston, Ioannis Filippidis, and Professors Eric Mazumdar, Yisong Yue, Georgia Gkioxari, Chris Umans, Sridhar Duggirala, and Lijun Chen for many insightful discussions that have shaped my research perspectives. I am also fortunate to have had the opportunity to mentor several talented undegraduates, including Ranai Srivastav, Kimia Hassibi, Jacob Alderete, Saeshu Karthika, Frida Moreno, Gerard Decker, Edward Zhang, Andy Dimnaku, and Berlin Del Aguila. Finally, I am grateful to members and alumni of the Murray Lab, including Kellan Moorse, Ioannis Mandralis, Nikos Mynhier, Sumanth Dathathri, and Professor ChuChu Fan. A special thanks to Shih-Hao Tseng and Ugo Rosolia for research discussions, and most especially their encouragement in the early years of my PhD.

I would like to thank, Adam Wierman, for his service as CMS ombudsperson, for his support for the CMS climate survey, and for taking the time to meet with me and share his advice on research. In the effort to make graduate school a more inclusive environment, I would like to thank Lisa Li, Sara Beery, and Natalie Bernat for our collaboration on hosting a department-wide townhall on these issues. One important outcome of that meeting was the establishment of a CMS Ombudsperson, which has helped several graduate students in the years since. I am also deeply grateful to the members of the CMS DEI Committee for their efforts in promoting an inclusive climate. I would especially like to acknowledge Elizabeth Qian and Adam Blank in helping craft the CMS Climate Survey and townhall. Finally, I want to acknowledge Karena Cai for envisioning and leading the Healthy Collaborations workshop, and including me in the process. I would like to thank the admins — Diana, Monica, Carmen, Roberta, Maria, Sydney, and Jolene — for solving many problems for me during my time here, and for the listening ear when I was having a rough time.

During this tough journey, spending time with friends and family has brought much joy and healing. I would like to thank my college friends, Maryam Zahid and Zoë Harvey; even if we have not spoken in months, we pick up like it was yesterday. I enjoyed the sparse, yet lovely and meaningful, visits to Minneapolis: they always fill my cup. Maryam, I still remember that one dinner with your mom, the conversation made it one of the loveliest memories of the last few years. Zoë, I loved visiting your place in St. Paul, our strolls around lakes, visits to coffee shops, listening to you speak about entomology, and most especially meeting Sylvie. I would like to thank my Caltech friends, Jacqueline Tawney and Gianfranco Canales, and I especially remember our bike rides during the pandemic, yoga classes, healing book club discussions, sharing my Ammamma's *avakaya* pickle, and cooking food together. I would like to thank my friend Josefine Graebener for all the morning coffees before we began work, playing with Lumi, and for all the small celebrations as we cheered ourselves through the PhD. I thank the first-year friends, Shreya, Christoph, Daniel, Alex, Matt, and Yousuf, for the laughter, fun conversations, and support in getting through the first year. Thanks to Nico and Jiaqing for being wonderful officemates. I am grateful to Carmen Amo Alonso and Anushri Dixit for their friendship and mentorship over the years. I am grateful to my friend and former neighbor, Samantha Harris, and her lovely dog, Ruthie, for fun adventures around town, and for chiding me when I worked on the weekends or did not take care of myself.

Post-pandemic, I was fortunate to have the chance to form some very meaningful friendships, for which I am deeply grateful. Angela Gao, thank you for being a wonderful friend. I love our hangouts, our conversations, cooking food together, and our adventures going bridal dress shopping. Thanks to you and Zach for all the warm invitations to your home. I am grateful to Andrew Taylor and Noosha Razavian for their warm friendship and meaningful conversations. I have a lot of respect for both of you, and I enjoy your company. I am grateful to Dohyeon Kim and Jiaqing Jiang for their friendly company, be it working out together or grabbing brunch and coffee. I am grateful to art nights, hikes, and hangouts with Lauren Conger, Joudi Hajar, Dohyeon Kim, Jiaqing Jiang. Though I have not been as regular, I would like to thank Ryan Cosner and the Caltech Triathlon Team, for always including me in their weekly workouts and cheering me on. I still look forward to participating in a triathlon one day. I found my love for Carnatic music late in my PhD, and I am grateful to my teacher, Madhavi Gattu, for conducting Zoom classes while being flexible with my schedule. I am also grateful to my Bhagvad Gita teachers, Sravani Jandhyala and Sudha, and the cohort. These experiences have enriched my PhD experience even if I could not fully commit myself.

I would like to thank my extended family for their support over the years. It has been years since I have seen some of you, and I cherish those rare moments of quality time. Many thanks to my cousins, Ruchi, Revanth and Anuhya, Nithin (Sonu), Akhil and Minu, Anil and Sridevi, Avani and Srinivas, Mahathi and Parth, Suchi, Teja, Poojitha, Vasisht, and Anirudh for the fun memories; it has always been a joy to spend time with you which I am very grateful for. I have to especially thank my uncle and aunt, Anil and Bharati Joshi, and my cousins Divya and Kashyap. Anil Mamayya, you encouraged and reassured my parents and I that coming to the US would be a good educational opportunity for me. Thank you all for the many visits to your home in Champaign, especially when I was doing a SURF at Purdue. Thanks to Seetha and Sujit for welcoming me whenever I visited Austin. I am grateful to Arvind and Katyayani Loke for helping me settle in at Caltech, for warmly inviting us to their home, and for the delicious homemade food. Many thanks to my maternal aunts and uncles: Sita Koo, Krishna Vojjala, Rajeshwar Vojjala, and Anita Jalavancha, and their families for embracing me. Many thanks to my paternal uncle, Prasad Badithela, for checking in on me over the years and for his blessings and encouragement. I would also like to thank the Badithela and Kommera families, and cousins for their support when I first came to the US.

I would like to thank Robin Ayling (Mama) and Corey Ayling (Papa) for welcoming and including me in their family. Mama, I am forever grateful for how you make everything work, and Papa, I have always found your advice very helpful. I love our holiday gatherings, the evening walks, the board games, and fun conversation. I am grateful to Alex's siblings, Claire, Lindsey, and John, for warmly accepting me, and for including my sister when she moved to Minneapolis. I am also grateful to Tom, AnnMarie, and Derek for making these family gatherings memorable. I am grateful to Alex's cousin, Sammie, for her warm friendship and for always checking in on me. Thank you to grandma Corinne Ayling for warmly welcoming me, telling me about the Ayling family history as I asked a million questions going over old photos, and for the relaxing visits to Palm Springs. I am grateful for the fun gatherings and laughter-filled conversations with Uncles Harry and Eben, and Auntie Holly and Uncle Mark.

To my dearest *Ammamma*, Prameela Vojjala, you helped raise me and loved me unconditionally, and in being yourself, you taught me to live life on my own terms. Little did I know that I would only see you twice after leaving home for college. It makes me sad that I did not have more time with you, and I miss you every day. I would like to thank the family matriarch, Savitri Badithela, my *Pedda Nainamma*, for always celebrating me and believing in my potential. I remember fondly the visits to your place in Manthani, and the care packages filled with homemade snacks that reminded me of home.

To my dearest little sister, Athreyi: you are so brave and strong, and you inspire me so much. Despite being the youngest, everyone in the family is always amazed at your witty and wise, and at times very funny remarks. You are unafraid to speak the truth, no matter how difficult, and you inspire me to be a better person. I find our phone conversations deeply healing, funny, and exciting, all at the same time.

To my dear parents, Uma Radha and Srinivas Badithela, who have always given me the best of everything. *Amma* and *Nana*, behind every one of your actions, I see your love and blessings for Athreyi, Alex, and me. Ten years ago, I left home to pursue my studies in the US and you have always encouraged and given me moral support from afar. These ten years have passed in the blink of an eye. There were so many times I wished that I could just come home in an instant and spend time with both of you. Every day, I miss you both deeply, and I hope for the future where we live closer and I can come home whenever my heart says so. Your love and support has given me wings to fly.

To my beloved husband, Alex, without your love and support I could not have come so far. No words can do justice to express how I feel. Being with you brings me peace. You bring out the best in me and make me very happy. You have always cheered me through my best and my worst moments. All the small things we did — grabbing coffee, going for walks and bike rides, watching movies, laughing at each others' jokes, or even just simply hanging out — made the tough days in grad school bearable. You have always believed in me, and have always encouraged me to not shy away from having a dream and pursuing it wholeheartedly. From you, I have learned to focus on what is truly important.

ABSTRACT

Autonomous robotic systems have potential for profound impact on our society — legged and wheeled robots for search and rescue missions, drones for wildfire management, self-driving cars for improving mobility, and robotic space missions for exploration and repair of spacecraft. The complexity of these systems implies that formal guarantees during the design phase alone is not sufficient; mainstream deployment of these systems requires principled frameworks for test and evaluation, and verification and validation. This thesis studies two such challenges to mainstream deployment of these systems.

First, we consider the problem of evaluating perception models in a manner relevant to the system-level specification and the downstream planner. Perception and planning modules are often designed under different computational and mathematical paradigms. This talk will focus on evaluating models for classification and detection tasks, and leverages confusion matrices which are popularly used in computer vision to evaluate object detection models to derive probabilistic guarantees at the system-level. However, not all perception errors are equally safety-critical, and traditional confusion matrices account for all objects equally. Thus, task-relevant metrics such as proposition labeled confusion matrices are introduced. These are constructed by identifying propositional formulas relevant to the downstream planning logic and the system-level specification, and result in less conservative system-level guarantees. Using this analysis, fundamental tradeoffs in perception models are reflected in the tradeoffs of probabilistic guarantees. This framework is illustrated on a car-pedestrian example in simulation, and the confusion matrices are constructed from state-of-the-art detection models evaluated on the nuScenes dataset.

Second, we consider the problem of automatically synthesizing tests for autonomous robotic systems. These systems reason over both discrete (e.g., navigate left or right around an obstacle) and continuous variables (e.g., continuous trajectories). This talk presents a flow-based approach for test environment synthesis which handles discrete variables and is also reactive to the system under test. Reactivity is important to account for uncertainties in system modeling, and to adapt to system behavior without knowledge of the system controller. These tests are synthesized from high-level specifications of desired behavior. Though the problem is shown to be NP-hard, a flow-based mixed-integer linear program formulation is used that scales well to medium-sized examples (e.g., >10,000 integer variables). The test environment can consist of static and reactive obstacles as well as dynamic test agents, whose strategies are synthesized to match the solution of the flow-based optimization. The overview of the approach is as follows. First, principles of automata theory are used to translate the high-level system and test objectives, and the non-deterministic abstraction of the system into a network flow optimization. The solution of this optimization is then parsed into GR(1) formulas in linear temporal logic. This GR(1) formula is used to synthesize reactive strategies of a dynamic test agent in a counterexample-guided fashion. We provide guarantees that the synthesized test strategy will realize the desired test behavior under the assumption of a well-designed system, the test strategy is reactive and not overly-restrictive. This framework is illustrated on several simulation and hardware experiments with quadrupeds, showing promise towards a layered approach to test and evaluation.

PUBLISHED CONTENT AND CONTRIBUTIONS

J. B. Graebener*, A. S. Badithela*, D. Goktas, W. Ubellacker, E. V. Mazumdar, A. D. Ames, R. M. Murray (2024). "Flow-Based Synthesis of Reactive Tests for Discrete Decision-Making Systems with Temporal Logic Specifications". arXiv preprint https://arxiv.org/abs/2404.09888 (In submission to Transactions on Robotics).
A. Badithela participated in the conception of the project, theoretical analysis and algorithm design, simulation code development, hardware experiments, and writing of the article. The contents of this paper are presented in Chapter 4.

I. Incer, A. Badithela, J. Graebener, P. Mallozzi, A. Pandey, S.-J. Yu, A. Benveniste, B. Caillaud, R. M. Murray, A. Sangiovanni-Vincentelli, and S. A. Seshia. (2024). "Evaluation Metrics of Object Detection for Quantitative System-Level Analysis of Safety-Critical Autonomous Systems." Conditionally accepted to: *The ACM Transactions on Cyber-Physical Systems (T-CPS).* arXiv preprint: https://arxiv.org/pdf/2303.17751.
A. Badithela led the case study on "Evaluating the end-to-end autonomy stack". For this case study, she participated in its conception, theoretical analysis, simulation code development, and writing of the article. The contents of this paper are presented in Chapter 2.

A. Badithela, T. Wongpiromsarn, R. M. Murray. (2023). "Evaluation Metrics of Object Detection for Quantitative System-Level Analysis of Safety-Critical Autonomous Systems." In: *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 8651–86581. DOI: 10.1109/IROS55552.2023.10342465.
A. Badithela participated in the conception of the project, theoretical analysis and algorithm design, simulation code development, and writing of the article. The contents of this paper are presented in Chapter 2.

A. Badithela*, J. B. Graebener*, W. Ubellacker, E. V. Mazumdar, A. D. Ames, R. M. Murray. (2023). "Synthesizing Reactive Test Environments for Autonomous Systems: Testing Reach-Avoid Specifications with Multi-Commodity Flows." In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 12430–12436. DOI: 10.1109/ICRA48891.2023.10160841.
A. Badithela participated in the conception of the project, theoretical analysis and algorithm design, simulation code development, hardware experiments, and writing of the article. The contents of this paper are presented in Chapter 4.

A. Badithela*, J.B Graebener*, I. Incer* , R. M. Murray. (2023). "Reasoning over Test Specifications Using Assume-Guarantee Contracts." In: *2023 NASA For-*

* denotes equal contribution.

mal Methods (NFM), pp. 278–294. DOI: 10.1007/978-3-031-33170-1_17.
A. Badithela participated in the conception of the project, theoretical analysis and algorithm design, simulation code development, and writing of the article. The contents of this paper are presented in Chapter 5.

J.B Graebener*, A. Badithela*, R. M. Murray. (2022). "Towards Better Test Coverage: Merging Unit Tests for Autonomous Systems." In: *2022 NASA Formal Methods (NFM)*, pp. 133–155. DOI: 10.1007/978-3-031-06773-0_7.
A. Badithela participated in the conception of the project, theoretical analysis and algorithm design, simulation code development, and writing of the article. The contents of this paper are presented in Chapter 5.

A. Badithela, T. Wongpiromsarn, R. M. Murray. (2021). "Leveraging Classification Metrics for Quantitative System-Level Analysis with Temporal Logic Specifications." In: *2021 60th IEEE Conference on Decision and Control (CDC)*, pp. 564–571. DOI: 10.1109/CDC45484.2021.9683611.
A. Badithela participated in the conception of the project, theoretical analysis and algorithm design, simulation code development, and writing of the article. The contents of this paper are presented in Chapter 2.

A. Badithela, R. M. Murray. (2020). "Synthesis of Static Test Environments for Observing Sequence-like Behaviors in Autonomous Systems." arXiv preprint: https://arxiv.org/pdf/2108.05911.
A. Badithela participated in the conception of the project, theoretical analysis and algorithm design, simulation code development, and writing of the article. The contents of this paper are presented in Chapter 3.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

LIST OF ALGORITHMS

Chapter 1

INTRODUCTION

1.1 Motivation

Autonomous robotic systems have the potential for profound impact on our society — legged and wheeled robots for search and rescue missions, drones for wildfire management, self-driving cars for improved mobility, and robotic space missions for exploration and repair of spacecraft. These systems are expected to *correctly reason* about and execute tasks in vast operational environments, including interactions with other agents, both human and autonomous. Furthermore, these systems are incredibly complex: they comprise of several subsystems which are designed under different algorithmic paradigms (e.g., learning-based to model-based) and operate at different timescales and abstractions (e.g., high-level reasoning and decision making to low-level control) to accomplish the different functionalities (e.g., perception, behavior prediction, planning, and control) necessary for correct system-level behavior.

In light of these complexities, formal guarantees of system behavior during the design phase alone is not sufficient; mainstream deployment of these systems requires principled *theoretical* and *algorithmic* frameworks for test and evaluation, and verification and validation, not just to validate software and hardware implementations of the system, but also to complement *formal guarantees* derived during system design. How do we derive a small number of tests that can provide high confidence that the system can operate safely? These operational tests should ideally cover salient features of the operating environment such as disturbance and uncertainty, discrete and continuous inputs, closed-loop behavior of agents in the environment among others. Furthermore, designing and testing systems for guarantees relies on definitions of *correct behavior*, *success*, or *good performance*, which differs for each subsystem and might not be easily identifiable. How do we evaluate subsystems with respect to system-level task requirements?

Driven by these questions, this thesis is focused on testing and evaluating high-level reasoning and decision making algorithms in safety-critical robotic systems. We will draw from fundamentals in control and systems theory, convex and combinatorial optimization, formal methods, and to address challenges in specification, testing, and evaluation of safety-critical autonomous systems.

1.2 Challenges

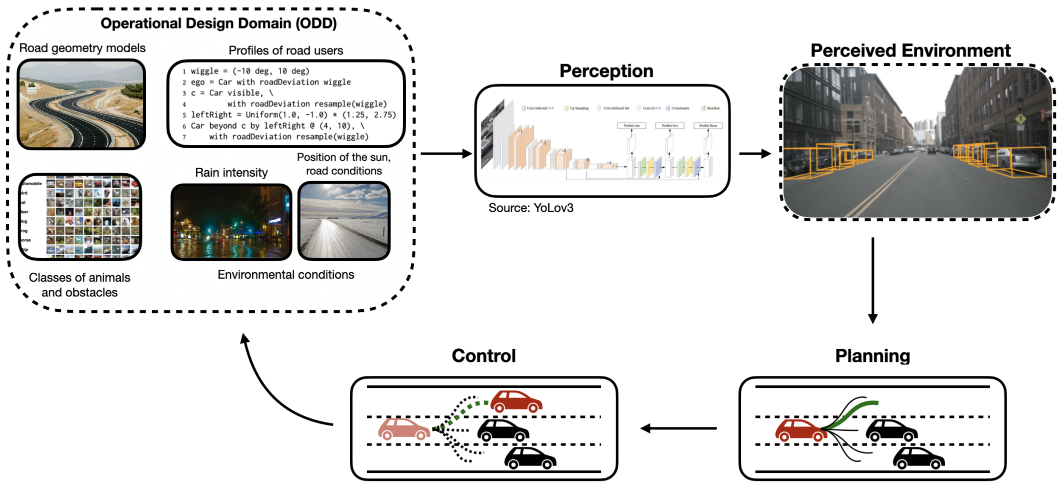
This thesis considers the following challenges in that are currently bottlenecks to safe deployment of complex autonomous systems, especially in safety-critical applications. We will take the example of self-driving to illustrate these challenges due to the richness of the example, but these challenges translate to other robotic applications as well.

Challenge 1: Evaluating Perception Performace with Respect to System-level Requirements

Consider the high-level overview of a classical software stack in a self-driving car as shown in Figure 1.1. Variations of this software stack differ in the neat separation of the perception and planning modules. Typically, the perception and planning modules are developed under different computational paradigms. The backbone of perception models is deep learning, while approaches to planning have traditionally included rulebooks and formal methods, sampling based planners, planning over occupancy grids, and model-based approaches such as model predictive control for mid-level planning. Due to this, these sub-systems are designed differently, often optimizing for different performance metrics. Therefore, it becomes important to establish a *safety case* that accounts for the interaction between perception and planning modules, and its impact on system-level safety. In its document, "A Blueprint for AV Safety: Waymo's Toolkit For Building a Credible Safety Case" [2, 3], Waymo defines a safety case as follows:

"A safety case for fully autonomous operations is a formal way to explain how a company determines that an AV system is safe enough to be deployed on public roads without a human driver, and it includes evidence to support that determination."

In an effort to establish such a safety case, we need to formally and quantitatively reason about how each subsystem contributes to the overall safety of the system. Advancements in perception models is often made along metrics that are not clearly aligned with system-level behavior. Yet, these state-of-the-art models are directly used in robotic systems such as self-driving cars, without standard methods of assessing whether it is indeed suited for the downstream planning and control task. For example, in object detection tasks, *recall* or *sensitivity* is a metric that quantifies how well a model can correctly classify a sample with a certain class label given all



```
wiggle = (-10 deg, 10 deg)
ego = Car with roadDeviation wiggle
c = Car visible, \
        with roadDeviation resample(wiggle)
leftRight = Uniform(1.0, -1.0) * (1.25, 2.75)
Car beyond c by leftRight @ (4, 10), \
    with roadDeviation resample(wiggle)
```

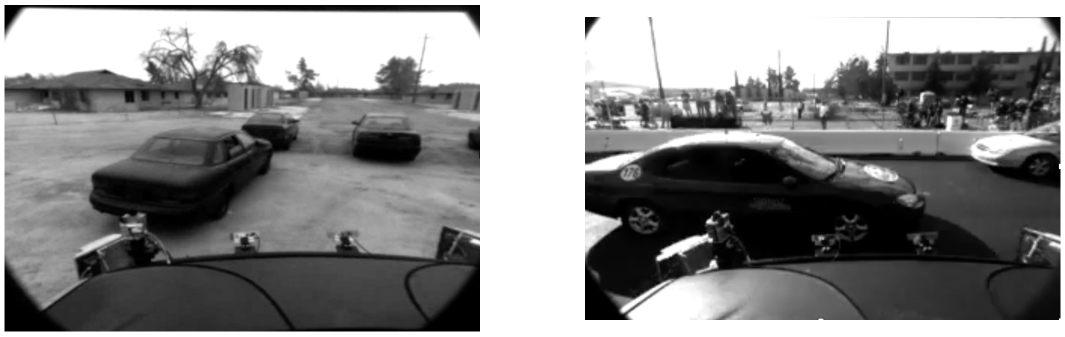
Figure 1.1: Typical software stack in a safety-critical system such as a self-driving vehicle.

relevant samples with that true class label. However, as we will see in this thesis, optimizing models with high *recall* with respect to pedestrians does not necessarily translate to better safety guarantees in all scenarios. Thus, we need new theoretical tools to formalize the interaction of perception errors, including detection and classification errors, localization errors, tracking errors, among others, on downstream planning tasks.

Challenge 2: Test and Evaluation, and Verification and Validation of Safety-Critical Autonomous Systems

For mainstream deployment of safety-critical systems, we need rigorous test and evaluation protocols to certify that autonomous systems comply with certain requirements. Testing can impact the certification process in by guiding regulators and designers to aspects of the design that need more careful evaluation.

Current approaches to safety certification can be broadly categorized as follows. The first category comprises of analysis techniques (e.g., fault tree analysis (FTA) and hazard analysis and risk assessment (HARA)), which cannot scale with the complexity in system design and in operational environments. The second category covers simulation-based testing such as Monte Carlo sampling, simulation-based falsification, and regression testing. These approaches typically sample continuous test parameters, and even if discrete parameters are sampled, they are typically kept fixed for the duration of the test (e.g., color of environment car) as opposed to a discrete test strategy that is reactive to system behavior. The third category involves collecting real-world experimental data (e.g., miles driven without disen-



(a) Static national qualifying test at 2007 Darpa Urban Challenge.

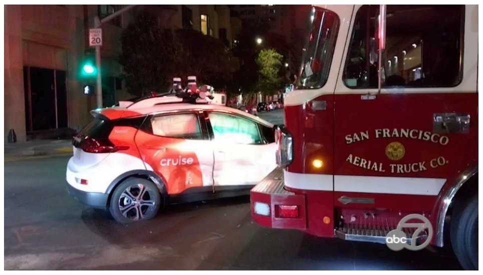
(b) Dynamic national qualifying test at 2007 Darpa Urban Challenge.

Figure 1.2: National Qualifying Events (NQEs) from the 2007 DARPA Urban Challenge. The photos are from the perspective of Alice, Caltech's entry in the competition, during the track tests.

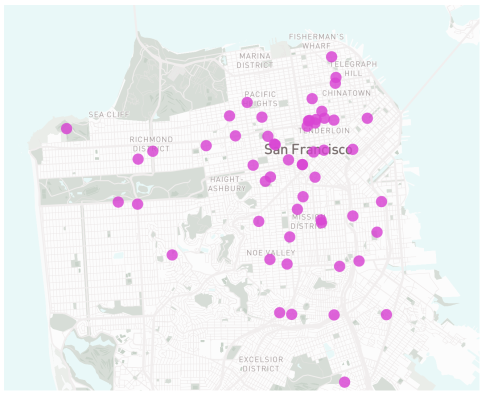
gagement) to build statistical confidence that the system is safe. This approach can be extremely inefficient in time and cost, and would have to be repeated after each design iteration [4]. The final approach of manual constructing tests requires test engineers to rely on their expertise to specify the high-level scenario as well as design the test harness (e.g., specifying the number and locations of obstacles, dynamic agents and their strategies). In the application of autonomous vehicles, there is ongoing effort to standardize requirement specification, and test and evaluation procedures [5–8]. Standards such as "ISO 21448:2022 Safety of the Intended Functionality (SOTIF)" [6] provide guidance on verification and validation methods to demonstrate that self-driving. Listed below are some approaches to testing in the self-driving industry today.

Track-testing at the DARPA Urban Challenge: The 2007 DARPA Urban Challenge ushered interest in autonomous driving in urban environments [9]. Participating vehicles had to pass three small-scale operational test-courses, national qualifying events or NQEs, that were designed to evaluate the autonomous car's ability to satisfy safety, basic and advanced navigation requirements, and basic and advanced traffic scenarios [10]. Exhaustive verification for such complex safety-critical systems is prohibitive, creating a need for a formal operational testing framework to certify reliability of these systems [11]. Figure 1.2 shows Caltech's entry, Alice, in the test tracks corresponding to a completely static test environment and a dynamic test environment with other live vehicles. These tests were designed by entirely by test engineers.

AV companies have long relied on testing on urban roads to demonstrate to gather



(a) Cruise vehicle driving in the path of a fire truck.



(b) Map by Will Jarrett at Mission Local [12] using data from the San Francisco Fire Department. This map shows locations where Cruise vehicles violated traffic rules during on-road testing in their interactions with fire trucks.

Figure 1.3: Cruise vehicles driving in the path of emergency responders. In just the first half of 2023, 55 such incidents were reported in the city of San Francisco.

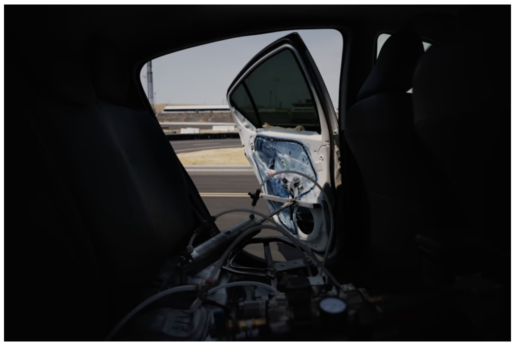
data for test and evaluation, and to demonstrate technological readiness. However, even test driving for millions of miles is not sufficient to demonstrate safety guarantees. In California in the year 2023 alone, six companies with permits for driverless testing have completed 3,267,792 miles in autonomous driving mode at SAE Level 4. However, it is still not sufficient to demonstrate required levels of safety. For example, in the first half of 2023, there were 55 incidents of Cruise vehicles driving in the path of emergency vehicles [12] (also see Figure 1.3). Recently, issues such as these have led to driverless permits being suspended by the California DMV.

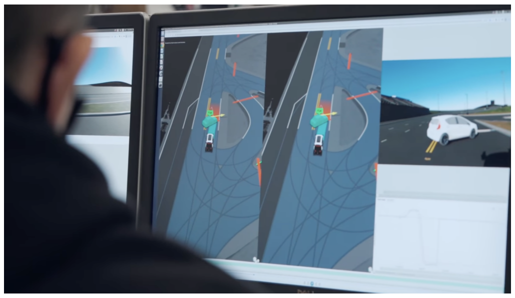
In addition to road testing, the AV industry heavily relies on track testing and simulation-based testing to ensure the safety of its vehicles. Waymo's safety methodology [5] lists the following methods to evaluate autonomous driving behavior on its vehicles: i) hazard analysis that tests for robustness against user-defined hazards, ii) scenario-based testing on an instrumented track and in simulation, and iii) extensive simulation testing that aggregates driving performance across several simulations. Aside from manually specified scenarios, the industry also relies on police reports to test its software in challenging scenarios [5]. The self-driving car company, Zoox, also released a highlight video demonstrating its approach to track testing, snapshots of which are shown in Figure 1.4. First, scenarios that are difficult are identified by test engineers, and these scenarios are recreated in simulation and on the closed-loop track.

(a) Road condition: bumpy

(b) Road condition: damp

(c) Testing high-speed maneuverability: obstacle course in simulation


(d) Instrumented door to test whether Zoox car can properly detect and avoid collision.


(e) Scenario design by test engineers prior to track test shown in Figure 1.4f.

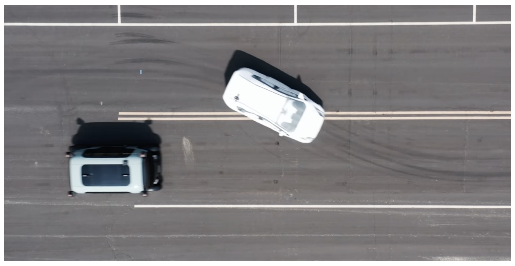

(f) Reactive test scenario in which Zoox car must respond correctly in reaction to the environment agent.

Figure 1.4: Instrumented track testing at Zoox. These images are taken from "Putting Zoox to the Test" [1].

These case studies illustrate the need for rigorous approaches to test and evaluation of these systems. Existing approaches do not provide a definitive answer to the certification of autonomous systems in safety-critical settings. Now, we will cover related work that is motivated by these challenges.

1.3 Related Work

Task-Relevant Evaluation of Perception

As discussed in the Challenges, perception and planning modules are typically designed under different computational paradigms. At the NVIDIA AV Team, empirical studies on how perception design choices affect overall system-level safety have been studied in a pedestrian jay-walking scenario [13]. These empirical studies reflect the need for studying this problem more rigorously. The design paradigms for planning and control submodules are usually backed by guarantees of safety and stability. For this related work, we will take the example of formal methods as a paradigm for control system design, but these insights can extend to other planning and control frameworks that provide guarantees of correctness.

Formal methods have been employed to construct provably correct planners and controllers given a system model and temporal logic specifications [14–18]. The correctness guarantee, typically specified using a temporal logic formula, relies heavily on the assumption that the input (i.e., the perceived world reported by perception) is perfect. Perception is important for *state estimation*, which is necessary for the downstream control and planning logic to effectively react to the environment. For example, if the perception component only reports the most likely class of each object, the control component assumes that the reported class is correct. Unfortunately, this assumption may not hold in most real-world systems, and the correctness guarantees might no longer hold.

In recent years, verifying neural networks with respect to safety and robustness properties has grown into an active research area [19–22]. Often, these methods apply to specific neural net architectures, such as those with piece-wise linear activation functions [19], or might require knowledge of the safe set in the output space of the neural network [20, 21]. Furthermore, these methods have been demonstrated on learning-based controllers with smaller input dimensions, and are not yet deployed for analysis of perception models. One reason for this is the difficulty in formally characterizing properties of ML-based perception models, as elaborated below.

First, recent work demonstrates that it is not realistically feasible to formally specify properties reflecting human-level perception for perception models, in particular, classification ML models, due to the high dimensional nature of the input, such as pixels in an image [23]. Finally, not all perception errors are equally safety-critical. Dreossi *et al.* reason that not all misclassifications are the same; some are more likely to result in system-level failure, and therefore, it is necessary to adopt system-level specifications and contextual semantics in developing a framework for quantitative analysis and verification of perception models [23, 24]. This has led to work on compositional analysis of perception models in finding system-level counter-examples [25]. The work in [26] introduced the concept of interaction zones using Hamilton-Jacobi reachability theory, and illustrated that perception errors in the interaction zone were more likely to result in system-level violations than those outside of it. This observation was further backed in [27], which demonstrated instances of both small perception errors (for the task of segmentation over RGB images) leading to closed-loop system-level failure, and large perception errors still resulting in safe system-level failures.

While there is work on evaluating performance of perception with temporal logic, those formal specifications are defined over image data streams, and must be manually formalized for each scenario / data stream [28, 29]. Often, there is high variability in the performance of perception models in seemingly similar environments, such as variations in sun angle [30]. Therefore, for any given scenario, it can be challenging to specify all realizations of the environment that a perception system might encounter. On the other hand, it is simpler, and more accurate, to define system-level specifications, such as "maintain a safe distance of 5 m from obstacles" [23, 31–33].

Testing for Autonomous Systems

As described in the Challenges section, tests are often manually designed by test engineers. This was seen in the DARPA Urban Challenge, and in current practices at AV companies such as Waymo and Zoox. Test scenarios are often constructed first in simulation using tools such as CARLA [34] and Scenic [35]. For example, Scenic is a probabilistic programming language to model environments of autonomous cyber-physical systems. A single Scenic program describes a distribution of environments by declaring random variables (e.g., position of parked car, location of pedestrians, color of obstacles) and specifying distributions of each of these random variables. A compiled Scenic program can be sampled to provide concrete scenes, and these concrete scenarios are related by the high-level scenario (e.g., number of cars and their approximate locations) used to define the Scenic program. However, Scenic cannot handle the generation of these Scenic programs from high-level specifications. The automated, reactive test synthesis framework in Chapters 3– 4 addresses this, and can potentially be interfaced to Scenic to automatically construct scenarios at all levels of the planning stack.

In the formal methods community, research on falsification aims to uncover bugs in the software of cyber-physical systems with access to just black-box models, and without any knowledge of the control design [36–40]. Oftentimes, specifications for these cyber-physical systems are characterized in metric temporal logic (MTL) and signal temporal logic (STL), which allow for specifying timed requirements and also lend themselves to quantitative metrics of robustness to characterize the degree to which a specification is satisfied or violated. The goal of falsification is search over a specified input domain (typically continuous) to identify an input that maximizes the degree of violation of the specified requirement. The community has introduced several toolboxes, e.g., Breach [41] and S-TaLiRo [36, 42], among others [43] for this effort. These falsification toolboxes can be interfaced with scenario definition programs such as Scenic to automatically construct test scenarios, and an example of such a tool is VerifAI [44]. Note that the user still needs to define the high-level scenario in Scenic — interfacing with the falsifier returns the worst-case concrete scenario from the distribution of scenarios.

Aside from traditional black-box optimization methods such as Bayesian optimization, cross-entropy method, reinforcement learning has been used to identify falsifying inputs [45–47]. Oftentimes, falsification algorithms are applied over continuous domains and metrics, and often cannot handle discrete input spaces. However, complex cyber-physical systems are expected to handle both continuous and discrete inputs, and reason over continuous and discrete state spaces [48]. Additionally, falsifying inputs are often open-loop signals that generate the worst-case trajectory in simlation. However, feedback is a fundamental principle in control theory that allows us to design systems that are robust to unmodeled dynamics, uncertainties, and disturbances. The contributions in thesis complements falsification — our focus is on synthesizing high-level test environments and reactive test strategies that operate over discrete state spaces. In future work, we can search over the continuous parameters of the synthesized test environment (e.g., continuous pose values of test agents, friction coefficients, exact timing of events) using falsification algorithms for further concretizing the test scenario.

1.4 Thesis Overview and Contributions

The principles underlying my past and current work are *reactive* test plans, *modular* test and evaluation of subsystems and interfaces between subsystems, and choosing relevant *specifications* and *evaluation criteria* at the system and subsystem levels by accounting for interactions between subsystems and their impact on system-level behavior. The theoretical contributions as well as its applications, in both algorithms and hardware, are outlined below.

Part I: System-level Reasoning for deriving Task-Relevant Metrics of Perception

Chapter 2 focuses on introducing task-relevant evaluation metrics for object detection and classification models for perception. This work identifies evaluation metrics of perception tasks that are useful in providing probabilistic guarantees on system-level behavior. At a high-level, the main contribution of this work is in identifying standard perception metrics that can be used in a quantitative system-level analysis, and in proposing new perception metrics that are relevant to the downstream planner and the system-level task.

First, we identify popularly used metrics in computer vision confusion matrices as a candidate model for sensor error, and leverage probabilistic model checking to quantify the probability of the overall system satisfying its requirements. Prior work [49] has shown how to leverage a probabilistic model of sensor error in probabilistic model-checking of the overall system with respect to system-level temporal logic specifications. The work in this chapter was the first to identify confusion matrices as a model of sensor error for detection and classification tasks, rigorously define probabilities of misdetection from the confusion matrix, and show how it can be leveraged in probabilistically model-checking system-level task specifications.

Confusion matrices are popularly used in computer vision to compare and evaluate models for detection tasks, and a wide-variety of metrics such as accuracy, precision, recall, among others, can be derived from the confusion matrix. The key idea was in identifying confusion matrices as a candidate for capturing requirements on detection tasks, and in rigorously defining probabilities to relate the confusion matrix to system-level performance with respect to temporal logic specifications. Even on simple examples, our approach highlighted fundamental insights: performance tradeoffs (e.g., precision-recall tradeoff) in detection tasks get reflected in system-level performance, and our method gives sanity checks – both qualitative and quantitative guidelines on selecting detection models and high-level planners, which in combination have probabilistic system-level guarantees.

For example, consider a car-pedestrian scenario in which the autonomous car needs to contend with multiple safety requirements — to stop for a pedestrian at a crosswalk and to not stop unnecessarily if there are no pedestrians at the crosswalk. While engineers training perception algorithms might optimize for high recall (i.e., to never miss a pedestrian even at the cost of false negatives), this will lead to the car stopping frequently. This intuition was captured quantitatively in my framework. Furthermore, if we have probabilistic system-level guarantees (e.g., meet a safety requirement to 99%) and given a specific planning logic, we can derive lower bounds on elements of the confusion matrix such as minimum true positive rate, minimum false negative rate, and encode requirements on perception tasks in this manner. The practical impact of this method is the ability to communicate quantitative requirements via confusion matrices, rather than temporal logic specifications, to engineers training perception algorithms for detection tasks.

The second contribution is in defining new metrics for detection tasks, informed by the system-level specification as well as the downstream planning logic. This work stemmed from the insight that not all perception errors are equally safety-critical, and that current methods to evaluate perception models do not account for this distinction. For instance, in evaluating models for object detection tasks in computer vision, all misdetections are given equal weight in the confusion matrix. However, not all misclassifications or misdetections will have the same impact on system-level safety. To account for this, we introduced a distance-parametrized, proposition-labeled confusion matrix, which: i) placed higher weight on correct detection of objects closer to the ego, and ii) replaced the object class labels of confusion matrices with atomic propositions that are more relevant to system-level safety specification.

Guarantees from the distance-parametrized, proposition-labeled confusion matrices is less conservative than the analysis that used the traditional class-based confusion matrix. Further extensions of the proposition-labeled confusion matrix, in which predictions are grouped according to the same level of abstraction used by the high-level planner, result in system-level satisfaction probabilities that are neither too relaxed and nor too conservative. Finally, the proposed metrics are used to evaluate a PointPillars on the real-world nuScenes dataset. The core message of this work is that metrics for evaluating perception tasks need to be carefully informed by both the system-level specification as well as the downstream planning and control logic. This work is being packaged as a Python toolbox, TRELPy: Task-Relevant Evaluation of Perception.

Part II: Reactive Test Synthesis

Chapters 3– 4 focus on reactive test synthesis. These chapters address the problem of synthesizing tests for high-level reasoning and decision-making in autonomous robotic applications. Instead of having the entire test be manually designed, we presume that it is easier for a test engineer to provide a formal description of the objective of the test. My work focused on automated construction of test scenarios from these high-level test objectives specified by the user/test engineer. Chapter 5 introduced preliminary directions on compositional test synthesis from unit tests via assume-guarantee contracts.

Chapters 3– 4: The first contribution of this work is introducing the notion of a test specification: a high-level description of the objective of the test. This test objective is not revealed to the system under test, but is consistent with safety and liveness assumptions the system has on its environment (e.g., there will always exist a path to the goal, the environment agents will not adversarially collide).

The second contribution is in automatic construction of a reactive test that is consistent with the test objective as well as minimally restrictive to the system. In particular, the constructed test harness involves placement of static and reactive constraints to system actions, and the smallest number of restrictions needed for the test objective are found. These restrictions on system actions are such that if the system under test is successful in meeting its requirements, the test objective is also met. The third contribution is in automatically mapping these reactive constraints to synthesize a reactive strategy of a dynamic test agent. Finally, we also prove that the reactive test synthesis problem is NP-hard via a reduction from 3-SAT.

Algorithms: Chapter 4 provides algorithms to automate each of the aforementioned tasks. First, leveraging automata theory and combinatorial graph algorithms, we formulate a network flow optimization to identify static and/or reactive test constraints for the system. The problem data for this algorithm includes the specifications the system is expected to satisfy, the test objective, and a discrete-state abstraction model of the system. Note that the system model is non-deterministic and does not carry knowledge of the system control; it is just a high-level abstraction representing all possible actions a system can take from any given state. Although this is a combinatorial problem, we take advantage of the structure in the resulting product graph to formulate a mixed-integer linear program with discrete variables representing interdiction of edges in the network that correspond to reactive test constraints. The choice of using network flows allows for the optimization to handle medium sized problems (5000 integer variables) with a runtime of around 30s to a few minutes. We prove that the optimal solution corresponds to a set of restrictions with the following guarantees: any trajectory of the system that satisfies the system objective will also satisfy the test objective.

Furthermore, it is easy to augment additional optimization constraints (e.g., some system actions cannot be constrained). This becomes prominent when synthesizing a test agent strategy to match the restrictions returned by the optimization. The optimization is solved offline, and the resulting solution is automatically mapped to a reactive test strategy for a given dynamic agent. If the solution is not dynamically feasible for the test agent, we use an efficient counterexample-guided approach to resolve the MILP. For this, the test constraints are mapped as safety formulas that the dynamic test agent is expected to satisfy. Additional safety formulas are found to ensure that the dynamic test agent does not restrict system actions other than the test constraints. Furthermore, we address livelocks by automatically identifying potential livelock states, and specify that the test agent, if it occupies these states, should only transiently occupy it. Finally, the synthesized test strategy chooses from a set of possible initial conditions and realizes the reactive test constraints found by the optimization.

Hardware Demos: This framework was demonstrated in hardware using quadrupeds for robot navigation examples such as search and rescue, and testing motion primitives. In addition to demonstrating the usefulness of this approach to real robotic systems, the hardware experiments were repeatable and successful immediately after the test strategy was generated in simulation. These experiments demonstrated that our framework can handle test objectives beyond simple abstractions of robot position (e.g., go to a particular cell), but can also capture more complex behaviors such as dynamic motion primitives (e.g., jump then stand). Our test synthesis framework had no knowledge of the control architecture for low-level motion primitives (e.g., standing, walking, jumping) or even the mid-level planning framework (e.g., waypoint following) on the quadruped. Despite this, the high-level, reactive test strategy resulted in successful demonstrations in hardware. This experimental success points to promising future directions in decoupling test synthesis for high-level reasoning and low-level control.

Finally, we also provide an argument for why traditional GR(1) synthesis techniques cannot be used to directly synthesize tests that are not overly-restrictive. There are two reasons. First, the synthesis of test constraints cannot be cast into an LTL synthesis problem. In reactive synthesis for LTL or similar temporal logics, synthesis assumes worst-case behavior of the other player, which is not consistent with our objectives. Our test harness is not fully cooperative nor fully adversarial: we do not help the system achieve its requirements yet ensure that there always exists a path for success. It is possible that this can be cast as a synthesis problem in a different temporal logic that reasons over path properties (e.g., computational tree logic (CTL), or hyperLTL). However, the synthesis in those specification languages is known to be computationally intractable. Second, our optimization finds the least restrictive set of test constraints, which traditional synthesis methods cannot provide.

Chapter 5: The main contributions of this chapter are as follows. We establish a mathematical framework for merging two unit test scenarios using assume-guarantee contracts. The merged test can be optimized according to an arbitrary difficulty metric, and we use a receding horizon approach to synthesize winning sets that guide the test strategy to optimize for the metric.

Chapter 2

EVALUATING PERCEPTION FOR SYSTEM-LEVEL TASK REQUIREMENTS

In safety-critical systems, the goal of perception is to aid downstream decision-making modules so that the overall system can meet its safety-critical requirements. Yet, the metrics we often use to evaluate perception performance do not account for system-level requirements or interactions between sub-systems. Usually, not all perception errors are equally safety-critical with respect to system-level requirements. this chapter argues for the importance of system-level reasoning in identifying metrics to evaluate perception. First, we show how existing evaluation metrics for object detection tasks, e.g., confusion matrices, can be leveraged to compute a probabilistic satisfaction of system-level specifications. However, confusion matrices, as traditionally defined, account for all detections equally. The second contribution of this chapter is in identifying that atomic propositions relevant to downstream planning logic and the system-level specification can be used to define new metrics for detection which result in less conservative system-level evaluations. Finally, we illustrate these ideas on a car-pedestrian example in simulation for confusion matrices constructed from the nuScenes dataset. We validate the probabilistic system-level guarantees in simulation.

This chapter is adapted from:

A. Badithela, T. Wongpiromsarn, R. M. Murray. (2021). "Leveraging Classification Metrics for Quantitative System-Level Analysis with Temporal Logic Specifications." In: *2021 60th IEEE Conference on Decision and Control (CDC)*, pp. 564–571. DOI: 10.1109/CDC45484.2021.9683611.

A. Badithela, T. Wongpiromsarn, R. M. Murray. (2023). "Evaluation Metrics of Object Detection for Quantitative System-Level Analysis of Safety-Critical Autonomous Systems." In: *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 8651–86581.
DOI: 10.1109/IROS55552.2023.10342465.

A. Badithela, R. Srivastav, T. Wongpiromsarn, R. M. Murray, "Task-relevant evaluation metrics for object detection." *In Preparation for submission to the International Journal of Robotics Research (IJRR).*

Section 2.7 has been adapted from:

I. Incer, A. Badithela, J. Graebener, P. Mallozzi, A. Pandey, S.-J. Yu, A. Benveniste, B. Caillaud, R. M. Murray, A. Sangiovanni-Vincentelli, and S. A. Seshia. (2024). "Evaluation Metrics of Object Detection for Quantitative System-Level Analysis of Safety-Critical Autonomous Systems." Conditionally accepted to: *The ACM Transactions on Cyber-Physical Systems (T-CPS).*
arXiv preprint: https://arxiv.org/pdf/2303.17751.

2.1 Introduction

The presence of deep neural network architectures in the software stack of safety-critical applications (e.g., self-driving vehicles) necessitates a comprehensive system-level evaluation of these systems. Figure 1.1 is an illustration of the software stack of the system in which the perception component involves a deep learning-based architecture, which perceives the environment and passes its observations as inputs to the downstream planning and control modules. Using this information, the control module computes a trajectory for the vehicle to follow and the corresponding actuation commands to keep the vehicle on the trajectory.

The perception and control modules are typically designed under different principles. For example, the perception module often relies on object classification that is based on deep learning such as the use of convolutional neural networks to distinguish objects of different classes. These learning-based algorithms are often evaluated based on the performance measures such as accuracy, precision, and recall [50, 51].

On the other hand, formal methods have been employed to construct a provably correct controller given a system model and temporal logic specifications [14–18]. The correctness guarantee, typically specified using a temporal logic formula, relies heavily on the assumption that the input (i.e., the perceived world reported by perception module) is perfect. For example, if the perception component only reports the most likely class of each object, the control component assumes that the reported class is correct. Unfortunately, this assumption may not hold in most real-world systems.

To reason about system-level safety, one might consider the paradigm of specifying formal requirements on the entire system and reasoning about it. However, specifying formal requirements on the object detection task of perception is not trivial. Even in the standard classification task of classifying handwritten digits, it is difficult to formally specify how the digits must be classified. Instead of taking this approach, we leverage metrics that are already used to evaluate learned models for their performance on object detection and classification tasks — confusion matrices. Confusion matrices are a statistical model of sensor error, constructed by evaluating a learned model against a large evaluation set.

The first contribution of this chapter is in identifying confusion matrices as a candidate model of sensor error. Leveraging confusion matrices, we can rigorously define transition probabilities representing the system's state evolution in the presence of detection error. On this model of the overall system, we can quantify system-level satisfaction of specifications via off-the-shelf probabilistic model-checking tools. An important insight gained from this analysis is that even in simple examples, intuitive design methodologies for detection models, such as maximizing recall with respect to pedestrians, might not result in safer systems overall.

However, traditionally defined confusion matrices do not account for the system-level task or the downstream controller. The second contribution of this chapter is proposing two new logic-based evaluation metrics that to account for the downstream planning logic and the system-level task. We replace the object class labels of a confusion matrix with logical formulas that are informed from the downstream controller and system-level guarantee.

Related Work

Evaluating and monitoring perception for safety-critical errors is an emerging research topic [26, 52, 53]. Perception is a complex subsystem responsible for tasks such as detection, localization, segmentation. These recent works have focused on evaluating object detection in the context of system-level safety. We follow this early work and focus on object detection task of perception, which refers to both detecting an object and classifying it correctly. As an initial stage of this study, we assume a static environment and perfect object localization. These assumptions can potentially be relaxed based on an analysis that takes into account partial observability of the environment [54], as discussed in Section 2.8.

The use of Markov chains for probabilistic reasoning about the correctness of high-level robot behaviors in the presence of perception errors was studied in [49]. However, the algorithms in [49] assumed knowledge of the probabilistic sensor model. Rigorously constructing these sensor models from confusion matrices was presented in [55]. In [56], this approach was further extended by providing confidence intervals on the probabilistic sensor models and was applied to a case study on guiding aircraft on taxiways introduced by Boeing [57].

For runtime monitoring of perception systems, Timed Quality Temporal Logic (TQTL) is used to specify spatio-temporal requirements on perception [28, 29] . However, to specify these requirements, the user has to label each scenario with critical objects that need to be detected. This approach is useful in evaluating perception in isolation with respect to the requirements defined on a specific scenario. In [52], temporal diagnostic graphs are proposed to identify failures in object detection during runtime.

In [26], Hamilton-Jacobi reachability was used to account for closed-loop interactions with agents in the environment to identify safety-critical perception zones in which correct detection is crucial. Our work can be viewed as a complementary approach to [26] by allowing crucial misclassifications, according to system-level analysis, to be identified. Task-relevant perception design has been studied in [58] and [59]. In [58], the codesign of control and perception modules has been explored for tasks such as state estimation [58] and behavior prediction [59].

2.2 Preliminaries

In this section, we give an overview of linear temporal logic (LTL), a formalism for specifying system-level requirements. We also describe the performance metrics used to evaluate object detection and classification models in the computer vision community. Finally, we setup a simple discrete-state car-pedestrian system as a running example to illustrate the role of these different concepts.

System-level Task Specifications

System Specification. We use the term system to refer to refer to the autonomous agent and its environment. The agent is defined by variables V_A, and the environment is defined by variables V_E. The valuation of V_A is the set of states of the agent S_A, and the valuation of V_E is the set of states of the environment S_E. Thus, the states of the overall system is the set $S := S_A \times S_E$. Let AP be a finite set of atomic propositions over the variables V_A and V_E. An atomic proposition $a \in AP$ is a statement that can be evaluated to *true* or *false* over states in S.

We specify formal requirements on the system in LTL (see [60] for more details).

Definition 2.1 (Linear Temporal Logic [60])**.** *Linear temporal logic* (LTL) is a temporal logic specification language that allows reasoning over linear-time trace properties. An LTL formula is defined by (a) a set of atomic propositions, (b) logical operators such as: negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), and implication ($\implies$), and (c) temporal operators such as: next ($\bigcirc$), eventually ($\diamond$), always ($\Box$), and until ($\mathcal{U}$). The syntax of LTL is given as:

$$\varphi ::= \textit{True} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1 \mathcal{U} \varphi_2,$$

with $a \in AP$, where AP is the set of atomic propositions, $\wedge$ (conjunction) and $\neg$ (negation) are the Boolean connectors from which other Boolean connectives such as $\rightarrow$ can be defined, and $\bigcirc$ (next) and $\mathcal{U}$ (until) are temporal operators. Let φ be an LTL formula over AP. We can define the operators $\Diamond$ (eventually) and $\Box$ (always) as $\Diamond\varphi = \textit{True}\,\mathcal{U}\varphi$ and $\Box\varphi = \neg\Diamond\neg\varphi$. The syntax of LTL is read as follows: (a) An atomic proposition p is an LTL formula, and (b) if φ and ψ are LTL formulae, then $\neg\varphi$, $\varphi \vee \psi$, $\bigcirc\varphi$, $\varphi\,\mathcal{U}\,\psi$ are also LTL formulae. For an execution $\sigma = s_0 s_1 \ldots$ and an LTL formula φ, $s_i \vDash \varphi$ iff φ holds at $i \geq 0$ of σ. More formally, the semantics of LTL formula φ are inductively defined over an execution $\sigma = s_0 s_1 \ldots$ as follows,

- for $a \in AP$, $s_i \vDash a$ iff a evaluates to *True* at s_i,
- $s_i \vDash \varphi_1 \wedge \varphi_2$ iff $s_i \vDash \varphi_1$ and $s_i \vDash \varphi_2$,
- $s_i \vDash \neg\varphi$ iff $\neg(s_i \vDash \varphi)$,
- $s_i \vDash \bigcirc\varphi$ iff $s_{i+1} \vDash \varphi$, and
- $s_i \vDash \varphi_1 \mathcal{U} \varphi_2$ iff $\exists k \geq i, s_k \vDash \varphi_2$ and $s_j \vDash \varphi_1$, for all $i \leq j < k$.

An execution/trace $\sigma = s_0 s_1 \ldots$ satisfies formula φ, denoted by $\sigma \models \varphi$, iff $s_0 \models \varphi$. A *strategy* π is correct (satisfies formula φ), if the trace σ_π resulting from the strategy satisfies φ.

For an infinite trace $\sigma = s_0 s_1 \ldots$, where $s_i \in 2^{AP}$, and an LTL formula φ defined over AP, we use $\sigma \models \varphi$ to denote that σ satisfies φ. For example, the formula $\varphi = \Box p$ represents that the atomic proposition $p \in AP$ is satisfied at every state in the trace, i.e., $\sigma \models \varphi$ if and only if $p \in s_t, \forall t$. In this chapter, these traces σ are executions of the system, which we model using a Markov chain.

Definition 2.2 (Labeled Markov Chain [60])**.** A discrete-time *labeled Markov chain* is a tuple $\mathcal{M} = (S, Pr, \iota_{init}, AP, L)$, where S is a non-empty, countable set of

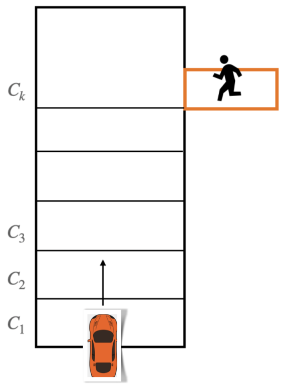


Figure 2.1: Running example of a car and pedestrian. If there is a pedestrian at crosswalk cell C_k, that is, $x_e \models \texttt{ped}$, then the car must stop at cell C_{k-1}. Otherwise, it must not stop.

states, $Pr : S \times S \to [0, 1]$ is the *transition probability function* such that for all states $s \in S$, $\Sigma_{s' \in S} Pr(s, s') = 1$, $\iota_{init} : S \to [0, 1]$ is the initial distribution such that $\Sigma_{s \in S} \iota_{init}(s) = 1$, AP is a set of atomic propositions, and $L : S \to 2^{AP}$ is a labeling function. The labeling function returns the set of atomic propositions that evaluate to true at a given state. Given an LTL formula φ (defined over AP) that specifies requirements of a system modeled by the Markov Chain $\mathcal{M}$, the probability that a trace of the system starting from $s_0 \in S$ will satisfy φ is denoted by $\mathbb{P}_{\mathcal{M}}(s_0 \models \varphi)$. The definition of this probability function is detailed in [60].

Example

Consider a car-pedestrian example, modeled using discrete transition system as illustrated in Figure 2.1. The true state of the environment is denoted by x_e. The state of the car is characterized by its position and speed, $s_a := (x_c, v_c) \in S_A$. The safety requirement on the car is that it "shall stop at the crosswalk if there is a waiting pedestrian, and not come to a stop, otherwise". The overall system specifications are formally expressed as safety specifications in equations (2.1)-(2.3).

1. If the true state of the environment is not a pedestrian, i.e. $x_e \neq \texttt{ped}$, then the car must not stop at C_{k-1}.

$$\varphi_1 = \square((x_e \neq \texttt{ped}) \to \neg(x_c = C_{k-1} \wedge v_c = 0)) \,. \tag{2.1}$$

2. If $x_e = \texttt{ped}$, the car must stop on C_{k-1}.

$$\varphi_2 = \Box\Big(x_e = \texttt{ped} \rightarrow ((x_c = C_{k-1} \wedge v_c = 0) \\ \vee \neg(x_c = C_{k-1}))\Big). \tag{2.2}$$

3. The agent should not stop at any cell C_i, for all $i \in \{1, \ldots, k-2\}$,

$$\varphi_3 = \Box\neg(\bigvee_{i=1}^{k-2}(x_c = C_i \wedge v_c = 0)). \tag{2.3}$$

The overall safety specification for the car is $\varphi := \varphi_1 \wedge \varphi_2 \wedge \varphi_3$. Since the car controller has been designed assuming perfect perception, the specification for the pedestrian and non-pedestrian environment simplifies to,

$$\begin{aligned}\varphi_{\texttt{ped}} =& \Box\neg(\bigvee_{i=1}^{k-2}(x_c = C_i \wedge v_c = 0)) \bigwedge \Box(\neg(x_c = C_{k-1}) \\ & \vee (x_c = C_{k-1} \wedge v_c = 0)), \\ \varphi_{\texttt{class}} =& \Box\neg(\bigvee_{i=1}^{k-1}(x_c = C_i \wedge v_c = 0)), \text{ if } \texttt{class} \in \{\texttt{obs}, \texttt{empty}\}.\end{aligned}$$

As mentioned previously, we assume a static environment. We also assume that the car knows the location of the crosswalk, e.g., from HD map information, and that it can coarsely localize whether the detected object is on the crosswalk. The evaluation framework presented in this chapter is valid for any discrete-state control strategy, both deterministic and probabilistic. To concretize the setup, we consider a car controller that acts corresponding to the detection model's prediction of the environment at the crosswalk. If the car at time step t detects a pedestrian, then it chooses its speed according to a control strategy for $\varphi_{\texttt{ped}}$ to come to a stop before the crosswalk at cell C_{k-1}. If the state of the car is such that it is impossible to find a controller that will bring it to a stop at cell C_{k-1}, then it decelerates as fast as possible. Similarly, if an obstacle or empty sidewalk is detected, then the car chooses its speed according to a control strategy designed correct-by-construction for φ_k.

2.3 Problem Statement

Here, we introduce and define the probability of satisfaction of an LTL formula starting from an initial state, given the true state of the environment.

Definition 2.3 (Model of Sensor Error)**.** Let S_E denote the set of possible environment states. Then, a model of sensor error in identifying the state of the environment $M : S_E \times S_E \to [0, 1]$ is defined as follows, $M(y, x) = p$, where p is the probability with which the sensor predicts the environment state to be $y \in S_E$ when its true state is $x \in S_E$.

Definition 2.4 (Transition Probability)**.** Let $s_1 = (s_{1,a}, x_e)$, $s_2 = (s_{2,a}, x_e) \in S$ be two states of the overall system, x_e be the true class label of the environment, and let M be a model of sensor error. Let $O(s_1, s_2)$ denote the set of environment observations $y_e \in V_E$ that result in the agent controller transitioning from $s_{1,a}$ to $s_{2,a}$. The *transition probability* $Pr : S \times S \to [0, 1]$ is defined as,

$$Pr(s_1, s_2) := \sum_{y_e \in O(s_1,s_2)} M(y_e, x_e)\,. \tag{2.4}$$

Since the controller is entirely informed by the outputs of the perception module, and for each output of the perception module, there is a corresponding control action, it is trivial to check that $\sum_{s_2 \in S} Pr(s_1, s_2) = 1$. Therefore, the transition probability between any two states is always in the range $[0, 1]$.

Definition 2.5 (Paths)**.** Choose a state $s_0 = (s_{a,0}, x_e) \in S$ for a fixed true environment state x_e. A finite path starting from s_0 is a finite sequence of states $\sigma(s_0) = s_0, s_1, \ldots, s_n$ for some $n \geq 0$ such that the probability of transition between consecutive states, $Pr(s_i, s_{i+1}) > 0$ for all $0 \leq i < n$ such that $s_i = (s_{a,i}, x_e) \in S$. Similarly, an infinite path $\sigma = s_0, s_1, \ldots$ is an infinite sequence of states such that $Pr(s_i, s_{i+1}) > 0$ for all $i \geq 0$. We denote the set of all paths starting from $s_0 \in S$ by $Paths(s_0)$, and the set of all finite paths starting from $s_0 \in S$ by $Paths_{fin}(s_0)$. For an LTL formula φ on AP, $Paths_\varphi(s_0) \subset Paths(s_0)$ is the set of paths $\sigma = s_0, s_1, \ldots$ such that $\sigma_S \models \varphi$.

Semantics

Now, we define probability of satisfaction of a temporal logic formula with respect to a formal specification based on the following definitions derived from [60]. Let $\Omega = Paths(s_0)$ represents the set of all possible outcomes, that is, the set of all paths of the agent, starting from state s_0. Let 2^Ω denote the powerset of Ω. Then, $(\Omega, 2^\Omega)$ forms a σ-algebra. For a path $\hat{\pi} = s_0, s_1, \ldots, s_n \in Paths_{fin}(s_0)$, we define a cylinder set as follows,

$$Cyl(\hat{\pi}) = \{\pi \in Paths(s_0) | \hat{\pi} \in pref(\pi)\}, \tag{2.5}$$

where $pref(\pi) = \{\pi_{...j} = s_0, \ldots, s_j | j \geq 0\}$ is the set of all finite prefix path fragments for $\pi = s_0, s_1, \ldots$, an infinite path. Let $\mathcal{C}_{s_0} = \{Cyl(\hat{\pi}) | \hat{\pi} \in Paths_{fin}(s_0)\}$. The following result can be found in [60], and can be derived from the fundamental definition of a σ-algebra.

Lemma 2.1. The pair $(Paths(s_0),\, 2^{\mathcal{C}_{s_0}})$ forms a σ-algebra, and is the smallest σ-algebra containing $\mathcal{C}_{s_0}$.

The σ-algebra associated with s_0 is $(Paths(s_0), 2^{\mathcal{C}_{s_0}})$. Then, there exists a unique probability measure $\mathbb{P}_{s_0}$ such that

$$\mathbb{P}_{s_0}(Cyl(s_0, \ldots, s_n)) = \prod_{0 \leq i \leq n} Pr(s_i, s_{i+1}). \tag{2.6}$$

Definition 2.6. Consider an LTL formula φ over AP with the overall system starting at state $s_0 = (s_{a,0}, x_e)$. Then, the probability that the system will satisfy the specification φ from the initial state s_0 given the true state of the environment is,

$$\mathbb{P}(s_0 \models \varphi) := \sum_{\sigma(s_0) \in \mathcal{S}(\varphi)} \mathbb{P}_{s_0}(Cyl(\sigma(s_0))), \tag{2.7}$$

where $\mathcal{S}(\varphi) := Paths_{fin}(s_0) \cap Paths_{\varphi}(s_0)$. Note that $\mathcal{S}(\varphi)$ need not be a finite set, but has to be countable.

Definition 2.7 (Controller)**.** For an initial condition $s_0 \in S$ of the system and environment, and the system specification φ, the system controller $K : S^{\omega}S \rightarrow S_A$ chooses the next system state based on the trace history of system states and environment observations.

Problem Formulation

Problem 2.1. Given a model of sensor error M for multi-class classification, a controller K, a temporal logic formula φ, the initial state of the agent $s_{a,0}$, and the true state of the static environment x_e, compute the probability $\mathbb{P}(s_0 \models \varphi)$ that φ will be satisfied for a system trace σ starting from initial condition $s_0 = (s_{a,0}, x_e)$?

2.4 Role of Detection Metrics in Quantitative System-level Evaluations

In this section, we will introduce x While the confusion matrix provides useful metrics for comparing and evaluating detection models, we would like to use these metrics in evaluating the overall system with respect to formal constraints in temporal logic. Not all detection errors are equally safety-critical [25, 26].

Confusion Matrix

We consider object detection to include both the detection and the classification tasks. In this section, we provide background on metrics used to evaluate performance with respect to these perception tasks. Let the evaluation dataset $\mathcal{D} = \{(f_i, b_i, d_i, x_i)\}_{i=1}^N$ consist of N objects across m image frames $F = \{F_1, \dots, F_m\}$. For each object, $f_i \in F$ represents the image frame token, b_i specifies the bounding box coordinates, d_i denotes the distance of the object to ego, and x_i denotes the true class of the object. When a specific object detection algorithm is evaluated on $\mathcal{D}$, each object has a predicted bounding box, $\tilde{b}_i$, and predicted object class $\tilde{x}_i$. We store these predictions in the set $\mathcal{E} = \{(\tilde{b}_i, \tilde{x}_i)\}_{i=1}^N$.

Definition 2.8 (Confusion Matrix)**.** Let $\mathcal{D}$ be an evaluation set of objects and $\mathcal{E}$ be the corresponding predictions by an object detection algorithm. Let $\mathcal{C} = \{c_1, \dots, c_n\}$ be a set of object classes in $\mathcal{D}$, and let n denote the cardinality of $\mathcal{C}$. The confusion matrix corresponding to the classes $\mathcal{C}$ and dataset $\mathcal{D}$, and predictions $\mathcal{E}$ is an $n \times n$ matrix $\texttt{CM}(\mathcal{C}, \mathcal{E}, \mathcal{D})$ with the following properties:

- $\texttt{CM}(\mathcal{C}, \mathcal{E}, \mathcal{D})[i, j]$ is the element in row i and column j of $\texttt{CM}(\mathcal{C}, \mathcal{E}, \mathcal{D})$, and represents the number of objects that are predicted to have class label $c_i \in \mathcal{C}$, but have the true class label $c_j \in \mathcal{C}$, and
- the sum of the j^{th}-column of $\texttt{CM}(\mathcal{C}, \mathcal{E}, \mathcal{D})$ is the total number of objects in $\mathcal{D}$ belonging to the class $c_j \in \mathcal{C}$.

Several performance metrics for object detection and classification such as true positive rate, false positive rate, precision, accuracy, and recall can be derived from the confusion matrix[50, 51, 61].

Definition 2.9 (Precision [50])**.** Given the confusion matrix $\texttt{CM}$ for a multi-class classification, the *precision* corresponding to class c_i is:

$$P(i) = \frac{\texttt{CM}(i, i)}{\texttt{CM}(i, i) + \frac{\sum_{j \neq i} \texttt{CM}(i,j) |\mathcal{D}_j|}{\sum_{j \neq i} |\mathcal{D}_j|}}, \tag{2.8}$$

where $\frac{\sum_{j \neq i} \texttt{CM}(i,j) |\mathcal{D}_j|}{\sum_{j \neq i} |\mathcal{D}_j|}$ is the false positive rate for class c_i, and $\texttt{CM}(i, i)$ is the true positive rate for class c_i.

Definition 2.10 (Recall [50])**.** Given the confusion matrix $\mathtt{CM}$ for a multi-class classification, the *recall* corresponding to class label c_i is:

$$R(i) = \frac{\mathtt{CM}(i,i)}{\mathtt{CM}(i,i) + \sum_{j \neq i} \mathtt{CM}(j,i)}, \tag{2.9}$$

where $\sum_{j \neq i} \mathtt{CM}(j,i)$ is the false negative rate for class c_i.

Maximizing precision typically corresponds to minimizing false positives while maximizing recall corresponds to minimizing false negatives. However, there is an inherent trade-off in minimizing both false positives and false negatives for classification tasks [50], and often, a good operating point is found in an *ad-hoc* manner. Typically, safety-critical systems are designed for optimizing recall, but as we will show in Section 2.6, this is not always the best strategy to satisfy formal requirements.

Remark 2.1. In this chapter, we use $c_n =$ (referring to the background class) as an auxiliary class label in the construction of confusion matrices. If an object has the true class label c_i but is not detected by the object detection algorithm, then this gets counted in $\mathtt{CM}(\mathcal{C}, \mathcal{E}, \mathcal{D})(n,i)$ as a false negative with respect to class c_i. If the object was not labeled originally, but is detected and classified to have class label c_i, then it gets counted in $\mathtt{CM}(\mathcal{C}, \mathcal{E}, \mathcal{D})(i,n)$ as a false negative of the `empty`class. We expect that in a properly annotated dataset, false negatives $\mathtt{CM}(\mathcal{C}, \mathcal{E}, \mathcal{D})(i,n)$ to be small. We ignore these extra detections in constructing the confusion matrix because by not being annotated, they are not relevant to the evaluation of object detection models.

Definition 2.11 (Transition Probability for Confusion Matrices)**.** Let $s_1 = (s_{1,a}, x_e)$, $s_2 = (s_{2,a}, x_e) \in S$ be two states of the overall system, x_e be the true class label of the environment, and $\mathtt{CM}$ be the known confusion matrix associated with the agent's perception model. Let $O(s_1, s_2)$ denote the set of environment observations $y_e \in V_E$ that result in the agent controller transitioning from $s_{1,a}$ to $s_{2,a}$. The transition probability $Pr : S \times S \to [0,1]$ is defined as,

$$Pr(s_1, s_2) := \sum_{y_e \in O(s_1, s_2)} \mathtt{CM}(y_e, x_e)\,. \tag{2.10}$$

From the definition, and consequently structure, of the confusion matrix in Definition 2.8, it is trivial to check that $\sum_{s_2 \in S} Pr(s_1, s_2) = 1$. Therefore, the transition probability between any two states is always in the range $[0,1]$.

Class-labeled, distance-parametrized Confusion Matrix

This performance metric builds on the class-labeled confusion matrix defined in Definition 2.8. As denoted previously, let $\mathcal{C} = \{c_1, \ldots, c_n\}$ be the set of different classes of objects in dataset $\mathcal{D}$. For every object in $\mathcal{D}_k$, the predicted class of the object will be one of the class labels $c_1, \ldots, c_n$. For each distance interval z_k, we define the class-labeled confusion matrix as $\texttt{CM}_{\text{class},k} := \texttt{CM}(\mathcal{C}, \mathcal{E}_k, \mathcal{D}_k)$. Algorithm 2 shows the construction of the class-labeled, distance-parametrized confusion matrix. Therefore, the outcomes of the object detection algorithm will be defined by the set $Outc = \{c_1, \ldots, c_n\}^m$, where m is the total number of objects in the true environment in the distance interval z_k. The tuple $(Outc, 2^{Outc})$ forms a σ-algebra for defining a probability function over the class-labeled confusion matrix $\texttt{CM}_{\text{class},k}$. Similar to the definition of a probability function, for every class label c_j, the probability function $\mu_{\text{class},k}(\cdot, c_j) : Outc \to [0, 1]$ is defined as follows,

$$\mu_{\text{class},k}(c_i, c_j) := \frac{\texttt{CM}_{\text{class},k}(c_i, c_j)}{\sum_{l=1}^{n} \texttt{CM}_{\text{class},k}(c_l, c_j)}. \tag{2.11}$$

Algorithm 1: Class-labeled Confusion Matrix

```
1: procedure ClassCM(Dataset $\mathcal{D} = \{(f_i, b_i, d_i, x_i)\}_{i=1}^{N}$, Classes $\mathcal{C}$, Distance Parameters $\{D_k\}_{k=0}^{k_{\max}}$)
2:     From $\{D_k\}_{k=0}^{k_{\max}}$, define distance intervals $\{z_k\}_{k=1}^{k_{\max}}$
3:     Run object detection algorithm to get predictions $\mathcal{E}$,
4:     Initialize $\mathcal{D}_1, \ldots, \mathcal{D}_{k_{\max}}$ as empty sets
5:     Initialize $\mathcal{E}_1, \ldots, \mathcal{E}_{k_{\max}}$ as empty sets
6:     for $(f_i, b_i, d_i, x_i) \in \mathcal{D}$ do
7:         if $d_i \in z_k$ then
8:             $\mathcal{D}_k \leftarrow \mathcal{D}_k \cup \{(f_i, b_i, d_i, x_i)\}$
9:             $\mathcal{E}_k \leftarrow \mathcal{E}_k \cup \{(\tilde{b}_i, \tilde{x}_i)\}$
10:    for $k \in \{0, \ldots, k_{\max}\}$ do
11:        Denote $\texttt{CM}_{\text{class}}(\mathcal{C}, \mathcal{E}_k, \mathcal{D}_k)$ as $\texttt{CM}_{\text{class},k}$
12:        $\texttt{CM}_{\text{class},k} \leftarrow$ zero matrix
13:        for $f_i \in \{f_1, \ldots, f_m\}$ do                    ▷ Loop over images
14:            for object in $\mathcal{D}_k$ do
15:                $c_i \leftarrow$ Predicted class label of object
16:                $c_j \leftarrow$ True class label of object in $\mathcal{E}_k$
17:                $\texttt{CM}_{\text{class},k}(c_i, c_j) \leftarrow \texttt{CM}_{\text{class},k}(c_i, c_j) + 1$
18:    $\texttt{CM}_{\text{class}}(\mathcal{C}, \mathcal{E}, \mathcal{D}) = \{\texttt{CM}_{\text{class}}(\mathcal{C}, \mathcal{E}_k, \mathcal{D}_k)\}_{k=0}^{k_{\max}}$
19:    return $\texttt{CM}_{\text{class}}(\mathcal{C}, \mathcal{E}, \mathcal{D})$
```

Definition 2.12 (Transition probability function for class-labeled confusion matrix)**.** Let the true environment be represented as a tuple x_e corresponding to class labels in the region z_k (class labels can be repeated in a tuple x_e when multiple objects of the same class are in region z_k). Let $s_{a,1}, s_{a,2} \in S$ be states of the car, and let $O(s_1, s_2)$ denote the set of all predictions of the environment that prompt the system to transition from $s_1 = (s_{a,1}, x_e)$ to $s_2 = (s_{a,2}, x_e)$. Likewise, the tuple y_e represents the object detection model's predictions of the environment. Then, the transition probability function from state s_1 to s_2 is defined as follows,

$$Pr(s_1, s_2) := \sum_{y_e \in O(s_1, s_2)} \prod_{i=1}^{|y_e|} \mu_{\text{class},k}(y_e(i), x_e(i)). \tag{2.12}$$

For both transition probability functions (2.12) and (2.14), we can check (by construction) that $\forall s_1 \in S, \sum_{s_2} Pr(s_1, s_2) = 1$. In the running example, if the crosswalk were to have another pedestrian and a non-pedestrian obstacle, then the probability of detecting each object is considered independently of the others. This results in the product of probabilities $\mu_{\text{class},k}(\cdot, x_e(i))$ in equation (2.12).

Proposition-labeled Confusion Matrix

In several instances, the high-level planner does not necessarily require correct detection of every single object in a frame to make a correct decision. For instance, for the planner to decide to stop for a cluster of pedestrians 20m away, knowledge that there are pedestrians, and not necessarily the exact number of pedestrians is sufficient for the planner to decide to slow down. Accounting for this in quantitative system-level evaluations would make the analysis less conservative. Therefore, we introduce the notion of using atomic propositions as class labels in the confusion matrix instead of the object classes themselves.

Let p_i be the atomic proposition: "*there exists an object of class* $c_i \in \mathcal{C}$," and let $\mathcal{P} = \{p_1, \ldots, p_n\}$ denote the set of all atomic propositions. Let $D_0 < D_1 < \ldots < D_k < \ldots < D_{k_{\text{max}}}$ denote progressively increasing distances from the autonomous vehicle. Let $\mathcal{D}_k \subset \mathcal{D}$ be the subset of the dataset that includes objects that are in the distance interval $z_k = (D_{k-1}, D_k)$ from the autonomous system. Let $\mathcal{E}_k$ denote the predictions of the object detection algorithm corresponding to dataset $\mathcal{D}_k$. For each parameter k, we define the proposition-labeled confusion matrix $\texttt{CM}_{\text{prop},k} = \texttt{CM}_{\text{prop}}(2^{\mathcal{P}}, \mathcal{E}_k, \mathcal{D}_k)$ where the classes are characterized by the powerset of atomic propositions $2^{\mathcal{P}}$. Algorithm 1 shows the construction of the proposition-labeled confusion matrix.

The true environment is associated with a set of atomic propositions that are a subset of $\mathcal{P}$ that evaluates to true. Suppose, there is a pedestrian and a trash can in the distance interval z_k from the ego, then the true class label is $\{p_{\texttt{ped}}, p_{\texttt{obs}}\}$ in the distance-parametrized confusion matrix $\texttt{CM}_{\text{prop},k}$. Note that for every possible environment, there is only one corresponding class in the proposition-labeled confusion matrix. Thus, for a given true environment, the predicted class of the environment at distance interval z_k could be any element of the set $2^{\mathcal{P}}$. Therefore, at each time step, the set of detection outcomes is $Outc = 2^{\mathcal{P}}$.

Algorithm 2: Proposition-labeled Confusion Matrix

```
1:  procedure PropCM(Dataset $\mathcal{D} = \{(f_i, b_i, d_i, x_i)\}_{i=1}^N$, Classes $\mathcal{C}$, Distance Parameters $\{D_k\}_{k=0}^{k_\text{max}}$)
2:      From $\{D_k\}_{k=0}^{k_\text{max}}$, define distance intervals $\{z_k\}_{k=1}^{k_\text{max}}$
3:      Run object detection algorithm to get predictions $\mathcal{E}$,
4:      Initialize $\mathcal{D}_1, \ldots, \mathcal{D}_{k_\text{max}}$ as empty sets
5:      Initialize $\mathcal{E}_1, \ldots, \mathcal{E}_{k_\text{max}}$ as empty sets
6:      for $(f_i, b_i, d_i, x_i) \in \mathcal{D}$ do
7:          if $d_i \in z_k$ then
8:              $\mathcal{D}_k \leftarrow \mathcal{D}_k \cup \{(f_i, b_i, d_i, x_i)\}$
9:              $\mathcal{E}_k \leftarrow \mathcal{E}_k \cup \{(\tilde{b}_i, \tilde{x}_i)\}$
10:     for $c_j \in \mathcal{C}$ do
11:         $p_j \equiv$ "there exists an object of class $c_j$"
12:     $\mathcal{P} \leftarrow \bigcup_j \{p_j\}$                    ▷ Set of atomic propositions
13:     for $k \in \{1, \ldots, k_\text{max}\}$ do
14:         Denote $\texttt{CM}_\text{prop}(2^{\mathcal{P}}, \mathcal{E}_k, \mathcal{D}_k)$ as $\texttt{CM}_{\text{prop},k}$
15:         $\texttt{CM}_{\text{prop},k} \leftarrow$ zero matrix
16:         for $f \in F$ do                                 ▷ Loop over image frames
17:             Group objects in $\mathcal{D}_k$ with image token $f$.
18:             Group predictions in $\mathcal{E}_k$ with image token $f$.
19:             $P_i \leftarrow$ Predicted set of propositions
20:             $P_j \leftarrow$ True set of propositions
21:             $\texttt{CM}_{\text{prop},k}(P_i, P_j) \leftarrow \texttt{CM}_{\text{prop},k}(P_i, P_j) + 1$
22:     $\texttt{CM}_\text{prop}(2^{\mathcal{P}}, \mathcal{E}, \mathcal{D})$= $\{\texttt{CM}_\text{prop}(2^{\mathcal{P}}, \mathcal{E}_k, \mathcal{D}_k)\}_{k=0}^{k_\text{max}}$
23:     return $\texttt{CM}_\text{prop}(2^{\mathcal{P}}, \mathcal{E}, \mathcal{D})$
```

The tuple $(Outc, 2^{Outc})$ forms a σ-algebra for defining a probability function over the proposition-labeled confusion matrix. Since the set $Outc$ is countable, we can define a probability function $\mu : Outc \to [0, 1]$ such that $\sum_{e \in Outc} \mu(e) = 1$. For a distance-parametrized confusion matrix $\texttt{CM}_{\text{prop},k}$ with class labels in the set $Outc$, and for every true environment class label P_j, define a probability function

$\mu_{\text{prop},k}(\cdot, P_j) : Outc \rightarrow 2^{Outc}$ as follows,

$$\mu_{\text{prop},k}(P_i, P_j) = \frac{\texttt{CM}_{\text{prop},k}[P_i, P_j]}{\sum_{l=1}^{|2^{\mathcal{P}}|} \texttt{CM}_{\text{prop},k}[P_l, P_j]}, \quad \forall P_i \in 2^{\mathcal{P}}, \tag{2.13}$$

where $\texttt{CM}_{\text{prop},k}[P_i, P_j]$ is the element of the confusion matrix $\texttt{CM}_{\text{prop},k}$ with predicted class label P_i and true class label P_j.

That is, for every confusion matrix $\texttt{CM}_{\text{prop},k}$ where $k \in \{1, \ldots, k_{\text{max}}\}$, we define a total of $2^{|\mathcal{P}|}$ different probability functions, one for each possible true environmentP_j. Thus, the probability function $\mu_{\text{prop},k}$ that characterizes the probability of detecting an environment satisfying propositions P_i, given that the true environment at z_k satisfies propositions P_j. This helps to formally define the state transition probability of the overall system as follows.

Definition 2.13 (Transition probability function for proposition-labeled confusion matrices)**.** Let x_e be the true environment state corresponding to propositions P_j evaluating to true, and let $s_{a,1}, s_{a,2} \in S$ be states of the car. Let $O(s_1, s_2)$ denote the set of all predictions of the environment that prompt the system to transition from $s_1 = (s_{a,1}, x_e)$ to $s_2 = (s_{a,2}, x_e)$. At state s_1, let z_k be the distance interval of objects in the environment causing the agent to transition from $s_{a,1}$ to $s_{a,2}$. The corresponding confusion matrix is $\texttt{CM}_{\text{prop},k}$. Then, the transition probability from state s_1 to s_2 is defined as follows,

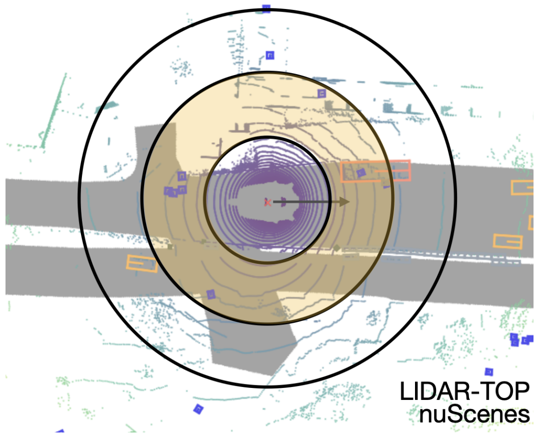
$$Pr(s_1, s_2) := \sum_{P_i \in O(s_1, s_2)} \mu_{\text{prop},k}(P_i, P_j). \tag{2.14}$$

For simplicity, we assume that objects at a specific distance interval influence the agent to transition from $s_{a,1}$ to $s_{a,2}$. However, Definition 2.13 can be extended to cases in which objects at multiple distances can influence transitions.

Choosing Proposition Labels

Generally, the set of atomic propositions $\mathcal{P}$ depends on the logic used by the planner to trigger different operation modes. In the running example, the planner outputs different actions depending on the environment, i.e., pedestrian or other objects. If the planner responds differently to other types of objects, e.g cars, bicycles, cones, those should be included in the set of atomic propositions $\mathcal{P}$. Thus, our approach can generalize to a wider range of scenarios by adapting the set $\mathcal{P}$ accordingly.

In particular, proposition labels of the confusion matrix can be chosen to match the set of environment observations S_E that are acceptable inputs to the controller (Definition 2.7). Proposition labels can be propositional formulas comprising of logical connectives, but not any temporal operators.



		True state of environment			
		$x_e \vDash ped$	$x_e \vDash obj$	$x_e \vDash \{ped, obj\}$	$x_e \vDash emp$
Predicted state	$y_e \vDash ped$	10	1	1	3
	$y_e \vDash obj$	1	25	20	1
	$y_e \vDash \{ped, obj\}$	2	2	11	2
	$y_e \vDash emp$	3	1	3	10

$\texttt{CM}_{prop}$

Figure 2.2: Proposition-labeled confusion matrices when evaluations are grouped solely by distance. Observe that detecting one pedestrian in the highlighted distance zone will amount to the proposition "there is a pedestrian" evaluating to true. This would not be an appropriate evaluation for the driving task.

Grouping Objects for Evaluation

The choice of evaluation metric for the detection model depends on the observations received by the downstream planner, and how the planner processes these observations to to control the system. Proposition labels are defined over objects in a group, and each group accounts for a single evaluation of the model. For meaningful evaluations of the perception system, the grouping of objects into propositional formulas should be at the right fidelity for the planning module. For example, in a robotic system that has a vision-based perception comprising of only forward-facing cameras and a planner tasked with driving forward, grouping objects by distance to the ego might be sufficient for effectively evaluating the perception with the system-level task. However, in robotic systems equipped with LiDAR sensors and tasked with navigating arbitrarily, the same evaluations might no longer be meaningful. In particular, since LiDAR sensor outputs 360° observations, grouping objects solely by ego-centric distance will be too coarse from a planning standpoint (see Figure 2.2). We denote this proposition-labeled confusion matrix as $\texttt{CM}_{prop,seg}$.

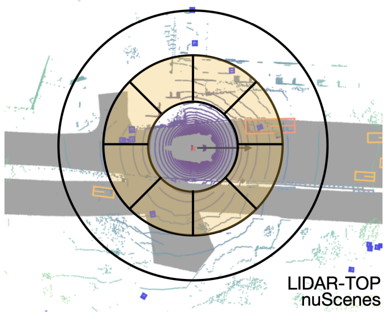


Figure 2.3: Grouping evaluations at the same level of abstraction used by the high-level planner. Evaluating the proposition "there is a pedestrian" in each segment of the distance zone.

The perception system localizes objects in the environment and returns detections in $\mathbb{R}^3$ upto some finite distance around the ego. Let $G = \{C_1, \dots, C_m\}$ be a finite, discretized ego-centric abstraction of $\mathbb{R}^3$. The labeling functions of the system and environment allow for mapping the states of the system and environment to G. Typically in high-level planning, detections from the perception system are mapped onto the discretized abstraction G, and this information is used by the planner to decide on next actions. Therefore, evaluation of the perception system using proposition labels at the fidelity of G would be as follows. Instead of using distance to group objects into radius bands (lines 17-21 of Algorithm 2), we group objects according to the ego-centric abstraction G, and evaluate proposition labels for each cell (see Figure 2.3) for an illustration).

2.5 Markov Chain Analysis

Our approach to solving Problem 2.1 is based on constructing a Markov chain that represents the state evolution of the overall system, taking into account the control logic as well as detection errors. This Markov chain is constructed for a particular true state of the environment. Given a Markov chain for the state evolution of the system, it is then straightforward to compute the probability of satisfying a temporal logic formula on the Markov chain from an arbitrary initial state [60]. Probabilistic model checking can be used to compute the probability that the Markov chain satisfies the formula using existing tools such as PRISM [62] and Storm [63], which have been demonstrated to successfully analyze systems modeled by Markov chains with billions of states. In addition to the efficient off-the-shelf probabilistic model checkers, our approach is computationally tractable because constructing the Markov chain from the confusion matrix is linear in the number of classes used for perception.

For each confusion matrix, we can synthesize a corresponding Markov chain of the system state evolution as per Algorithm 3. Using off-the-shelf probabilistic model checkers such as Storm [63], we can compute the probability that the trace of a system satisfies its requirement, $\mathbb{P}(s_0 \models \varphi)$, by evaluating the probability of satisfaction of the requirement φ on the Markov chain. Let $O(x_e)$ be the set of all possible predictions of true environment state x_e by the the object detection model. The system controller $K : S \times O(x_e) \to S$ accepts as inputs the current state of the agent and the environment, $s_0 \in S$, and the environment state predictions $y_e \in O(x_e)$ from object detection. Based on the predictions, it actuates the agent resulting in the end state $s_f \in S$. At each time step, the agent makes a new observation of the environment (y_e) and chooses a control action corresponding to y_e.

Remark 2.2. Markov chain construction aids in evaluating the overall system. In future work, we plan to address the issue of tracking, in which perception errors are tracked over multiple temporal frames.

Definition 2.14 (Labeled Markov Chain [60])**.** A discrete-time labeled Markov chain is a tuple $\mathcal{M} = (S, \mathbf{P}, \iota_{init}, AP, L)$, where S is a non-empty, countable set of states, $\mathbf{P} : S \times S \to [0,1]$ is the *transition probability function* such that for all states $s \in S$, $\Sigma_{s' \in S}\mathbf{P}(s, s') = 1$, $\iota_{init} : S \to [0,1]$ is the initial distribution such that $\Sigma_{s \in S}\iota_{init}(s) = 1$, AP is a set of atomic propositions, and $L : S \to 2^{AP}$ is a labeling function.

The σ-algebra of Markov chain $\mathcal{M}$ is $(Paths(\mathcal{M}, 2^{\mathcal{C}_\mathcal{M}}))$, where $\mathcal{C}_\mathcal{M} = \{Cyl(\hat{\pi}) | \hat{\pi} \in Paths_{fin}(\mathcal{M})\}$ [60]. Let $\mathcal{S}_\mathcal{M}(\varphi)$ denote all paths of the MC $\mathcal{M}$ in $Paths_{fin}(\mathcal{M}) \cap Paths(\mathcal{M})$.

Definition 2.15 (Probability on a Markov Chain)**.** Given an LTL formula φ over AP, a true state of the environment, x_e, an initial system state, $s_0 = (s_{a,0}, x_e)$, and a Markov chain $\mathcal{M}$ describing the dynamics of the overall system, we denote the probability that the system will satisfy φ starting from state s_0 as $\mathbb{P}_\mathcal{M}(s_0 \models \varphi_s)$. This probability can be computed using standard techniques as described in [60].

Algorithm 3: Markov Chain Construction

```
1: procedure Labeled Markov Chain($S, x_e, s_0, K,$ CM)
Input: Product states $S$, True environment $x_e$, Initial condition $s_0 := (s_{a,0}, x_e) \in S$, Controller $K$ synthesized for $s_0$ and $\varphi$, Confusion matrix CM,
Output: Markov Chain $\mathcal{M}$ carrying the probability of detection error
2:     $Pr(s, s') = 0, \forall s, s' \in S$
3:     $K \leftarrow$ Initialize Controller for initial state $s_0$
4:     for $s_i \in S$ do
5:         $\iota_{init}(s_i) = 1$                                ▷ Initial Distribution
6:         for $y_e \in O(x_e)$ do
7:             $s_f \leftarrow K(s_i, y_e)$                        ▷ Controller
8:             Identify $z_k$ according to Definitions 2.12, 2.13
9:             $\mu_{\text{class},k}, \mu_{\text{prop},k} \leftarrow$ Equations (2.11), (2.13).
10:            if class-labeled then
11:                $p \leftarrow \prod_{i=1}^{|y_e|} \mu_{\text{class},k}(y_e(i), x_e(i))$
12:            if proposition-labeled then
13:                $P_j \leftarrow$ Propositions for true $x_e$
14:                $P_i \leftarrow$ Propositions for predicted $y_e$
15:                $p \leftarrow \mu_{\text{prop},k}(P_i, P_j)$
16:            $Pr(s_i, s_f) \leftarrow Pr(s_i, s_f) + p$
17:    return $\mathcal{M} = (S, Pr, \iota_{init}, AP, L)$
```

Proposition 2.1. Given φ as a temporal logic formula over the agent and the environment states, true state of the environment x_e, agent initial state $s_{a,0}$, and a Markov chain $\mathcal{M}$ constructed via Algorithm 3, then $\mathbb{P}(s_0 \models \varphi)$ is equivalent to computing $\mathbb{P}_{\mathcal{M}}(s_0 \models \varphi)$, where $s_0 = (s_{a,0}, x_e)$.

Proof. We begin by considering the transition probabilities Pr and the transition probabilities on the Markov chain $\mathbf{P}$. Since misclassification errors are the only source of non-determinism in the evolution of the agent state, by construction, we have that $\mathbf{P}(s_i, s_j) = Pr(s_i, s_j)$ for some $s_i, s_j \in S$. Next, we compare the σ-algebra of Markov chain $\mathcal{M}$ with the σ-algebra associated with state s_0. By construction of the Markov chain, observe that any path $p \in Paths(s_0)$ is also a path on the MC $\mathcal{M}$, $p \in Paths(\mathcal{M})$, and as a result $\mathcal{C}_{s_0} \subset \mathcal{C}_{\mathcal{M}}$. Similarly, by construction, there is no finite trace on the Markov chain starting from s_0, $\sigma(s_0) \in \mathcal{S}_{\mathcal{M}}$ that is not in $\mathcal{S}(\varphi)$.

$$\begin{aligned}
\mathbb{P}(s_0 \models \varphi) &= \sum_{\sigma(s_0)\in\mathcal{S}(\varphi)} \mathbb{P}_{s_0}(Cyl(\sigma(s_0))) \\
&= \sum_{\sigma(s_0)\in\mathcal{S}(\varphi)} \prod_{0\leq i<n} Pr(\sigma_i, \sigma_{i+1}) \\
&= \sum_{\sigma(s_0)\in\mathcal{S}(\varphi)} \prod_{0\leq i<n} \mathbf{P}(\sigma_i, \sigma_{i+1}) \\
&= \sum_{\sigma(s_0)\in\mathcal{S}_{\mathcal{M}}(\varphi)} \prod_{0\leq i<n} \mathbb{P}_{\mathcal{M}}(Cyl(\sigma(s_0))) \\
&= \mathbb{P}_{\mathcal{M}}(s_0 \models \varphi)
\end{aligned}$$

□

2.6 Experiments

In this section, we conduct various experiments the car-pedestrian example in simulation. We present system-level evaluations for the car pedestrian example for various types of confusion matrices.

Fundamental Tradeoffs. Even in the simplest setting of the traditional class-based confusion matrix, we can show that these quantitative evaluations highlight fundamental tradeoffs in detection, and that the right operating point must be informed by system-level specifications as well as the down-stream control logic. Often in autonomous driving applications, maximizing recall is prioritized over precision for safety purposes. In our example, maximizing recall would correspond with increasing tendency to stop at C_{k-1}, even if $x_e \neq \texttt{ped}$. In Figure 2.4, we show how varying precision/recall affects the probability of satisfaction for $V_{max} = 6$. These precision/recall pairs were chosen to reflect the general precision/recall tradeoff trends for classification tasks [50]. For the results presented in this chapter, we construct a confusion matrix as a function of precision (p) and recall (r) as shown in $\texttt{CM}(p, r)$ of Table 2.1, and are in reference to the class label *ped*. In Table 2.1, TP, FP, TN, FN are the number of true positives, false positives, true negatives, and false negatives, respectively, of the ped class label. These are derived from precision p and recall r as follows,

$$\begin{aligned}
&\texttt{TP} = r\,, && \texttt{FP} = \texttt{TP}(\frac{1}{p} - 1)\,, \\
&\texttt{TN} = 2 - \texttt{FP}\,, && \texttt{FN} = 1 - \texttt{TP}\,.
\end{aligned} \tag{2.15}$$

Note that this is one of many possible confusion matrices that could be constructed; we have chosen one of them for illustration, and we use it consistently across all

Table 2.1: Confusion matrices used in simulation for various precision-recall pairs, where TP, TN, FP, FN are given according to equation (2.15).

Predicted	True ($\texttt{CM}(p, r)$)		
	ped	obs	empty
ped	TP	FP/2	FP/2
obs	FN/2	4TN/10	TN/10
empty	FN/2	TN/10	4TN/10

precision/recall pairs.

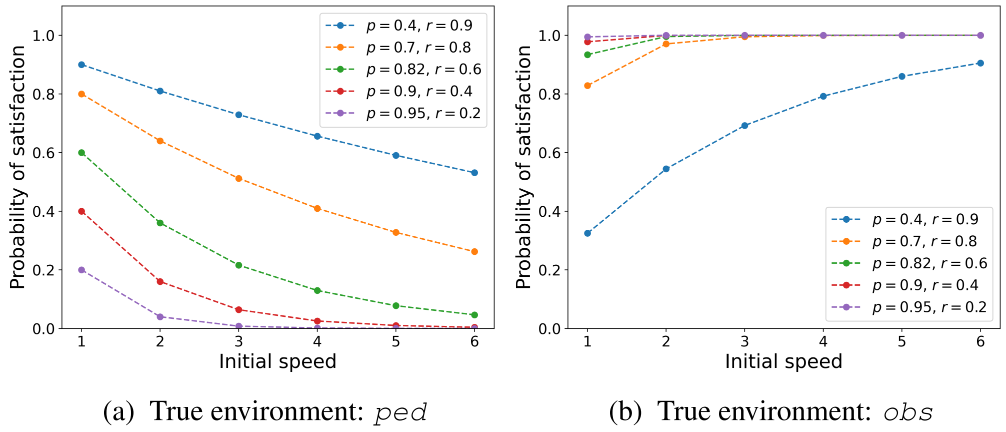


Figure 2.4: For class-labeled confusion matrices with precision-recall values derived according to Table 2.1. (a) Satisfaction probabilities that the car stops at C_{k-1} for $x_e = \texttt{ped}$ under various initial speeds and maximum speeds V_{max} such that $1 \leq V_{max} \leq 6$. (b) Satisfaction probabilities that the car does not stop at C_{k-1} for $x_e = \texttt{obs}$ under various initial speeds and maximum speeds V_{max} such that $1 \leq V_{max} \leq 6$.

nuScenes Dataset: We choose nuScenes [64] to illustrate the metrics introduced in this chapter on a real-world dataset. We choose a state-of-the-art PointPillars detection model for nuScenes that uses the LiDAR modality [65, 66]. The pre-trained model[1] is evaluated on the validation split of the full nuScenes dataset. The resulting dataset has 6019 pointcloud samples, with annotated objects common to urban settings such as pedestrians, cars, trucks, among others. For this detection model, we tabulate the evaluation results according to the various confusion matrices discussed so far. For the car-pedestrian example and its controller described previously, we compute the system-level guarantees, i.e., the probability that the car will satisfy the safety requirements in equations (2.1)-(2.3), given the confu-

[1]Available open source at this Github repositoryhttps://github.com/open-mmlab/mmdetection3d/tree/main/configs/pointpillars.

Prediction	True Label								
	$1 \le d \le 10$			$11 \le d \le 20$			$21 \le d \le 30$		
	`ped`	`obs`		`ped`	`obs`		`ped`	`obs`	
`ped`	1849	11	369	5443	87	963	4290	271	943
`obs`	56	5697	47	45	12406	354	191	12939	762
	1002	621	6117	2734	1949	12668	2406	3647	8969
	$31 \le d \le 40$			$41 \le d \le 50$			$51 \le d \le 60$		
	`ped`	`obs`		`ped`	`obs`		`ped`	`obs`	
`ped`	3302	382	213	0	0	0	0	0	0
`obs`	358	10670	345	0	6981	252	0	0	0
	1824	4358	1285	0	4346	709	0	0	0

Table 2.2: Class labeled confusion matrix, parametrized by distance computed from the full nuScenes dataset for the Pointpillars model

sion matrices from various evaluations of the detection model. Each discrete state corresponds abstracts a 1m distance on the road.

Each scene is 20 seconds long, with 3D object annotations made at 2 Hz for 23 different classes. All objects with nuScenes annotation “human” are clustered under the class `ped`, and all objects annotated as “vehicle”, static obstacles, and moving obstacles are annotated as `obs`. We use all 40 pointcloud frames from the `LIDAR-TOP` sensor in each scene to form our dataset $\mathcal{D}$. The LiDAR sweeps accompanying each scene provides distances of annotated objects from the ego vehicle. We use the birds-eye-view to compare predicted bounding boxes to the ground truth, comparing for both l_2-norm in x, y-positions as well as orientation error. These evaluations are used to construct the (distance-parametrized) class-labeled and proposition-labeled confusion matrices from Algorithms 1 and 2 with 10m distance intervals with parameters $D_0 = 0$ and $D_{k_{max}} = 100\text{m}$. The class-labeled and proposition-labeled confusion matrices for each distance bin are listed in Tables 2.2 and 2.3, respectively.

For proposition-labeled confusion matrices, ground truth annotations and predictions are grouped according to an occupancy patch that roughly covers the area occupied by the ego. Concretely, the radius band $D_k = (z_k, z_{k+1})$ is split into occupancy patches covering the area every $\theta = \frac{z_k}{2.5}$ radians. The arc length of 2.5 m is a user-specified parameter; here, it is chosen to roughly approximate the width of a car. Table 2.4 is the proposition-labeled confusion matrix, where in addition to distance, evaluations are grouped by the occupancy patch size. There is a considerable difference between the proposition-labeled confusion matrix that is and is not

Prediction	True Label							
	$1 \leq d \leq 10$				$11 \leq d \leq 20$			
	{}	{ped}	{obs}	{ped, obs}	{}	{ped}	{obs}	{ped, obs}
{}	0	141	94	6	0	110	139	24
{ped}	54	373	9	17	34	363	18	81
{obs}	20	3	2122	210	36	1	2301	388
{ped, obs}	0	3	104	415	1	18	233	1400
	$21 \leq d \leq 30$				$31 \leq d \leq 40$			
	{}	{ped}	{obs}	{ped, obs}	{}	{ped}	{obs}	{ped, obs}
{}	0	84	253	27	0	106	331	48
{ped}	34	246	34	74	31	241	45	128
{obs}	25	14	2109	443	17	12	2200	489
{ped, obs}	8	37	343	1565	0	42	245	1240
	$41 \leq d \leq 50$				$51 \leq d \leq 60$			
	{}	{ped}	{obs}	{ped, obs}	{}	{ped}	{obs}	{ped, obs}
{}	0	0	905	0	0	0	0	0
{ped}	0	0	0	0	0	0	0	0
{obs}	42	0	3396	0	0	0	0	0
{ped, obs}	0	0	0	0	0	0	0	0

Table 2.3: Proposition labeled confusion matrix, parametrized by distance computed from the full nuScenes dataset for the pretrained Pointpillars model

grouped according to an occupancy patch that is planner consistent. For example, consider the label $\{ped\}$ in the distance range $1 \leq d \leq 10$ in in the ungrouped (see Table 2.3) and the grouped proposition labeled confusion matrices (see Table 2.4). The true positive rate of matching the label is higher when atomic propositions are not grouped (see Tables 2.3 and 2.4). This is because the proposition must match in every occupancy patch, which is finer, as opposed to every radius band.

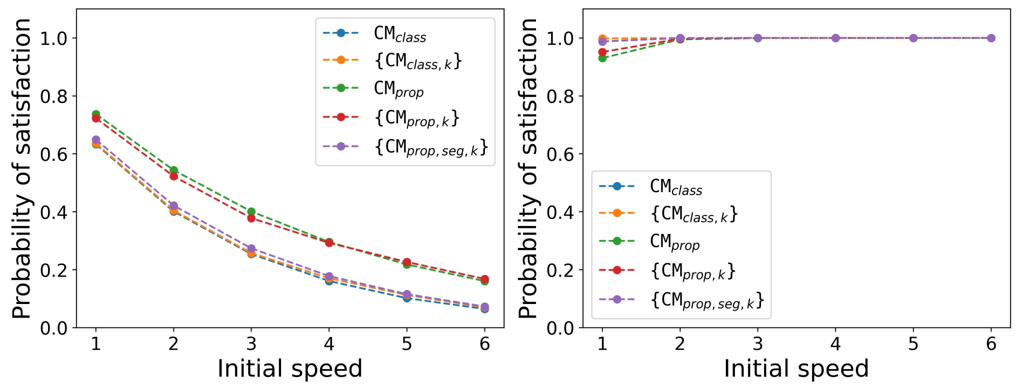
The satisfaction probabilities for the pedestrian case is shown in Figure 2.5a. The system-level satisfaction probability in the case of the true environment not having a pedestrian is given in Figure 2.5b. The full class and proposition labeled confusion matrices are given in Tables 2.5 and 2.6, respectively. The code for this chapter is given in the Python package, TRELPY and is available on GitHub[2]. In both the class labeled and proposition labeled confusion matrices, notice that after a distance of $50m$, there are no more detections output by the model, beyond which the nuScenes LiDAR data is sparse and cannot be reliably inferred from [64]; this is also nuScenes threshold for evaluation and objects beyond $50m$ are in the far-field and not annotated [67].

Figure 2.5 illustrates the importance of choosing perception metrics at the right level of fidelity. The proposition-labeled confusion matrix (green curve) and its dis-

[2]https://github.com/IowaState-AutonomousSystemsLab/TRELPy

Prediction	True Label							
	$1 \leq d \leq 10$				$11 \leq d \leq 20$			
	{}	{ped}	{obs}	{ped, obs}	{}	{ped}	{obs}	{ped, obs}
{}	0	344	280	6	0	1689	1658	10
{ped}	145	649	10	11	582	3183	77	28
{obs}	29	8	4006	133	262	29	11241	94
{ped, obs}	2	3	44	256	20	12	36	175
	$21 \leq d \leq 30$				$31 \leq d \leq 40$			
	{}	{ped}	{obs}	{ped, obs}	{}	{ped}	{obs}	{ped, obs}
{}	0	1615	3150	16	0	1316	3912	13
{ped}	697	2987	260	17	188	2368	353	22
{obs}	658	149	11878	58	317	286	9978	37
{ped, obs} 9	29	64	71	2	30	26	57	
	$41 \leq d \leq 50$				$51 \leq d \leq 60$			
	{}	{ped}	{obs}	{ped, obs}	{}	{ped}	{obs}	{ped, obs}
{}	0	0	4069	0	0	0	0	0
{ped}	0	0	0	0	0	0	0	0
{obs}	245	0	6706	0	0	0	0	0
{ped, obs}	0	0	0	0	0	0	0	0

Table 2.4: Proposition labeled confusion matrix, in which evaluations are groupedboth by distance as well as orientation from the ego. This matrix is derived for the full nuScenes dataset for the pre-trained Pointpillars model.



(a) True environment: *ped* (b) True environment: *obs*

Figure 2.5: System-level probabilistic guarantees for the car-pedestrian example. Figure 2.5a shows the satisfaction probability that the car stops at C_{k-1} for x_e = ped under various initial speeds and maximum speeds V_{max} such that $1 \leq V_{max} \leq 6$. Figure 2.5b shows the satisfaction probability that the car does not stop at C_{k-1} for x_e = obs under various initial speeds and maximum speeds V_{max} such that $1 \leq V_{max} \leq 6$.

Prediction	True Label		
	`ped`	`obs`	
`ped`	14884	751	2488
`obs`	650	48693	1760
	7966	14921	29748

Table 2.5: Class Labeled Confusion Matrix computed from the full nuScenes dataset for the Pointpillars model

Prediction	True Label			
	$\{empty\}$	{`ped`}	{`obs`}	{`ped`, `obs`}
{}	0	441	1722	105
{`ped`}	153	1223	106	300
{`obs`}	140	30	12128	1530
{`ped`, `obs`}	9	100	925	4620

Table 2.6: Proposition Labeled Confusion Matrix computed from the full nuScenes dataset for the Pointpillars model

tance parametrized counterpart (red curve) result in the highest system-level guarantees for the pedestrian case (see Figure 2.5a). In comparison, the class-labeled and proposition-labeled confusion matrices with grouped evaluations result in lower probabilities of satisfaction. While the class-labeled confusion matrix can result in overly conservative results, the proposition-labeled confusion matrices (without the grouped evaluations) might result in overly relaxed guarantees. For example, suppose there are multiple pedestrians in the radius band D_k, and the model detects just one pedestrian from the LiDAR data. If the pedestrian detected is one that is not going to interact with the car (e.g., it is located laterally distant from or behind the vehicle), then this detection is not safety-critical. However, this still gets counted as a true positive in the proposition-labeled confusion matrix. This coarseness is reduced when evaluations are grouped, especially in a manner consistent with the high-level planner's discrete abstraction. This can be seen in the satisfaction probabilities of the proposition-labeled confusion matrix computed from grouped evaluations (brown curve). This satisfaction probability lies between probability curves for the class-labeled and ungrouped proposition-labeled counterparts, thus illustrating the importance of choosing the right fidelity in grouping abstractions.

Sensitivity Analysis. this chapter is focused on highlighting the importance of system-level reasoning of determining perception metrics that are best suited for system-level analysis. The choice of a stronger object detection model would better highlight the strength of our evaluation framework, as illustrated in Figure 2.6. For each true positive rate for the pedestrian class, 20 random instances of the 4×4 proposition-labeled confusion matrix were generated. Even though the class-labeled confusion matrix is the most conservative, we observe that system-level satisfaction probability is close to 1 when the true positive rate is high ($> 99\%$).

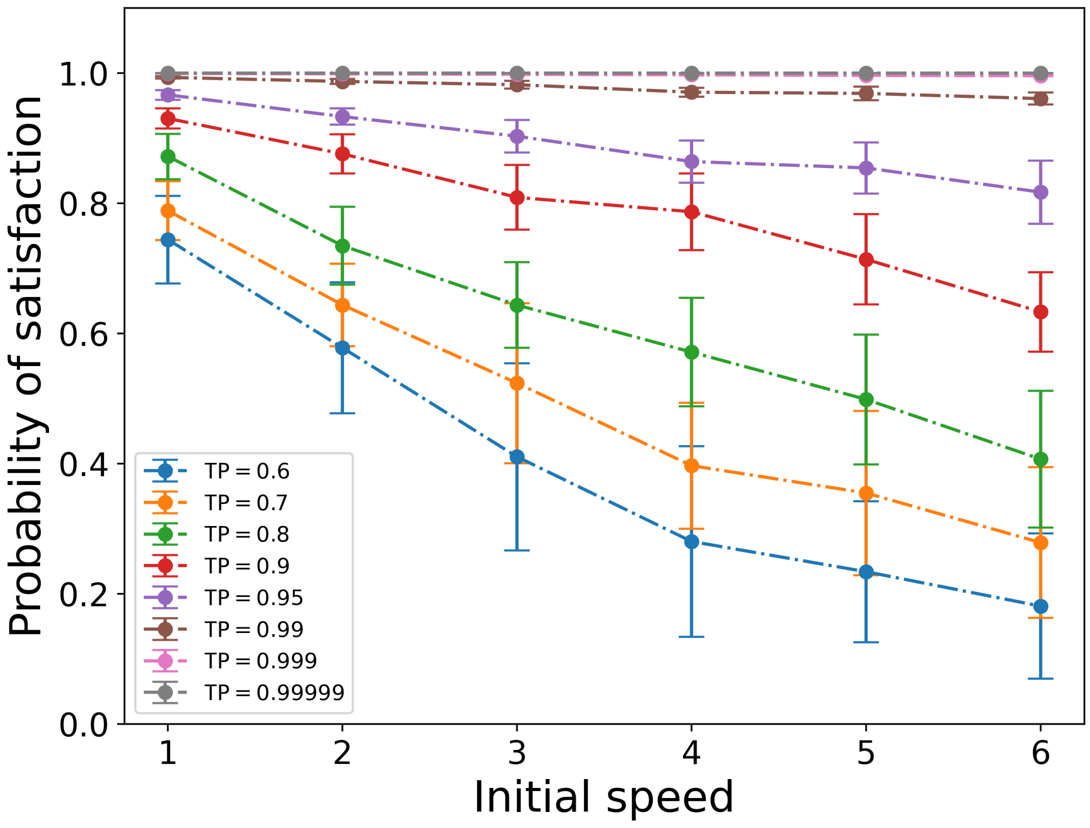


Figure 2.6: Sensitivity plots for satisfaction probability derived from proposition labeled confusion matrix for the specification that the car does not stop at C_{k-1} for $x_e = \texttt{ped}$ under various initial speeds and maximum speeds V_{max} such that $1 \leq V_{max} \leq 6$. The sensitivity is shown for varying true positive rates of detecting pedestrians.

2.7 Lower Bounds for Detection Metrics from System-level Guarantees

In this section, we will cover a case study to illustrate how system-level probabilistic guarantees can inform quantitative evaluation metrics for perception. In particular, we will derive lower bounds on detection metrics from desired system-level guarantees. This was implemented as a case study in the system design and analysis tool, Pacti [68].

Assume-guarantee contracts are a useful formalism to specify assumptions and guarantees of individual sub-systems or scenario viewpoints. Building on fundamentals in category theory, operators for composition, conjunction, refinement, quotient, and others can be rigorously defined over assume-guarantee contracts. This allows for formal reasoning about interactions between component implementations that respect assume-guarantee contracts, allowing for rigorous system

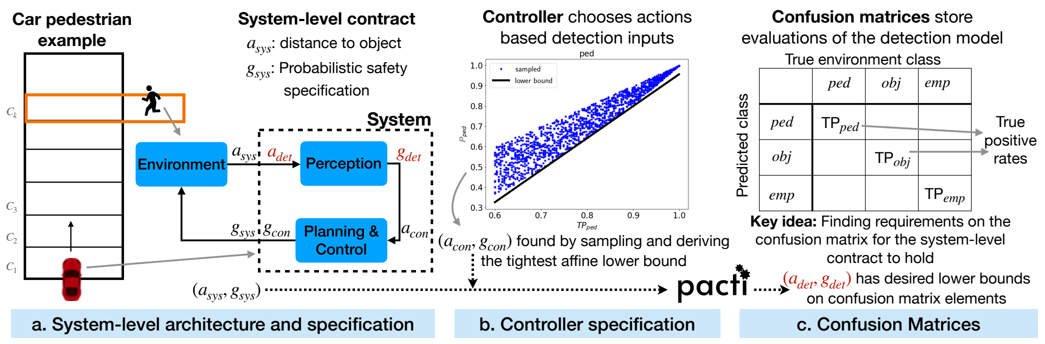

Car pedestrian example
System-level contract
a_{sys}: distance to object
g_{sys}: Probabilistic safety specification
System
Environment
Perception
Planning & Control
a_{sys} a_{det} g_{det} g_{sys} g_{con} a_{con}
(a_{sys}, g_{sys})
Controller chooses actions based detection inputs
sampled
lower bound
(a_{con}, g_{con}) found by sampling and deriving the tightest affine lower bound
pacti
Confusion matrices store evaluations of the detection model
True environment class
Predicted class
ped obj emp
TP$_{ped}$ TP$_{obj}$ TP$_{emp}$
True positive rates
Key idea: Finding requirements on the confusion matrix for the system-level contract to hold
(a_{det}, g_{det}) has desired lower bounds on confusion matrix elements
a. System-level architecture and specification
b. Controller specification
c. Confusion Matrices

Figure 2.7: Given a system-level specification $\mathcal{C}_{\text{sys}} = (a_{\text{sys}}, g_{\text{sys}})$, and a specification for the controller $\mathcal{C}_{\text{con}} = (a_{\text{con}}, g_{\text{con}})$, derive the object detection specification $\mathcal{C}_{\text{det}} = (a_{\text{det}}, g_{\text{det}})$.

design. In addition to identifying requirements on perception systems from system-level guarantees of safety, we will use this formalism in Chapter 5 for designing compositional test plans.

Definition 2.16 (Assume-Guarantee Contract)**.** Let $\mathcal{B}$ be a universe of behaviors, then a *component* M is a set of behaviors $M \subseteq \mathcal{B}$. A *contract* is the pair $\mathcal{C} = (A, G)$, where A are the assumptions and G are the guarantees. A component E is an *environment* of the contract $\mathcal{C}$ if $E \models A$. A component M is an *implementation* of the contract, $M \models \mathcal{C}$ if $M \subseteq G \cup \neg A$, meaning the component provides the specified guarantees if it operates in an environment that satisfies its assumptions. There exists a partial order of contracts, we say $\mathcal{C}_1$ is a refinement of $\mathcal{C}_2$, denoted $\mathcal{C}_1 \leq \mathcal{C}_2$, if $(A_2 \leq A_1)$ and $(G_1 \cup \neg A_1 \leq G_2 \cup \neg A_2)$. We say a contract $\mathcal{C} = (A, G)$ is in canonical, or saturated, form if $\neg A \subseteq G$.

In this case study, we consider the design of a vehicle that has to satisfy a safety property with a given probability. We understand the vehicle as a system that consists of two subsystems: a perception component (for object detection) and a controller, as shown in Figure 2.7a. From knowledge of a system-level safety contract and of the specification of the control component, the *quotient* operator is used to derive a specification for the perception component.

Consider the car-pedestrian example once again. We encode the notion of safety in linear temporal logic formulas φ_c, where $c \in \{\texttt{ped}, \texttt{obs}, \texttt{empty}\}$. This way, we can specify safe behavior when an element of each class is present on the crosswalk. We synthesized controllers to satisfy these safety properties, assuming perfect perception. The details of the properties and our synthesis approach can be found in [55].

System-level contract. Let $\mathbb{P}_c$ be the probability that the car will satisfy requirement φ_c when the crosswalk object has true class c. We set the system-level contract to

$$\mathcal{C}_{\text{sys}} = (d_l \leq d \leq d_u,\ g_{\texttt{ped}} \wedge g_{\texttt{obs}} \wedge g_{\texttt{empty}}),$$

where $d_l,\ d_u$ are bounds on the distance d to the object in the crosswalk, and g_c characterizes an affine lower bound of $\mathbb{P}_c$. This system-level contract assumes bounded distance to the object of interest, and guarantees affine lower bounds (as a function of d) on probabilistic satisfaction of safety properties. In other words, at the system-level we allow the probability of satisfaction of the safety property to degrade if the vehicle is far away from the crosswalk.

Controller contract. As mentioned, we synthesize three controllers, each making sure that property φ_c would be satisfied under perfect perception. In order to write a contract for each of the controllers, we make use of the fact that the perception component is not perfect. As a result, the controller satisfies its safety specification probabilistically.

To correlate probabilities of property satisfaction to perception errors, we base our approach on [55, 69]. The satisfaction probability $\mathbb{P}_c$ for the safety property φ_c is computed by constructing a Markov chain with transition probabilities derived from the *true positive rates*[3] of the perception component, and then invoking standard statistical model-checking tools.

For this example, $\mathbb{P}_c$ depends mainly on the true positive rate $\texttt{TP}_c$ of the class c. We determine a tight affine lower bound for $\mathbb{P}_c$ as a function of $\texttt{TP}_c$ by sampling and solving a linear program. The data for the linear program is generated by sampling false negatives for each value of $\texttt{TP}_c$ and computing the corresponding $\mathbb{P}_c$ (see Figure 2.7b). This procedure yields the following controller contract corresponding to each object class *c*:

$$\mathcal{C}_{\text{c}} = (l_c \leq \texttt{TP}_c,\ a_c(\texttt{TP}_c) + b_c \leq \mathbb{P}_c), \tag{2.16}$$

where l_c, a_c, and b_c are reals. The three contracts are composed to find the overall control contract: $\mathcal{C}_{\text{con}} = \mathcal{C}_{\texttt{ped}} \parallel \mathcal{C}_{\texttt{obs}} \parallel \mathcal{C}_{\texttt{empty}}$.

[3]The true positive rate of a perception component for an object class is defined as the probability that the component correctly detects an object to be of that class.

Object detection contract. Now that we have the specifications for the system and for the three controllers, we use contract operations to obtain the specification of the perception component. The detection component contract is found via the quotient $\mathcal{C}_{\text{det}} = \mathcal{C}_{\text{sys}}/\mathcal{C}_{\text{con}}$, where $\mathcal{C}_{\text{sys}}$ is the system-level contract and $\mathcal{C}_{\text{con}}$ is the controller contract. $\mathcal{C}_{\text{det}}$ imposes lower bounds on the true positive rates $\texttt{TP}_c$ of each object class c. We illustrate the results numerically for an instance of the car-pedestrian example. The system contract is set to

$$\begin{aligned}\mathcal{C}_{sys} = (1 \leq d \leq 10,\ 0.99(1-0.1d) \leq \mathbb{P}_{\texttt{ped}} \wedge 0.8(1-0.1d) \leq \mathbb{P}_{\texttt{obs}} \\ \wedge\, 0.95(1-0.1d) \leq \mathbb{P}_{\texttt{empty}}),\end{aligned} \tag{2.17}$$

that is, the contract assumes the distance to the crosswalk is bounded between 1 and 10 units, and specifies desired system-level probabilities $\mathbb{P}_c$ as a function of distance d. The controller contracts are computed to be $\mathcal{C}_{\texttt{ped}} = (0.6 \leq \texttt{TP}_{\texttt{ped}},\ 1.58\texttt{TP}_{\texttt{ped}} - 0.622 \leq \mathbb{P}_{\texttt{ped}})$, $\mathcal{C}_{\texttt{obs}} = (0.3 \leq \texttt{TP}_{\texttt{obs}},\ 0.068\texttt{TP}_{\texttt{obs}} + 0.93 \leq \mathbb{P}_{\texttt{obs}})$, and $\mathcal{C}_{\texttt{empty}} = (0.6 \leq \texttt{TP}_{\texttt{empty}},\ 0.2\texttt{TP}_{\texttt{empty}} + 0.799 \leq \mathbb{P}_{\texttt{empty}})$. These contracts impose affine lower bounds on $\mathbb{P}_c$ with respect to the true positive rates $\texttt{TP}_c$. The quotient results in an object detection contract with true positive rates lower bounded by affine functions of the distance d:

$$\begin{aligned}\mathcal{C}_{\text{det}} = (1 \leq d \leq 10,\ (1.02 - 0.063d \leq \texttt{TP}_{\texttt{ped}}) \wedge (0.6 \leq \texttt{TP}_{\texttt{ped}}) \wedge \\ (0.3 \leq \texttt{TP}_{\texttt{obs}}) \wedge (0.6 \leq \texttt{TP}_{\texttt{empty}})).\end{aligned} \tag{2.18}$$

Now that we have obtained the contract for the perception component, we can give this contract to designers responsible for object detection. The designers can develop the perception component and verify that it satisfies the requirements on true positive bounds as in $\mathcal{C}_{\text{det}}$. If it does, we can infer that the overall system with controller designed according to $\mathcal{C}_{\text{con}}$ will satisfy the system-level requirements.

2.8 Conclusion

The main takeaway of this chapter is that evaluation metrics for perception tasks should be informed by the downstream control logic as well as system-level metrics of safety. We focused on the object detection and classification task of perception, and made the following contributions. First, we proposed the idea of using confusion matrices as probabilistic models of sensor error to inform how system-level guarantees must be computed. Second, we replaced the labels of the confusion matrix with atomic propositions that are used in the system-level specifications and the downstream planner. Third, we finetuned the proposition-labeled confusion matrix by grouping evaluations to an abstraction that is consistent with the occupancy size of the vehicle. Fourth, we illustrated how assume-guarantee contracts, or system design optimization tools in general, can leverage our framework to inform desired evaluation criteria for the percpetion module from system-level guarantees. Finally, we evaluated a state-of-the art detection model on the nuScenes dataset according to these metrics, and computed the corresponding system-level guarantees for a discrete-state car-pedestrian example.

There are several exciting directions for future work. As illustrated in Figure 2.5, the satisfaction probabilities of safety requirements are still relatively low compared to the high levels of safety guarantees (e.g., $1 - 10^{-6}$ to $1 - 10^{-9}$) that are often expected in these applications. This is for several reasons. First, we evaluated a model trained on one modality (3D object detection from pointcloud); typically the best models are multi-modal and use data from several different sensors. Secondly, we do not consider tracking in our evaluation; once an object is detected, it is tracked across frames and an object misdetected in a single frame need not drastically change the high-level plan. Given the sensitivity analysis, we expect the satisfaction trends to improve with the aforementioned extensions and with better object detection models.

In addition, this paradigm can be extended to evaluate other perception tasks in a task-relevant manner, to handle scenarios with dynamic environments, to synthesize controllers that are optimal for a given perception model, and to validate the framework via experimental demonstrations. For this, we will consider building on the work in [54], which studies quantitative analysis of systems that operate in partially known dynamic environments. It assumes that the environment model belongs to a set $\mathbf{M}^{env}$ of Markov chains. The system does not know the true model of the environment, and instead maintains a belief, which is defined as a probability distribution over all possible environment models in $\mathbf{M}^{env}$. We will extend our work to derive the belief update function based on the perception performance.

Chapter 3

AUTOMATED TEST SYNTHESIS VIA NETWORK FLOWS: AN INTRODUCTION

3.1 Introduction

This chapter explores reactive test synthesis for discrete decision-making components in autonomous systems. This chapter originated from thinking about how to find a small set of difficult test cases for discrete decision-making behaviors. For safety as well as satisfying system requirements, a full-stack autonomous system must reason over its *own state* as well as about how the environment might *react* to its actions. Oftentimes, this involves reasoning over inputs and states that are both discrete and continuous valued, and implementations of autonomous systems accomplish this at various levels of abstraction. In this chapter, we formulate the test synthesis problem, and introduce the concept of a test objective.

This chapter is adapted from:

A. Badithela, R. M. Murray. (2020). "Synthesis of Static Test Environments for Observing Sequence-like Behaviors in Autonomous Systems." arXiv preprint: https://arxiv.org/pdf/2108.05911.

3.2 Related Work

Due to robustness metrics from their quantitative semantics, signal temporal logic (STL) and metric temporal logic (MTL), are natural paradigms for reasoning over trajectories of low-level continuous dynamics [37, 70]. In many instances, the term testing is used inter-changeably with falsification [39]. Falsification is the problem of finding initial conditions and input signals that lead to violation of a temporal logic formula with the goal of finding such failures quickly and for black-box models [36, 41, 71, 72]. Furthermore, the black-box approaches in the related topics of falsification of hybrid systems [36], and simulation-based test generation [38, 42], rely on stochastic optimization algorithms to minimize the robustness of temporal logic satisfaction. Since dense-time temporal logics better encapsulate the range of system behaviors at the with continuous dynamics, these techniques are successful at falsification at the low-level. However, some of the complexity can be attributed to the coupling between continuous dynamics with high-level discrete decision-making behaviors, a hierarchical approach to test-case generation could be effective.

Typically, high-level choices of autonomous robotic systems exhibit discrete decision-making [73, 74], and LTL specifications are often used to capture mission objectives at higher levels of abstraction. Covering arrays have been used to initialize discrete parameters of the test configuration at the start of the falsification procedure in [36, 42, 75], but are not reactive. Furthermore, linear temporal logic (LTL) model checkers for testing has been explored in [71, 76–78], in which counterexamples found via model-checking are used to exactly construct test cases. However, these are usually applied to deterministic systems, thus relying on the knowledge of the system controller, and become inconclusive if the system behavior deviates from the expected model. In this chapter and next, we focus on a framework for testing of high-level specifications in linear temporal logic (LTL) without assuming knowledge of the system controller.

3.3 Motivation

Here we adopt a different notion of testing – one that is focused on observing the autonomous agent undertake a certain behavior in its mission. The DARPA Urban Challenge test courses, that mainly comprised of static obstacles and (dynamic) human-driven cars, were carefully designed to observe the agent undertaking certain behaviors [10]. For example, a part of the test course was designed for assessing parking behavior. The static obstacles – barriers blocking the region in front of the parking lot and other parked cars – were placed such that the agent had to repeatedly reverse/pull-in to incrementally adjust its heading angle before successfully parking in the designated spot. The clever placement of static obstacles in this scenario made it a challenging test for the agent, as opposed to an environment in which the agent pulls-in straight into the parking spot. Similarly, carefully designed scenarios with human-driven cars sought to observe other behaviors of the agent. In many, but not necessarily all, of these scenarios, the high-level behavior of the agent can be described as a sequence of waypoints. In the parking lot example, the sequence of waypoints can be characterized as a sequence of agent states, which can be characterized as a product of position and heading angle in the high-level abstraction. As a step towards automatically synthesizing these test scenarios, this chapter asks the following question.

Problem (Informal): Given a valid, user-defined sequence of waypoints, a reacha-

bility objective for the mission specification, find a set of possible initial conditions for the agent (if not specified by user) and determine a set of static constraints, characterized by transitions that are blocked/restricted, such that:

i. *the agent must visit the sequence of waypoints in order before its goal, and*

ii. *the test environment is minimally restricted.*

3.4 Preliminaries

Automata Theory and Temporal Logic

Definition 3.1 (Finite Transition System)**.** A *finite transition system* (FTS) is the tuple

$$TS := (S, A, \delta, S_0, AP, L),$$

where S denotes a finite set of states, A is a finite set of actions, $\delta : S \times A \to S$ the transition relation, S_0 the set of initial states, AP the set of atomic propositions, and $L : S \to 2^{AP}$ denotes the labeling function. We denote the transitions in TS as $TS.E := \{(s, s') \in S \times S \mid \text{ if } \exists a \in A \text{ s.t. } \delta(s, a) = s'\}$. We refer to the states of TS as $TS.S$, and similarly denote the other elements of the tuple. An execution σ is an infinite sequence $\sigma = s_0 s_1 \ldots$, where $s_0 \in S_0$ and $s_k \in S$ is the state at time k. We denote the finite prefix of the trace σ up to the current time k as σ_k. A strategy π is a function $\pi : (TS.S)^* TS.S \to TS.A$.

Definition 3.2 (System)**.** The *system under test* is modeled as a finite transition system T_{sys} with a singleton initial set, $|T_{\text{sys}}.S_0| = 1$.

A directed graph $G = (V, E)$ can be induced from T_{sys} in which the vertices represent states $T_{\text{sys}}.S$ and the edges represent the transitions $T_{\text{sys}}.E$, and the labeling function assigns propositions that are true at each vertex. For a proposition $p \in AP$, and vertex $v \in V$, $v \vdash p$ means that p evaluates to $True$ at v. A *run* $\sigma = s_0 s_1 \ldots$ on the graph is an infinite sequence of its nodes where $s_i \in T_{\text{sys}}.S$ represents the system state at time step i.

We introduce the notion of a test harness to specify how the test environment can interact with the system. A test harness is used to constrain a state-action (s, a) pair of the system in the sense that the system is prevented from taking action a from state $s \in T_{\text{sys}}.S$. Let the actions $A_H \subseteq T_{\text{sys}}.A$ denote the subset of system actions that can be restricted by the test harness. The test harness $H : T_{\text{sys}}.S \to 2^{A_H}$ maps states of the transition system to actions that can be restricted from that state.

In the examples considered in this thesis, every state of the system has a self-loop transition corresponding to stay-in-place action, though the framework does not require this. Note that in our examples, A_H does not contain self-loop actions.

In this work, we synthesize tests for high-level decision-making components of the system under test and therefore model it as a discrete-state system. Linear temporal logic (LTL) has been effective in formally specifying safety and liveness requirements for discrete-decision making [14, 15, 18]. For our problem, we use LTL to capture the system and test objectives. The reach-avoid fragment of LTL is restricted to the use of logical operators and the *next*, *always*, and *eventually* temporal operators, and can capture a rich set of behaviors such as safety and coverage properties. Every LTL formula can be transformed into an equivalent non-deterministic Büchi automaton, which can then be converted to a deterministic Büchi automaton [60].

Flow Networks

We will leverage network flows to model the test synthesis problem. Flow networks are used in computer science to model several problems on graphs [79]. One of the main contributions of this thesis is in using flow networks for test synthesis.

Definition 3.3 (Flow Network [80])**.** A *flow network* is a tuple $\mathcal{N} = (V, E, c, (V_s, V_t))$, where V denotes the set of nodes, $E \subseteq V \times V$ the set of edges, $c \geq 0$ represents edge capacity, $V_s \subseteq V$ the source nodes, and $V_t \subseteq V$ the sink nodes. On the flow network $\mathcal{N}$, we can define the *flow* vector $\mathbf{f} \in \mathbb{R}^{|E|}_{\geq 0}$ to satisfy the following constraints: i) the capacity constraint

$$0 \leq f^e \leq c, \forall e \in E, \tag{3.1}$$

ii) the conservation constraint

$$\sum_{u \in V} f^{(u,v)} = \sum_{u \in V} f^{(v,u)}, \forall v \in V \setminus \{V_s, V_t\}, \text{ and} \tag{3.2}$$

iii) no flow into the source or out of the sink

$$f^{(u,v)} = 0 \text{ if } u \in V_t \text{ or } v \in V_s. \tag{3.3}$$

The flow value on the network $\mathcal{N}$ is defined as

$$F := \sum_{\substack{(u,v) \in E, \\ u \in V_s}} f^{(u,v)}. \tag{3.4}$$

In the following chapters, we will primarily be using the framework of network flows to identify restrictions on system actions, which will be analogous to cuts on a graph. An edge represents a transition the system can make. For this reason, it can suffice to define flow networks that have unit capacity $c = 1$ on all edges. Unless otherwise mentioned, all references to a flow network hereafter will assume unit edge capacities.

Definition 3.4 (Cut [80])**.** Given a graph $G = (V, E)$, a *cut* of G is the tuple $\texttt{Cut} := (S, T)$ that partitions the vertices of G into disjoint sets $S \subset V$ and $T \subset V$, that is, $S \cup T = V$ and $S \cap T = $. The *cut-set* $C \subseteq E$ of the cut $\texttt{Cut} = (S, T)$ is the set of edges that when removed from G results in the disjoint node sets S and T:

$$C := \{(u, v) \in E \,|\, u \in S,\ v \in T\}. \tag{3.5}$$

The expression $G_{cut}(C) := (V, E \setminus C)$ refers to the graph resulting from remove the edges in C from G. We will use the same expression to refer to any graph from which edges C are removed, even if the set C does not correspond to a $\texttt{Cut}$ (i.e., complete partition of the graph G). Any edge that is removed from G is referred to as an *edge-cut*.

In finding maximum flow, it becomes important to identify edges on the graph through which flow can be pushed through and track edges which have already been saturated. This is the concept of a residual network which is defined below. For a more detailed exposition with illustrations, see pages 726–727 of [80].

Definition 3.5. Given a graph $G = (V, E)$ and a flow f, the set of residual edges E_f are defined as

$$E_f := E \cup \{(u, v) \,|\, (v, u) \in E \text{ and } f(v, u) > 0\}. \tag{3.6}$$

The corresponding *residual network* $G_f = (V, E_f)$ is a flow network with edge capacities $c_f : E_f \to [0, 1]$ defined as follows:

$$c_f(u, v) = \begin{cases} 1 - f(u, v) & \text{if } (u, v) \in E \text{ and } f(u, v) < 1, \\ f(v, u) & \text{if } (v, u) \in E, \\ 0 & \text{otherwise.} \end{cases} \tag{3.7}$$

Standard algorithms such as Edmonds-Karp [79] use residual networks to find the maximum-flow (and equivalently, minimum-cut) of a single source-sink flow problem in a graph $G = (V, E)$ in $O(|V||E|^2)$ time. Roughly, starting with zero flow, the Edmonds-Karp algorithm iteratively updates the residual network as flow from the source to target is found. Initially, the residual network is exactly the same as the original graph G. Then, the algorithm finds paths from source to target on the residual network until no such paths remain. These paths are known as augmenting paths, and are used to construct a realization of the maximum flow.

Definition 3.6 (Augmenting Path [80])**.** Given a graph $G = (V, E)$ with flow f, an *augmenting path* from a source $s \in V$ to target $t \in V$ is a simple path $Path(s,t)$(i.e., without any cycles) from s to t on the corresponding residual network G_f.

The time complexity of the Edmonds-Karp algorithm comes from finding the shortest augmenting path on the residual network in each iteration until no paths remain. If G has a maximum possible flow $F_{\max}$, then the *set of augmenting paths* $\mathtt{AP} = \{Path_1(s,t), \ldots, Path_{F\max}(s,t)\}$ has cardinality $F_{\max}$ since all edges in G have unit capacities. That is, once an augmenting path is identified, all of the edges in the augmenting path are saturated on the residual network. As a result, any two paths $Path_i(s,t), Path_j(s,t) \in \mathtt{AP}$ are always *edge-disjoint* in the sense that there does not exist any edge $e \in E$ that is in both $Path_i(s,t)$ and $Path_j(s,t)$. The set of augmenting paths found by the Edmonds-Karp algorithm is denoted as the *set of shortest augment paths* SAP. It can be proven that finding the shortest augmenting paths and updating the residual network accordingly results in finding a realization of the maximum flow [80].

Proposition 3.1. A set of shortest augmenting paths AP comprises of edge-disjoint paths.

3.5 Test Objective

We begin by considering reachability specifications as mission objectives for the agent under test. For the test itself, we wish to observe a sequence-like behavior of the agent in its attempt to satisfy its mission objectives. Formally, this test behavior can be described by the temporal logic formula given below.

Definition 3.7 (Test Objective (strict sequence))**.** The *test objective* for strict sequenced visit is given by the LTL formula:

$$\varphi_{\text{test}} := \Diamond(p_1 \wedge \Diamond(p_2 \wedge \Diamond(\cdots \wedge \Diamond p_n))) \bigwedge_{i=1}^{n-1} (\neg p_{i+1} \mathsf{U}\, p_i)\,, \tag{3.8}$$

where $p_1, \dots, p_n \in AP$ are propositional formulas. This is a sequence-like formula since the agent has to eventually visit every v_i, but it cannot visit v_{i+1} before visiting v_i, where $v_i \vdash p_i$, for all $i = 1, \dots, n$.

The system under test does not have access to the test objective φ_{test}. Since LTL formulae cannot be evaluated on finite test runs, the length of the test run depends on the time the agent takes to satisfy its mission objective.

Example 3.1. Consider the gridworld in Figure 3.1 on which the agent can transition between states (up, down, left, right) with the mission specification of reaching some goal state (formalized as $\varphi_g = \Diamond g$). Of the many possible paths the agent can take to meet its objective, we are interested in observing it navigate to the goal while restricted to a class of paths described by the test specification $\varphi_{test} = \Diamond(p_1 \wedge \Diamond p_2)$. How would we constrain actions of the agent in certain states, such that it navigates through the sequence of waypoints before reaching the goal? Furthermore, is it possible to synthesize these constraints such that the sequence flow value from p_1 to g is maximized?

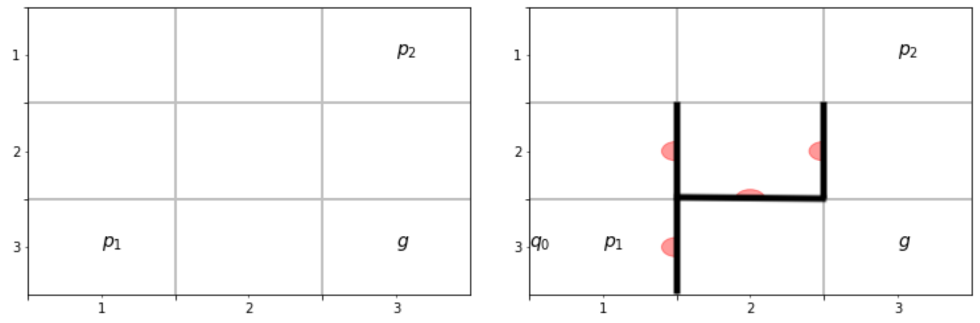


Figure 3.1: Left: Unrestricted gridworld labeled by propositional formulas. Right: A test environment synthesized by our algorithm where the transitions $(2,1) \to (2,2)$, $(3,1) \to (3,2)$, $(2,2) \to (3,2)$, and $(2,2) \to (2,3)$ blocked. Red semi-circle patches illustrate one-way constraints, that is, transition from state u to state v is restricted, but v to u is allowed, if arch of the semi-circle is in the grid corresponding to u along the transition from u to v.

Problem Statement

Now, we formalize the test environment synthesis problem. We limit our focus to *static* test environments, by which we mean that the test environment does not react to the actions of the agent during the test, leaving the reactive test synthesis problem for later in this chapter and the next chapter.

Definition 3.8 (Test Graph)**.** Given a labeled directed graph $G = (V, E)$, a mission/agent specification $\varphi_{sys} = \diamondsuit p_{n+1}$, a test specification φ_{test} (equation (3.8)), a *test graph* $G_{cut}(C) = (V, E \backslash C)$ is the directed graph obtained by removing set of edges C from the original graph G. On $G_{cut}(C)$, a run σ starting from state v_1 will satisfy the specification,

$$\tag{3.9}$$

Definition 3.9 (Minimally Restricted Test Graph)**.** A test graph $G_{cut}(C)$ is *minimally restricted* if the total sequence flow value from $v_1 \models p_1$ to $v_{n+1} \models p_{n+1}$ on $G_{cut}(C)$ are maximized.

Remark 3.1. In this chapter, the definition of a *minimally restricted* graph does not relate to the actual number of cuts in the cut-set C, but only whether the flow on $G_{cut}(C)$ is maximized.

Problem 3.1. Given a system specification $\psi_{\text{sys}} = \diamondsuit v_{n+1}$, a labeled directed graph $G = (V, E)$ induced by the non-deterministic transition model $\mathcal{T}$ of the agent, a test specification $\varphi_{test} = \diamondsuit(p_1 \wedge \diamondsuit(p_2 \wedge \diamondsuit(\cdots \wedge \diamondsuit p_n)))$, static constraints $C \subseteq E$ such that on the resulting test graph $G_{cut}(C) = (V, E \backslash C)$ is a minimally restricted test graph.

Here we aim to find a cut that maximizes the flow from a waypoint p_i to its consecutive waypoint p_{i+1}, while eliminating any flow to waypoints p_j (for $j > i + 1$) for all $i = 1, \cdots, n$. In other words, some flows need to be cut while other flows should be maximized. The problem of constructing a minimally restricted test graph for observing a sequence-like specification can be cast as the following optimization,

$$\begin{aligned} \max_{C \subset E} \quad & f_{G_{cut}(C)} \\ \text{s.t.} \quad & f_{G_{cut}(C)} \leq f_{G_{cut}(C)}(v_i, v_{i+1}) && \forall i = 1, \cdots, n-1, \\ & f_{G_{cut}(C)}(v_i, v_j) = 0 && \forall i = 1, \cdots, j-2, \forall j = 3, \cdots, n, \end{aligned} \tag{3.10}$$

where the variables $C \subset E$ are the set of edges to be restricted resulting in the unit-capacity graph $G_{cut}(C) = (V, E \backslash C)$, the scalar $f_{G_{cut}(C)}$ represents the total flow on $G_{cut}(C)$ from v_1 to v_n, the scalars $f_{G_{cut}(C)}(v_i, v_j)$ represent the total flow from source v_i to sink v_j. The problem data is the original graph $G = (V, E)$ and the sequence nodes $v_1, \ldots, v_n$. Solving this optimization directly will require constructing an integer linear program (ILP), for which constructing the constraint set is not straightforward. Furthermore, it would require solving the integer program with $|E|$ number of integer variables.

3.6 Algorithm for Synthesizing Static Test Environments

Let $G = (V, E)$ be a directed graph, with unit capacity on every edge, induced by the transition system T_{sys} of the system under test. Assuming that the test environment has complete freedom to "block" any transition in the graph G allowed by the test harness H, Algorithm 4 returns a set of edges, $C \subset E$, of the graph G that must be removed before the test run. First, we make following assumptions on G. Let $d_G(v_1, v_2)$ denote the length of the shortest path from vertex v_1 to vertex v_2 on graph G.

Assumption 3.1. For each $i \in \{1, \dots, n+1\}$, let v_i denote the vertex $v \in V$ s.t $v \vdash p_i$. Assume $|v_i|= 1$, for all $i = \{1, \dots, n+1\}$.

Informally, Assumption 3.1 states that every propositional formula, $p_1, \dots, p_{n+1}$, has a single vertex in G associated with it.

Assumption 3.2. There exists a set of edges $C \subseteq E$ such that the modified graph obtained by removing these edges, $G_{cut}(C) = (V, E \backslash C)$, is such that

$$d_{G_{cut}(C)}(v_1, v_{n+1}) > \cdots > d_{G_{cut}(C)}(v_n, v_{n+1}) > d_{G_{cut}(C)}(v_{n+1}, v_{n+1}) = 0. \quad (3.11)$$

Assumption 3.2 is equivalent to the statement that by removing some edges (or restricting certain transitions) from the original graph G, there exists some set of initial conditions Q_0 for which the only path(s) to the goal g is through the behavior φ_{test}. This assumption is imperative since there might be instances for which it is impossible to construct a test graph. For example, in the following simple labeled graph (Figure 3.2), it is impossible to construct a test graph for the test specification $\varphi_{\text{test}} = \diamondsuit(p_1 \wedge \diamondsuit p_2)$. Once the system is in state v_1, it can directly proceed to the goal state v_g without visiting v_2. For instances such as this one, a reactive test environment is necessary.

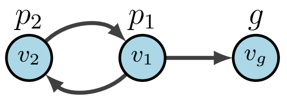


Figure 3.2: An invalid configuration of propositional formulas for test specification $\varphi_{\text{test}} = \diamondsuit(p_1 \wedge \diamondsuit p_2)$

Overview of the Approach: At a high-level, we identify all edge disjoint path combinations through the sequence specification to find edge restrictions. This can be seen as a brute-force approach to solving the problem. As we will discuss in the following chapter, this problem is NP-hard, and therefore, it is not possible to find a polynomial-time algorithm to solve this problem if we assume that P is not NP. Later in this chapter and next, we will cover a more efficient optimization formulations that can capture a wide-range of specification types with faster runtimes.

Definition 3.10 (Sequence Path)**.** Given a graph G and atomic propositions characterizing the sequence specification, $p_1, \dots, p_n$, where $Path_G(v_i, v_{i+1})$ represent a simple path from v_i to v_{i+1}, for all $i = 1, \dots, n$. The *sequence path* from v_1 to v_{n+1} can be constructed from the individual path segments as $Path_G(v_1, v_{n+1}) = Path_G(v_1, v_2), \dots, Path_G(v_n, v_{n+1})$. The sequence path $Path_G(v_1, v_{n+1})$ is *valid* if it does not have a cycle involving two or more path segments. That is, if there are no edges $(u, w), (w, v) \in E$ such that edge $(u, w) \in Path_G(v_i, v_{i+1})$ and $(w, v) \in Path_G(v_j, v_{j+1})$ for some $i + 1 \leq j \leq n + 1$, except for the case in which both $w = v_{i+1}$ and $j = i + 1$. In other words, except for the sequence nodes $v_1 \dots, v_n$ that link individual segments, there are no common nodes linking an earlier path segment to a later segment. Observe that existence of a valid sequence path implies that Assumption 3.2 is true.

Finding Combinations of Augmenting Paths

Maximum flow realizations on a graph need not be unique; there can exist more than one set of augmenting paths to capture maximum flow between a source and target. For some graph G, we will denote the maximum flow from source s to target t as $F_{max_G}(s, t)$ and a set of augmenting paths by $\texttt{AP}_G(s, t)$ or $\texttt{SAP}_G(s, t)$ for the set of shortest augmenting paths. Let $\mathcal{AP}_G(s, t)$ correspondingly denote the *set of sets of augmenting paths*, and let $\mathcal{SAP}_G(s, t)$ correspondingly denote the *set of sets of shortest augmenting paths*. Note that $\mathcal{AP}_G(s, t)$ captures all possible realizations of maximum flow from s to t on G, and $\mathcal{SAP}_G(s, t) \subseteq \mathcal{AP}_G(s, t)$. Intuitively, since the shortest path need not be unique, the set $\mathcal{SAP}_G(s, t)$ could have multiple elements.

By definition, on a test graph G_{cut}, the maximum sequence flow value will be bounded as follows:

$$f_{G_{cut}}(v_1, v_{n+1}) \leq \min_{i=1,\dots,n} f_{G_{cut}}(v_i, v_{i+1}). \tag{3.12}$$

On a minimally restricted test graph, the total sequence flow $f_{G_{cut}}(v_1, v_{n+1})$ is maximized. For each $1 \leq i \leq n$, note that $\mathcal{SAP}_G(v_i, v_{i+1})$ is finite since the number of edges in G are finite, but is combinatorial in the number of edges since it requires enumerating simple paths from v_i to v_{i+1}. The total number of augmenting path combinations from v_1 to v_{n+1} that can realize the maximum flow $f_{G_{cut}}(v_1, v_{n+1})$ will be at most $\Pi_{i=1}^{n}|\mathcal{AP}_G(v_i, v_{i+1})|$. Not every augmenting path combination might lead to a valid test graph since there could exist a combination of augmenting paths that violates equation (5.2) by resulting in an invalid sequence path $Path(v_1, v_n)$. Consider the simple example of the 3×3 grid in Figure 3.4. The combination of sequence flows $(\mathtt{AP}_G(v_1, v_2), \mathtt{AP}_G^2(v_2, v_3), \mathtt{AP}_G(v_3, v_4))$ will give us $f_{G_{cut}}(v_1, v_{n+1}) = 1$, but the combination of $(\mathtt{AP}_G(v_1, v_2), \mathtt{AP}_G^1(v_2, v_3), \mathtt{AP}_G(v_3, v_4))$ does not have any valid sequence paths.

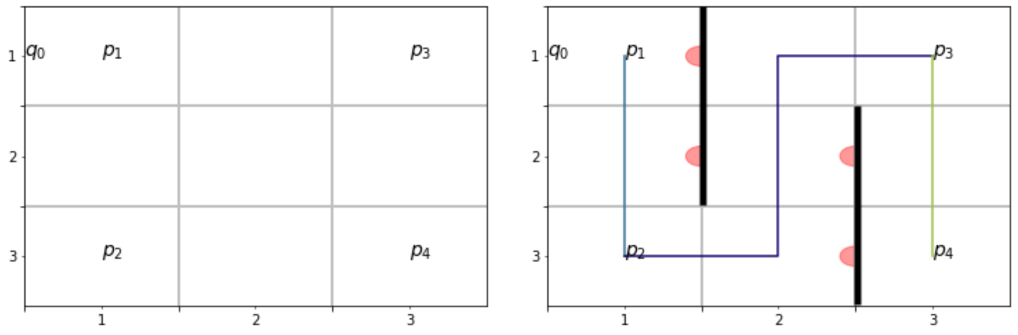


Figure 3.3: Left: 3×3 grid for the sequence specification with atomic proposition p_1, p_2, and p_3. Right: Illustrated with cuts that route the flow from p_1 to p_4.

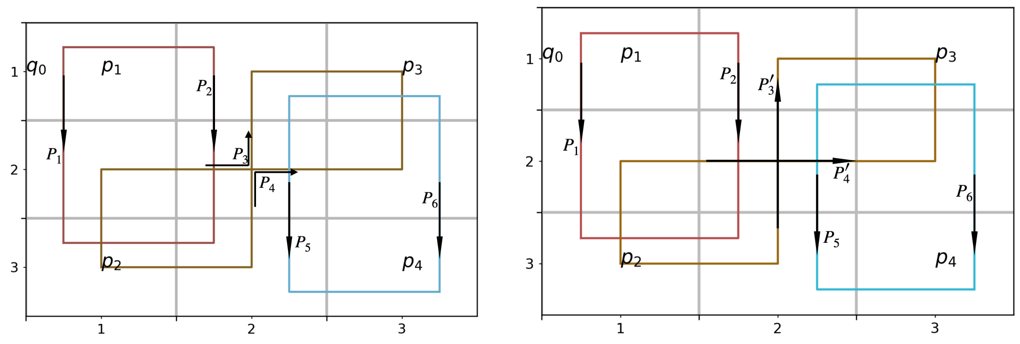


Figure 3.4: In this 3×3 grid, the left and right figures illustrate two different augmenting path combinations. Each grid shows a realization of the flow for each pair of nodes: red: (p_1, p_2), gold: (p_2, p_3), and blue: (p_3, p_4). The main difference between the two figures is in the flow from p_2 to p_3. In both figures, the augmenting paths characterizing the flow from p_1 to p_2 and p_3 to p_4 are the same: $\mathtt{AP}_G(v_1, v_2) = \{P_1, P_2\}$ characterizes the maximum flow from p_1 to p_2, and $\mathtt{AP}_G(v_3, v_4) = \{P_5, P_6\}$ characterizes the maximum flow from p_3 to p_4. On the left, $\mathtt{AP}_G^1(v_2, v_3) = \{P_3, P_4\}$, and on the right, $\mathtt{AP}_G^2(v_2, v_3) = \{P_3', P_4'\}$. It is possible to form a sequence path on the right with $Path_G(v_1, v_4) = P_1, P_3', P_6$, but not on the left. This sequence path is exactly illustrated in Fig. 3.3 (right).

Algorithm 4: Restrict Transitions

```
Input: $\varphi_{\text{test}}, \varphi_a, G = (V, E, L)$.
Output: $C \subseteq E$.
  $p \leftarrow \{p_1, \dots, p_n, p_{n+1}\}, V_p \leftarrow \{v_1, \dots, v_n, v_{n+1}\}$    ▷ $v_i \vdash p_i$
  $P_{cut} \leftarrow$Find-Cut-Paths$(G, p)$
  $C = \{\}$
  if Assumption 3.3 then
      $flg \leftarrow 1$
  while $P_{cut} \neq \emptyset$ do    ▷ Repeat until all cuts are found
      $E \leftarrow$ Edges in $P_{cut}$
      $\mathcal{A}, \mathcal{P}_{keep}, |\mathcal{A}|, F_{max} \leftarrow$ Sequence-Flows$(G, p, flg = 0)$    ▷ Combinations of
  sequence flows
      for all $j = 0, \dots, |\mathcal{F}|$ do
          $\mathcal{A}_f \leftarrow \mathcal{A}(j)$    ▷ Selecting a combination $(S_1, \dots, S_n)$
          $P_{keep} \leftarrow \mathcal{P}_{keep}(j)$    ▷ Augmenting paths for each $v_i$ to $v_{i+1}$
          $MC_{keep} \leftarrow$ Min-Cut-Edges$(G, p, P_{keep})$
          $D_{keep} \leftarrow diag(A_{keep}\mathbb{1})$
          for all $A_f \in \mathcal{A}_f$ do
              $D_f \leftarrow diag(A_f\mathbb{1})$
              $A_{cut}, A_{keep}, D_{keep} \leftarrow$ ILP-params$(P_{cut}, P_{keep}, MC_{keep})$
              $x^*, f^*, b^* \leftarrow$ ILP$(A_{cut}, A_{keep}, D_{keep}, A_f, D_f)$    ▷ Call to ILP (3.10)
              if $\mathbb{1}^T f^* = F_{max}$ then
                  $C_{new} \leftarrow \{e_i | x_i^* = 1\}$
                  $C \leftarrow C \cup C_{new}$
                  break    ▷ Breaking out of both for loops
      $G \leftarrow G \backslash C_{new}$
      $P_{cut} \leftarrow$ Find-Cut-Paths$(G, p)$
```

To avoid this issue, the algorithm searches through all combinations of sequence flows before constructing the input to the ILP (3.10). Since this is an expensive computation, a further assumption on the input graph and set of propositions can ease this bottleneck.

Assumption 3.3. Let f_i be the maximum flow on G from source v_i to target v_{i+1})/ Let $\mathcal{SAP}_G(v_i, v_{i+1}) = \{\mathtt{SAP}_G^1 = \{P_1, \dots, P_{f_i}\}\}$, represent the set of sets of shortest augmenting paths that characterizes the flow from v_i to v_{i+1} on G. Then, there exists a combination $(\mathtt{SAP}_G^1, \dots, \mathtt{SAP}_G^n)$ on which a maximum sequence flow can be characterized.

In other words, Assumption 3.3 allows us to reason over combinations of shortest augmenting path flows, which is combinatorial in all shortest paths, instead of combinations of the set of augmenting flows, which exacerbates the combinatorial complexity by enumerating all possible paths. All shortest paths are a subset of all simple paths between two nodes.

3.7 Iterative Synthesis of Constraints

Algorithm 4 details how edge cuts are computed by iteratively solving the following integer linear program.

$$\begin{aligned} \max_{\substack{x\in\mathbb{B}^n,\, f\in\mathbb{B}^l \\ b\in\mathbb{B}^m}} \quad & \mathbb{1}^T f \\ \text{s.t.} \quad & \mathbb{1} \le A_{cut} x \\ & A_{keep} x \le D_{keep} b \\ & b \le A_{keep} x \\ & D_f f \le A_f(\mathbb{1} - b)\,, \\ & A_f(\mathbb{1} - b) - D_f \mathbb{1} + \mathbb{1} \le f, \end{aligned} \tag{3.13}$$

where (x, b, f) are the optimization variables, and $A_{cut} \in \mathbb{B}^{k\times n}$, $A_{keep} \in \mathbb{B}^{m\times n}$, $D_{keep} \in \mathbb{B}^{m\times m}$, $D_f \in \mathbb{B}^{l\times l}$, $A_f \in \mathbb{B}^{l\times m}$ are problem data described in more detail below.

Variables: The variable $x \in \mathbb{B}^n$, where $n = |E_{cut}|$, is the Boolean vector corresponding to edges E_{cut} such that for some $k \le n$, if $x_k = 1$, then the corresponding edge is restricted, and $x_k = 0$ means that it is left in the graph for future iterations. Given $P_{keep} = (\texttt{SAP}^1_G, \ldots, \texttt{SAP}^n_G) \in \mathcal{P}_{keep}$, a combination of set of shortest augmenting paths, the variable $b \in \mathbb{B}^m$ keeps track of whether an augmenting path in some $\texttt{SAP}^i_G$ $(1 \le i \le n)$ is restricted or not.

For some $k \le m$, if $b_k = 1$, then the corresponding augmenting path in some $\texttt{SAP}^i_G$ has minimum-cut edge(s) restricted by the ILP, and $b_k = 0$ if none of the minimum-cut edges of that augmented path have been restricted. The variable $f \in \mathbb{B}^l$ is the sequence flow vector for a given sequence flow, S_f, such that $l = |S_f|$ is the number of edge-disjoint paths constituting the sequence flow.

Constraints: The first constraint of the ILP, $A_{cut}x \ge \mathbb{1}$, enforces the requirement that each path in $P \in P_{cut}$ is restricted. Each row of A_{cut} corresponds to a path $P \in P_{cut}$. The q-th row of A_{cut} is constructed as follows:

$$(A_{cut})_{q,r} = \begin{cases} 1 & \text{if } E_{cut}(r) \in P = P_{cut}(q) \\ 0 & \text{otherwise.} \end{cases} \tag{3.14}$$

In the second and third constraints, $A_{keep}x \leq D_{keep}b$ and $b \leq A_{keep}x$, is used to determine the variable b from the variable x. Each row of $A_{keep} \in \mathbb{B}^{m\times n}$ corresponds to some path $P \in \texttt{SAP}^i_G$, and $D_{keep} \in \mathbb{B}^{m\times m}$ is a diagonal matrix. Suppose the q-th row of A_{keep} corresponds to a path $P \in \texttt{SAP}^i_G$ for $P_{keep} = (\texttt{SAP}^1_G, \ldots, \texttt{SAP}^n_G)$, and $MC_{keep}(i)$ is the set of minimum-cut edges on some path in $\texttt{SAP}^i_G$, then the q-th row is constructed as follows:

$$(A_{keep})_{q,r} := \begin{cases} 1, & \text{if } E_{cut}(r) \in P \cap MC_{keep}(i). \\ 0, & \text{otherwise.} \end{cases} \tag{3.15}$$

The q-th diagonal entry of D_{keep} stores the total number of minimum-cut edges in the path corresponding to the q-th row of A_{keep}:

$$D_{keep} := diag(A_{keep}\mathbb{1}) \tag{3.16}$$

These two constraints ensure that for some $q \leq n$, $b_q = 1$ iff at least one minimum-cut edge on the path corresponding to the q-th row of A_{keep} is restricted, and $b_q = 0$ iff none of the minimum-cut edges on the path corresponding to the q-th row of A_{keep} are restricted.

The fourth and fifth constraints, $D_f f \leq A_f(\mathbb{1} - b)$ and $f \geq A_f(\mathbb{1} - b) - D_f\mathbb{1} + \mathbb{1}$, determine the flow value for a given set of sequence flow paths, S_f. Suppose the q-th row of the matrix $A_f \in \mathbb{B}^{l\times m}$ corresponds to some sequence flow path $P = (P_1, \ldots, P_n) \in S_f$. Let $R = (r_1, \ldots, r_n)$ denote the indices of the paths $P_1, \ldots, P_n$ according to the ordering of the paths constituting all $\texttt{SAP}^i_G$ that is consistent with the construction of A_{keep} and D_{keep}. Then, the q-th row of A_f is defined as follows:

$$(A_f)_{q,r} := \begin{cases} 1, & \text{if } r = r_i \text{ for some } 1 \leq i \leq n. \\ 0, & \text{otherwise.} \end{cases} \tag{3.17}$$

The q-th diagonal entry of matrix $D_f \in \mathbb{B}^{l\times l}$ stores the total number of ones in the q-th row of A_f:

$$D_f := diag(A_f\mathbb{1}). \tag{3.18}$$

The fourth constraint ensures that if any of the constituent paths, $P_1, \ldots, P_n$, in the q-th sequence flow path $P = (P_1, \ldots, P_n) \in S_f$ (for $1 \leq q \leq l$), is restricted, then the flow value, $f_q = 0$. The last constraint ensures that if none of the constituent paths, $P_1, \ldots, P_n$, in the q-th sequence flow path $P = (P_1, \ldots, P_n) \in S_f$ (for $1 \leq q \leq l$), are restricted, then the flow value, $f_q = 1$.

Parameters: The parameters used to construct the problem data for the ILP (3.13) are the set of paths that need to be restricted, P_{cut}, the set of paths whose combination constitutes sequence flow and should not be restricted, P_{keep}, and the set of minimum-cut edges, MC_{keep}, on the paths constituting P_{keep}. The set $\mathcal{P}_{keep} = \{(\texttt{SAP}^1_G, \ldots, \texttt{SAP}^n_G) \mid \texttt{SAP}^i_G \in \mathcal{SAP}^i_G\}$ is a set of all combinations of shortest augmenting paths in the sequence. For a given combination of sets of augmenting paths, $P_{keep} = (\texttt{SAP}^1_G, \ldots, \texttt{SAP}^n_G)$, with the cardinality of $\texttt{SAP}^i_G$ being denoted as follows: $k_i := |\texttt{SAP}^i_G|$, and $m := \Sigma^n_{i=1} k_i$. In P_{keep}, suppose a combination of augmenting paths, $S_f = \{P = (P_1, \ldots, P_n) \mid P_i \in \texttt{SAP}^i_G\}$, represents a sequence flow, then a matrix $A_f \in \mathbb{B}^{|S_f| \times m}$ can be constructed to represent the sequence flow S_f. This construction is outlined in the descriptions of Constraints of the ILP. An instance of P_{keep} can have several sequence flows, S_f, and correspondingly, several matrices, A_f, all of which are collectively denoted by $\mathcal{A}_f$. The set of all such $\mathcal{A}_f$ is denoted by $\mathcal{A}$, which has cardinality $|\mathcal{A}| = |\mathcal{P}_{keep}|$, since each $\mathcal{A}_f$ corresponds to an instance of P_{keep}. The maximum sequence flow value is given by $F_{\max}$.

Cost Function: The cost function computes the maximum sequence flow value. Algorithm 4 does not proceed to the next iteration of P_{cut} until it finds the set of static constraints that return the maximum possible sequence flow value, F_{max}. To guarantee completeness of Algorithm 4, we need to prove that the cuts synthesized in prior iterations do not preclude feasibility of further iterations with regards to assumption 3.2. See Section 3.8 for complexity of the subroutines in Algorithm 4.

Sub-routines of Algorithm 4

The Min-Cut-Edges sub-routine takes as input a graph G, a list of propositions $\{p_1, \ldots, p_n\}$, and for each $1 \leq i \leq n$, a non-empty set of shortest augmenting paths for the source-sink pair (v_i, v_{i+1}). This sub-routine returns as output the set of minimum cut-set on those augmenting paths, which is then used in constructing the problem data for the ILP.

The Sequence-Flows sub-routine takes as input a graph G, a list of propositions $\{p_1, \ldots, p_n\}$, and a parameter to indicate if Assumption 3.3 holds. It then computes the combination of all augmenting flows (or all shortest augmenting flows) that can result in a non-zero sequence flow from v_1 to v_{n+1}. It returns as output the set of all sets of matrices that capture sequence-flow paths, $\mathcal{A}$, a set of $\mathcal{P}_{keep} = \{(\texttt{SAP}^1_G, \ldots, \texttt{SAP}^n_G) | \texttt{SAP}^i_G \in \mathcal{SAP}^i_G\}$, the total number of combinations, $|\mathcal{A}|$, and the maximum possible sequence flow value, F_{max}, which is determined when $\mathcal{A}$ is constructed.

The sub-routine FIND-BYPASS-PATHS takes as input a graph G and list of propositions, $\{p_1, \ldots, p_n\}$, and uses the Edmonds-Karp algorithm to find bypass paths for every source-sink pair (v_i, v_j), where $i + 1 \leq j \leq n + 1$. Specifically, this sub-routine finds a set of shortest augmenting paths from v_i to v_j, and to ensure that they are bypass paths, the sub-routine is applied on $G_{ij} = G = (V \setminus V_k, E \setminus E(V_k))$, where:

$$V_k \coloneqq \{v_k \vDash p_k \mid 1 \leq k \leq n+1 \text{ and } k \neq i, k \neq j\},$$

and the edges associated with V_k are denoted by E_k:

$$E_k \coloneqq \{(u, v) \in E \mid u \in V_k \text{ or } v \in V_k\}.$$

All of these augmenting paths are collectively returned as the output P_{cut}, and the edges constituting these cuts are denoted by E_{cut}. Note that P_{cut} does not return all simple paths from v_i to $v_{j>i+1}$, but just a set of edge-disjoint paths. As a result, transitions are iteratively restricted until P_{cut} is empty.

3.8 Characteristics of the Algorithm

Lemma 3.1. In a graph $G = (V, E)$, let $\mathcal{P}$ represent a maximal set of sequence flow paths from v_1 to v_n. Let $\mathcal{P}_{cut}$ be the set of paths that need to restricted, with the edges constituting the paths in $\mathcal{P}_{cut}$ denoted by $E_{cut} \subset E$. Then, the set of constraint edges $C \subseteq E_{cut}$ can be found such that C does not constrain any path in $\mathcal{P}$.

Proof. A path $P_{cut} \in \mathcal{P}_{cut}$ can be restricted by removing at least one of its constituent edges. The number of edges of P_{cut} that are not in some path $P \in \mathcal{P}$ is non-zero, since otherwise it would imply that $P_{cut,i} \in \mathcal{P}$, and would not need to be restricted. The set C can simply be chosen by selecting one or more edges on every $P_{cut} \in \mathcal{P}_{cut}$ that are not a part of some path in $\mathcal{P}$. $\square$

Proposition 3.2. Let $G_m = (V, E_m)$ denote the graph for which the m-th iteration of the ILP (3.13) synthesizes new cuts $C_m \subset E_m$. Then, Assumption 3.2 is satisfied on $G_{m+1} = (V, E_m \backslash C_m)$.

Proof. In the first iteration, from Assumption 3.2, we know there exists at least one test graph $G' = (V, E \backslash C)$ that satisfies equation (3.11). Assume that the m-th iteration graph $G_m = (V, E_m)$ also satisfies Assumption 3.2. We will show by induction that the graph resulting from the the $(m+1)$-th iteration, $G_{m+1} = (V, E_m \backslash C_m)$, also satisfies Assumption 3.2. By construction, Algorithm 4 chooses a combination of set of shortest augmenting paths $(\mathtt{SAP}_G^1, \ldots, \mathtt{SAP}_G^n)$, such that there exists a non-empty set of sequence flow paths $\mathcal{F} = \{(P_1, \ldots, P_n) | \, P_i \in \mathtt{SAP}_G^i\}$ such that the simple path from v_1 to v_n characterized by $\Gamma = (P_1, \ldots, P_n) \in \mathcal{F}$ does not form an ij-cycle for some $i < j \leq n$. This implies that on the subgraph comprising of the edges in Γ, equation (3.11) is satisfied.

If the maximum possible sequence flow in a minimally restricted test graph is $F_{\max}$, then we can find a combination $(\mathtt{SAP}_G^1, \ldots, \mathtt{SAP}_G^n)$ such that for each $i = 1, \ldots, n$, there exists a subset of edge-disjoint paths carrying flow $F_{\max}$:

$$\mathtt{S}_i = \{P_i^1, \ldots, P_i^{F_{\max}}\} \subseteq \mathtt{SAP}_G^i,$$

from which we can construct the set of sequence flow paths:

$$\mathcal{F}' = \{(P_1^{k_1}, \ldots, P_n^{k_n}) | P_i^{k_i} \in \mathtt{S}_i, \, 0 \leq k_i \leq F_{\max}\} \subseteq \mathcal{F}.$$

By construction of the input variables to the ILP (3.13), the constraints of ILP (3.13) require that the sequence flow variable f has atleast one element that is 1. This is possible only if there exists a set of edges C_m that constrain $A_{cut,m}$ such that there exists at least one sequence path $P \in \mathcal{F}$ that does not have any of its minimum-cut edges restricted, which is true as shown in Lemma (3.1). Therefore, the new graph $G_{m+1} = (V, E_m \backslash C_m)$ satisfies Assumption (3.2). □ □

Theorem 3.1. Under Assumption (3.2), Algorithm 4 is complete and returns a test graph G' from Definition 3.8 that satisfies equation (3.11).

Proof. Consider iteration m of the outer while loop in Algorithm 4, and let the graph at the m-th iteration be $G_m = (V, E_m)$. Denote $V_p = \{v_i | v_i \vdash p_i, \, \forall 1 \leq i \leq n+1\}$. Let $F_{\max}^{i,j}$ denote the maximum flow value from v_i to v_j on $G_{ij} = (V \backslash (V_p \backslash \{v_i, v_j\}), E_m)$, for some i, j such that $1 \leq i < j - 1 \leq n$. That is, G_{ij} is a copy of G_m, but with nodes in V_p, except for source v_i and sink v_j, removed.

This implies that there is a set $\mathtt{SAP}_{G_{ij}}^{i,j}$ of $F_{\max}^{i,j}$ edge-disjoint paths that characterize the maximum flow from v_i to v_j on G_{ij}. Let $(\mathcal{P}_{i,j}(k))_m$ be the set of all simple paths from v_i to v_j that share an edge with the k-th path in $\mathtt{SAP}_{G_{ij}}^{i,j}$. Let $(MC_{i,j})_m$ be the set of minimum-cut edges on the paths in $\mathtt{SAP}_{G_{ij}}^{i,j}$ and let $(E_{i,j})_m \subset E_m$ be the set of all edges on some path from v_i to v_j on G_{ij}. Clearly, $(MC_{i,j})_m \subseteq (E_{i,j})_m$.

For every $m \geq 1$, we can claim that $|(E_{i,j})_{m+1}| < |(E_{i,j})_m|$ because edges are removed to constrain $\texttt{SAP}^{i,j}_{G_{ij}}$ in the m-th iteration. Let $\tilde{m}$ be the number of iterations for G_{ij} to become disjoint. In the worst-case, edges continue to be restricted until iteration $\tilde{m}$ at which $(E_{i,j})_{\tilde{m}} = (MC_{i,j})_{\tilde{m}}$, at which point constraining edges to cut $(\texttt{SAP}^{i,j}_{G_{ij}})_{\tilde{m}}$ results in a cut separating v_i and v_j. Thus, $\tilde{m}$ has to be finite for every such i, j.

At the same time, from Proposition 3.2, the synthesized cuts are such that Assumption 3.2 is maintained as an invariant. Therefore, when the last set of paths $\texttt{SAP}^{i,j}_{G_{ij}}$ are restricted, the final test graph G' is such that $d_{G'}(v_1, v_{n+1}) > \ldots > d_{G'}(v_n, v_{n+1})$. □

In addition to Assumption 3.2, if Assumption 3.3 holds, Algorithm 4 can be modified by a parameter setting. The proof of Theorem 3.1 still holds.

Lemma 3.2. On the test graph G', any test run σ starting from state v_1 will satisfy the specification ((5.2)).

Proof. From Assumption 3.1, there is only one node in G' for each proposition in characterizing the test specification ((5.2)), and node satisfying proposition p_i is labeled as v_i. For every $i \in \{1, \cdots, n\}$, v_i is the only state in test graph G' that is successor to all states v on paths $Paths(v_{j<i}, v_{n+1})$ for which $d_{G'}(v, v_{n+1}) = d_{G'}(v_i, v_{n+1}) + 1$. This is true by construction of the ILP constraints. All paths in the set $Paths(p_{j<i}, g)$ on the test graph G' must pass through v_i.

Let σ denote the test run of the agent starting at v_1. We define a metric on the test graph G': $m_t := \min_t d_{G'}(\sigma_t, v_{n+1})$ to be the closest distance to node v_{n+1} in the first t steps of the test run. Note three properties of this metric m_t: (a) $m_t \geq 0$, (b) m_t decreases: $m_{t+1} := \min\{\sigma_{t+1}, m_t\} \leq m_t$, and (c) there exists a successor q_{t+1} to $\sigma_t = q_t$ on G' such that $d_{G'}(q_{t+1}, v_{n+1}) = d_{G'}(q_t, v_{n+1}) - 1$ that decreases m_t. The metric m_t starts at $m_0 \geq d_{G'}(v_1, v_{n+1})$ and decreases to 0 at the end of the test run. Thus, we can observe that $\sigma \models \Diamond(p_1 \wedge \Diamond(p_2 \cdots \wedge \Diamond\, p_{n+1})) \wedge_{i=1}^{n} (\neg p_{i+1} \mathsf{U}\, p_i) \iff \sigma \models \Diamond\, v_{n+1}$. □ □

From Theorem 3.1 and Lemma 3.2, Algorithm 4 synthesizes a test graph G' for the test specification (5.2), solving Problem 3.1.

Lemma 3.3. Consider the test graph G' from Definition 3.8 for the test specification σ from (5.2). If Assumption 3.2 holds, Algorithm 4 returns a minimally restricted test graph.

Proof. By construction, the inputs to the ILP (3.13) are constructed based on a maximal set of sequence flow paths from v_1 to v_n. By Lemma 3.1, at each iteration of the ILP (3.13) from which constraint edges are chosen, the maximum sequence flow value does not decrease at each iteration. Since there are a finite number of edges, there are a finite number of iterations until test graph is found. Therefore, the Algorithm 4 returns a minimally restricted test graph. □ □

Complexity of Subroutines in Algorithm 4

Since Find-Cut-Paths is determining a set of augmenting paths for a single source-sink flow, it has a complexity of Edmonds-Karp algorithm, $O(|V||E|^2)$ time for graph $G = (V, E)$ [79]. The complexity of Min-cut-Edges is $O(|V||E|^3)$ time since it runs a max-flow algorithm for each edge in the worst-case. The main computational bottleneck is in the Sequence-Flows subroutine, which constructs sets of augmenting flows by computing combinations of all simple paths and all shortest paths. In the worst-case, enumerating all simple paths between two nodes is $O|V!\,|$, and enumerating all shortest paths is slightly better in several cases.

3.9 Examples

We illustrate the iterative synthesis of restrictions on a simple graph and a small gridworld, and then show runtimes of Algorithm 4 on random gridworld instances for both the case for which Assumption 3.2 is true, and the case for which Assumptions 3.2 and 3.3 are true.

Simple graph: Consider a simple non-deterministic Kripke structure representing an autonomous agent, shown in Figure 3.5, with propositional formulas labeled adjoining the states. The agent mission objective is to reach g while being restricted to start from state q_0. The test environment seeks to restrict transitions such that the agent is prompted to pass through waypoint w in its trajectory to g.

Inputs to Algorithm 4 include the labeled graph G induced by the Kripke structure, the agent specification $\diamondsuit p_3$, the test specification $\diamondsuit p_2$, and the initial condition constraint $\diamondsuit p_1$. Algorithm 4 constrains the edges $\{(v_2, v_4), (v_4, v_6)\}$ in the first iteration, and the edges $\{(v_2, v_5), (v_5, v_6)\}$ in the second iteration. Although in this simple example, searching the set of all augmented paths becomes searching over all paths, in larger examples discussed below, each augmented path represents a class of paths that share some edge(s) with it.

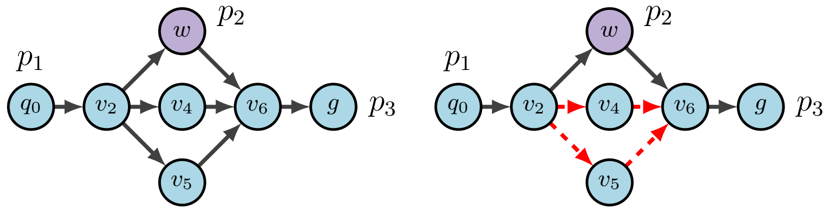


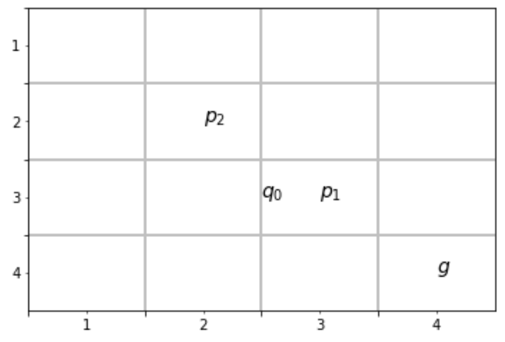
Figure 3.5: *Left:* Simple Kripke structure representing states that the agent can occupy. The waypoint, w, is highlighted in purple to indicate that transitions are restricted corresponding to propositional formula $p_2 = L(w)$. *Right:* A test graph. Dashed edges in red illustrate transitions that have been restricted/removed from the Kripke structure above.

Simple Gridworld: In Figure 3.6, we illustrate the iterative synthesis of obstacles in a gridworld instance. Note that this configuration can be synthesized only by considering all sets of augmenting paths between (p_1, p_2) and (p_2, p_3). Since there is no shortest augmenting path from p_2 to p_3 that does not form a cycle with some (in this example, there is only one) shortest augmenting path flow from p_1 to p_2, it is imperative to use all sets of augmenting paths in the Sequence-Flows subroutine.

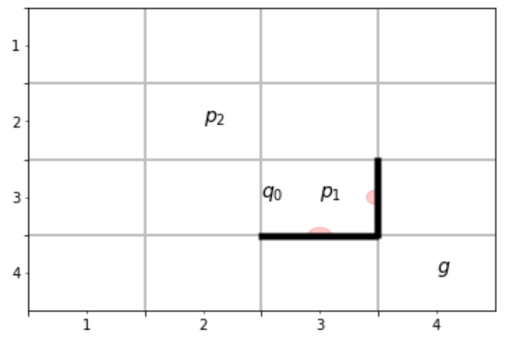
Random Gridworld Instances: For the case of setting all augmenting paths in the Sequence-Flows subroutine, we ran 50 random instances each for small gridworlds and propositions and plotted the average runtimes in Figure 3.7a. The number of propositions are limited by the size of the gridworld instances, which is restricted by the combinatorial nature of finding all sets of augmenting paths, and all combinations of sets of augmenting paths.

If we choose initial gridworld instances that satisfy Assumption 3.3, then Algorithm 4 can synthesize static constraints for slightly larger $t \times t$ grid sizes. The average runtimes for 50 random iterations for various grid sizes t is plotted in Figure 3.7b. The small increase to larger grid size is due to the Sequence-Flows subroutine reasoning over shortest augmenting paths, and not all augmenting paths.

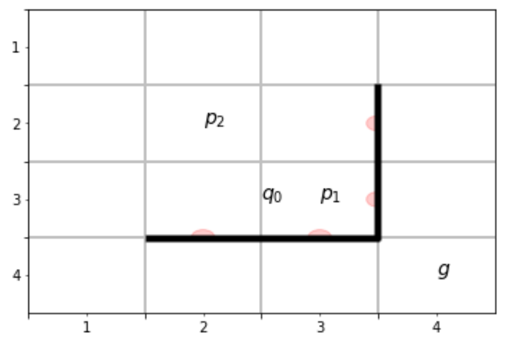
The average runtimes increase exponentially with the size of the grid. The number of propositions, denoted by $|P|$, is labeled n if the test specification φ_{test} (5.2) is comprised of propositions $(p_1, \ldots, p_n)$. In both Figures 3.7a and 3.7b, the average runtime for fewer propositions is at times higher that the average runtime for more propositions. This can be attributed to the Sequence-Flows subroutine



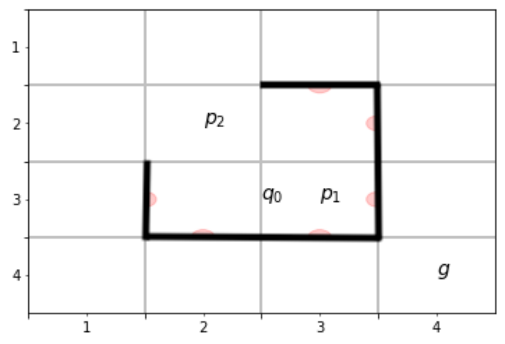
(a) Initial grid



(b) Iteration 1



(c) Iteration 2



(d) Iteration 3. The colored paths highlight a sequence flow from p_1 to g.

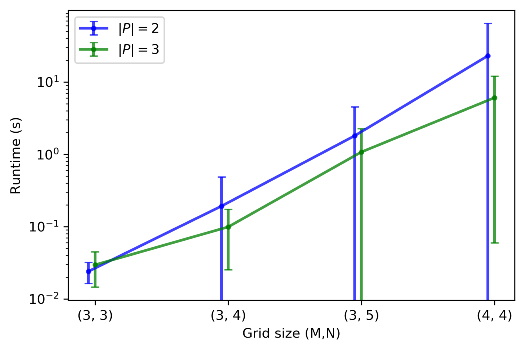
Figure 3.6: Synthesizing static test environment for $\varphi_{test} = \diamondsuit(p_1 \wedge \diamondsuit p_2) \wedge \neg p_2 \mathsf{U}\, p_1$ and $\varphi_a = \diamondsuit g$.

taking longer to enumerate all simple paths (or all shortest paths in case of Assumption 3.3) between two nodes, which could be greater in number due to fewer propositions constraining the graph.

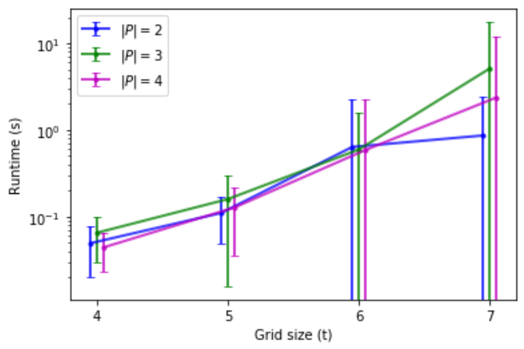
Another paradigm for the problem of synthesizing static test environments for sequence behaviors could be multi-commodity network flows, which will be explored later in this chapter. The multi-commodity flow setting typically considers multiple source-sink flows simultaneously drawing from the capacity of each edge, and here we compute separate network flows for every source-sink pair of nodes.

3.10 Conclusions

An algorithm to synthesize a static test environment to observe sequence-like behavior in a discrete-transition system was introduced. First, we formulated this test environment synthesis problem as a problem of synthesizing cuts on graphs using concepts of flow networks. Then, we proposed an algorithm which synthesized the cuts iteratively using an integer linear program. We proved that this algorithm is



(a) Small gridworld configurations using all augmenting flows.



(b) Gridworld configurations using only shortest augmenting flows.

Figure 3.7: Average runtime over 50 random instances. The number of propositions in φ_{test} is denoted by $|P|$ in the legend. Error bars represent standard deviation of runtimes.

complete and that the edges restricted by the ILP at each iteration maintain feasibility of the constraint in the next iteration. Finally, we conducted numerical experiments on random gridworld instances to assess the runtime of our algorithm. Simulation results preclude this algorithm from being tractable to larger examples.

However, the integer linear program requires reasoning over all possible paths on the transition system in order to identify the set of augmenting paths with the highest flow. This essentially becomes a brute-force approach to finding a set of edge disjoint paths from S to T that are routed through the propositions $p_1, \ldots, p_n$ in a sequence. The poor scalability is due to the exponential number of constraints (in the number of edges of T_{sys}) in the ILP formulation. To alleviate this, we will present an alternative flow-based formulation in the form of a min-max game with coupled constraints.

Chapter 4

FLOW-BASED REACTIVE TEST SYNTHESIS

The previous chapter introduced the problem of automated test strategy synthesis based on network flow optimization for user-specified temporal logic objectives. In this chapter, we will consider an bigger class of temporal logic objectives, and propose an automated test synthesis framework rooted in automata theory and flow networks. Especially, the routing optimization will be reformulated using two flow-based optimizations: i) a min-max Stackelberg game with coupled constraint sets, and ii) a mixed-integer linear programming formulation. The second flow-based reformulation lends itself to tractable implementations. Additionally, we study how these automatically found test strategies can be used to synthesize a strategy for a dynamic test agent.

4.1 Introduction

In this chapter, we will expand the class of temporal logic objectives to include reachability, avoidance, and reaction sub-tasks that commonly occur in high-level specifications of robotic missions [81]. A test strategy is *feasible* if a well-designed system can succeed in the test. We will formalize notions of *feasibility* and *restrictiveness* of a test strategy to handle these expanded class of specifications. Furthermore, in addition to the previous chapter, we will formally present the assumptions and guarantees that the system places on its test environment.

Next, we revisit the routing problem for the expanded class of specifications, and

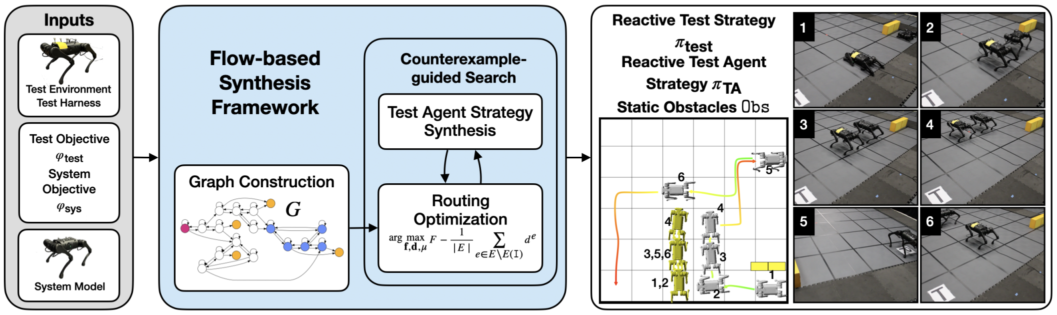

Inputs
Test Environment
Test Harness
Test Objective
φ_{test}
System Objective
φ_{sys}
System Model
Flow-based Synthesis Framework
Graph Construction
G
Counterexample-guided Search
Test Agent Strategy Synthesis
Routing Optimization
$\arg\max_{\mathbf{f},\mathbf{d},\mu} F - \frac{1}{|E|} \sum_{e \in E \setminus E(\mathrm{I})} d^e$
Reactive Test Strategy π_{test}
Reactive Test Agent Strategy π_{TA}
Static Obstacles Obs
6
5
4
4
3,5,6
3
1,2
2
1
1
2
3
4
5
6

Figure 4.1: Overview of the flow-based test synthesis framework which consists of three key parts: i) graph construction, ii) routing optimization, and iii) test environment synthesis (e.g., reactive test strategy / test agent strategy, static obstacles).

use automata theory to construct a product graph representing system state evolution along with progress of the temporal logic objectives. We formulate the routing problem on this product graph, first as a special class of Stackelberg games, and then as a mixed-integer linear program. We will motivate the mixed-integer formulation from the drawbacks of the game formulation. Using the MILP formulation, we can automatically find a test strategy for different environment types: static obstacles, reactive obstacles, and dynamic test agents. Even feasible solutions of the MILP return test strategies that satisfy the temporal logic objectives, and optimal solutions are guaranteed to not be overly-restrictive. Moreover, this routing optimization is proven to be NP-hard in the size of the product graph, thus supporting the MILP formulation. Finally, given a test agent, we are able to match the solution of the MILP to synthesize a strategy for the test agent. We use a simple counerexample-guided approach to ensure that the MILP solutions are dynamically feasible for the test agent.

Finally, the test synthesis framework is demonstrated on simulated grid world settings and on hardware with a pair of quadrupedal robots. For all experiments, our framework synthesizes test strategies that place the fewest possible restrictions on the system over the course of the test either by obstacle placement or a dynamic agent. In experiments with reactive obstacles and dynamic agents, the reactive test strategy results in a different test execution depending on system behavior. Despite this, the system is always routed through the test objective (e.g., being put in low-fuel state or having to walk over challenging terrain).

This chapter is adapted from:

J. B. Graebener*, A. S. Badithela*, D. Goktas, W. Ubellacker, E. V. Mazumdar, A. D. Ames, R. M. Murray (2024). "Flow-Based Synthesis of Reactive Tests for Discrete Decision-Making Systems with Temporal Logic Specifications". arXiv preprint https://arxiv.org/abs/2404.09888 (In submission to Transactions on Robotics).

A. Badithela*, J. B. Graebener*, W. Ubellacker, E. V. Mazumdar, A. D. Ames, R. M. Murray. (2023). "Synthesizing Reactive Test Environments for Autonomous Systems: Testing Reach-Avoid Specifications with Multi-Commodity Flows." In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 12430–12436. DOI: 10.1109/ICRA48891.2023.10160841.

4.2 Related Work

Following the previous chapter, we synthesize test environments from LTL specifications. We generate tests without specific knowledge of the system controller such that the test environment *adapts* or *reacts* to system behavior at runtime. Our test synthesis framework requires knowledge of a nondeterministic model of the system, but is agnostic to the high-level controller of the system, and is completely black-box to models and controllers at lower levels of abstraction.

Reactive, specification-based testing over discrete logics has been studied in [82–86]. In [82], reactive synthesis [87] is used to find a test strategy from LTL specifications of the system and a user-defined fault model, with guarantees that the resulting test trace will show the fault if the system implementation is faulty with respect to the fault model. However, this method requires fault models to be carefully specified over the output states of the system. Though very beneficial for specifying and catching sub-system level faults, it becomes intractable for specifying complex system-level faults, especially when the set of output states is large. The test synthesis framework in this chapter is also specification-based and adaptive to system behavior, but we specify desired test behavior in terms of system and test objectives. Additionally, the procedure in [82] does not account for the freedom of the system to satisfy its own requirements. In this chapter, we will synthesize reactive test strategies that demonstrate the test objective while placing minimal restrictions on the system. The automata-theoretic tools used in this chapter build on concepts used in correct-by-construction synthesis and model checking [60, 88], and will be covered in the next section.

Game-based formulations of testing either presume entirely cooperative or entirely adversarial settings. In [89], testing of reactive systems was formalized as a game between two players, where the tester and the system try to reveal and hide faults, respectively. In [90], the test strategy is found by optimizing for reachability and coverage metrics over a game modeling the system and the tester. Test case generation in cooperative settings is studied in [91, 92]. However, the test synthesis problem considered here is neither fully *adversarial* nor fully *cooperative*; a well-designed system is cooperative with the test environment in realizing the system objective, but since the system is agnostic to test objective, it need not cooperate with the test environment in realizing it.

In this chapter, in addition to static obstacles that restrict the system throughout the test, we will also consider reactive obstacles, and a dynamic test agent that is reactive to system behavior at runtime. Leveraging network flows, we will first pose the test synthesis problem as a Stackelberg game, and then present a more efficient formulation as an MILP. In recent years, network flow optimization frameworks with tight convex relaxations have led to massive computational speed-ups in solving robot motion planning problems [93, 94]. Network flow-based mixed integer programs have also been to synthesize playable game levels in video games [95], which was then applied to construct playable scenarios in robotics settings [96].

4.3 Preliminaries

In this section, we will revisit concepts from automata theory, and build on the background of automata theory and flow networks introduced in Chapter 3.

Consider the finite transition system TS, introduced in previous chapter. Once again, we use LTL to describe the system and test objectives. However, we place the following additional constraint on the system, requiring it to have at least one terminal state, to simplify the test objective that we synthesize for.

Definition 4.1 (System)**.** The *system under test* is modeled as a finite transition system T_{sys} with a single initial state, that is, $|T_{\text{sys}}.S_0|= 1$. Furthermore, at least one of the system states is terminal (i.e., no outgoing edges).

The system designers provide the states S, actions A, transitions δ, and a set of possible initial conditions S_0, set of atomic propositions, AP_{sys} and a corresponding label function $L_{\text{sys}} : S \to 2^{AP_{\text{sys}}}$. We require a unique initial condition $s_0 \in S_0$ to synthesize the test. If the test designer wishes to select an initial condition, then they can synthesize the test for each $s_0 \in S_0$ and choose accordingly. In addition to AP_{sys}, the test designer can choose additional atomic propositions AP_{test} and define a corresponding labeling function $L : S \to 2^{AP}$, where $AP := AP_{\text{sys}} \cup AP_{\text{test}}$. For test synthesis, the system model is $T_{\text{sys}} = (S, A, \delta, \{s_0\}, AP, L)$ is defined for the specific initial condition s_0 chosen by the test designer. The terminal state is used for defining test termination when the system satisfies its objective.

Assumption 4.1. Except for sink states, transitions between states of the system are bidirectional: $\forall(s, s') \in T_{\text{sys}}.E$ where s' is not a terminal state, we also have $(s', s) \in T_{\text{sys}}.E$.

This assumption is for a simpler presentation, and the framework can be extended to transition systems without this assumption (see Remark 4.8).

Definition 4.2 (Test Environment)**.** The *test environment* consists of one or more of the following: static obstacles, reactive obstacles, and dynamic test agents. A *static obstacle* on $(s, s') \in T_{\text{sys}}.E$ is a restriction on the system transition (s, s') that remains in place for the entire duration of the test. A *reactive obstacle* on $(s, s') \in T_{\text{sys}}.E$ is a temporary restriction on the system transition (s, s') that can be enabled/disabled over the course of the test. A *dynamic test agent* can occupy states in $T_{\text{sys}}.S$, thus restricting the system from entering the occupied state.

The desired test behavior can be captured via sub-tasks that are defined over atomic propositions AP. Table 4.1 lists the sub-task specification patterns that are considered. These specification patterns are commonly used to specify robotic missions [81]. The desired test behavior is characterized by the system and test objectives, defined over the set of atomic propositions AP that can be evaluated on system states $T_{\text{sys}}.S$.

Table 4.1: Sub-task specification patterns defined over atomic propositions.

Name	Formula	
Visit	$\bigwedge_{i=1}^{m} \diamondsuit p^i$	(s1)
Sequenced Visit	$\diamondsuit(p^0 \wedge (\diamondsuit(p^1 \wedge \ldots \diamondsuit p^m)))$	(s2)
Safety	$\square \neg p$	(s3)
Instantaneous Reaction	$\square(p \rightarrow q)$	(s4)
Delayed Reaction	$\square(p \rightarrow \diamondsuit q)$	(s5)

Definition 4.3 (Test Objective)**.** The *test objective* φ_{test} comprises of at least one visit or sequenced visit sub-task or a conjunction of these sub-tasks. The Büchi automaton $\mathcal{B}_{\text{test}}$ corresponds to the test objective φ_{test}.

Definition 4.4 (System Objective)**.** The *system objective* φ_{sys} contains at least one visit or sequenced visit sub-task. In addition, it can also contain some conjuction of safety, instantaneous and/or delayed reaction, and visit and/or sequenced visit sub-tasks. The Büchi automaton $\mathcal{B}_{\text{sys}}$ corresponds to the system objective φ_{sys}. We say that the *system reaches its goal*, or the *system execution is successful*, if the system trace is accepted $\mathcal{B}_{\text{sys}}$.

Typically, some aspects of a test are not revealed to the system until test time such as testing the persistence of a robot or prompting it to exhibit a difficult maneuver by placing obstacles in its path. This is formalized as a test objective and is not known to the system. In contrast, the system is aware of the system objective, which captures its requirements. For example, to test for *safety*, the system should know to avoid unsafe areas (4.3). To test a *reaction*, $\square(p \rightarrow q)$, the system needs to be aware of the reaction requirement (4.4), and the test objective needs to contain the corresponding visit requirement $\diamondsuit p$ to trigger the reaction. Furthermore, the test objective can contain standalone reachability (visit and/or sequenced visit) sub-tasks that are not associated with a system reaction sub-task, but require the system to reach/visit certain states. The test objective is accomplished by restricting system actions in reaction to the system state via the test harness.

In addition to the system objective, the system must interact safely with the test environment. The system must also obey the initial condition set by the test designer. For each obstacle/agent of the test environment, the system controller must respect the corresponding restrictions on its actions (i.e., cannot crash into obstacles/agents). Furthermore, for a valid system implementation, all lower-level planners and controllers of the system must simulate transitions on T.

Definition 4.5 (System Guarantees)**.** The system guarantees are a conjunction of the system objective, initial condition, safe interaction with the test environment, and a system implementation respecting the model T_{sys}.

Definition 4.6 (System Assumptions)**.** The system *assumes* that the test environment satisfies the following conditions:
A1. The test environment can consist of: i) static obstacles (e.g., wall), ii) reactive obstacles (e.g., door), and iii) test agents whose dynamics are provided to the system.
A2. The test environment will not take any action that will inevitably lead to unsafe behavior (e.g., not restricting a system action after the system has committed to it, test agents not colliding into the system).
A3. The test environment will not take any action that will inevitably block all paths for the system to reach its goal (e.g., restrictions will not completely the enclose the system or block it from progressing to its goal).
A4. If the system and test environment are in a livelock, the system will have the option to break the livelock and take a different path toward its goal.

A *correct system strategy* satisfies the system guarantees when the test environment satisfies the system assumptions. Therefore, a correct system strategy would result in a successful system execution. The system's specification cannot be expressed as an LTL formula. This is because, in an LTL synthesis setting, the system can assume that the test harness can behave in a worst-case manner and will never synthesize a satisfying controller. However, the system can assume that the test harness will always ensure that a path to achieving the system specification remains. This existence of a satisfying path cannot be easily captured in an LTL assumption.

Now, we will cover background on Büchi automata from the system and test objectives, and its usefulness for constructing a product graph that tracks both the evolution of the system state as well as the automaton state as it makes progress in satisfying its specifications.

Definition 4.7 (Deterministic Büchi Automaton)**.** A *non-deterministic Büchi automaton* (NBA) [60, 97] is a tuple $\mathcal{B} := (Q, \Omega, \delta, Q_0, F)$, where Q denotes the states, $\Omega := 2^{AP}$ is the set of alphabet for the set of atomic propositions AP, $\delta : Q \times \Sigma \to Q$ denotes the transition function, $Q_0 \subseteq Q$ represents the initial states, and $F \subseteq Q$ is the set of acceptance states. The automaton is a *deterministic Büchi automaton* (DBA) iff $|Q_0| \leq 1$ and $|\delta(q, A)| \leq 1$ for all $q \in Q$ and $A \in \Omega$.

Remark 4.1. We use *deterministic* Büchi automata since each input word corresponding to a test execution should have a unique run on the automaton. While there are several different automata representations, deterministic Büchi automata are a natural choice for many LTL specifications.

Since the objectives are reach-avoid specifications and do not encode behaviors that occur "infinitely often", *deterministic finite automata* (DFAs) [60] would have sufficed. The intuition behind this is that we are only using the automata for tracking the automaton state on the product graph (see the following paragraphs on graph construction). Using Büchi automata was an implementation choice, and to leave the possibility for expanding to objectives that can only be characterized by DBAs and not DFAs. The tool Spot [98] was used to construct a deterministic Büchi automaton from an LTL formula, which had excellent documentation that made it easy to access and use. LTL to DFA tools are not as common, though there are a few tools that translate LTL formulas on finite traces (LTLf) to DFAs such as LTL_f2DFA [99] and Lisa [100]. Both tools rely on MONA [101], which translates LTL formulas to finite-state automata; Lisa additionally also depends on the DBA conversion from Spot to return a DFA.

To track progress with respect to the system and test objectives, we introduce the specification product, which will be used to construct the product graph.

Definition 4.8 (Specification Product)**.** A *product* of two Büchi automata, $\mathcal{B}_1$ and $\mathcal{B}_2$ over the alphabet Ω, is defined as $\mathcal{B}_1 \otimes \mathcal{B}_2 := (Q, \Omega, \delta, Q_0, F)$, with states $Q := \mathcal{B}_1.Q \times \mathcal{B}_2.Q$, initial state $Q_0 := \mathcal{B}_1.Q_0 \times \mathcal{B}_2.Q_0$, acceptance states $F := \mathcal{B}_1.F \times \mathcal{B}_2.F$. The transition relation δ is defined as follows, for all $(q_1, q_2) \in Q$, for all $A \in \Omega$, $\delta((q_1, q_2), A) = (q_1', q_2')$ if $\mathcal{B}_1.\delta(q_1, A) = q_1'$ and $\mathcal{B}_2.\delta(q_2, A) = q_2'$. The *specification product* is the product $\mathcal{B}_\pi := \mathcal{B}_{\text{sys}} \otimes \mathcal{B}_{\text{test}}$, where $\mathcal{B}_{\text{sys}}$ is the Büchi automaton corresponding to the system specification, and $\mathcal{B}_{\text{test}}$ is the Büchi automaton corresponding to the test objective. The states $(q_{\text{sys}}, q_{\text{test}}) \in \mathcal{B}_\pi.Q$, where $q_{\text{sys}} \in \mathcal{B}_{\text{sys}}$ and $q_{\text{test}} \in \mathcal{B}_{\text{test}}$, capture the event-based progression of the test and are referred to as history variables.

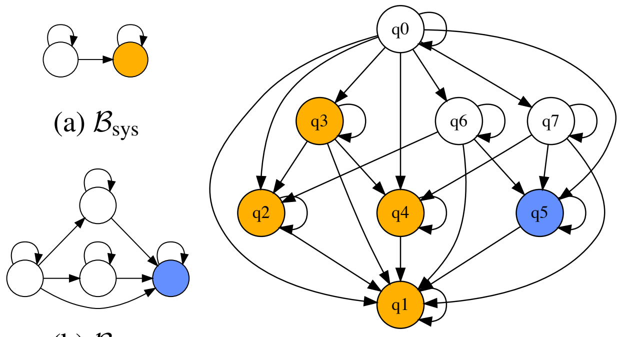
The system reaching its goal would typically mark the end of a test execution. However, the test engineer can also decide to terminate the test if the system appears to be stuck or enters an unsafe state. We assume that the test engineer gives the system a reasonable amount of time to complete the test. Upon test termination in state s_n, we augment the trace σ with the infinite suffix s_n^ω for evaluation purposes.

Remark 4.2. As tests have a defined start and end point, we need to bridge the gap between the finiteness of test executions and the infinite traces that are needed to evaluate LTL formulae. Augmenting the trace with the infinite suffix allows us to leverage useful tools available for LTL.

Remark 4.3. The states of the specification product automaton track the states of the individual Büchi automata, $\mathcal{B}_{\text{sys}}$ and $\mathcal{B}_{\text{test}}$, in the form of the Cartesian product to remember accepting states of the individual automata, which will be necessary for our framework (see Definitions 4.8, 4.15).

We use the synchronous product operator to construct a product of a transition system and a Büchi automaton. In particular, we will use this operator to construct the virtual product graph and the system product graph (see Section 4.4).

Definition 4.9 (Synchronous Product)**.** The *synchronous product* of a DBA $\mathcal{B}$ and a FTS T_{sys}, where the alphabet of $\mathcal{B}$ is the labels of T_{sys}, is the transition system



(a) $\mathcal{B}_{\text{sys}}$

(b) $\mathcal{B}_{\text{test}}$

(c) $\mathcal{B}_{\pi}$

Figure 4.2: Automata for Example 4.2. Yellow ● and blue ● nodes in $\mathcal{B}_{\text{sys}}$ and $\mathcal{B}_{\text{test}}$ are the respective accepting states. In the product $\mathcal{B}_{\pi}$, we continue to track these states for the system and test objectives. States in the product $\mathcal{B}_{\pi}$ that are accepting to both objectives (e.g., q1) are also shaded yellow.

$P \coloneqq T_{\text{sys}} \otimes \mathcal{B}$, where:

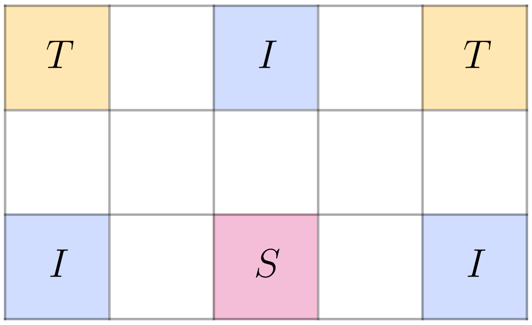
$$\begin{aligned}
&P.S \coloneqq T_{\text{sys}}.S \times \mathcal{B}.Q,\\
&P.\delta((s,q),a) \coloneqq (s',q') \text{ if } \forall s,s' \in T_{\text{sys}}.S, \forall q,q' \in B.Q,\\
&\quad \exists a \in T_{\text{sys}}.A, \text{ s.t. } T_{\text{sys}}.\delta(s,a) = s' \text{ and } \mathcal{B}.\delta(q, T_{\text{sys}}.L(s')) = q',\\
&P.S_0 \coloneqq \{(s_0,q) \,|\, s_0 \in T_{\text{sys}}.S_0, \; \exists q_0 \in \mathcal{B}.Q_0 \text{ s.t.}\\
&\quad \mathcal{B}.\delta(q_0, T_{\text{sys}}.L(s_0)) = q\},\\
&P.AP \coloneqq \mathcal{B}.Q,\\
&P.L((s,q)) \coloneqq \{q\}, \quad \forall (s,q) \in P.S.
\end{aligned}$$

We denote the transitions in P as

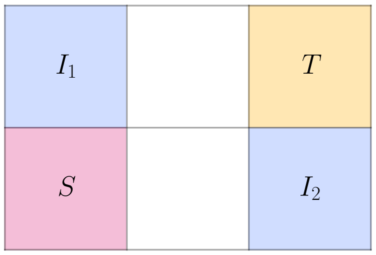
$$P.E \coloneqq \{(s,s') \,|\, s,s' \in P.S \text{ if } \exists a \in P.A \text{ s.t. } P.\delta(s,a) = s'\}. \tag{4.6}$$

An infinite sequence on P corresponds to a state-history trace $\vartheta = (s_0,q_0),\ldots,$ $(s_n,q_n)^{\omega}$. We refer to $(s,q) \in P.S$ as the state-history pair and define the corresponding path to be the finite prefix: $\vartheta_{fin} = (s,q)_0,(s,q)_1,\ldots,(s,q)_n$.

Example 4.1. The system under test can transition (N-S-E-W) on the grid world as illustrated in Fig. 4.3a. The initial condition of the system is marked by `S`, and the system is required to visit one of the goal states marked by T, $\varphi_{\text{sys}} = \diamondsuit T$. The test objective is to observe the system visit at least one of the I states before the system reaches its goal, encoded as $\varphi_{\text{test}} = \diamondsuit I$.



(a) Example 4.1.



(b) Example 4.2.

Figure 4.3: Grid world layouts for examples.

Example 4.2. In this example, the system under test can transition (N-S-E-W) on the grid world as illustrated in Fig. 4.3b. The initial condition of the system is marked by `S`, and the system objective is to visit `T`, $\varphi_{\text{sys}} = \diamondsuit T$. The test objective is to observe the system visit states `I`$_1$ and `I`$_2$: $\varphi_{\text{test}} = \diamondsuit I_1 \wedge \diamondsuit I_2$.

4.4 Problem Statement

For the improved class of specifications, the test environment synthesis problem can be stated as follows. Assuming that a test engineer specifies the desired test behavior (i.e., system and test objectives), we seek to synthesize a reactive test strategy under which every successful system execution will also be a successful test execution — every system trace that satisfies the system objective will also satisfy the test objective. The reactive test strategy restricts system actions that are available from the test harness. Formally, a reactive test strategy is defined as follows.

Definition 4.10 (Reactive Test Strategy)**.** A *reactive test strategy* $\pi_{\text{test}} : (T_{\text{sys}}.S)^* T_{\text{sys}}.S \to 2^{A_H}$ defines the set of restricted system actions at the current system state based on the prefix of the system trace up to the current state. For some finite prefix $\sigma_{0:k} = s_0, \dots, s_k$ starting from the system initial state $s_0 \in T_{\text{sys}}.S_0$, $\pi_{\text{test}}(\sigma_{0:k}) \subseteq H(s_k)$ is the set of restricted system actions from state s_k. A test environment *realizes* the strategy π_{test} if it restricts system actions according to π_{test}. The resulting system trace is denoted as $\sigma(\pi_{\text{sys}} \times \pi_{\text{test}})$.

More concretely, for some finite prefix $s_0, \dots, s_i$ of a system execution σ starting from an initial state $s_0 \in T.S_0$, $\pi_{\text{test}}(s_0, \dots, s_i) \subseteq H(s_i)$ denotes the set of actions unavailable to the system from state s_i of execution σ. These actions can be restricted by the test environment via static and reactive obstacles, and a dynamic test agent. The reactive test strategy must be such that it respects the system assumptions **A1–A4**. In turn, a correct system strategy must choose from actions available following the restrictions placed by the test environment. Formally, suppose $\Sigma_{\text{fin}} := (T_{\text{sys}}.S)^* T_{\text{sys}}.S$ be the set of all possible finite trace prefixes for the system, and at each time step $k \geq 0$, the system strategy $\pi_{\text{sys}} : \Sigma_{\text{fin}} \rightarrow T_{\text{sys}}.A \setminus \pi_{\text{test}}(\Sigma_{\text{fin}})$ must pick from unrestricted system actions from its current state.

By use of obstacles and/or test agents, the test environment externally blocks system transitions, and the system must correctly observe these obstacles and recognize the corresponding actions to be unsafe. We assume that the system can observe all restricted actions on its current state before it commits to an action, and therefore, a correct system strategy π_{sys} must choose from the available actions at each time step. Depending on the implemnentation, the system might have to re-plan its strategy π_{sys} in response to obstacles placed by the test environment.

Definition 4.11 (Feasibility of a Test Strategy)**.** Given a test environment, system T_{sys}, system and test objectives, φ_{sys} and φ_{test}, a reactive test strategy π_{test} is said to be *feasible* iff: i) the test environment can realize π_{test}, ii) there exists a correct system strategy π_{sys}, and iii) any execution corresponding to a correct π_{sys} satisfies the system and test objectives: $\sigma(\pi_{\text{sys}} \times \pi_{\text{test}}) \vDash \varphi_{\text{test}} \wedge \varphi_{\text{sys}}$.

The reactive test strategy does not help the system in achieving the system objective φ_{sys}; it only restricts the system such that the test objective can be realized. The system can choose an incorrect strategy π_{sys}, and in such a case, we cannot provide any guarantees. Furthermore, in routing the system to the test objective, it would be ideal if the test strategy does not overly restrict the system for the system to demonstrate decision-making when given the freedom to choose from multiple possible actions, including those that might be unsafe or lead the system to a livelock. For this reason, we will revisit the notion of the restrictiveness of tests defined over test executions. Given any system trace σ, every finite prefix $\sigma_{0:k} = s_0, \ldots, s_k$ maps to some history variable $q \in \mathcal{B}_\pi.Q$. Therefore, we can track this history variable along with the evolution of the system in a state-history trace $\vartheta = (s_0, q_0), (s_1, q_1), \ldots$, where the history variable q_k corresponds to the finite prefix $\sigma_{0:k}$. From now on, we refer to ϑ as the test execution, and proceed to define the restrictiveness of a test strategy in terms of the number of possible test executions. In the previous chapter, restrictiveness of a test was only defined on the system trace. Since we now have a broader class of specifications and use Büchi automata to track temporal events, we will define restrictiveness over test executions.

Definition 4.12 (Restrictiveness of a Test Strategy)**.** State-history traces ϑ_1 and ϑ_2 are *unique* if they do not share any consecutive state-history pairs — any two state-history pairs (s, q) and (s', q') do not appear in consecutive time steps in both ϑ_1 and ϑ_2. For a feasible π_{test}, let Σ be the set of all executions corresponding to correct system strategies, and let Θ be the set of all state-history traces corresponding to Σ. Let $\Theta_u \subseteq \Theta$ be a set of unique state-history traces. A test strategy π_{test} is *not overly restrictive* if the cardinality of Θ_u is maximized.

Remark 4.4. The set of all state history traces Θ can be infinite. However, the set Θ_u is finite because: i) the system has a finite number of states and the specification product has a finite number of history variables, and ii) every state-history trace in Θ_u is *unique* with respect to any other trace in Θ_u.

Problem 4.1 (Finding a Test Strategy)**.** Given a high-level abstraction of the system model T, test harness H, system objective φ_{sys}, test objective φ_{test}, find a feasible, reactive test strategy π_{test} that is not overly-restrictive.

To realize the test strategy, the test environment can place obstacles and use dynamic agents to restrict actions of the system. In the case of dynamic agents, the agent strategy must be found such that it *simulates* the restrictions set forth by the test strategy.

Problem 4.2 (Reactive Test Agent Strategy Synthesis)**.** Given a high-level abstraction of the system model T, test harness H, system objective φ_{sys}, test objective φ_{test}, and a test agent modeled by transition system T_{TA}. Find the test agent strategy π_{TA} and the set of static obstacles $\texttt{Obs}_{\text{static}}$ that: i) satisfy the system's assumptions on its environment, and ii) realize a reactive test strategy π_{test} that is not overly-restrictive and feasible. If the test agent cannot realize at least one reactive test strategy π_{test} that is not overly-restrictive, then find the π_{TA} that realizes the best possible π_{test}.

4.5 Graph Construction

To reason about executions of the system in relation to the system and test objectives, automata theory is leveraged to construct the following product graphs.

Definition 4.13 (Virtual Product Graph and System Product Graph)**.** A *virtual product graph* is the product transition system $G := T_{\text{sys}} \otimes \mathcal{B}_\pi$. Similarly, the system product graph is defined as $G_{\text{sys}} := T_{\text{sys}} \otimes \mathcal{B}_{\text{sys}}$.

The virtual product graph G tracks the test execution in relation to both the system and test objectives while the system product graph G_{sys} tracks the system objective. We will find the restrictions on system actions on G, while G_{sys} represents the system's perspective concerning the system objective during the test execution. For each node $u = (s, q) \in G.S$, we denote the corresponding state in $s \in T_{\text{sys}}.S$ as $u.s := s$. Similarly, the state corresponding to $v \in G_{\text{sys}}.S$ is denoted by $v.s := s$. For practical implementation, we remove nodes on the product graphs that are not reachable from the corresponding initial states, $G.S_0$ or $G_{\text{sys}}.S_0$.

Definition 4.14 (Projections)**.** States from G to G_{sys} are related via the *projection* map $\mathcal{P}_{G\to G_{\text{sys}}} : G.S \to G_{\text{sys}}.S$ as

$$\mathcal{P}_{G\to G_{\text{sys}}}((s, (q_{\text{sys}}, q_{\text{test}}))) = (s, q_{\text{sys}}). \tag{4.7}$$

These projections help us to reason about how restrictions found on G map to the system T_{sys} and the system product graph G_{sys}. We can now define the edges on G that we can restrict with the test harness as follows,

$$\begin{aligned} E_H = &\{((s, q), (s', q')) \in G.E \,|\, \forall s \in T_{\text{sys}}.S, \\ &\forall a \in H(s) \text{ s.t. } s' = T_{\text{sys}}.\delta(s, a)\}. \end{aligned} \tag{4.8}$$

Lemma 4.1. For every path $(s, q_{\text{sys}})_0, (s, q_{\text{sys}})_1, \ldots, (s, q_{\text{sys}})_n$ on G_{sys}, there exists at least one corresponding path on G.

Proof. Suppose there exists some $q_{\text{test}\,0}, \ldots, q_{\text{test}\,n} \in \mathcal{B}_{\text{test}}.Q$ such that $(s, (q_{\text{sys}}, q_{\text{test}}))_0, \ldots, (s, (q_{\text{sys}}, q_{\text{test}}))_n$ is a path on G. Then, by construction, there exists a path on G_{sys} where $(s, (q_{\text{sys}}, q_{\text{test}}))_k$ maps to $(s, q_{\text{sys}})_k$ for all $0 \le k \le n$. □

Paths on the virtual product graph G correspond to possible test executions. This is illustrated in Figures 4.4 and 4.5 for the example 4.2. We identify the nodes on G that capture the acceptance conditions for the system and test objectives.

Definition 4.15 (Source, Intermediate, and Target Nodes)**.** The *source node* S represents the initial condition of the system. The *intermediate nodes* I correspond to system states in which the test objective acceptance conditions are met. Finally, the *target nodes* T represent the system states for which the acceptance condition for the system objective is satisfied. Formally, these nodes are defined as follows,

$$\begin{aligned} &\mathtt{S} := \{(s_0, q_0) \in G.S \,|\, s_0 \in T_{\text{sys}}.S_0, q_0 \in \mathcal{B}_\pi.Q_0\}, \\ &\mathtt{I} := \{(s, (q_{\text{sys}}, q_{\text{test}})) \in G.S \,|\, q_{\text{test}} \in \mathcal{B}_{\text{test}}.F,\ q_{\text{sys}} \notin \mathcal{B}_{\text{sys}}.F\}, \\ &\mathtt{T} := \{(s, (q_{\text{sys}}, q_{\text{test}})) \in G.S \,|\, q_{\text{sys}} \in \mathcal{B}_{\text{sys}}.F\}. \end{aligned}$$

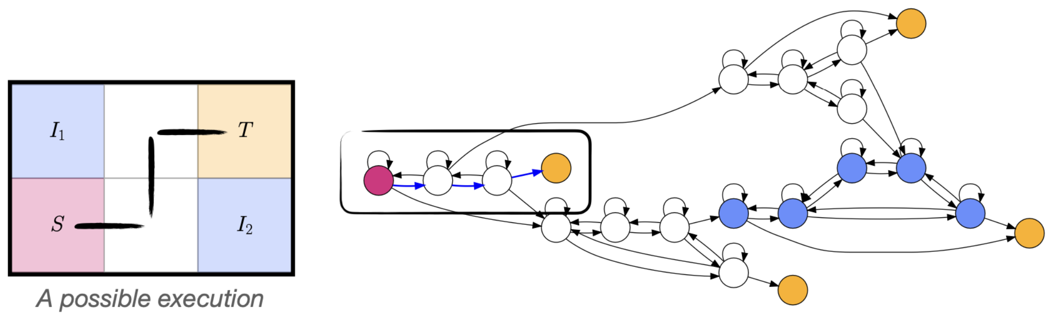


Figure 4.4: A possible execution of the system for Example 4.2 as illustrated on the transition system T_{sys} and the corresponding product graph G.

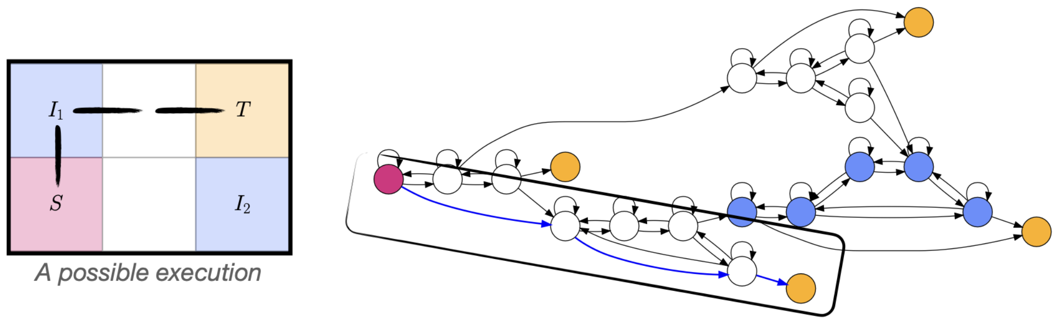


Figure 4.5: A possible execution of the system for Example 4.2 as illustrated on the transition system T_{sys} and the corresponding product graph G.

In addition, we define the set of states corresponding to the system acceptance condition on G_{sys} as $\mathtt{T}_{\text{sys}} := \{(s,q) \in G_{\text{sys}}.S \mid q \in \mathcal{B}_{\text{sys}}.F\}$.

Proposition 4.1. Every test execution corresponds to a path $\vartheta_n = (s,q)_0, \ldots, (s,q)_n$ on G where $(s,q)_0 \in \mathtt{S}$. The corresponding system trace σ_n satisfies the system objective, $\sigma \models \varphi_{\text{sys}}$ iff $(s,q)_n \in \mathtt{T}$. Furthermore, if $\sigma \models \varphi_{\text{test}}$, then the path ϑ_n contains a state-history pair $(s,q)_i \in \mathtt{I}$ for some $0 \leq i \leq n$.

Provided that there exists a path on G from $\mathtt{S}$ to $\mathtt{T}$, identifying a feasible reactive test strategy corresponds to identifying edges to cut on G. These edge cuts correspond to restricted system actions. In particular, these edge cuts are such that all paths on G from source $\mathtt{S}$ to target $\mathtt{T}$ visit the intermediate $\mathtt{I}$. Now, we will go over two new flow-based formulations to solve the routing problem.

4.6 Part I: Flow-based Optimization via Min-Max Stackelberg Games with Coupled Constraints

In the previous chapter, we covered flow-based test synthesis in which flow networks that were defined on the transition system of the system under test. The temporal logic objectives were also limited to a simple sequence of waypoints defined on the states of the system.

This new flow-based formulation has the following key advances: i) edge-cuts or restrictions are found on a product graph of the system transition system and a Büchi automaton representing the system and test objectives, ii) the new flow-based optimization no longer has exponential number of constraints, and is tractable to encode, and iii) the synthesized test is *reactive* to system behavior and no longer limited to static obstacles. Furthermore, this new flow-based reformulation represents edge-cuts as continuous variables, but the complexity still remains since the formulation becomes a min-max Stackelberg game with coupled constraints. In this part of the chapter, we will introduce the product graph and setup the flow-based reactive synthesis formulation. This work serves as a prelude to the next section in which we propose an MILP approach to solving the problem that comes with guarantees of synthesizing a test that is feasible and not overly-restrictive, and also lends to a more tractable implementation.

Stackelberg Game Formulation

Leveraging automata theory to represent product graphs leads to the intermediate node becoming analogous to the waypoints. Instead of a sequence of waypoints on $T_{\text{sys}}.S$, the nodes $\texttt{I}$ become the waypoint that the system must be routed through. Instead of defining flows for every pair of propositions, we will define three flows: from source to intermediate ($\mathbf{f}_{\texttt{S}\to\texttt{I}}$), from intermediate to sink ($\mathbf{f}_{\texttt{I}\to\texttt{T}}$), and a bypass flow ($F_{\texttt{S}\to\texttt{T}}$). The test strategy synthesis problem can be seen as placing restrictions such that flows $\mathbf{f}_{\texttt{S}\to\texttt{I}}$ and $\mathbf{f}_{\texttt{I}\to\texttt{T}}$ are preserved while the bypass flow is cut.

Definition 4.16 (Constrained Min-Max Optimization with Coupled Constraints [102])**.** A *constrained min-max optimization with dependent feasible sets*, also referred to as a min-max Stackelberg game, between the lead player X with strategy space $\mathcal{X}$ and a follower player Y with strategy space $\mathcal{Y}$ can be represented as the following optimization:

$$\begin{aligned} \min_{x\in\mathcal{X}} \max_{y\in\mathcal{Y}} \quad & f(x,y) \\ \text{s.t.} \quad & g(x,y) \geq 0, \end{aligned} \tag{4.9}$$

where $f(x,y) : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ is the objective function and $g(x,y) : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^k$ represents the constraints.

In this Stackelberg formulation, the outer (min) player is the test environment controls the flow variables $\mathbf{f}_{\texttt{S}\to\texttt{I}}$ and $\mathbf{f}_{\texttt{I}\to\texttt{T}}$, edge cuts $\mathbf{d}$, and the auxiliary variable t. Let $0 < t \leq 1$ be an auxiliary variable defined as: $t = \frac{1}{F}$, given that $F > 0$. Hereafter, all flow variables are normalized with respect to the total flow by multiplying with t. The objective function is such that the tester maximizes the total flow $F = \min\{F_{\texttt{S}\to\texttt{I}}, F_{\texttt{I}\to\texttt{T}}\}$, and minimizes the total bypass flow $F_{\texttt{S}\to\texttt{T}}$. Likewise, the system player maximizes bypass flow $F_{\texttt{S}\to\texttt{T}}$. Next, the constraints of the bilevel optimization are briefly described. A lot of these constraints are formally explained in the following section with MILPs, and are intuitively described here for brevity. The objective, which is given as, $t + \gamma F_{\texttt{S}\to\texttt{T}}$, where $\gamma > 0$ is a regularization parameter that penalizes the test environment for any non-zero bypass flow through the network. The auxiliary variable t is useful because the outer (min) player can minimize this term, thus maximizing the total flow value F. This objective also works for the inner (max) player, which in the worst-case, seeks to take a bypass path.

Table 4.2: List of outer player constraints used in Optimization 4.21 with normalized flows.

Outer Player Constraints	**Equation** $k \in \{\texttt{S} \to \texttt{I},\ \texttt{I} \to \texttt{T}\}$	
Capacity (exact)	$\forall e \in E, \quad d^e \in \{0, t\}, \quad 0 \leq f_k^e \leq t.$	(oc10)
Capacity (approx.)	$\forall e \in E, \quad 0 \leq d^e \leq t, \quad 0 \leq f_k^e \leq t.$	(oc11)
Conservation	$\forall v \in S \setminus \{\texttt{S}, \texttt{T}\}, \sum_{\substack{u:(u,v) \\ \in E}} f_k^{(u,v)} = \sum_{\substack{u:(v,u) \\ \in E}} f_k^{(v,u)}.$	(oc12)
Cut	$\forall e \in E, \quad d^e + f_k^e \leq t.$	(oc13)
Minimum Total Flow	$1 \leq F_{\texttt{S}\to\texttt{I}}$ and $1 \leq F_{\texttt{I}\to\texttt{T}}$	(oc14)
Feasibility	$F_{\mathcal{G}_{\text{sys}}}(q) \geq 1 \forall q \in \mathcal{B}_\pi.Q$	(oc15)
Static Obstacles	$d^{(i,j)} = d^{(k,l)}$ if $i.s = k.s$ and $j.s = l.s$	(oc16)

The three flows satisfy standard network flow constraints concerning capacity and conservation. The only difference: i) all standard flow constraints are normalized by multiplying through t, and ii) the cut variable $\mathbf{d}$ is relaxed as opposed to being restricted to vectors with binary elements. Furthermore, the cut variable restricts flows as follows, for all $k \in \{\texttt{S} \to \texttt{I}, \texttt{I} \to \texttt{T}, \texttt{S} \to \texttt{T}\}$, the cut constraints are:

$$\forall e \in \mathcal{G}.E, \quad d^e + f_k^e \leq t.$$

Table 4.3: List of inner player constraints used in Optimization 4.21 with normalized flows.

Inner Player Constraints	Equation	
Cut	$\forall e \in E, \quad d^e + f^e_{\mathtt{S}\to\mathtt{T}} \leq t.$	(ic17)
Capacity (approx.)	$\forall e \in E, \quad 0 \leq f^e_{\mathtt{S}\to\mathtt{T}} \leq t.$	(ic18)
Conservation	$\forall v \in S \setminus \{\mathtt{S},\mathtt{T}\}, \sum_{u\in V} f^{(u,v)}_{\mathtt{S}\to\mathtt{T}} = \sum_{u:(v,u)\in E} f^{(v,u)}_{\mathtt{S}\to\mathtt{T}}.$	(ic19)
No $\mathtt{I}$ Flow	$f^{(u,v)}_{\mathtt{S}\to\mathtt{T}} = 0$ if $u \in \mathtt{I}$ or $v \in \mathtt{I}$.	(ic20)

That is, despite having multiple flows, they do not compete for capacity; a cut d^e will equally restrict all flows. Note that the above cut constraint for the case of bypass flow $k = \mathtt{S} \to \mathtt{T}$ is the coupling constraint between inner and outer players; the test environment controls the cut and the inner player controls bypass flow, but they must together respect this cut constraint. The constraints on the outer and inner players for the game-based network flow optimization are given in Tables 4.2 and 4.3. The constraint on minimum total flows in equation (ic(4.14)) is applied to normalized total flow values $F_{\mathtt{S}\to\mathtt{I}}$ and $F_{\mathtt{I}\to\mathtt{T}}$, and implies that the total flow $F > 0$. If this constraint is violated, the constraints are infeasible and no solution is returned. A detailed approach on these network flow constraints (e.g., feasibility constraints in Eq. (ic(4.15))) is given in the next section when we re-use these constraints to setup the final MILP formulation to solve the routing problem. The game based network flow optimization formulation is given below.

MCF-OPT(γ):

$$\min_{\substack{\mathbf{f}_{\mathtt{S}\to\mathtt{I}}\mathbf{f}_{\mathtt{I}\to\mathtt{T}},\mathbf{d},t,\\ \mathbf{f}_{\mathtt{S}_{\text{sys}}\to\mathtt{T}_{\text{sys}}}(q),\forall q\in\mathcal{B}_\pi.Q}} \quad \max_{\mathbf{f}_{\mathtt{S}\to\mathtt{T}}} \quad t + \gamma \sum_{v:e=(\mathtt{S},v)\in\mathcal{G}.E} f^e_{\mathtt{S}\to\mathtt{T}} \tag{4.21}$$
$$\text{s.t.} \quad \text{(oc(4.11))-(oc(4.15)), (ic(4.17))-(ic(4.20)).}$$

This optimization is in the form of a min-max Stackelberg game with dependent constraint sets studied in [103]. However, there are no known polynomial-time solutions to solve this optimization since its value function is not convex. Given below, the value function outputs the optimal value of the inner optimization problem in Optimization (4.21) for any choice of $(\mathbf{f}, \mathbf{d}, t)$. The flow values F and $F_{\mathtt{S}\to\mathtt{T}}$

are functions of the edge cuts $\mathbf{d}$ — they represent max-flow on their corresponding flow networks with the capacities reduced by $\mathbf{d}$.

$$\begin{aligned} V(\mathbf{f}, \mathbf{d}, t) = \max_{\mathbf{f}_{\mathtt{S}\to\mathtt{T}}} \quad & t + \gamma F_{\mathtt{S}\to\mathtt{T}} \\ \text{s.t.} \quad & (\text{ic}(4.17)) – (\text{ic}(4.20)). \end{aligned}$$

The *value function* is defined over the space of outer player variables that satisfy constraints (oc(4.17)), (oc(4.18)), (oc(4.19)), (oc(4.20)). If the value function is convex in the outer player variables, then there exist efficient algorithms to converge to the Stackelberg equilibrium [102]. However, our value function is not convex because the inner problem corresponds to solving a max flow problem parameterized by cuts $\mathbf{d}$, which is not convex. Note that this problem complexity is despite the fact that the objective and constraints are all affine and defined over continuous valued domains. The beaver rescue and motion primitive hardware experiments are derived from solutions to **MCF-OPT**(γ) for $\gamma = 1000$, which is solved using Pyomo [104]. Also note that all Stackelberg equilibria for **MCF-OPT**(γ) need not correspond to bypass flow value $F_{\mathtt{S}\to\mathtt{T}}$ being zero. This shortcoming is handled in the MILP formulation presented in the next section, and is largely driven by insights from taking the dual of the inner maximization, as we shall see below.

However, this approach did not scale to solving medium-sized examples that we will see later in this chapter. In an effort to address this, the first attempt was to take the Lagrange dual of the inner maximization, and solve a minimization instead of a min-max game. In traditional max-flow problems, the Lagrange dual of the max flow linear program is the minimum cut linear program with Lagrange multipliers corresponding to edge cuts and partitions [105]. However, in our case, since the outer player modifies edge cuts $\mathbf{d}$, the Lagrange multipliers correspond to paritions on the modified graph but do not inform the actual partition that we seek. The Lagrangian $\mathcal{L}$ associated with the inner player is:

$$\begin{aligned} \mathcal{L}\left(\mathbf{f}, \mathbf{d}, t, \mathbf{f}_{\mathtt{S}\to\mathtt{T}}, \boldsymbol{\lambda}, \boldsymbol{\mu}, \boldsymbol{\nu}\right) = t + \gamma F_{\mathtt{S}\to\mathtt{T}} + \sum_{v \in V \setminus \{\mathtt{S},\mathtt{I},\mathtt{T}\}} \mu^i \left(\sum_{u \in V \setminus \mathtt{I}} \mathbf{f}^{(u,v)}_{\mathtt{S}\to\mathtt{T}} - \sum_{u \in V \setminus \mathtt{I}} \mathbf{f}^{(v,u)}_{\mathtt{S}\to\mathtt{T}} \right) \\ + \sum_{\substack{(u,v) \in E \setminus E(\mathtt{I}) \\ u \in \mathtt{T}, v \in \mathtt{S}}} \nu^{(u,v)} \mathbf{f}^{(u,v)}_{\mathtt{S}\to\mathtt{T}} + \sum_{e \in E \setminus E(\mathtt{I})} \lambda^e \left(t - d^e - \mathbf{f}^e_{\mathtt{S}\to\mathtt{T}} \right), \quad (4.22) \end{aligned}$$

where $\boldsymbol{\lambda}, \boldsymbol{\mu}, \boldsymbol{\nu}$ are the Lagrange variables; $\boldsymbol{\lambda}$ is associated with edge-cuts and $\boldsymbol{\mu}$ represents the partition of nodes. Finding the optimal Lagrange multipliers results in following dual problem:

$$\begin{aligned} \min_{\boldsymbol{\lambda}\in\mathbb{R}_+^{|E|},\boldsymbol{\mu}\in\mathbb{R}_+^{|V|}} \quad & t+\gamma\sum_{e\in E}\lambda^e(t-d^e) \\ \text{s.t.} \quad & \mu^{\mathtt{S}}-\mu^{\mathtt{T}}\geq 1, \\ & \lambda^{(u,v)}-\mu^u+\mu^v\geq 0,\ \forall(u,v)\in E\setminus E(\mathtt{I}). \end{aligned} \tag{4.23}$$

This dual corresponds to the dual of the max-flow problem on the graph (without nodes $\mathtt{I}$) with edge capacities $t-\mathbf{d}$. In the canonical max-flow problem, $\boldsymbol{\mu}$ represents node partitioning corresponding to minimum cut. In our case, the solution to this dual problem returns optimal Lagrange multipliers $\boldsymbol{\mu}$ as a function of the outer player variable $\mathbf{d}$. Therefore, the choice of $\mathbf{d}$ by the outer player affects $\boldsymbol{\mu}$. The partition of the graph $G\setminus\mathtt{I}$, characterized by $\boldsymbol{\mu}^*(\mathbf{d}^*)$, will be the optimal lagrange multiplier at the equilibrium $\mathbf{d}^*$. Due to strong duality, we can rewrite the **MCF-OPT**(γ) equivalently as follows:

OPT-MIN(γ)

$$\begin{aligned} \min_{\mathbf{f},\mathbf{d},t,\boldsymbol{\lambda}\in\mathbb{R}_+^{|E|},\boldsymbol{\mu}\in\mathbb{R}_+^{|V|}} \quad & t+\gamma\sum_{e\in E}\lambda^e(t-d^e) \\ \text{s.t.} \quad & \mu^{\mathtt{S}}-\mu^{\mathtt{T}}\geq 1, \\ & \lambda^{(u,v)}-\mu^u+\mu^v\geq 0,\ \forall(u,v)\in E\setminus E(I), \\ & \text{(oc(4.11))-(oc(4.15))}. \end{aligned} \tag{4.24}$$

Despite being a single minimization, **OPT-MIN**(γ) has the structure of a *bilinear* program, which is also a consequence of our min-max Stackelberg game not being convex-concave. In general, solving a bilinear program is NP-hard in the problem data [106]. For zero bypass flow, the second bilinear term must be zero. This was also empirically observed when solving **OPT-MIN**(γ). At zero bypass flow, the term λ becomes equal to the unnormalized cut value d; if $d^{(u,v)}=t$, then $\lambda^{(u,v)}$ must equal 1 to partition nodes u and v into separate groups, and if $d^{(u,v)}=0$, then $\lambda^{(u,v)}=0$ to ensure that the nodes remain in the same group. First, we can recognize that the dual variables $\boldsymbol{\lambda}$ and $\boldsymbol{\mu}$ can take integer values [105]. This allows us to formulate the problem as a mixed-integer linear program, where we can encode zero bypass flow as a constraint by setting $\lambda^e=d^e$ for edges $e\in E\setminus E(\mathtt{I})$. Furthermore, this formulation also allows the use of the unnormalized flow and capacity values, removing the need for the auxiliary parameter t.

4.7 Part II: Flow-based Optimization via Mixed-Integer Linear Programming

Once again, we revisit a network flow formulation to solve the routing optimization for the expanded class of specifications. Recall that $Paths(\mathtt{S}, \mathtt{T})$ on the graph G are the set of possible test executions. Therefore, the set of edge cuts on G must be such that all $Paths(\mathtt{S}, \mathtt{T})$ are routed to visit at least one node in the intermediate set $\mathtt{I}$. The set of edge cuts to achieve this need not be unique, and therefore, we also require that resulting test strategy to not be overly-restrictive. Additionally, we also minimize the cardinality of the set of edge cuts to remove unnecessary restrictions. In comparison to the previous chapter, the formulation presented defines just a single flow network. Furthermore, the routing problem is solved as an MILP as opposed to a min-max game with coupled constraints. This new formulation allows us to derive guarantees that the optimal solution will provide a test strategy that solves Problem 4.1. Furthermore, this formulation easily lends itself to extensions (e.g., adding auxiliary constraints, excluding certain solutions, accommodating various types of test environments).

Consider the flow network $\mathcal{G} = (V, E, (\mathtt{S}, \mathtt{T}))$ defined based on the graph G: nodes are defined exactly as $V := G.V$, and edges $E := G.E \backslash \{(u, u) \in G.E \,|\, u \in V\}$ are the same as G with the exception of self-loops, and the source and target correspond to $\mathtt{S}$ and $\mathtt{T}$, respectively. The reason for introducing flow networks separately is to maintain a representation without self-loop transitions which are not relevant when computing the flow on a graph. Maintaining self-loops on G, however, is important since it is the product between a transition system and a Büchi automaton. For simplicity, notation for nodes (V) and edges (E) is shared between the graph G and its network $\mathcal{G}$ since self-loop transitions are also not a part of the test harness. If self-loops become important, the notation $G.E$ will be explicitly used. On $\mathcal{G}$, we introduce the flow vector $\mathbf{f} \in \mathbb{R}^{|E|}$ and a Boolean vector $\mathbf{d} \in \mathbb{B}^{E \backslash E(\mathtt{I})}$ carries the edge-cut value for each edge. For some $e \in E$, $d^e = 1$ denotes that edge e is cut and the corresponding system action is restricted, and $d^e = 0$ denotes that the edge remains. Immediately, it can be specified that all edges outside the test harness cannot be restricted:

$$\begin{aligned} &d^e \in \{0, 1\}, \quad \forall e \in E, \text{ and} \\ &d^e = 0, \quad \forall e \notin E_H. \end{aligned} \tag{c1}$$

The set $E(\mathtt{I}) = \{(u, v) \in E \,|\, u \in \mathtt{I} \text{ or } v \in \mathtt{I}\}$ is the set of edges that enter or exit from a node in set $\mathtt{I}$.

Objective. Among all possible sets of edge cuts that route test executions through $\mathtt{I}$ (corresponding to satisfying the test objective), we seek a test strategy that is not overly-restrictive. Thus, we optimize for a set of edge cuts that maximizes the total flow from $\mathtt{S}$ to $\mathtt{T}$ on $\mathcal{G}$. Since the edges have unit capacity, the set of edge cuts that maximize the total flow will also result in the largest set Θ_u (see Lemma 4.3). Thus, maximizing the flow alone is sufficient to get a test that is not overly-restrictive (see Remark 4.7). In addition, unnecessary edge-cuts can be reduced by introducing a second term of subtracting the fraction of edges that are cut from the flow value:

$$\sum_{\substack{(u,v)\in E,\\ u\in\mathtt{S}}} f^{(u,v)} - \frac{1}{|E|}\sum_{e\in E} d^e. \tag{4.25}$$

The regularize $\frac{1}{|E|}$ is chosen to avoid trade-off between the terms. Due to binary edge cuts and integer edge capacity, the maximum flow value on the graph will always be an integer. The second term, however, will always take a fractional value between 0 and 1 corresponding to none of the edges being cut to all of the edges being cut. Thus, maximizing the objective will always favor increasing the first term as much as possible, and then minimizing the number of edges that are cut.

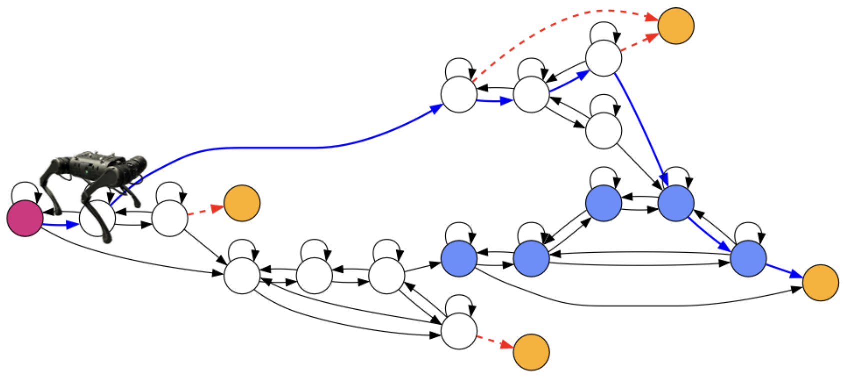
Network Flow Constraints. The flow vector $\mathbf{f}$ is subject to the standard network flow constraints:

$$\text{Flow constraints (3.1), (3.2), and (3.3) on flow network } \mathcal{G}. \tag{c2}$$

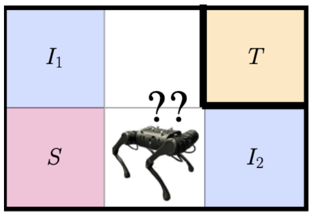
Next, an edge that is cut restricts the flow on that edge completely. However, an edge that is not cut may or may not have flow pushed through it:

$$\forall e \in E, \quad d^e + f^e \leq 1. \tag{c3}$$

Partition Constraints. In standard max-flow problems, the dual min-cut formulation has partition constraints that group nodes across a cut into two groups, one of which contains the source and the other that contains the sink [105]. The max-flow problem (and equivalently, the min-cut problem) is totally unimodular, implying that there exists an optimal integer solution. Our problem differs from the standard max-flow/min-cut problem in that we seek all $Paths(\mathtt{S},\mathtt{T})$ to be routed through $\mathtt{I}$. Alternatively, the set of edge cuts should be such that there exists a positive total flow on $\mathcal{G}$, but the network $(V \setminus \mathtt{I}, E \setminus E(\mathtt{I}), (\mathtt{S},\mathtt{T}))$ is fully partitioned. To capture this partitioning requirement, the partition conditions from standard settings is



(a) A path on G exists despite the cuts not being feasible for the system.



(b) Restrictions from the system perspective before it visits I_1 or I_2.

Figure 4.6: Illustration of why feasibility constraints are important for identifying a reactive test strategy that respects the system's assumptions. Since the system is not aware of the test objective, such a placement of constraints would lead to all paths being blocked from the system perspective.

adapted as follows. All nodes except those in I must be partitioned into the source S group or the sink T group. For this, introduce the variable $\boldsymbol{\mu} \in \mathbb{R}^{|V \setminus \mathtt{I}|}$ such that:

$$\begin{aligned} \mu^{\mathtt{S}} - \mu^{\mathtt{T}} &\geq 1, \\ 0 \leq \mu^v &\leq 1, \forall v \in V \setminus \mathtt{I}. \end{aligned} \tag{c4}$$

The partition condition is applied to edges that are not incoming or outgoing from I:

$$d^{(u,v)} - \mu^u + \mu^v \geq 0, \; \forall (u,v) \in E \setminus E(I). \tag{c5}$$

Despite the vector $\boldsymbol{\mu}$ being real-valued, it only appears in these constraints. Therefore, the partition conditions form a block diagonal in the constraint matrix, and this block diagonal sub-matrix is totally unimodular. Therefore, we preserve the partitioning properties from standard min-cut despite adapting the constraint to our problem.

Feasibility Constraints. These constraints ensure that the synthesized test is feasible from the system's perspective (see Figure 4.6), that is, restrictions placed by the test strategy should be such that the system still has a chance of successfully navigating to the goal if it has not committed to an incorrect action (either unsafe or one that inevitably leads to livelock) up to that point. For this, the reactive edge cuts must respect system assumptions **A2 – A4**. For every history variable $q \in \mathcal{B}_\pi.Q$, the function $\mathtt{S}_G : \mathcal{B}_\pi.Q \to G.S$ is the set of states on G when the the test execution can enter the history variable q, and is defined as follows,

$$\mathtt{S}_G(q) := \{(s, q) \in G.S \mid \forall((\bar{s}, \bar{q}), (s, q)) \in G.E,\ \bar{q} \neq q\}. \tag{4.26}$$

On the system product graph, these states map to the set:

$$\mathtt{S}_{G_{\text{sys}}}(q) := \{u \in G_{\text{sys}}.S \mid u = \mathcal{P}_{G \to G_{\text{sys}}}(v),\ v \in \mathtt{S}_G(q),\ \text{and} \ \exists\, Path(u, \mathtt{T}_{\text{sys}})\}, \tag{4.27}$$

where this set is empty if no path from the node u to $\mathtt{T}_{\text{sys}}$ exists on G_{sys}. For each $q \in \mathcal{B}_\pi.Q$, for each source in $\mathtt{s} \in \mathtt{S}_{G_{\text{sys}}}(q)$, define a flow network $\mathcal{G}_{\text{sys}}^{(q,\mathtt{s})} := (V_{\text{sys}}, E_{\text{sys}}, (\mathtt{s}, \mathtt{T}_{\text{sys}}))$, where nodes are $V_{\text{sys}} := G_{\text{sys}}.S$, and edges are $E_{\text{sys}} := G_{\text{sys}}.E \setminus \{(u, u) \in G_{\text{sys}}.E | u \in G_{\text{sys}}.V\}$. On graph $\mathcal{G}_{\text{sys}}^{(q,\mathtt{s})}$, flow vector is denoted as $\mathbf{f}_{\text{sys}}^{(q,\mathtt{s})}$. All such flow vectors are subject to the standard network flow constraints:

$$\begin{aligned} &\forall q \in \mathcal{B}_\pi.Q, \forall \mathtt{s} \in \mathtt{S}_{G_{\text{sys}}}(q), \\ &\text{Flow constraints (3.1), (3.2), and (3.3) on network } \mathcal{G}_{\text{sys}}^{(q,\mathtt{s})}. \end{aligned} \tag{c6}$$

The edge cut vector $\mathbf{d}$ from $\mathcal{G}$ is directly related to mapped to edges on each graph $\mathcal{G}_{\text{sys}}^{(q,\mathtt{s})}$. Then, it is checked whether there exists a $Path(\mathtt{s}, \mathtt{T}_{\text{sys}})$ on the system product graph copy $\mathcal{G}_{\text{sys}}^{(q,\mathtt{s})}$. Since edge-cuts/restrictions are placed reactively, only edge cuts starting from a state-history pair with history variable q applies to the graph $\mathcal{G}_{\text{sys}}^{(q,\mathtt{s})}$. Edges are grouped by the history variable using the mapping $\mathtt{Gr} : \mathcal{B}_\pi.Q \to 2^{G.E}$:

$$\mathtt{Gr}(q) := \{((s, q), (s', q')) \in G.E\}. \tag{4.28}$$

Then, the edge-cut values $\mathbf{d}$ are mapped onto system product graph copies, impacting the flow $f_{\text{sys}}^{(q,\mathtt{s}}$ by the cut constraint:

$$\begin{aligned} &\forall q \in \mathcal{B}_\pi.Q, \forall \mathtt{s} \in \mathtt{S}_{G_{\text{sys}}}(q), \forall (u, v) \in \mathtt{Gr}(q), \forall (u', v') \in E_{\text{sys}}, \\ &\quad d^{(u,v)} + f_{\text{sys}}^{(q,\mathtt{s})^{(u',v')}} \leq 1, \ \text{if } u'.s = u.s \text{ and } v'.s = v.s. \end{aligned} \tag{c7}$$

Once the cuts have been mapped, the check for a $Path(\mathtt{s}, \mathtt{T}_{\text{sys}})$ is ensured by requiring the flow value on each system product graph copy to be at least 1:

$$\sum_{(\mathtt{s},v) \in E_{\text{sys}}} f_{\text{sys}}^{(q,\mathtt{s})^{(\mathtt{s},v)}} \geq 1,\ \forall q \in \mathcal{B}_\pi.Q,\ \forall \mathtt{s} \in \mathtt{S}_{G_{\text{sys}}}(q). \tag{c8}$$

In the reactive obstacle setting, the feasibility constraints (c6)-(c8) group edge cuts for each history variable q, and check if there is a feasible path for the system. This feasibility check is carried out for every history variable q and every possible node $\mathtt{s} \in \mathtt{S}_G(q)$ at which the test execution enters history variable q. Since the system controller is unknown, it becomes imperative to check feasibility on every copy $\mathcal{G}_{\text{sys}}^{(q,\mathtt{s})}$.

Finally, the routing optimization is characterized as the following mixed-integer linear program (MILP) with the edge-cut vector $\mathbf{d}$ being the integer variable, and the flow $\mathbf{f}$ and partition $\boldsymbol{\mu}$ variables being continuous:

MILP-REACTIVE:

$$\begin{aligned} \max_{\substack{\mathbf{f},\mathbf{d},\boldsymbol{\mu},\\ \mathbf{f}_{\text{sys}}^{(q,\mathtt{s})}\ \forall q\in\mathcal{B}_\pi.Q\ \forall \mathtt{s}\in\mathtt{S}_{\mathcal{G}_{\text{sys}}}(q)}} \quad & F - \frac{1}{|E|}\sum_{e\in E} d^e \\ \text{s.t.} \quad & \text{(c1)-(c3)}, \text{(c4)-(c5)}, \text{(c6)-(c8)}. \end{aligned} \tag{4.29}$$

Static Constraints. The feasibility constraints are simplified in test environments comprising only of static obstacles. A static obstacle is one that remains for the entire duration of the test. That is, an restriction on a system action at a particular state should always be in place regardless of the current state-history pair of the test execution. To specify this, the edges in G that correspond to the same transition of the system in $T_{\text{sys}}.E$ will have the same edge cut value:

$$d^{(u,v)} = d^{(u',v')},\ \forall (u,v),(u',v') \in E,\ \text{if } u.s = u'.s \text{ and } v.s = v'.s. \tag{c9}$$

With Eq. (c9), a separate check for feasible paths on copies of the system product graph is not needed. By the projection map $\mathcal{P}_{G\to G_{\text{sys}}}$, a path on G implies a path on G_{sys}, and since restrictions do not change with q, the path on G_{sys} always remains. Therefore, the MILP formulation for static obstacles can be simplified to be:

MILP-STATIC:

$$\begin{aligned} \max_{\mathbf{f},\mathbf{d},\boldsymbol{\mu}} F - \frac{1}{|E|}\sum_{e\in E} d^e \\ \text{s.t.} \quad \text{(c1)-(c3)}, \text{(c4)-(c5)}, \text{(c9)}. \end{aligned} \tag{4.30}$$

The following lemma and proof was taken from [107].

Lemma 4.2. For the case of static constraints, due to (c9), ensuring feasibility from the system's perspective is guaranteed by checking $F > 0$ on G. That is, $F > 0$ on G is equivalent to checking (c6)-(c8).

Proof. Under (c9), the edge groupings $\texttt{Gr}(q)$ become the same for all $q \in \mathcal{B}_\pi.Q$. Thus, the constraints (c6)-(c8) can be reduced onto a single flow network $\mathcal{G}_{\text{sys}} = (V_{\text{sys}}, E_{\text{sys}}, (\texttt{S}_{\text{sys}}, \texttt{T}_{\text{sys}}))$, where $\texttt{S}_{\text{sys}} := G_{\text{sys}}.I$. Equation (c8) being satisfied on $\mathcal{G}_{\text{sys}}$ implies that there is a path on G from $\texttt{S}$ to $\texttt{T}$ via Lemma 4.1. Additionally, if there is a path on G from $\texttt{S}$ to $\texttt{T}$ with the static constraints (c9), then it must be that there exists a path from $\texttt{S}_{\text{sys}}$ to $\texttt{T}_{\text{sys}}$ on G_{sys}. $\square$

Remark 4.5. The reactive feasibility check involved checking for feasible paths on copies of system product graphs. Alternatively, this check can also be carried out using copies of the network $\mathcal{G}$, as we will see later in the section on computational complexity (Section 3.8). For implementation purposes, we choose the feasibility formulation presented in this section since it results in fewer variables and constraints in the optimization.

Mixed Constraints. In test environments with a mix of static obstacles and reactive obstacles and/or dynamic test agents, we require the static area $T_{\text{sys}}.E_{\text{static}} \subseteq T_{\text{sys}}.E$ to be given. Transitions in $T_{\text{sys}}.E_{\text{static}}$ can be restricted using static obstacles. In such mixed settings, the feasibility constraints (c6)–(c8) can be applied as normal, and the static constraints given in (c9) can be applied on edges $(u, v) \in E$ whose mapping onto the system transitions is in the static area, i.e., $(u.s, v.s) \in T_{\text{sys}}.E_{\text{static}}$.

MILP-MIXED:

$$\begin{aligned} \max_{\substack{\mathbf{f},\mathbf{d},\boldsymbol{\mu},\\ \mathbf{f}_{\text{sys}}^{(q,\mathtt{s})}\,\forall q\in\mathcal{B}_\pi.Q\,\forall \mathtt{s}\in\mathtt{S}_{\mathcal{G}_{\text{sys}}}(q)}} \quad & F - \frac{1}{|E|}\sum_{e\in E} d^e \\ \text{s.t.} \quad & \text{(c1)-(c3)}, \text{(c4)-(c5)}, \text{(c6)-(c8)}, \text{(c9)}. \end{aligned} \tag{4.31}$$

Auxiliary Constraints. Auxiliary constraints are any additional affine constraints that are not required but can be added to the optimization to accommodate the test environment. For example, in some instances such as placing static obstacles like doors or fences, restricting a directed edge would also require the transition in the reverse direction to be blocked. This specific affine constraint can be written as

$$d^{(u,v)} = d^{(u',v')},\ \forall (u,v),\ (u',v') \in E, \text{if } u.s = v'.s \text{ and } v.s = u'.s. \qquad \text{(c10)}$$

Algorithm 5: Finding the test strategy π_{test}

```
1:  procedure FindTestStrategy($T_{\text{sys}}, H, \varphi_{\text{sys}}, \varphi_{\text{test}}$)
      Input: transition system $T_{\text{sys}}$, test harness $H$, system objective $\varphi_{\text{sys}}$, test objective $\varphi_{\text{test}}$
      Output: test strategy $\pi_{\text{test}}$
2:      $\mathcal{B}_{\text{sys}} \leftarrow \text{BA}(\varphi_{\text{sys}})$                ▷ System Büchi automaton
3:      $\mathcal{B}_{\text{test}} \leftarrow \text{BA}(\varphi_{\text{test}})$              ▷ Tester Büchi automaton
4:      $\mathcal{B}_{\pi} \leftarrow \mathcal{B}_{\text{sys}} \otimes \mathcal{B}_{\text{test}}$                ▷ Specification product
5:      $G_{\text{sys}} \leftarrow T_{\text{sys}} \otimes \mathcal{B}_{\text{sys}}$                  ▷ System product
6:      $G \leftarrow T_{\text{sys}} \otimes \mathcal{B}_{\pi}$                     ▷ Virtual Product Graph
7:      $\texttt{S}, \texttt{I}, \texttt{T} \leftarrow \text{IdentifyNodes}(G, \mathcal{B}_{\text{sys}}, \mathcal{B}_{\text{test}})$
8:      $\mathcal{G} \leftarrow \text{DefineNetwork}\ (G, \texttt{S}, \texttt{T})$
9:      $\mathfrak{G}_{\text{sys}} \leftarrow \texttt{set}()$                       ▷ System Perspective Graphs
10:     for $q \in \mathcal{B}_{\pi}.Q$ do
11:         for $\texttt{s} \in \texttt{S}_{G_{\text{sys}}}(q)$ do
12:             $\mathcal{G}_{\text{sys}}^{(\texttt{s},q)} \leftarrow \text{DefineNetwork}(G_{\text{sys}}, \texttt{s}, \texttt{T}_{\text{sys}})$
13:             $\mathfrak{G}_{\text{sys}} \leftarrow \mathfrak{G}_{\text{sys}} \cup \mathcal{G}_{\text{sys}}^{(\texttt{s},q)}$
14:     $\mathbf{d}^* \leftarrow \text{MILP}(\mathcal{G}, T, \mathfrak{G}_{\text{sys}}, \texttt{I}, H)$          ▷ Reactive, static, or mixed.
15:     $C \leftarrow \{(u,v) \in G.E \mid \mathbf{d}^{*(u,v)} = 1\}$                 ▷ Cuts on $G$
16:     $\pi_{\text{test}} \leftarrow$ Define test strategy according to equation (4.33)
17:     return $\pi_{\text{test}}$
```

Characterizing Optimization Results

The flow value (3.4) of the network is always integer-valued since the edge cuts are binary, and therefore, any strictly positive flow value corresponds to at least one valid test execution. In the following cases, the problem data are *inconsistent* and a flow value ≥ 1 cannot be found.

Case 1: There is no path from $\texttt{S}$ to $\texttt{T}$ on G (and equivalently, no path from $\texttt{S}_{\text{sys}}$ to $\texttt{T}_{\text{sys}}$ on G_{sys}). In this case, the optimization will not have to place any cuts because the only possible maximum flow value is 0.

Case 2: There is a path from $\texttt{S}$ to $\texttt{T}$ on G, but there is no path $\texttt{S}$ to $\texttt{T}$ in G visiting an intermediate node in $\texttt{I}$. In this case, the partition constraints will cut all paths from $\texttt{S}$ to $\texttt{T}$, while by Lemma 4.1 the feasibility constraints require a path to exist from $\texttt{S}$ to $\texttt{T}$—a contradiction. The optimization is infeasible in this instance.

For each MILP, the set of edges that are cut are found from the optimal $\mathbf{d}^*$ as follows, $C \coloneqq \{(u,v) \in E \setminus E(\mathtt{I}) \,|\, d^{*(u,v)} = 1\}$, resulting in the cut network $\mathcal{G}_{\text{cut}} = (V, E \setminus C, (\mathtt{S}, \mathtt{T}))$. The bypass flow value is computed on the network $\mathcal{G}_{\text{byp}} \coloneqq (V_{\text{byp}}, E_{\text{byp}}, (\mathtt{S}, \mathtt{T}))$, where $V_{\text{byp}} \coloneqq V \setminus \mathtt{I}$, and $E_{\text{byp}} \coloneqq E \setminus (E(\mathtt{I}) \cup C)$. A strictly positive bypass flow value indicates the existence of a $Path(\mathtt{S}, \mathtt{T})$ on $\mathcal{G}_{\text{cut}}$ that does not visit an intermediate node in $\mathtt{I}$.

Theorem 4.1. The optimal or feasible cuts C returned by each MILP result in a bypass flow value of 0.

Proof. The partition constraints (c4) and (c5) partition the set of vertices $V \setminus \mathtt{I}$ into two groups: nodes with potential $\mu = 0$ (e.g., $\mathtt{T}$) and nodes with potential $\mu = 1$ (e.g., $\mathtt{S}$). On any path $v_0 \dots v_k$ on $\mathcal{G}_{\text{byp}}$, where $v_0 = \mathtt{S}$ and $v_k = \mathtt{T}$, the difference in potential values can be expressed as a telescoping sum: $\sum_{i=0}^{k-1}(\mu^i - \mu^{i+1}) = \mu^{\mathtt{S}} - \mu^{\mathtt{T}}$. Then, by partition constraints (c4) and (c5),

$$\sum_{i=0}^{k-1} d^{(v_i, v_{i+1})} \geq \sum_{i=0}^{k-1}(\mu^i - \mu^{i+1}) = \mu^{\mathtt{S}} - \mu^{\mathtt{T}} \geq 1.$$

Therefore, for at least one edge (v_i, v_{i+1}) on the path, where $0 \leq i \leq k-1$, the corresponding cut value is $d^{(v_i, v_{i+1})} = 1$. These edges belong to the set of cut edges C. Thus, the flow value on $\mathcal{G}_{\text{byp}}$ is zero. □

Theorem 4.2. For each MILP, the returned cuts C are such that there always exists a path to the goal from the system's perspective.

Proof. First, consider the MILP in the reactive setting. The optimal cuts C satisfy the feasibility constraints (c6), (c7), and (c8). These constraints ensure that for each history variable $q \in \mathcal{B}_\pi.Q$, there exists a path for the system from each state $\mathtt{s} \in \mathtt{S}_{G_{\text{sys}}}(q)$ to $\mathtt{T}_{\text{sys}}$ on G_{sys}. The edge cuts C are grouped by their history variable (see equation (4.28)) and mapped to the corresponding $\mathcal{G}_{\text{sys}}^{(q,\mathtt{s})}$ (see equation (c7)). Then, each copy $\mathcal{G}_{\text{sys}}^{(q,\mathtt{s})}$ represents all the cuts that can be simultaneously applied when the state of the test execution is at history variable q. Thus, all restrictions on system actions at history q are captured by the cuts on $\mathcal{G}_{\text{sys}}^{(q,\mathtt{s})}$. Since this is true for every q and every source state $\mathtt{s}$ at which the test execution enters into q, there always exists a path to the goal by equation (c8). The proof for the static and mixed settings follows similarly. □

Remark 4.6. Note that Theorems 4.1 and 4.2 are not limited to optimal solutions of the MILP, but apply to feasible solutions as well. That is, any time termination of the MILP provided that a feasible solution has been found is sufficient to find a test strategy with guarantees that the system assumptions are satisfied, and that there are no bypass paths. However, only an optimal solution can return a test strategy that is not overly-restrictive. However, the following lemma only applies to optimal solutions.

Lemma 4.3. For each MILP, the optimal cuts C correspond to maximizing the cardinality of Θ_u.

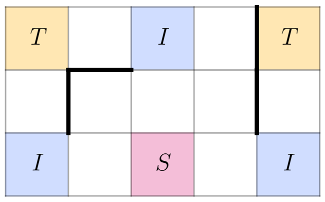
Proof. By construction, a realization of the flow $\mathbf{f}$ on $\mathcal{G}$ corresponds to a set of unique state-history traces Θ_u. The MILP objective maximizes the flow, and therefore the cardinality of Θ_u is maximized.

Additionally, the feasibility constraints do not induce any conservativeness in terms of finding a test strategy that is not overly-restrictive. Let $Path(\mathtt{S}, \mathtt{s})$ be a path from the source of the product graph G to node $\mathtt{s}$, where $\mathtt{s} \in \mathtt{S}_{G_{\text{sys}}}(q)$ for $q \in mathttB_{\pi}.Q$ is some source at which the execution updates to history variable q. Since the number of edge-cuts are minimized in the optimization objective, no $Path(\mathtt{S}, \mathtt{s})$ will be restricted unless if necessary to cut a bypass path. Even in this instance, checking that there exists a $Path(\mathtt{s}, \mathtt{T})$ in the feasibility constraints will not be an issue. If all $Path(\mathtt{s}, \mathtt{T})$ are bypass paths, then the optimization will choose to cut all $Path(\mathtt{S}, \mathtt{s})$. Thus, despite the feasibility constraints, the optimal solution of the MILP still corresponds to a not overly-restrictive strategy. $\square$

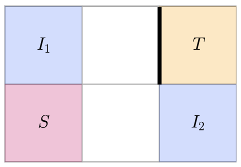
Remark 4.7. The definition of a not overly-restrictive test strategy, both in this and the previous chapter, did not account for the number of restrictions placed. In this chapter, the routing optimization, in addition to providing optimal edge-cuts corresponding to not overly-restrictive test strategies, also returns the minimum the number of such restrictions required to realize the strategy. Overly restricting the system, especially when not necessary, could potentially increase testing effort.

4.8 Test Strategy Synthesis

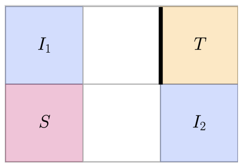
This section outlines how edge-cuts found solving the optimizations can help construct a test strategy. This section is split into two parts: i) construction of a test strategy involving static and reactive obstacles matched to the optimization solution, and ii) synthesis of a test agent strategy for a given dynamic agent such that



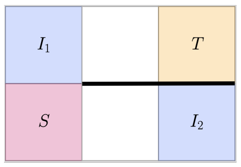
(a) Static Obstacles in black.



(b) q_0



(c) q_6



(d) q_7

Figure 4.7: Static and reactive obstacle placement for running examples. Figure 4.7a shows static obstacles synthesized for Example 4.1. Figures 4.7b, 4.9c, and 4.7d show a test environment implementation of a reactive test strategy for Example 4.2.

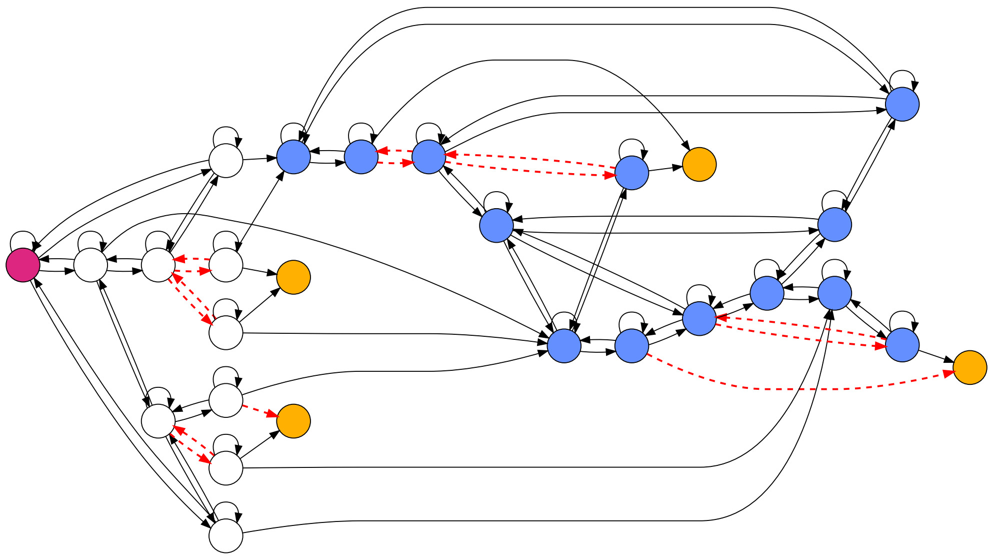
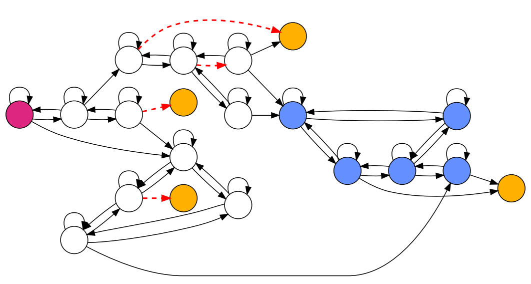


Figure 4.8: Virtual product graph with static cuts in dashed red for the medium example 4.1. Static obstacles in Fig. 4.7a corresponding to edge cuts found on this product graph for Example 4.1. States marked S, I, and T illustrated in Fig. 4.7a correspond to states S (magenta ●), I (blue ●), and T (yellow ●) on $\mathcal{G}$ as shown here. There are three edge-disjoint paths on this graph from the source to the target nodes.


(a) Virtual product graph G.

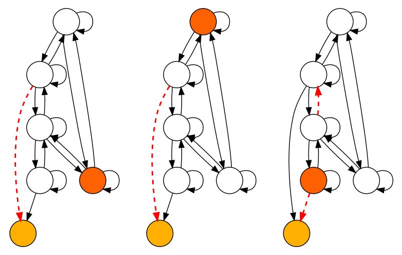

(b) $G_{\text{sys}}^{(q_0,s_3)}$ (c) $G_{\text{sys}}^{(q_6,s_1)}$ (d) $G_{\text{sys}}^{(q_7,s_{11})}$

Figure 4.9: Virtual product graph and system product graphs for Example 4.2. Fig. 4.9a shows the virtual product graph G, with the source S (magenta ●), the intermediate nodes I (blue ●), and the target nodes (yellow ●). Edge cut values for each edge in G are grouped by their history variable q and projected to the corresponding copy of G_{sys}. Figs. 4.9b—4.9d show the copies of G_{sys} with their source (orange ●) and target node (yellow ●). The graphs in Figs. 4.9b—4.9d correspond to the history variables q_0, q_6, and q_7 from $\mathcal{B}_\pi$ shown in Fig. 4.2c. The constraints (c6)—(c8) ensure that the edge cuts are such that a path from each source to the target node exists for each history variable q.

the synthesized strategy matches the restrictions on system actions from the optimization solution.

Test Environments with Static and/or Reactive Obstacles

In this section, we will detail the construction of a test strategy from a solution of the MILP solved in the static, reactive, or mixed settings. First, consider the more general reactive setting. A solution (not necessarily optimal) of the MILP in each setting returns a set of edge-cuts C that can be parsed into a reactive map $\mathcal{C} : \mathcal{B}_\pi.Q \to T_{\text{sys}}.E$ of system restrictions:

$$\mathcal{C}(q) := \{(s, s') \in T_{\text{sys}}.E \mid ((s, q), (s', q')) \in C\}. \tag{4.32}$$

The argument of the reactive map is the state history variable q, and intuitively, $\mathcal{C}(q)$ represents the set of all system restrictions that can be active when the test execution is at the state history variable q. Formally, when the test execution ϑ arrives at a state (s, q) at some time $k \geq 0$ (and correspondingly the system trace σ is at s at $k \geq 0$), and the system restriction $(s, s') \in \mathcal{C}(q)$, the test environment must restrict the system action corresponding to (s, s') at this event. Therefore, the test strategy is constructed as

$$\begin{aligned}\pi_{\text{test}}(\sigma_{0:k}) \coloneqq \{a \in T_{\text{sys}}.A \,|\, s' \in T_{\text{sys}}.\delta(s,a),\ q = \textsc{Hist}(\sigma_{0:k}),\\ (s,s') \in \mathcal{C}(q)\}.\end{aligned} \tag{4.33}$$

Practically, π_{test} can be realized by the test environment by placing obstacles during the test execution in reaction to system behavior (given by the trace σ). The *set of active obstacles* at time step k denoted by $\texttt{Obs}(\sigma_{0:k})$ is the set of state-action restrictions that are placed at time k. Note that the set of active obstacles can contain more restrictions than the test strategy. For example, an action corresponding to transition (s', s'') of the system can be restricted even though the system is not at s' at time k. Intuitively, this might correspond to a static obstacle that is far away from the system and not blocking it immediately. The set of active obstacles represent different implementations of the same reactive test strategy. A few implementations of the reactive test strategy as a set of active obstacles are:

1. **Exact Reactive Placement**: In this setting, the set of active obstacles correspond exactly the set of actions restricted by the test strategy: $\texttt{Obs}(\sigma_{0:k}) \coloneqq \{(s,a) | s = \sigma_k,\ a \in \pi_{\text{test}}(\sigma_{0:k})\}$. The obstacle is only active when the system is in a state from which an action is restricted.

2. **Instantaneous Placement**: In this setting, the test environment instantaneously places all obstacles or the restrictions in $\mathcal{C}(q)$ are realized "at once" when the test execution enters a state-history trace with history variable q. Concretely, let (s_k, q_k) be the state-history of the test execution at some time $k \geq 0$, then the set of active obstacles are

$$\texttt{Obs}(\sigma_{0:k}) \coloneqq \{(s,a) \,|\, (s,s') \in \mathcal{C}(q_k) \text{ and } s' \in T_{\text{sys}}.\delta(s,a)\}.$$

3. **Accumulative Placement**: In this setting, active obstacles are accumulated as the system trace evolves as long as the history variable does not change. For some $k > 0$, let (s_{k-1}, q_{k-1}) and (s_k, q_k) be the state-history pairs at time steps $k-1$ and k, respectively. If $q_{k-1} \neq q_k$, then the set of active obstacles becomes $\texttt{Obs}(\sigma_{0:k}) \coloneqq \{(s_k, a) \,|\, a \in \pi_{\text{test}}(\sigma_{0:k})\}$. As the system trace evolves to state-history pairs (s_l, q_l), where $l > k$ and $q_l = q_k$, the set of active obstacles are accumulated: $\texttt{Obs}(\sigma_{0:l}) = \bigcup_{j=k}^{l} \texttt{Obs}(\sigma_{0:j})$. When the history variable advances, i.e., $q_l \neq q_k$, then the set of active obstacles are *reset*: $\texttt{Obs}(\sigma_{0:l}) \coloneqq \{(s_l, a) \,|\, a \in \pi_{\text{test}}(\sigma_{0:l})\}$.

All three methods of determining the set of active obstacles will *simulate* the reactive test strategy; they are varied implementations of the test environment. The placement of obstacles need not coincide with when the system observes these obstacles, which depends on the system implementation. However, we assume that the system can observe all restrictions placed by the test environment on its current state before it commits to an action.

Remark 4.8 (On Relaxing the Assumption 4.1)**.** Roughly speaking, the feasibility constraints (c8) ensure that placing obstacles does not block the system from its goal. This condition is checked by ensuring that there exists feasible path for the system from every possible source $\mathtt{s} \in \mathtt{S}_{G_{\text{sys}}}(q)$ for every history variable q when all restrictions in $C(q)$ have been placed. With Assumption 4.1, the above feasibility constraint is a sufficient check since the system can backtrack to the source and find an alternative path if it encounters a restriction placed by the test environment. However, this assumption is not necessary and can be relaxed in one of two ways. First, if a restriction were to cause a livelock (i.e., system has no choice but to remain in the same state or be stuck in a cycle), then the restriction must be revealed to the system before the livelock becomes inevitable. Second, for every cut $((s,q),(s',q'))$ in the set of edge-cuts C, we can check that there exists a $Path((s,q),\mathtt{T})$ on $\mathcal{G}$ after edges C have been removed. If this is not the case, then the solution corresponding to C can be added as a counterexample constraint to the MILP, which will then be resolved. This process is repeated until the set of cuts C are accepted. Implementation of a counterexample constraint is detailed in section 4.8.

Proposition 4.2. In both the instantaneous and accumulative settings, as long as no new restrictions that are not in $\mathcal{C}(q)$ are introduced, the flow value F remains the same.

Example 4.2 (Small Reactive (continued))**.** Fig. 4.7 illustrates a reactive example on gridworld introduced previously. The reactive test strategy is constructed from the optimal solution of **MILP-REACTIVE**. The optimization returns cuts on $\mathcal{G}$, which is realized as follows: when the system is at the initial state and the test execution history variable is at q0, the test environment places a restriction as shown in Fig. 4.7b. If the system chooses to visit $\mathtt{I}_1$ first, the restriction does not change even as the test execution history variable updates to q6 (see Fig. 4.7c). Alternatively, if the system visits $\mathtt{I}_2$ first, the test execution history variable updates to q7, and to prevent direct access to goal cell T, the test environment places the restrictions shown in Fig. 4.7d. These restrictions can be implemented either in the instantaneous or the accumulative setting.

Static and Mixed Test Environments: In the special case of test environments consisting of only static obstacles, the solution of **MILP-STATIC** returns a set of edge-cuts which result in a reactive map $\mathcal{C}$ in which restrictions do not change based on the history variable: $\mathcal{C}(q) = \mathcal{C}(q'), \forall q, q' \in \mathcal{B}_\pi.Q$. All system transitions constitute the static area: $T_{\text{sys}}.E = T_{\text{sys}}.E_{\text{static}}$, and the test environment instantaneously places all static obstacles at the start of the test execution:

$$\texttt{Obs}_{\text{static}} \coloneqq \{(u.s, v.s) \in T_{\text{sys}}.E_{\text{static}} \mid (u, v) \in C\}, \; \forall k \geq 0. \tag{4.34}$$

In the mixed setting, the test strategy is constructed according to Eq. (4.33), and the set of active obstacles are constructed similar to the reactive setting. Restrictions that are in the static area $T_{\text{sys}}.E_{\text{static}}$ can be implemented by placing static obstacles.

Example 4.1 (continued)**.** For the medium-sized grid world example illustrated in Fig. 4.3a, the static test environment is illustrated in Fig. 4.7a. Figure 4.8 illustrates edge-cuts that correspond to static obstacles. There are 14 edge-cuts on $\mathcal{G}$ that correspond to 4 static obstacles on T_{sys}. On $\mathcal{G}$, observe that there is no bypass flow, and the maximum flow after the cuts is $F^* = 3$, corresponding to the three different ways in which the system can be routed through the intermediates.

Algorithm 5 summarizes the following aspects of the framework discussed so far: i) graph construction, ii) routing optimization using flow networks, and iii) construction of a reactive test strategy from the optimization solution. Finally, the following theorem (taken from [107]) shows that the reactive test strategy is feasible and not overly-restrictive when constructed from the optimal solution of the MILP.

Theorem 4.3. If the problem data are not inconsistent (see Section 4.7), the reactive test strategy π_{test} found by Algorithm 5 solves Problem 4.1.

Proof. The test environment informs the choice of the MILP (static, reactive, or mixed). Therefore, the resulting π_{test} will be realizable by the test environment. By construction of G_{sys}, any correct system strategy corresponds to a $\texttt{Path}(\texttt{S}_{\text{sys}}, \texttt{T}_{\text{sys}})$. By Theorem 4.2, at any point during the test execution, if the system has not violated its guarantees, there exists a path on G_{sys} to $\mathtt{T}_{\text{sys}}$. Therefore, there exists a correct system strategy π_{sys}, and resulting trace $\sigma(\pi_{\text{sys}} \times \pi_{\text{test}})$, which corresponds to the path $\vartheta_{\text{sys},n} = (s,q)_0 \dots (s,q)_n$ on G_{sys}, where $(s,q)_0 \in \mathtt{S}_{\text{sys}}$ to $(s,q)_n \in \mathtt{T}_{\text{sys}}$. By Lemma 4.1 any $\mathtt{Path}(\mathtt{S}_{\text{sys}}, \mathtt{T}_{\text{sys}})$ on G_{sys} has a corresponding $\mathtt{Path}(\mathtt{S}, \mathtt{T})$ on G and by Theorem 4.1, the cuts ensure that all such paths on G are routed through the intermediate $\mathtt{I}$. Therefore, for a correct system strategy π_{sys}, the trace $\sigma(\pi_{\text{sys}} \times \pi_{\text{test}}) \models \varphi_{\text{sys}} \wedge \varphi_{\text{test}}$. Thus, π_{test} is feasible and by Proposition 4.2 and Lemma 4.3, π_{test} is not overly-restrictive. Thus, Problem 4.1 is solved. $\square$

The resulting test strategy ensures that as long as the system does not take an incorrect action, there will always exist a path to its goal. However, the system is not aided in reaching the goal either — the test strategy will not block actions that lead to unsafe states. Therefore, a correctly implemented system should be able pass the test, and if the test fails, then it is the fault of the system design.

Algorithm 6: Reactive Test Synthesis

```
1:  procedure TEST SYNTHESIS($T_{\text{sys}}, T_{\text{TA}}, H, \varphi_{\text{sys}}, \varphi_{\text{test}}$)
      Input: system $T_{\text{sys}}$, test agent $T_{\text{TA}}$, test harness $H$, system objective $\varphi_{\text{sys}}$, test
    objective $\varphi_{\text{test}}$
      Output: test agent strategy $\pi_{\text{TA}}$
2:    $T_{\text{sys}}.E_{\text{static}} \leftarrow$ Define from $T_{\text{sys}}, T_{\text{TA}}$          ▷ Static area (Eq. (4.35)
3:    $\mathcal{G}, \mathfrak{G}_{\text{sys}}, \mathtt{I}, G \leftarrow$ Setup arguments          ▷ Lines 2-13 in Alg. 5
4:    $\mathtt{C}_{\text{ex}} \leftarrow \emptyset$          ▷ Initialize empty set of excluded solutions
5:    $F_{\max} \leftarrow$ MILP-AGENT($\mathcal{G}, \mathfrak{G}, \mathtt{I}, T_{\textbf{sys}}, H, \mathtt{C}_{\textbf{ex}} = \{\}$)
6:    while True do
7:        $F^*, \mathbf{d}^* \leftarrow$ MILP-AGENT($\mathcal{G}, \mathfrak{G}, \mathtt{I}, T_{\text{sys}}, H, \mathtt{C}_{\text{ex}}$)
8:        if STATUS(MILP) $=$ infeasible then
9:            return infeasible
10:       $C \leftarrow \{(u,v) \in G.E \mid \mathbf{d}^{*(u,v)} = 1\}$          ▷ Cuts on $G$
11:       $\mathtt{Obs} \leftarrow$ Define from $C$          ▷ Static Obstacles (Eq. (4.34))
12:       $\mathcal{R} \leftarrow$Define from $C$          ▷ Reactive map (Eq. (4.36))
13:       $\mathbf{A} \leftarrow$ Assumptions (a1)–(a5) from $T_{\text{sys}}$, $T_{\text{TA}}$, $G$, $\varphi_{\text{sys}}$
14:       $\mathbf{G} \leftarrow$ Guarantees (g1)–(g7) from $T_{\text{sys}}$, $T_{\text{TA}}$, $\mathcal{R}$
15:       $\varphi \leftarrow (\mathbf{A} \rightarrow \mathbf{G})$          ▷ Construct GR(1) formula
16:       if REALIZABLE($\varphi$) then
17:           $\pi_{\text{TA}} \leftarrow$ GR1Solve($\varphi$)
18:           return $\pi_{\text{TA}}, \mathtt{Obs}$
19:       $\mathtt{C}_{\text{ex}} \leftarrow \mathtt{C}_{\text{ex}} \cup C$
```

Strategy Synthesis for a Dynamic Test Agent

In some test scenarios, it might be beneficial to make use of an available dynamic test agent. Thus, the challenge is to find a tester strategy that corresponds to $\mathcal{C}$ while ensuring that the system's operational environment assumptions are satisfied. To accomplish this, we adapt the **MILP-MIXED** using information about the dynamic test agent. Then, we find the test agent strategy using reactive synthesis and counter-example guided search. From the optimal cuts of **MILP-MIXED** and the resulting reactive map $\mathcal{C}$, we can find states that the test agent must occupy in reaction to the system state. Then, we synthesize a strategy for the dynamic test agent using the Temporal Logic and Planning Toolbox (TuLiP) [108, 109]. If we cannot synthesize a strategy, we use a counterexample-guided approach to exclude the current solution and resolve the MILP to return a different set of optimal cuts until a strategy can be synthesized. Suppose we are given a test agent whose dynamics are given by the transition system T_{TA}, where $T_{\text{TA}}.S$ contains at least one state that is not in $T.S$, denoted as `park`. During the test execution, the test agent can navigate to these `park` states, if necessary. These states are required to synthesize a test agent strategy. From the test agent's transition system T_{TA}, we determine which states in T the test agent can occupy. Static obstacles are used to restrict transitions to states that cannot be occupied by the test agent, and thus the static area is defined as

$$T_{\text{sys}}.E_{\text{static}} \coloneqq \{(u, v) \in T_{\text{sys}}.E \mid v \notin T_{\text{TA}}.S\}. \tag{4.35}$$

Assumption 4.2. In the mixed setting with static obstacles, and a reactive dynamic agent, static obstacles can only restrict transitions in $T_{\text{sys}}.E_{\text{static}}$ as defined in Eq. (4.35).

Adapting the MILP for test agent: Since an agent can only occupy a single state at a time, solutions in which multiple edge cuts can be realized by occupying the same state are incentivized. For this, we introduce the variable $\mathbf{d}_{\text{state}} \in \mathbb{R}_{+}^{|V|}$, which represents whether an incoming edge into a state is cut. This is captured by the constraint

$$\forall (u, v) \in E, \quad d^{(u,v)} \leq d^{v}_{\text{state}}, \tag{c11}$$

where $d^{v}_{\text{state}} \geq 1$ corresponds to at least one incoming edge being cut. The adapted objective is then defined as

$$F - k \sum_{e \in E} d^{e} - m \sum_{v \in V} d^{v}_{\text{state}},$$

where $k \leq \frac{1}{1+|E|}$ and $m \leq \frac{1}{|V|(1+|E|)}$. The objective is chosen such that the total number of edge cuts, and the number of nodes that are blocked are minimized. The regularizers are chosen to reflect this order of priority: once the number of edges are minimized, the number of nodes that are cut are minimized. The optimal cuts from the resulting MILP are used to synthesize a reactive test agent strategy as follows. From the optimal cuts C^*, we find the set of static obstacles $\texttt{Obs} \subseteq T.E_{\text{static}}$ according to Eq. (4.34) and the reactive map $\mathcal{R} : \mathcal{B}_\pi.Q \to T.E$ as follows:

$$\mathcal{R}(q) := \{(s, s') \in T.E \mid (s, s') \notin T.E_{\text{static}} \text{ and } ((s, q), (s', q')) \in C^*\}. \tag{4.36}$$

The reactive map $\mathcal{R}$ is used to synthesize a strategy for the test agent. If no strategy can be found, a counter-example guided approach is used to resolve the MILP.

Reactive Synthesis: From the solution of the MILP, we now construct the specification to synthesize the test agent strategy using TuLiP. In particular, we construct a GR(1) formula with assumptions being our model of the system and the guarantees capturing requirements on the test agent. Note that we are synthesizing a strategy for the test agent, where the environment is the system under test. The variables needed to define the GR(1) formula consist of variables capturing the system's state $\mathtt{x}_{\text{sys}} \in T.S$ and $\mathsf{q}_{\text{hist}} \in \mathcal{B}_\pi.Q$, which track how system transitions affect the history variable q. The test agent state is represented in the variable $\mathtt{x}_{\text{TA}} \in T_{\text{TA}}.S$.

First, we set up the subformulae constituting the assumptions on the system model. The initial conditions of the system are defined as

$$(\mathtt{x}_{\text{sys}} = s_0 \wedge \mathsf{q}_{\text{hist}} = q_0), \tag{a1}$$

where $s_0 \in T.S_0$ and $\mathcal{B}_\pi.Q_0$. We define the dynamics of the system and the history variable for each state $(s, q) \in G.S$ as follows:

$$\square\Big((\mathtt{x}_{\text{sys}} = s \wedge \mathsf{q}_{\text{hist}} = q) \to \bigvee_{\substack{(s',q')\in \\ \texttt{succ}(s,q)}} \bigcirc(\mathtt{x}_{\text{sys}} = s' \wedge \mathsf{q}_{\text{hist}} = q')\Big), \tag{a2}$$

where $\texttt{succ}(s, q)$ denotes the successors of state $(s, q) \in G.S$. For simplicity, we choose a turn-based setting, in which each player will only take their action if it is their turn. To track this, we introduce the variable $\texttt{turn} \in \mathbb{B}$ as a test agent variable. For the system, this is encoded as remaining in place when $\texttt{turn} = 1$:

$$\bigwedge_{s\in T.S} \square\Big((\mathtt{x}_{\text{sys}} = s \wedge \texttt{turn} = 1) \to \bigcirc(\mathtt{x}_{\text{sys}} = s)\Big). \tag{a3}$$

If a turn-based setup is not used, we need to synthesize a Moore strategy for the test agent since it should account for all possible system actions. The system objective φ_{sys} can be encoded as the formula

$$\square \lozenge (\mathtt{x}_{\text{sys}} = x_{\text{goal}}) \wedge \varphi_{\text{aux}}, \tag{a4}$$

where x_{goal} is the terminal state of the system and a reachability objective specified in φ_{sys}. The other objectives specified in φ_{sys} are transformed to their respective GR(1) forms in φ_{aux}. This transformation of LTL formulas into GR(1) form is detailed in [110]. In addition, the system is expected to safely operate in the test agent's presence. The set of states where collision is possible is denoted by $S_{\cap} := T.S \cap T_{\text{TA}}.S$. Thus, the safety formula encoding that the system will not collide into the tester is given as:

$$\bigwedge_{s \in S_{\cap}} \square \Big(\mathtt{x}_{\text{TA}} = s \rightarrow \bigcirc \neg (\mathtt{x}_{\text{sys}} = s) \Big). \tag{a5}$$

Equations (a1)– (a5) represent the test agent's assumptions on the system model. Next, we describe the subformulas for the guarantees of the GR(1) specification. The initial conditions for the test agent are

$$\bigvee_{s \in T_{\text{TA}}.S_0} \mathtt{x}_{\text{TA}} = s. \tag{g1}$$

The test agent dynamics are represented by

$$\square \Big((\mathtt{x}_{\text{TA}} = s) \rightarrow \bigvee_{(s,s') \in T_{\text{TA}}.E} \bigcirc (\mathtt{x}_{\text{TA}} = s') \Big). \tag{g2}$$

The test agent can also move only in its turn and will remain stationary when $\mathtt{turn} = 0$:

$$\bigwedge_{s \in T_{\text{TA}}.S} \square \Big((\mathtt{x}_{\text{TA}} = s \wedge \mathtt{turn} = 0) \rightarrow \bigcirc (\mathtt{x}_{\text{TA}} = s) \Big). \tag{g3}$$

The $\mathtt{turn}$ variable alternates at each step:

$$(\mathtt{turn} = 1) \rightarrow \bigcirc (\mathtt{turn} = 0) \ \wedge (\mathtt{turn} = 0) \rightarrow \bigcirc (\mathtt{turn} = 1). \tag{g4}$$

To satisfy the system assumptions (Def. 4.6), the test agent should not adversarially collide into the system. This is captured via the following safety formula,

$$\bigwedge_{s \in S_{\cap}} \square \Big(\mathtt{x}_{\text{sys}} = s \rightarrow \bigcirc \neg (\mathtt{x}_{\text{TA}} = s) \Big). \tag{g5}$$

Now, we enforce the optimal cuts found from the MILP. To enforce cuts reactively during the test execution, the states occupied by the system are defined as follows,

$$\bigwedge_{q \in \mathcal{B}_\pi.Q} \bigwedge_{(s,s') \in \mathcal{R}(q)} \square\Big((\mathtt{x}_{\text{sys}} = s \wedge \mathtt{q}_{\text{hist}} = q \wedge \mathtt{turn} = 0) \rightarrow (\mathtt{x}_{\text{TA}} = s') \Big). \tag{g6}$$

Essentially, for some history variable q, if $(s, s') \in \mathcal{R}(q)$ is an edge cut, then the test agent must occupy the state s' when the system is in the state s when the test execution is at history variable q. However, the test agent should not introduce any additional restrictions on the system, which is formulated as

$$\bigwedge_{q \in \mathcal{B}_\pi.Q} \bigwedge_{\substack{(s,s') \in T.E \\ (s,s') \notin \mathcal{R}(q)}} \square\Big((\mathtt{x}_{\text{sys}} = s \wedge \mathtt{q}_{\text{hist}} = q \wedge \mathtt{turn} = 0) \rightarrow \neg(\mathtt{x}_{\text{TA}} = s') \Big). \tag{g7}$$

Intuitively, this corresponds to the requirement that the tester agent shall not restrict system transitions that are not part of the reactive map $\mathcal{R}$. A test agent strategy that satisfies the above specifications is guaranteed to not restrict any system action unnecessarily. However, the test agent can occupy a state that is not adjacent to the system and block all paths to the goal from the system's perspective. This could lead the system to not making any progress towards the goal at all, resulting in a livelock. To avoid this, we characterize the livelock condition as a safety constraint that the test agent must satisfy (e.g., if it occupies a livelock state, it must not occupy it in the next step). The specific safety formula that captures the livelock depends on the example. We find the states where the tester would block the system from reaching its goal $T.S_{\text{block}} \subseteq T_{\text{TA}}.S$. The following condition ensures that it will only transiently occupy blocking states:

$$\bigwedge_{s \in T.S_{\text{block}}} \square\Big(\mathtt{x}_{\text{TA}} = s \rightarrow \bigcirc \neg(\mathtt{x}_{\text{TA}} = s) \Big). \tag{g8}$$

Therefore, we synthesize a test agent strategy π_{TA} for the GR(1) formula with assumptions (a1)–(a5) and guarantees (g1)–(g8).

Counterexample-guided Search: The MILP can have multiple optimal solutions, some of which may not be realizable for the test agent. If the GR(1) formula is unrealizable, we exclude the solution and re-solve the MILP until we find a realizable GR(1) formula. In particular, every new set of optimal cuts C that is unrealizable is added to the set $\mathtt{C}_{\text{ex}}$. Then, the MILP is resolved with an additional set of affine constraints as follows,

$$\sum_{e \in E} d^e - \sum_{e \in C} d^e \geq 1, \; \forall C \in \mathtt{C}_{\text{ex}}. \tag{c12}$$

This corresponds to removing the solution C from the constraint set. The adapted MILP is then defined as follows:

MILP-AGENT:

$$\begin{aligned}\max_{\substack{\mathbf{f},\mathbf{d},\mathbf{d}_{\text{state}},\boldsymbol{\mu},\\ \mathbf{f}_{\text{sys}}^{(q,\mathsf{s})}\,\forall q\in\mathcal{B}_\pi.Q,\\ \forall \mathsf{s}\in \mathsf{S}_{\mathcal{G}_{\text{sys}}}(q).}} \quad & F - \frac{1}{1+|E|}\sum_{e\in E} d^e - \frac{1}{(1+|E|)|V|}\sum_{v\in V} d^v_{\text{state}} \\ \text{s.t.} \quad & \text{(c1)-(c9), (c11), (c12).}\end{aligned} \tag{4.37}$$

This process is repeated until a strategy is synthesized or the **MILP-AGENT** becomes infeasible. Algorithm 6 summarizes the approach for synthesizing the test agent strategy. The terms $F_{\max}$ and F^* (lines 5 and 7 in Algorithm 6) denote the maximum possible flow before and after accounting for counterexamples, respectively. The following lemma and theorem are adapted from [107].

Lemma 4.4. Let π_{TA} be the test agent strategy and let $\texttt{Obs}$ satisfying Assumption 4.2 be the set of static obstacles synthesized from the optimal solution C of **MILP-AGENT** according to the GR(1) formula with assumptions (a1)–(a5) and guarantees (g1)–(g8). Let π_{test} be the reactive test strategy corresponding to the optimal cuts C^*. Then π_{TA} and $\texttt{Obs}$ realize π_{test}.

Proof. By construction in Eqs. (4.32), (4.34), (4.36), we have that $\mathcal{C}(q) = \mathcal{R}(q) \cup \texttt{Obs}$ for all history variables $q \in \mathcal{B}_\pi.Q$. Due to guarantee (g6), the synthesized test agent strategy restricts the transitions in $\mathcal{R}(q)$. The test agent is also prohibited from restricting any other transitions by the guarantee (g7). Therefore, at each step of the test execution, the system actions restricted as a result of π_{TA} and static obstacles $\texttt{Obs}$ directly correspond to the system actions restricted by the test strategy π_{test}. $\square$

Theorem 4.4. Algorithm 6 is sound with respect to Problem 4.2.

Proof. The test agent strategy is synthesized to satisfy guarantees (g1)-(g8). The guarantees (g1)-(g4) specify the dynamics of the test agent, which satisfies **A1**. The safety guarantee (g5) satisfies **A2**. Guarantees (g6) and (g7) realize the optimal cuts from **MILP-AGENT**. Due to constraint (c8) the optimal cuts ensure that there always exists a path on G_{sys}. Together with guarantee (g8), this results in π_{TA} satisfying assumptions **A3** and **A4**. By Lemma 4.4, π_{TA} is a realization of a feasible π_{test} that is not overly-restrictive.

In Algorithm 6, each iteration of **MILP-AGENT** is solved to optimality while excluding the counterexamples. If **MILP-AGENT** returns with $F^* = F_{\max}$, then π_{TA} corresponds to a π_{test} that is not overly-restrictive. By iteratively removing counterexamples, the agent strategy is synthesized for a reactive test strategy with the highest possible $F^* \leq F_{\max}$. This is valid under Assumption 4.2, which allows static obstacles only on transitions that cannot be restricted by the test agent. In **MILP-AGENT**, this condition is enforced by applying constraint (c9) on the static area $T_{\text{sys}}.E_{\text{static}}$. $\square$

If a matching test agent strategy is found for the maximum possible F, the test agent strategy and obstacles, π_{TA} and `Obs`, correspond to a not overly-restrictive reactive test strategy π_{test} possible for that test environment. In future work, we will exploring relaxing Assumption 4.2.

4.9 Complexity Analysis

This framework comprises of three parts: automata-theoretic graph construction, flow-based MILP to solve the routing optimization, and finally reactive synthesis to match the solution of the optimization to a test agent strategy. The automata-theoretic framework includes construction of Büchi automata from specifications, which can be doubly exponential in the length of the formula in the worst-case [60]. Then, construction of the product graphs relies on building the Cartesian product of the transition system and the automaton. The Cartesian product implementation in this work has a worst-case time complexity of $O(|T.S|^2|\mathcal{B}_\pi.Q|^2)$. In this section, I will discuss the computational complexity of the routing optimization, and prove that the routing optimization is an NP-hard problem. Finally, the solution of the routing optimization is mapped to a strategy of the test agent via GR(1) synthesis, which has time complexity $O(|N|^3)$, where N is the number of states required to define the GR(1) formula.

To establish the computational complexity of the routing optimization, we will first look at the special case of static obstacles, and then extend the proof to the setting with reactive obstacles. Consider the problem data of the routing optimization once again: a graph $G = (V, E)$ with specially denoted nodes $\mathtt{S}$, $\mathtt{I}$, and $sink$, and the corresponding flow network $\mathcal{G}$. A bypass path on G is some $Path(\mathtt{S}, \mathtt{T})$ which does not contain an intermediate node $v \in \mathtt{I}$. For all edges $e \in E \setminus E(\mathtt{I})$, the Boolean variable d^e carries information on whether the edge e is cut (i.e., $d^e = 1$), and the set $C \subset E$ represents the set of all edges that are cut. The flow F on $\mathcal{G}$ is the maximum flow value from source S to T, computed after accounting for the edge cuts.

The static and reactive obstacle settings are based on the grouping of edges on G, which become important for checking system feasibility. Static obstacles are grouped by the corresponding transition in the system transition T since they are present for the entire test duration. In particular, the static grouping $\texttt{Gr}_{\text{static}} : T.E \to G.E$ groups all edges in G that correspond to the same system transition in T:

$$\texttt{Gr}_{\text{static}}((s, s')) := \{(u, v) \in G.E \mid u.s = s, v.s = s'\}. \tag{4.38}$$

For some edge $(s, s') \in T.E$, all the corresponding edges in G, that is, all edges $e \in \texttt{Gr}_{\text{static}}((s, s'))$ must have the same d^e value. Similarly, in the reactive setting, edges are grouped by the history variable q, as given in Eq. (4.28). System feasibility can then be checked by applying these groupings onto copies of G or G_{sys}, as detailed in Remark 4.5.

For purposes of clearly stating the static the optimization and decision versions of the routing problem, we introduce the label of a valid set of edge cuts. In the static setting, a *valid* set of edge cuts C when applied to G is such that: i) there are no bypass paths, ii) there exists at least one path from S to T, and iii) edges of G respect the static grouping $\texttt{Gr}_{\text{static}}$. Note that there can exist graphs G for which there does not exist a valid set of edge cuts in the static setting. These are graphs for which we cannot synthesize a test comprising only static obstacles to realize the test objective. One such example is Beaver Rescue. Now, the optimization version of the routing problem for the special case of static obstacles is stated as follows,

Problem 4.3 (Routing Problem, Static Setting (Optimization))**.** Given a graph G, find a valid set of edge cuts C in the static setting such that the resulting maximum flow F is maximized over all possible sets of edge cuts, and such that $|C|$ is minimized for the flow F.

In other words, the optimization follows a two-step procedure: first, identify a valid set of edge cuts C to maximizes the flow F, and second, tie-break between the optimal candidates C to choose one with the smallest cardinality $|C|$. The decision version of Problem 4.3 can be stated as follows.

Problem 4.4 (Routing Problem, Static Setting (Decision))**.** Given a graph G and an integer $M \geq 0$, does there exist a valid set of edge cuts C in the static setting such that $|C| \leq M$?

Lemma 4.5. A solution to Problem 4.3 can be used to construct a solution for Problem 4.4 in polynomial time.

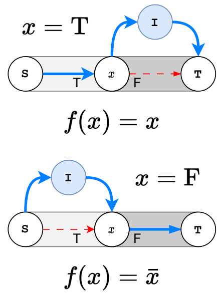
Proof. The solution of Problem 4.3 returns a set of valid edge cuts C. Thus for any given integer $M \geq 0$, we can check in polynomial time if $|C| \leq M$. $\square$

Basics of Complexity Theory: Finding the complexity class of Problem 4.4 will help in determining the complexity of Problem 4.3 because by Lemma 4.5, we can infer that Problem 4.3 is at least as hard as Problem 4.4. The class of NP problems consists of those that are verifiable in a time polynomial to the size of the input to the problem [80]. A problem is said to be in the class of NP-complete problems if: i) it is in the class NP, and ii) it is as hard as any problem in NP. Polynomial-time algorithms for solving NP-complete problems would exist only if P=NP. The class of NP-hard problems are those that are as hard as a problem in NP. In this section, I will show that Problem 4.4 is NP-complete, and by extension that Problem 4.6 is an NP-hard problem in the size of the input: product graph G. This would also support the choice of a mixed-integer linear programming framework to solve the routing optimization, since MILPs belong to the class of NP-hard problems as well.

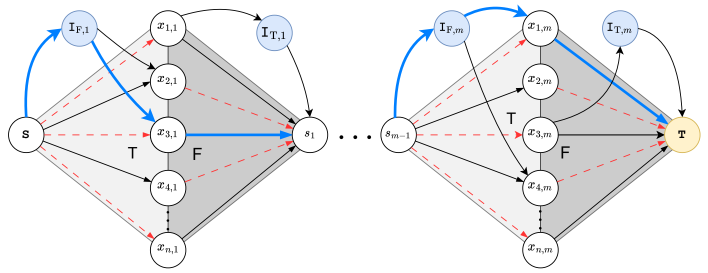
To show that Problem 4.4 is NP-complete, we have to establish its membership in NP, and then give a polynomial-time reduction of a problem in NP to Problem 4.4. We will choose the 3-SAT problem and give a polynomial-time reduction algorithm that maps any instance of the 3-SAT problem to an instance of Problem 4.4. This reduction algorithm is such that a solution to the constructed instance of Problem 4.4 can be transformed in polynomial-time to a solution of the 3-SAT instance.

Lemma 4.6. Problem 4.4 is in the class NP.

Proof. Given a solution C, we need to show that verifying that it is a valid set of edge-cuts for the static setting can be done in polynomial-time. In reference to the definition of a valid set of edge-cuts, it can be checked in polynomial-time that there are no bypass paths when the edges in C are cut from G. This would involve a simple check (e.g., via any max-flow algorithm) to verify zero maximum flow from $\mathtt{S}$ to $\mathtt{T}$ on the bypass network $\mathcal{G}_{byp} = (V \setminus \mathtt{I}, E \setminus E(\mathtt{I}), \mathtt{S}, \mathtt{T})$. Similarly, condition (ii) can also be checked in polynomial-time by running a max-flow algorithm on $\mathcal{G}$ and verifying that the max-flow is at least 1. Finally, condition (iii) can be checked in polynomial-time by iterating over all edges $e \in T.E$, and checking that exactly one of the following in true: a) $\mathtt{Gr}_{\text{static}}(e) \subseteq C$ or b) $\mathtt{Gr}_{\text{static}}(e) \cap C = \emptyset$. $\square$



(a) Graphs matching formulas with a single variable x.


$\mathrm{I}_{\mathrm{F},1}$
$\mathrm{I}_{\mathrm{T},1}$
s_1
s_{m-1}
$\mathrm{I}_{\mathrm{F},m}$
$\mathrm{I}_{\mathrm{T},m}$

(b) Graph resulting from a reduction of the 3SAT formula $f(x_1, \ldots, x_5)$, where the resulting edge cuts correspond to the truth assignment of the variables $x_1, \ldots, x_5$.

Figure 4.10: Graphs constructed from a 3SAT formula, where a truth assignment for the variables can be found using the network flow approach for static obstacles.

Now, we introduce the 3-SAT problem .

Definition 4.17 (3-SAT [111])**.** Let $x_1, \ldots, x_n$ be propositions that can evaluate to true or false. A literal is a proposition x_i or its negation. The propositional logic formula $f(x_1, \ldots, x_n) := \bigwedge_{j=1}^{m} c_j$ is a conjunction of clauses $c_1, \ldots, c_m$, where each clause is a disjunction of three Boolean literals. A solution to the 3-SAT problem is an algorithm that returns *True* if there exists a satisfying assignment to $f(x_1 \ldots, x_n)$ and *False*.

Outline of the Reduction Algorithm: Given any 3-SAT formula, we will construct a product graph, an instance of 4.4 in polynomial-time. The product graph is constructed modularly — each clause in the 3-SAT formula corresponds to a sub-graph in the larger product graph (Construction 1). Then, using Construction **??**, the sub-graphs are connected to form the product graph instance to Problem 4.4. Finally, we will prove that any algorithm used to solve Problem 4.4 can be used to solve the 3-SAT problem, thus showing that Problem 4.4 is at least as hard as a problem in NP. Consequently, Problem 4.4 can be solved in polynomial-time in the size of the product graph only if there exists a polynomial-time algorithm for 3-SAT which is only possible if P=NP.

Construction 1 (Clause to Sub-graph)**.** For each clause c_j in the given 3-SAT formula, construct the following sub-graph. First, introduce nodes $x_{1,j}, \ldots, x_{n,j}$ for each of the Boolean propositions $x_1, \ldots, x_n$ that constitute the 3-SAT formula. If $j = 0$, introduce the nodes s_0 and s_1, otherwise introduce the node s_j. For all $j \in \{1, \ldots, m\}$, the nodes s_{j-1} and s_j represent the beginning and end of each sub-graph. Additionally, introduce two more nodes: $\mathtt{I}_{\mathrm{T},j}$ and $\mathtt{I}_{\mathrm{F},j}$. These nodes will serve as intermediate nodes in the constructed graph.

Second, the edges of the sub-graph are added as follows. The intermediate nodes are connected to the start and end nodes of the sub-graph via the directed edges: $(s_{j-1}, \mathtt{I}_{\mathrm{F},j})$ and $(\mathtt{I}_{\mathrm{T},j}, s_j)$. Next, for each $x_{i,j}$, add the directed edges: $(s_{j-1}, x_{i,j})$ and $(x_{i,j}, s_j)$. If neither x_i nor its negation $\bar{x}_i$ appear in the clause c_j, then these are the only directed edges connected to the node $x_{i,j}$ in the sub-graph. If the literal x_i appears, then we add the directed edge $(x_{i,j}, \mathtt{I}_{\mathrm{T},j})$, and if a negated literal $\bar{x}_i$ appears, we add the directed edge $(\mathtt{I}_{\mathrm{F},j}, x_{i,j})$.

From Construction 1, edge-cuts on the sub-graph are related to the Boolean valuations of the propositions as follows. Either the incoming or outgoing edge to each node $x_{i,j}$ must be cut. As illustrated in Fig. 4.10a, if the edge $(s_{j-1}, x_{i,j})$ remains in the sub-graph (and $(x_{i,j}, s_j)$ is cut), this implies that the proposition x_i is assigned the value $True$. Similarly, if the edge $(x_{i,j}, s_j)$ remains, then the proposition x_i is assigned the value $False$. Construction 1 ensures that a satisfying assignment to the clause c_j implies that there exists a $Path(s_{j-1}, s_j)$ and all such paths are routed through the intermediates $(s_{j-1}, \mathtt{I}_{\mathrm{F},j})$ and $(\mathtt{I}_{\mathrm{T},j}, s_j)$ (see Fig. 4.10b). An assignment that evaluates clause c_j to $False$ would only be possible if there was no $Path(s_{j-1}, s_j)$. The full graph can be constructed by stitching together the individual sub-graphs built using Construction 1.

Construction 2 (Reduction)**.** Given a 3-SAT problem with n Boolean propositions and m clauses, construct the sub-graph for each clause according to Construction 1. Denote the node s_0 as the source $\mathtt{S}$ and s_m as the sink $\mathtt{T}$. The node s_{j-1} is common between the sub-graphs of clauses c_{j-1} and c_j. Let the integer variable M be set to $m \times n$. Note that the constructed graph has $O(mn)$ edges and is constructed in polynomial-time in the number of propositions and clauses of the given 3-SAT formula.

In addition to constructing the graph, two groups of edges for each Boolean proposition x_i are tracked: i) an incoming group of edges $\{(s_{j-1}, x_{i,j}) \mid 1 \leq j \leq m\}$, and ii) an outgoing group of edges $\{(s_{j-1}, x_{i,j}) \mid 1 \leq j \leq m\}$. All edges in a group must share the same edge-cut value, corresponding to the static grouping map $\mathtt{Gr}_{\text{static}}$. By imposing this constraint, the truth assignment to Boolean propositions across literals can be guaranteed to be the same.

The reduction algorithm takes as input a 3-SAT formula, and applying Constructions 1 and 2, returns a graph with source $\mathtt{S} = s_0$, $\mathtt{I} = \bigcup_{j=1}^{m}\{\mathtt{I}_{\mathrm{T},j}, \mathtt{I}_{\mathrm{F},j}\}$, and edges are grouped according to Construction 2. Now, we will show that for the static setting, the routing problem is NP-hard. The following proof of Theorem 4.5 is taken from [107].

Theorem 4.5. Problem 4.4 is NP-complete.

Proof. We will show that Problem 4.4 is NP-hard by showing that Construction 2 is a correct polynomial-time reduction of the 3-SAT problem to Problem 4.4 i.e., any polynomial-time algorithm to solve Problem 4.4 can be used to solve 3-SAT in polynomial-time. Consider the graph constructed by Construction 2 for any propositional logic formula. The valid set of edge cuts C on this graph with cardinality $|C| \leq M$ is a witness for Problem 4.4. A witness for the 3-SAT formula is an assignment of the variables $x_1, \dots, x_n$. A witness to a problem is *satisfying* if the problem evaluates to *True* under that witness. Next, we show that a valid set of edge cuts C is a satisfying witness for Problem 4.4 iff the corresponding assignment to variables $x_1, \dots, x_n$ is a satisfying witness for the 3-SAT formula.

First, consider a satisfying witness for Problem 4.4. By Construction 2, the cardinality of the witness, $|C| = m \times n$ will be exactly M, which is the minimum number of edge cuts required to ensure no bypass paths on the constructed graph. This implies that each variable x_i has a Boolean assignment. By Construction 1, a strictly positive flow on the sub-graph of clause c_j implies that c_j is satisfied. By Construction 2, a strictly positive flow through the entire graph implies that all clauses in the 3-SAT formula are satisfied. Therefore, a satisfying witness to the 3-SAT formula can be constructed in polynomial-time from a satisfying witness for an instance of Problem 4.4.

Next, we consider a satisfying witness for the 3-SAT formula. The Boolean assignment for each variable x_i corresponds to edge cuts on the graph (see Fig. 4.10b). Any Boolean assignment ensures that there is no bypass path on the graph since either all incoming edges or all outgoing edges for each variable x_i are cut. This also corresponds to the minimum number of edge cuts required to cut all bypass paths, corresponding to $|C| = m \times n$. By Construction 1, a satisfying witness corresponds to a Path(s_{j-1}, s_j) on the sub-graph for each clause c_j. By Construction 2, observe that there exists a strictly positive flow on the graph. Thus, we can construct a satisfying witness to an instance of Problem 4.4 in polynomial time from a satisfying witness to the 3-SAT formula. Therefore, any 3-SAT problem reduces to an instance of Problem 4.4, and thus, Problem 4.4 is NP-hard. Additionally, Problem 4.4 is NP-complete since we can check the cardinality of C, and whether C is a valid set of edge cuts in polynomial time. □

Corollary 4.1. Problem 4.3 is NP-hard [112].

Proof. By Theorem 4.5, Problem 4.4 is NP-complete, and therefore by Lemma 4.5, Problem 4.3 is NP-hard. □

These insights can be extended to the complexity analysis for the reactive setting. In the reactive setting, a valid set of edge cuts is defined similar to the static setting, except in the grouping constraint that the edges must respect, as detailed in Remark 4.5, and restated below for clarity.

Recall that $\mathtt{Gr}(q)$ is the set of possible system transitions in the history variable q. All restrictions during history variable q are a subset of $\mathtt{Gr}(q)$, and in the accumulative placement of reactive constraints, they are all realized in the worst-case. Therefore, the reactive feasibility constraints (c8) checks if there is a $Path(\mathtt{s}_{\text{sys}}, T_{\text{sys}})$ on $\mathcal{G}_{\text{sys}}$ (i.e., from the system perspective, are we ensuring that there is a path to the goal) when all reactive constraints are accumulated and projected onto $\mathcal{G}_{\text{sys}}$. Instead of checking on $\mathcal{G}_{\text{sys}}$, we can verify the same condition by checking on $\mathcal{G}$ by statically mapping the edges $\mathtt{Gr}(q)$.

Definition 4.18 (Static Mapping)**.** For a network $\mathcal{G}$, let $E' \subseteq \mathcal{G}.E$ be a set of edges in which each edge has an associated edge-cut value d^e. The network $\mathcal{G}$ is *statically mapped* with respect to E' if for every edge $(u, v) \in E'$, the following is true:

$$d^{(u',v')} = d^{(u,v)}, \; \forall (u', v') \in \mathtt{Gr}_{\text{static}}((u.s, v.s)). \tag{4.39}$$

The static mapping connects restrictions on the same system action across history variables. The feasibility networks are necessary to ensure that the system restrictions do not block the system from its goal.

Definition 4.19 (Reactive Feasibility Networks)**.** For each $q \in \mathcal{B}_\pi.Q$ and for every possible system source $\mathtt{s} \in \mathtt{S}_{G_{\text{sys}}}(q)$, introduce a copy of $\mathcal{G}_{\text{sys}}$ denoted $\mathcal{G}_{\text{sys}}^{(q,\mathtt{s})} = (V_{\text{sys}}, E_{\text{sys}}, \mathtt{s}, \mathtt{T}_{\text{sys}})$. The set of edges in $\mathcal{G}$ that map to some edge $(u_{\text{sys}}, v_{\text{sys}}) \in E_{\text{sys}}$ is

$$\mathcal{P}_{E_{\text{sys}} \to E}((u_{\text{sys}}, v_{\text{sys}})) = \{(u, v) \in E | \mathcal{P}_{G \to G_{\text{sys}}}(u) = u_{\text{sys}} \text{ and } \mathcal{P}_{G \to G_{\text{sys}}}(v) = v_{\text{sys}}\}.$$

Note that multiple edges on $\mathcal{G}$ can map to the same edge on $\mathcal{G}_{\text{sys}}$. Furthermore, reactive restrictions at history variable q are all contained in $\texttt{Gr}(q)$. Therefore, if one of the edges in $\mathcal{P}_{E_{\text{sys}} \to E}((u_{\text{sys}}, v_{\text{sys}})) \cap \texttt{Gr}(q)$ is restricted, then the edge $(u_{\text{sys}}, v_{\text{sys}})$ is restricted on the copy $\mathcal{G}_{\text{sys}}^{(q,\texttt{s})}$. Let $\mathbf{d}_{\text{sys}} \in \mathbb{B}^{|E_{\text{sys}}|}$ denote the cut values of edges on $\mathcal{G}_{\text{sys}}^{(q,\texttt{s})}$. The system restrictions on $\mathcal{G}$ are mapped to edge-cuts on $\mathcal{G}_{\text{sys}}^{(q,\texttt{s})}$ only for the history variable q:

$$\begin{aligned} d_{\text{sys}}^{(u_{\text{sys}}, v_{\text{sys}})} = \max \quad & d^{(u,v)} \\ \text{s.t.} \quad & (u,v) \in \mathcal{P}_{E_{\text{sys}} \to E}((u_{\text{sys}}, v_{\text{sys}})) \cap \texttt{Gr}(q). \end{aligned} \tag{4.40}$$

The *reactive feasibility networks* $\mathfrak{G}_{\text{sys}}$ is the set of graphs $\mathcal{G}_{\text{sys}}^{(q,\texttt{s})}$ whose edge cut values are mapped according to Eq. (4.40):

$$\begin{aligned} \mathfrak{G}_{\text{sys}} := \{ \mathcal{G}_{\text{sys}}^{(q,\texttt{s})} = & (V_{\text{sys}}, E_{\text{sys}}, \texttt{s}, \texttt{T}_{\text{sys}}) | q \in \mathcal{B}_\pi.Q, \ \texttt{s} \in \texttt{S}_{G_{\text{sys}}}(q) \text{ and} \\ & \mathcal{G}_{\text{sys}}^{(q,\texttt{s})} \text{ is mapped according to Eq. (4.41)} \}. \end{aligned} \tag{4.41}$$

In Alg. 5, lines 9–13 correspond to the construction of $\mathfrak{G}_{\text{sys}}$. In the implementation of the optimizations, edge cut variables $\mathbf{d}_{\text{sys}}$ for system feasibility networks are not defined since this would dramatically increase the number of integer variables. Instead, edge cuts on $\mathcal{G}$ are directly used to cut the flow on the feasibility networks (see Eq. (c8)).

When the cut-set C is found for $\mathcal{G}$, the *reactive feasibility condition* requires that for every reactive feasibility network $\mathcal{G}_{\text{sys}}^{(q,\texttt{s})} \in \mathfrak{G}_{\text{sys}}$, there exists a path from source $\texttt{s}$ to target T_{sys} after the cuts C are applied to $\mathcal{G}^{(q,\texttt{s})}$ via the mapping in Eq. (4.40). For the purpose of proving computational complexity, it is easier to reduce from 3-SAT if reasoning over graphs with similar structures. Thus, we consider the following check for reactive feasibility which reasons over copies of $\mathcal{G}$ instead of $\mathcal{G}_{\text{sys}}$.

Definition 4.20 (Statically mapped Reactive Feasibility Networks)**.** For each $q \in \mathcal{B}_\pi.Q$ and for every possible source $\texttt{s} \in \texttt{S}_G(q)$, introduce a copy of $\mathcal{G}$ denoted $\mathcal{G}^{(q,\texttt{s})} := (V, E, \texttt{s}, \texttt{T})$. Each network $\mathcal{G}^{(q,\texttt{s})}$ is statically mapped with respect to the edges $\texttt{Gr}(q)$. The *statically mapped reactive feasibility networks* $\mathfrak{G}$ is the set of all $\mathcal{G}^{(q,\texttt{s})}$:

$$\begin{aligned} \mathfrak{G} := \{ \mathcal{G}^{(q,\texttt{s})} = & (\mathcal{G}.V, \mathcal{G}.E, \texttt{s}, \texttt{T}) | q \in \mathcal{B}_\pi.Q, \ \texttt{s} \in \texttt{S}_G(q) \text{ and} \\ & \mathcal{G}^{(q,\texttt{s})} \text{ is statically mapped with respect to } \texttt{Gr}(q) \}. \end{aligned} \tag{4.42}$$

In other words, all edges restricted (i.e., $d^e = 1$) at history variable q are statically mapped on $\mathcal{G}^{(q,\texttt{s})}$. When the cut-set C is found for $\mathcal{G}$, the *reactive feasibility* *condition (via static mapping)* requires that for every statically mapped reactive feasibility network $\mathcal{G}^{(q,\mathtt{s})} \in \mathfrak{G}$, there exists a path from source $\mathtt{s}$ to target T after the cuts C are applied to $\mathcal{G}^{(q,\mathtt{s})}$ via the static mapping. Checking for the reactive feasibility condition (via static mapping) on $\mathfrak{G}$ is equivalent to checking the reactive feasibility condition on $\mathfrak{G}_{\text{sys}}$.

Theorem 4.6. The reactive feasibility condition (via static mapping) on $\mathfrak{G}$ is true iff the reactive feasibility condition on $\mathfrak{G}_{\text{sys}}$ is true.

Proof. Suppose the reactive feasibility condition (via static mapping) is true. Then, the corresponding graphs $\mathcal{G}_{\text{sys}}$ will also have a path by theorem relating to why static checks are enough. If the reactive feasibility condition on $\mathfrak{G}_{\text{sys}}$ is true, then there exists a corresponding path on $\mathcal{G}$ as well due to lemma 4.1. $\square$

For each history variable $q \in \mathcal{B}_\pi.Q$ and for every possible system source $\mathtt{s} \in \mathtt{S}_G(q)$, introduce a copy of $\mathcal{G}$ denoted $\mathcal{G}^{(q,\mathtt{s})} = (V, E, \mathtt{s}, \mathtt{T})$. On this copy $\mathcal{G}^{(q,\mathtt{s})}$, we statically map the edges $\mathtt{Gr}(q)$, and check if the flow from $\mathtt{s}$ to $\mathtt{T}$ is at least 1. In the reactive setting, a *valid* set of edge cuts C when applied to G is such that: i) there are no bypass paths, ii) there exists at least one path from $\mathtt{S}$ to $\mathtt{T}$, and iii) edges of G respect the reactive grouping condition (via static mapping). Finally, the optimization and decision problems in the reactive setting can be stated as follows.

Problem 4.5 (Routing Problem, Reactive Setting (Optimization))**.** Given a graph G, find a valid set of edge cuts C in the reactive setting such that the resulting maximum flow F is maximized over all possible sets of edge cuts, and such that $|C|$ is minimized for the flow F.

Problem 4.6 (Routing Problem, Reactive Setting (Decision))**.** Given a graph G and an integer $M \geq 0$, does there exist a valid set of edge cuts C in the reactive setting such that $|C| \leq M$?

Once again, consider the 3-SAT reduction from the static setting. This reduction will be adapted to construct a polynomial-time reduction of 3-SAT to an instance of Problem 4.6 with a single history variable q. Similar to the static setting, I will prove that Problem 4.6 is NP-complete in the size of the product graph G. To do this, we will first establish that Problem 4.6 is in the class of NP problems.

Lemma 4.7. Problem 4.6 is in the class of NP problems.

Proof. This proof follows similarly to the proof of Lemma 4.6. Given a solution C, we need to show that verifying C to be a valid set of edge cuts for the reactive setting can be carried out in polynomial-time. Conditions (i) and (ii) can be checked similarly as in, and condition (iii) can also be checked in polynomial-time. □

Similar to the static setting, 3-SAT can be reduced to an instance of Problem 4.6. In this chapter, I will construct the reduction to an instance of Problem 4.6 with a single history variable.

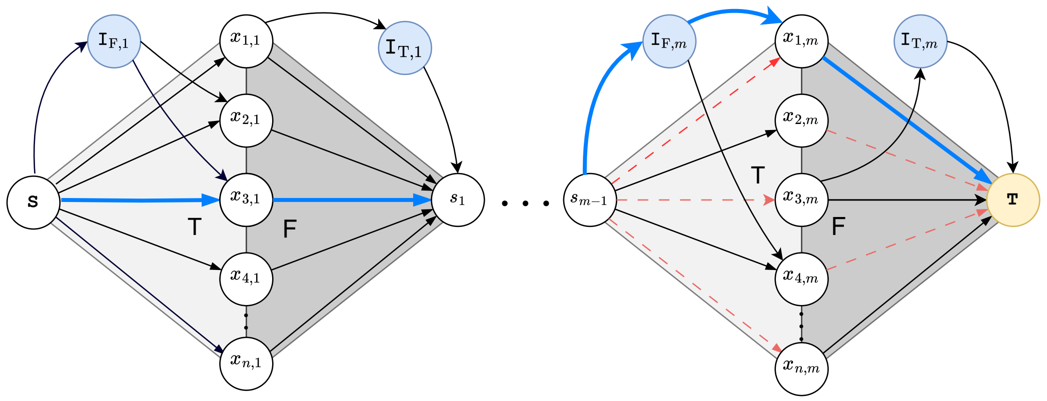
Construction 3 (Reduction from 3-SAT to Problem 4.6 with single history variable q)**.** Given a 3-SAT formula with n propositions and m clauses. Construct a graph, denoted $G^{(q,\mathtt{S})}$, according to Construction 2. In addition, construct a graph G with nodes and edges according to Construction 2 without applying the constraint that all edges in a group must have the same edge-cut value. The graph $G^{(q,\mathtt{S})}$ serves a reactive feasibility network (via static mapping) where S is the source at history variable q. On the other hand, edges on G are not grouped together. In addition to graphs, the edge cuts on G and $G^{(q,\mathtt{S})}$ are mapped as follows: the edge-cut value of a group in $G^{(q,\mathtt{S})}$ is set to the maximum edge-cut value in the equivalent group in G.

Figures 4.11a and 4.11b are constructed from a 3SAT formula for the reactive optimization problem, where a truth assignment for the variables can be found by solving **MILP-REACTIVE**.

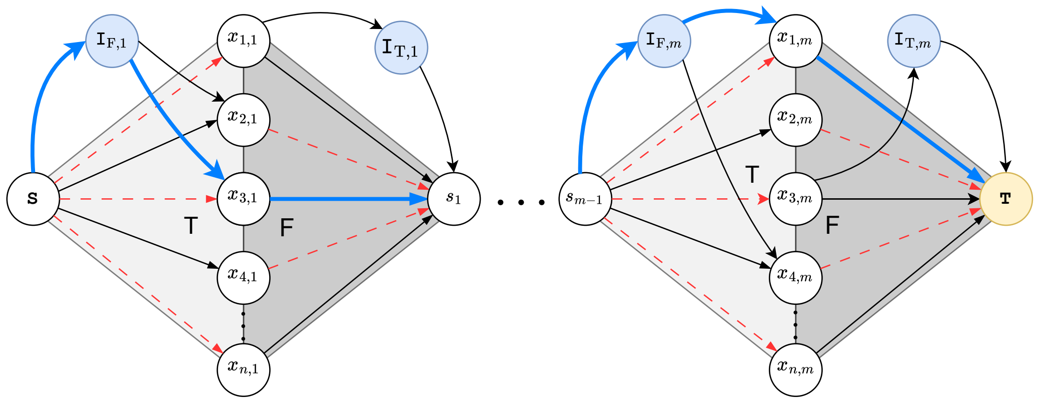
The following theorem and proof is taken from [107].

Theorem 4.7. Problem 4.6 is NP-complete and Problem 4.5 is NP-hard.

Proof. The proof follows similarly from Theorem 4.5. In this setting, a witness for Problem 4.6 comprises the maximum edge cut value of each group in G. Construction 3 relates edge cuts on G and $G^{(q,\mathtt{S})}$. This implies that edge cuts on G are found under the condition that there is a strictly positive flow on $G^{(q,\mathtt{S})}$ under a static mapping of edges. The minimum set of edge cuts which ensures no bypass paths on G has cardinality n, corresponding to only one of the sub-graphs having edge cuts. Furthermore, for each x_i, there will be one edge-cut in one of the two groups (incoming or outgoing edges). Therefore, for each x_i, only the incoming or the outgoing edge group will have a maximum edge cut value of 1, corresponding to the Boolean assignment for x_i. A minimum cut on G found under the conditions of no bypass paths on G and a positive flow on $G^{(q,\mathtt{S})}$ results in a Boolean assignment



(a) Constructed graph G for an arbitrary 3SAT formula $f(x)$.



(b) Statically mapped cuts on G for every subgraph.

Figure 4.11: (a) Graph G and (b) graph G_{sys} constructed according to Construction 3.

that is a satisfying witness to the 3-SAT formula. Thus, we have polynomial-time construction of a satisfying witness to the 3-SAT formula from a satisfying witness to Problem 4.6. This follows similarly to Theorem 4.5.

Likewise, a satisfying witness to the 3-SAT formula can be mapped to edge cuts on one of the sub-graphs of G. These edge cuts will be such that there is no bypass path on G, and will be the minimum set of edge cuts to accomplish this task, corresponding to $|C| = n$. Additionally, by construction of the graphs, this will correspond to a strictly positive flow on $G^{(q,\mathtt{S})}$. Thus, we can construct a satisfying witness to Problem 4.6 in polynomial time from a satisfying witness of the 3-SAT formula. Therefore, any 3-SAT problem reduces to an instance of Problem 4.6. As a result, Problem 4.6 is NP-complete and following similarly to Corollary (4.1), Problem 4.5 is NP-hard. $\square$

4.10 Comparison to Reactive Synthesis

We presented an approach to solve Problems 4.1 and 4.2 leveraging tools from automata theory and network flow optimization. In particular, for Problem 4.2, we rely on the optimization solution to construct a GR(1) specification to reactively synthesize a test agent strategy. One indication of the optimization step being necessary is the computational complexity of the problem. If the problem data are consistent, there exists a GR(1) specification for the test agent that would solve the problem, but directly expressing this specification is impractical. Essentially, the challenge is in finding the restrictions on system actions, which are then captured in the sub-formulas of the GR(1) specification. In this section, we argue that we cannot solve Problems 4.1 and 4.2 solely via synthesis from an LTL specification.

To the authors' knowledge, directly capturing the different perspectives of the system and the tester in this neither fully adversarial nor fully cooperative setting is not possible with current state-of-the-art approaches in GR(1) synthesis. Particularly in the reactive setting, the test strategy must ensure that from the system's perspective, there always exists a path to the system goal. To capture this constraint, we reason over a second product graph that represents the system perspective. It is not obvious how this semi-cooperative setting can be directly encoded as a synthesis problem in common temporal logics.

In the static setting, the problem can be posed on a single graph. However, it is difficult to find the set of static obstacles directly from GR(1) synthesis. Every state in the winning set describes an edge-cut combination, but qualitative GR(1) synthesis cannot maximize the flow or minimize the cuts. Furthermore, the winning set can include states that vacuously satisfy the formula, i.e., not allowing the system any path to the goal. Finally, the combinatorial complexity of the problem would manifest as follows. Although the time complexity of GR(1) synthesis is $O(N^3)$ in the number of states N, we require an exponential number of states to characterize the GR(1) formula. For example, in Figure 4.12, this is illustrated for the GR(1) formula:

$$\square\varphi_{\text{sys}}^{\text{dyn}} \wedge \square\lozenge\, \mathtt{T} \rightarrow \square\varphi_{\text{test}}^{\text{dyn}} \wedge \square\varphi_{\text{test}}^{\text{aux\_dyn}} \wedge \square\lozenge\, I_{\text{aux}},$$

where $\varphi_{\text{sys}}^{\text{dyn}}$ captures the system transitions on the gridworld, $\varphi_{\text{test}}^{\text{dyn}}$ are the dynamics of the test environment, and $\varphi_{\text{test}}^{\text{aux\_dyn}}$ and I_{aux} capture the $\lozenge I$ condition in GR(1) form. In this example, each edge in the system transition system T can take 0/1 values, and once an edge is cut, it remains cut and the system cannot take a transition that corresponds to a cut edge. Due to this, the number of states N to describe

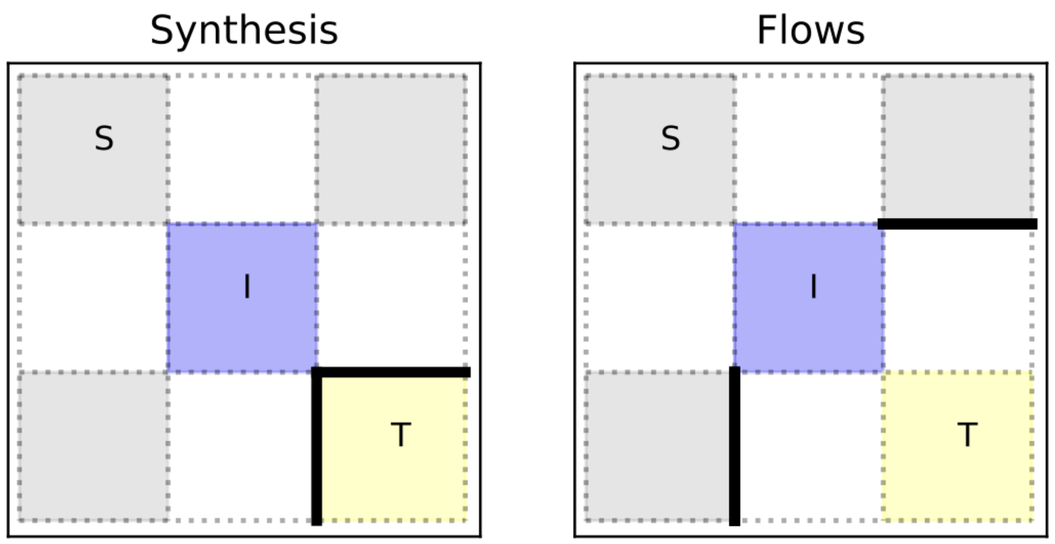


Figure 4.12: Solution returned by GR(1) synthesis and the network flow optimization in the case of static constraints

the GR(1) formula includes the $2^{|T.E|}$ states that characterize the edge cuts. As seen in Figure 4.12, the direct GR(1) synthesis approach returns a trivial solution corresponding to an impossible setting for the system. Finally, even when an acceptable solution is returned, the problem being at least NP-hard will result in the combinatorial complexity manifesting in the synthesis approach.

One key advantage of the network flow optimization is reasoning over flows as opposed to paths, which allows for tractable implementations. These insights from network flow optimization in this work can help in driving further research along these directions.

4.11 Experiments

We illustrate the synthesized test strategy in simulation and hardware with Unitree A1 quadrupeds. These experiments show that the high-level abstraction models is useful in high-level test synthesis as long as the lower levels of the system are implemented to simulate the high-level abstraction. The low-level control of the quadruped is managed at the motion primitive layer, which abstracts the underlying dynamics and facilitates transitions between primitives as described in [113]. These primitives include behaviors such as lying down, walking at various speeds, jumping, standing, and reduced-order model-based tracking of waypoints that rely on a unicycle or single integrator model. These motion primitives can directly be commanded from a high-level controller implemented by a temporal logic planning

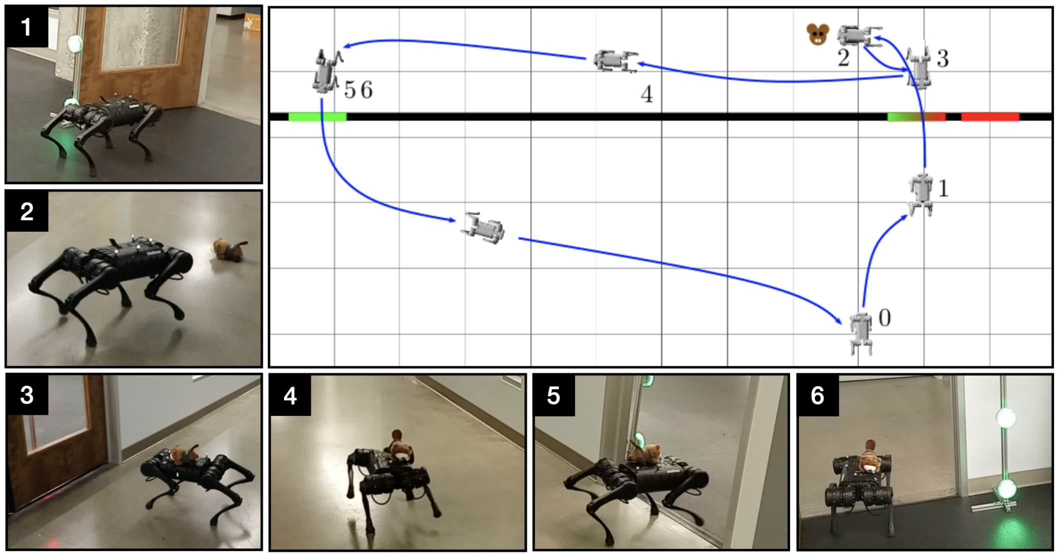


Figure 4.13: Beaver Rescue Hardware Experiment with door$_1$ on the right and door$_2$ on the left.

toolbox such as TuLiP [109]. Each motion primitive is implemented in C++, with control laws, sensing, and estimation executed at 1kHz.

For experiments demonstrating the flow-based synthesis in the MILP formulation, examples with static test environments solve the routing optimization **MILP-STATIC**, examples with reactive test environments solve **MILP-REACTIVE**, and those with reactive dynamic agents solve **MILP-AGENT**, unless otherwise stated. These optimizations are solved using Gurobipy [114]. The reactive test agent strategies are synthesized using the temporal logic planning toolbox TuLiP [109].

Hardware Experiments for Tests Synthesized from solving the Min-Max Stackelberg Game

We will see two hardware experiments — beaver rescue and motion primitive examples — for the flow-based synthesis formulated via min-max Stackelberg games. These optimizations were implemented in Pyomo [104], which interfaces to Gurobipy. These examples will be revisited in simulation and solved using the MILP framework.

Beaver Rescue Example: This example is inspired by a search and rescue mission and the hardware trace is shown in Fig. 4.13. In this example, the quadruped (system under test) is tasked with picking up a beaver (located in the corridor), and returning to lab safely: $\varphi_{\text{sys}} = \diamondsuit\, goal$, where $goal$ is satisfied when the beaver is brought into the lab. The lab has two doors which the quadruped can use to navigate into the corridor. In our implementation of the discretized abstraction of this experiment, the transitions of the quadruped are as follows,

$$\begin{aligned} T_{\text{sys}}.E =&\{(s_0,d_1),(s_0,d_2),(d_1,d_2),(d_2,d_1),(d_1,b),(d_2,b),(b,p_1),\\ &(b,p_2),(p_1,p_2),(p_2,p_1),(p_1,g),(p_2,g)\}, \end{aligned} \tag{4.43}$$

where i) states d_1 and d_2 are states in the lab adjacent to doors: $T_{\text{sys}}.L(d_1) = \{\text{door}_1\}$ and $T_{\text{sys}}.L(d_2) = \{\text{door}_2\}$, ii) states p_1 and p_2 are states in the *corridor* adjacent to doors: $T_{\text{sys}}.L(p_1) = \{\text{door}_1\}$ and $T_{\text{sys}}.L(p_2) = \{\text{door}_2\}$, iii) state b in the hallway is the rescue location of the beaver, and iv) states s_0 and g represent the lab. The test objective is to route the system to visit both doors: $\varphi_{\text{test}} = \Diamond \text{door}_1 \wedge \Diamond \text{door}_2$.

There are several ways in which the test environment could have routed the system. If the system visits door$_1$ from d_1 (or likewise door$_2$ from d_2), the door could then be blocked; forcing the system to re-plan to exit into the corridor through the other door. Alternatively, the system could visit door$_1$ from d_1 (or likewise door$_2$ from d_2) and exit into the corridor, and on its return with the beaver, door$_1$ from p_1 (or likewise door$_2$ from p_2) can be blocked while leaving the other door at p_2 (or likewise at p_1) open for the quadruped to re-enter the lab. Our algorithm found edge-cuts that resulted in the latter test case that allows the system to exit through the door of its choice and blocks that door on the return path. The synthesized test is reactive to the choice of system actions — depending on the door approached by the system, the synthesized constraints are placed accordingly.

Motion Primitive Example: In this example, the quadruped is can execute the following motion primitives: "jump", "stand", "lie", and "walk". Once again consider the lab-corridor setup. The quadruped's goal is to reach the beaver in the corridor: $\varphi_{\text{sys}} = \Diamond \text{goal}$. The test objective is $\varphi_{\text{test}} = \Diamond \text{jump} \wedge \Diamond \text{lie} \wedge \Diamond \text{stand}$ in order to test the system to demonstrate the three motion primitives.

Unlike the previous example in which doors were closed to restrict the system, in this example each door has three lights located at different heights to signal motion primitives that might unlock the door. There are three such doors, and the states p_i (for $i = 1, 2, 3$) represents the state of the quadruped standing in front of door$_i$ before demonstrating a motion primitive. The state prim$_i$ for motion primitive prim $\in \{\text{lie}, \text{stand}, \text{jump}\}$ represents the abstract state of the quadruped performing the motion primitive in front of door$_i$. After it performs a motion primitive, the quadruped state transitions to $d_{i,\text{prim}}$, from which it can proceed to the goal or be

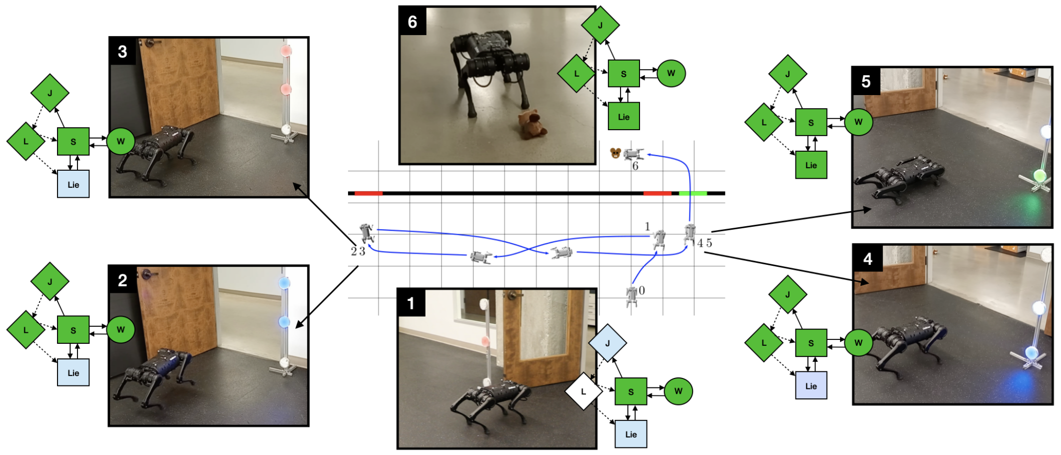


Figure 4.14: Motion Primitive example: Snapshots of the hardware test execution on the Unitree A1 quadruped.

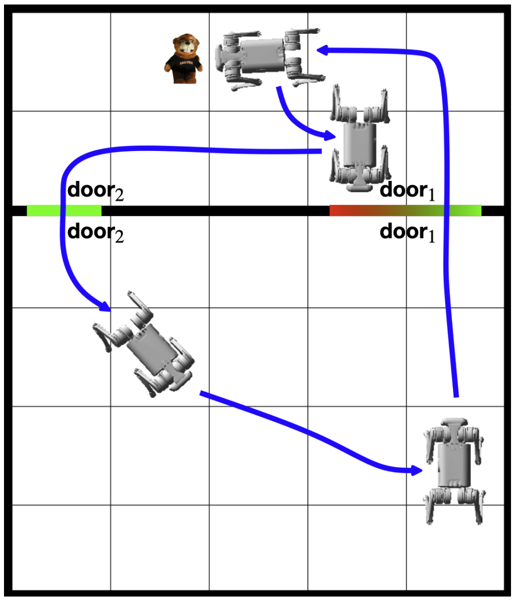
returned to the state p_i. The test harness comprises of system actions corresponding to the following transitions: $\{(d_{i,\text{prim}}, goal)\}$.

For example, if the middle light is blue, it implies that demonstrating the stand motion primitive could unlock the door (by the light turning green). The test strategy is reactive to the system; depending on the order in which the quadruped approaches the doors and demonstrates motion primitives, the lights turn red/green to restrict/permit the system to pass.

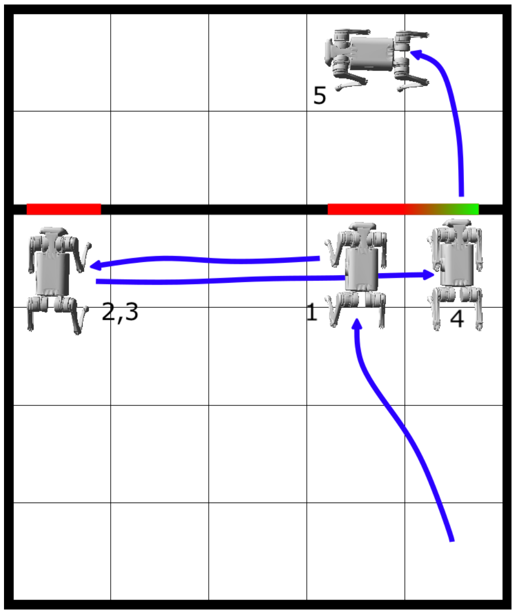
In this test execution, the system chooses to approach the middle door (door$_2$) first which can only be unlocked by the stand motion primitive. The quadruped successfully demonstrates this (panel 1 of Fig. 4.14), but the light turns red. Following this, the quadruped approaches (door$_1$) and demonstrates the jump and stand motion primitives, but is still not permitted to pass (panels 2 and 3 of Fig. 4.14). Finally, after approaching door$_3$ on the right, the system demonstrates the lie motion primitive, after which the corresponding light turns green (panels 4 and 5 of Fig. 4.14), and the quadruped finally navigates to the corridor. In this manner, the test strategy reacts to system behavior and routes the test execution to lead the system to demonstrate all three motion primitives before being allowed to pass.

Simulated Experiments

First, we will revisit the beaver rescue and motion primitive examples, in which test strategies will be implemented by the test environment using reactive obstacles For the beaver rescue example, the test harness consists of doors that connect the lab and corridor, and system transitions can be restricted by closing the doors. For the



(a) Beaver rescue.



(b) Motion primitive example.

Figure 4.15: Simulated experiment results with test strategy found by solving **MILP-REACTIVE**. In (b), system (gray) demonstrates primitives in the order: stand (1), stand (2), jump (3), and lie (4), before advancing to goal (5).

motion primitive example, the test harness consists of restricting transitions after motion primitives have been demonstrated. Figure 4.15 shows the simulated experiments for these examples where the test strategy was found by solving **MILP-REACTIVE**. The simulated test executions are qualitatively similar to the hardware demos discussed previously, even with the new MILP formulation. As shown in Table 4.4, the graph size $|G|$ for these examples is relatively small compared to other examples, with the exception of the running examples 4.2 and 4.1. Despite this, the MILP approach is faster by three orders of magnitude. Both the game formulation and the MILP formulation aim to solve the problem exactly. However, defining a single set of flows and directly solving the problem as an MILP makes the problem much more tractable. A large part of this can be attributed to Gurobi, and the algorithms for solving min-max stackelberg games with coupled constraints have not been optimized for efficiency as much in comparison to solving MILPs.

Maze 1: This example consists of a system quadruped (gray) navigating on the grid shown in Fig. 4.17a, and a dynamic test agent (yellow) that can traverse the middle column. The test agent is restricted to only walk up or stay in a cell. From the middle cell of the top row, the test agent can navigate off the grid into a parking state. The system objective is to reach the goal on the top-left corner of the grid,

$\varphi_{\text{sys}} = \Diamond\, goal$, and the test objective is to route the system through intermediate states I_1, I_2, and I_3: $\varphi_{\text{test}} = \Diamond\, I_1 \wedge \Diamond\, I_2 \wedge \Diamond\, I_3$.

Figures 4.16a– 4.16c visualize a counterexample that is not dynamically realizable by the test agent. This solution is added as a counterexample, and the MILP is resolved until a realizable solution (see Figures 4.16d– 4.16f) is found.

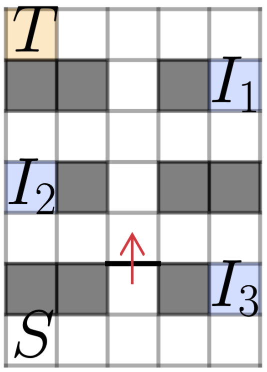
Brief explanation for counterexample: In Figure 4.16a, an agent would have to occupy cell $(4, 2)$ when the system occupies the cell $(5, 2)$. This would result in a livelock — from the system perspective, there is no incentive to back up and navigate around through $\mathtt{I}_3$ since the test agent would block all paths to the goal, and if the test agent moves out of cell $(4, 2)$, the system can navigate to the goal without being routed through $\mathtt{I}_3$.

The test quadruped first begins in the lower-most row and moves out of the way but still blocking the path through the center column so that the system is routed through I_3. Once the system visits I_3, the test agent walks up to the middle cell in the grid to block it so it is routed through I_2. Similarly, the test agent routes the system through I_1. After the system visits I_1 but before it reaches the center cell in the first row, the test agent walks off the grid, and into its parking state. This is due to the temporal logic constraint to not over-restrict the system (equation (g7)). When any cell occupied by the test agent (say v) is adjacent to the system (say occupying cell u), then the transition (u, v) is registered as a restriction on the system. To avoid over restricting the system, the test agent navigates of the grid.

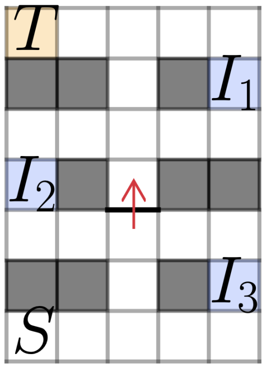
Hardware Experiments

Running Example 4.1: The experiment trace for the medium example is given in Fig. 4.17c. The corresponding solution is shown in Fig. 4.7.

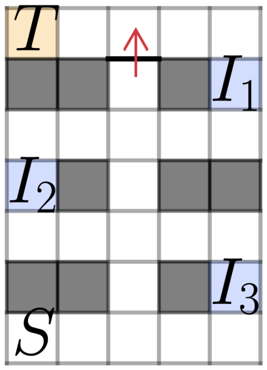
Refueling Example: In this example, the system quadruped (gray) navigates on the grid shown in Figure 4.18a. In addition to coordinates $\mathbf{x} = (x, y)$, the system state also includes a discretized fuel state $fuel$. The maximum value of $fuel$ is 10, and every cell transition on the grid decreases this value by 1. Visiting the refueling station in the bottom-right corner of the grid resets $fuel$ to its maximum value. The desired test behavior is to place the system in a state in which its fuel level is not sufficient for it to directly navigate to the goal. The system objective is given as $\varphi_{\text{sys}} = \Diamond\, T \wedge \Box\neg(fuel = 0)$. The test objective is set to $\varphi_{\text{test}} = \Diamond(y < 4 \wedge fuel < 2)$, which seeks to place the system in the lower 3 cells of the grid with less than two units of fuel. The sub-tasks used in describing these objectives are safety and



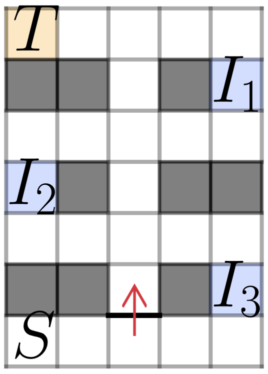
(a) q0



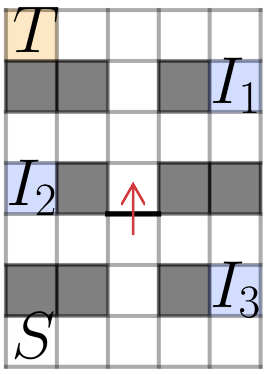
(b) q15



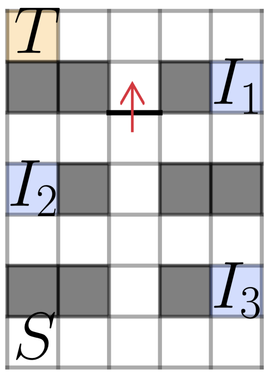
(c) q12



(d) q0

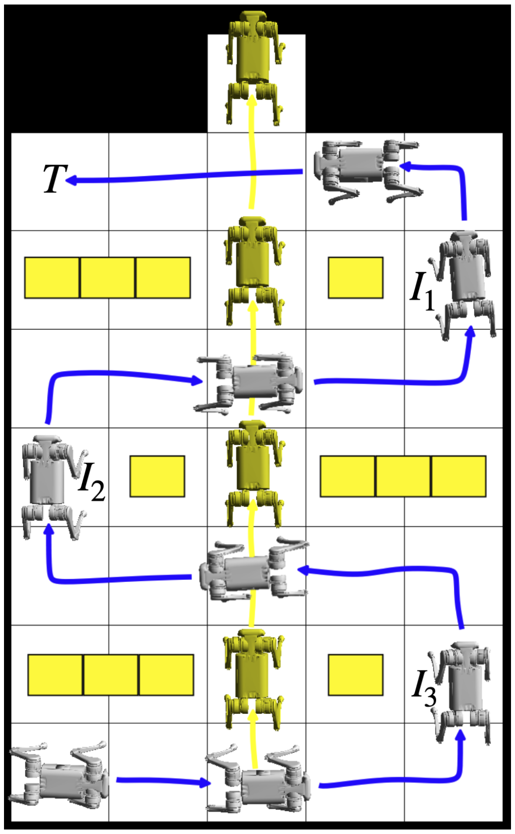


(e) q15

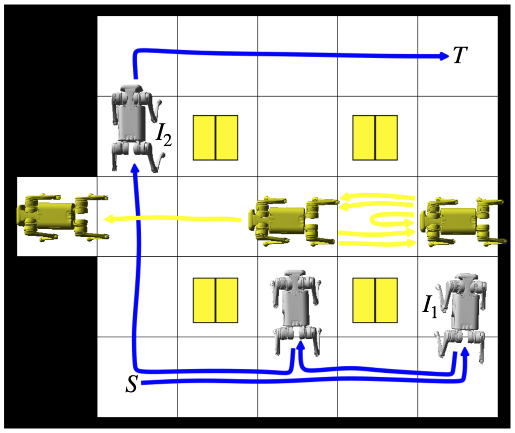


(f) q12

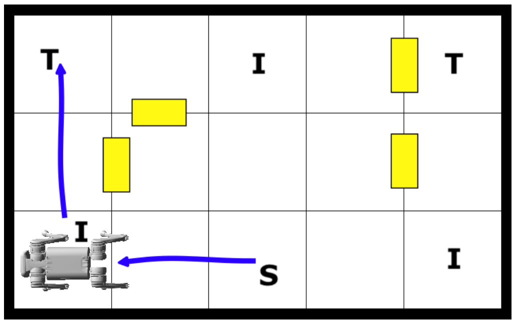
Figure 4.16: Illustration of dynamically unrealizable (top (a)–(c)) and dynamically realizable reactive obstacles (bottom (d)–(f)). In Figures 4.16a– 4.16c: Reactive obstacles returned by **MILP-REACTIVE** that cannot be realized by a dynamic test agent. In Figures 4.16d– 4.16f: Accepted solution for which a test agent strategy is synthesized. Red arrow indicates the direction of the restriction; the edge-cuts found by **MILP-REACTIVE** are not subject to the (optional) bidirectional cut constraint. History variable q0 refers to the state of the test execution before I_3 is visited by the system, q15 is the state of the test execution after only I_3 is visited, and q12 is the state of the test execution after I_3 and I_2 have been visited.



(a) Simulated experiment for Maze 1.



(b) Simulated alternative trace, Maze 2.

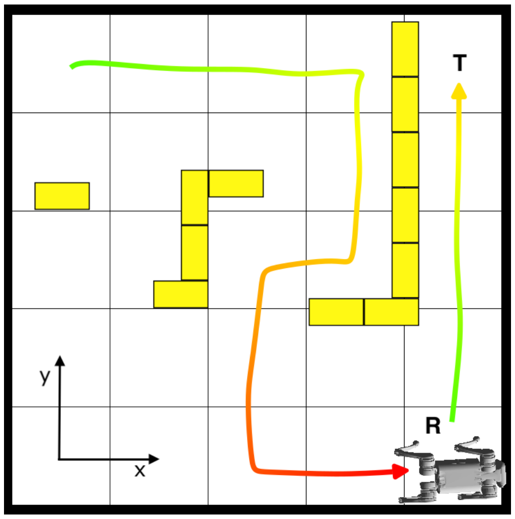


(c) Hardware trace for the medium example 4.1 with static obstacles found by test strategy.

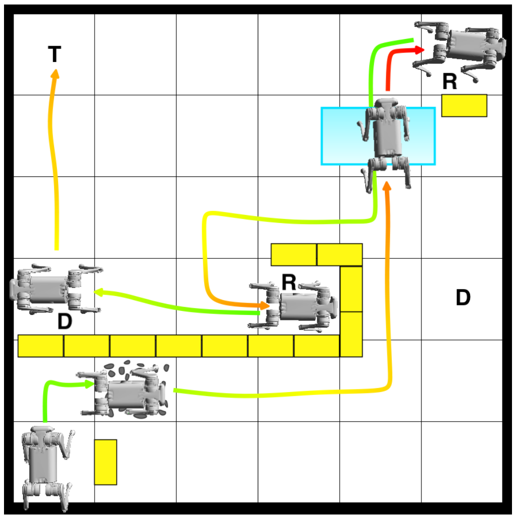
Figure 4.17: Yellow boxes in (a) and (b) are pre-defined obstacles to indicate states that are not navigable in T_{sys}. Yellow obstacles in (c) are static obstacles placed by the test environment. Gray quadruped is the system, and yellow quadruped in (a) and (b) is the test agent, which chooses to navigate off-grid after the test objective is realized.

reachability. Note that the intermediate states resulting from this test objective also include states with $fuel = 0$, but the restrictions from the MILP will not force the system into these *unsafe* states, giving the system the option to have a fuel level of 1 and refuel. This still satisfies the test objective without making it impossible for the system to satisfy the test objective.

The experiment trace of the test execution in shown in Fig. 4.18a, in which the color of the trace indicates the comparative fuel level at that state. The yellow boxes represent static obstacles placed to restrict transitions according to the solution of **MILP-STATIC**. As given in Tables 4.4 and 4.5, the product graph has over 1000 edges resulting in around 1000 binary variables for the routing optimization. The optimization is solved to optimality in 0.87 seconds, and the maximum flow is found



(a) Refueling example experiment trace.



(b) Mars exploration experiment trace.

Figure 4.18: Traces of hardware demos with test environment consisting of static obstacles.

to be 2. The 199 cuts on G correspond to the 8 transitions restricted (bidirectionally) on the transition system. This example illustrates the usefulness of our framework — test objectives are not limited to being defined over atomic propositions of the pose $\mathbf{x}$ of the system. The solution to this specific example is not one that can be easily identified like the previous examples we have discussed thus far.

Mars Exploration Example: This example is inspired by a sample collection mission on Mars. The sub-tasks reachability, avoidance, and delayed reaction are used to characterize system and test objectives. The system quadruped (gray) can traverse the grid shown in Figure 4.20a, which has states with "rock" and "ice" samples, and states designated as sample drop-off locations D, and refueling stations denoted R. The system is required to reach the goal in the top-level corner (labeled T), and must drop-off any samples collected during its navigation without running out of fuel. The system state carries a fuel level $fuel$ in addition to its pose state $\mathbf{x} = (x, y)$. Similar to the refueling example, the maximum fuel value is 10, it decreases by 1 for every transition on the grid, and it can be refueled by visiting the refueling states R.

The system objective is given by the formula:

$$\varphi_{\text{sys}} = \diamondsuit T \wedge \square\neg(f = 0) \wedge \square(\text{ice} \vee \text{rock} \rightarrow \diamondsuit \text{ drop-off}).$$

The test objective consists of reachability sub-tasks that include triggers of the reaction sub-task of the system objective, and also a sub-task to place the system in a

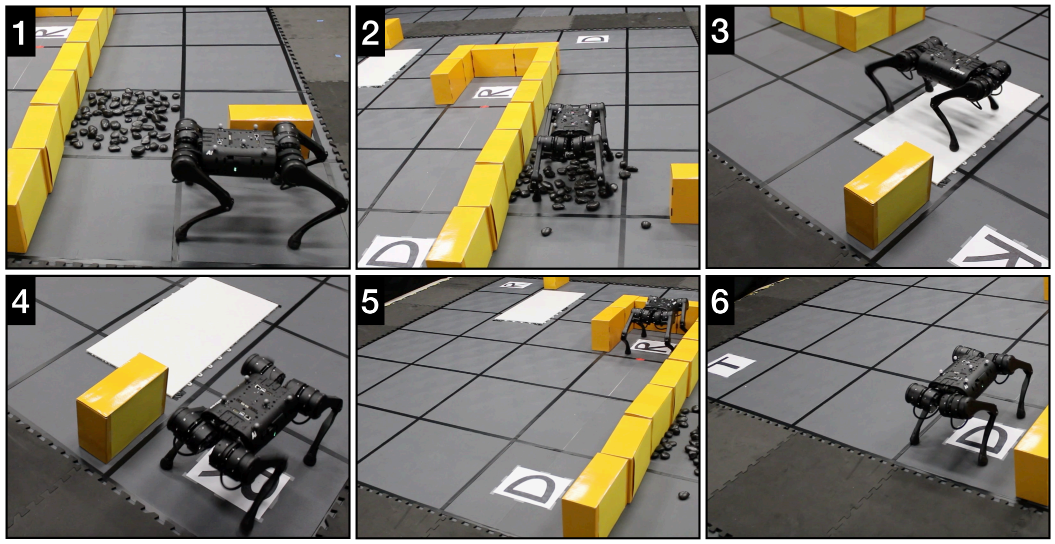


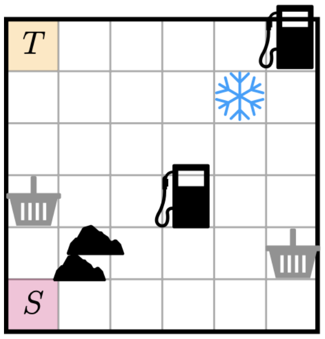
Figure 4.19: Mars exploration experiment snapshots from resulting on the Unitree A1 quadruped for static test environments. The overview is shown in Fig. 4.18b.

low-fuel state:

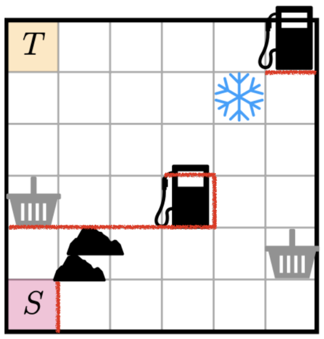
$$\varphi_{\text{test}} = \Diamond \, \text{rock} \wedge \Diamond \, \text{ice} \wedge \Diamond (d > f),$$

where $d = |\mathbf{x} - \mathbf{x}_{\text{goal}}|$ is the distance to the goal and f is the fuel level. Figure 4.20b shows a sub-optimal placement of static obstacles with maximum flow $F = 1$, and Figure 4.20c shows the optimal placement permitting a maximum flow of $F = 2$ on the product graph. The experiment trace (Figure 4.18b) and accompanying demo (Figures 4.19) are test executions in the test environment realizing the sub-optimal test strategy. The system begins in the bottom-left corner of the grid with a full fuel tank. From these figures, we can observe the quadruped being routed to pick up the rock sample close to the initial condition. Then, the placement of static obstacles in both the sub-optimal and the optimal settings is such that the system needs to visit the top-right refueling station at least once. In order to visit that refueling station without running out of fuel, the system must navigate the state with ice samples. In the test execution from the experiment, the system is routed to visit states with rock and ice samples, after which it refuels twice — first at the top-right refueling station and then at the refueling station at the center of the grid — and finally, drops off the samples before navigating to goal T. Table 4.5 lists this example as one of the largest with around $13,000$ integer variables. Despite this, the routing optimization is solved optimally in about 45 seconds.

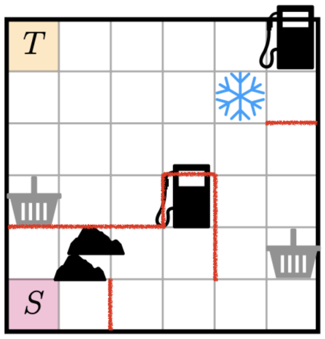
Patrolling Quadruped: This examples involves a dynamic test agent whose strategy is synthesized to be consistent with the solution of the routing optimization in



(a) Empty grid for Mars exploration example. Refueling stations, rock and ice sample locations are denoted. Drop-off locations are denoted by the basket.



(b) Static obstacles for Mars exploration example with a feasible solution of **MILP-STATIC** (max-flow F=1).

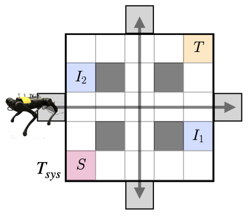


(c) Static obstacles for Mars exploration example with an optimal solution of **MILP-STATIC** (max-flow $F = 2$).

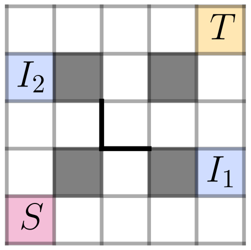
Figure 4.20: Feasible and optimal solutions for the Mars exploration example. The hardware experiment corresponded to the feasible solution.

MILP-AGENT. The context of this example is similar to the refueling example except that the test environment can now consist of static obstacles and a dynamic test agent (see Figure 4.1). The system (gray quadruped) is tasked with beginning in the lower right corner of the grid, and reaching the target cell in the lower-left corner without running out of fuel. Additionally, the system must not collide with obstacles. The test objective is to put the system in a low-fuel state similar to the Mars exploration example. The test agent dynamics allow it to traverse up and down the center column of the grid, and from the center cells of the top and bottom rows, it can choose to move off the grid into a parking state. Thus, the system and test objectives are given by the formulas: $\varphi_{\text{sys}} = \Diamond T \wedge \Box \neg (fuel = 0)$, and the test objective is $\varphi_{\text{test}} = \Diamond (d > f)$, where $d = |\mathbf{x} - \mathbf{x}_{\text{goal}}|$ is the distance to the goal.

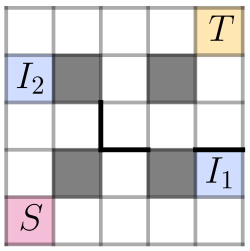
As seen in Figure 4.1, the test environment places a static obstacle near the initial state of the system (panel 1 snapshot). Then, as the system proceeds to go to the goal, the test agent blocks the quadruped from crossing the center column of the grid — in panels 1 and 2, the test agent blocks the system in the lowermost row, and when the system advances up in panels 3 and 4, the test agent continues to block the system. The test agent blocks the system until the it can no longer directly navigate to the goal, and must refuel. Thus, the system refuels and is then able to navigate to T without any further interactions with the test agent. Some implementation details are as follows. the system controller in this test execution resynthesizes its strategy each time it is restricted by the test agent. Furthermore, the optimization **MILP-**



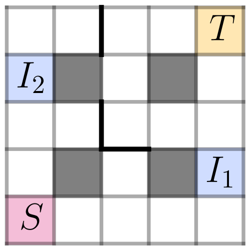
(a) Grid world layout.



(b) Reactive cuts in q_0.

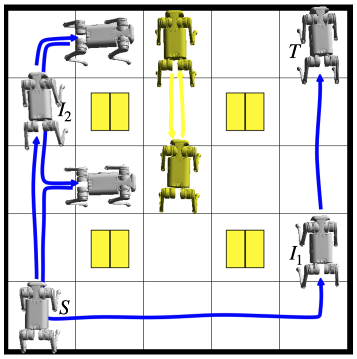


(c) Reactive cuts in q_6.

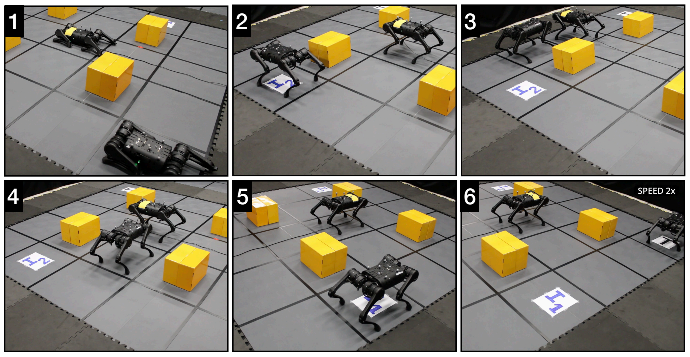


(d) Reactive cuts in q_7.

Figure 4.21: Grid world layout and reactive cuts corresponding to the history variables for the Maze 2 experiment. (a) Grid world layout with cells traversible by the test agent marked. Dark gray cells are not traversible by either agent. (b)–(d) Black edges indicate reactive cuts corresponding to the history variables for the Maze 2 experiment. Note that the cuts are not bidirectional. The history variable states q0, q6, and q7 can be inferred from $\mathcal{B}_\pi$ illustrated in Fig. 4.2c, and correspond to initial state, visiting I_1 first, and visiting I_2 first.



(a) Maze 2 trace.



(b) Maze 2 experiment snapshots.

Figure 4.22: Resulting test execution for the Maze 2 experiment with a dynamic test agent.

AGENT with the modified objective is solved to minimize the occupied states.

Maze 2: The grid world layout for this example is shown in Figure 4.21a, in which the gray boxes denote states that the system cannot navigate to. The system is tasked with navigating to goal T from state S: $\varphi_{\text{sys}} = \Diamond T$. The test objective is to route the system to visit states I_1 and I_2: $\varphi_{\text{test}} = \Diamond I_1 \wedge \Diamond I_2$. The test environment has access to a dynamic test agent that can traverse the center column and row of the grid as illustrated in yellow lines in Figure 4.21a. In addition, the test agent can walk off the grid into a parking state from the four cells at the boundaries of the grid (top and bottom cell of center column, and left-most and right-most cells of the center row). While the test environment can also place static obstacles, it chooses to restrict the system using just the test agent.

The specification product is exactly the same as the running example 4.2, and is illustrated in Figure 4.2c. Observe that to route the test execution through the test objective acceptance states, we need to find cuts for the history variables q0 (initial state), q6 (I_1 has been visited but not I_2), and q7 (I_2 has been visited but not I_1). The reactive cuts found by the flow-based synthesis procedure are shown in Figs. 4.21b-4.21d. The trace and snapshots of the resulting test execution is shown in Figs. 4.22a and 4.22b. We observe that the system quadruped decides to take the top path first, visits I_2 (see panel 2 in Fig. 4.22b), and is blocked by the test agent (see panel 3). It then decides to try navigating through the center of the grid, and is again blocked by the test agent (see panel 4). Subsequently, it decides to try the bottom path, visits I_1 (see panel 5), and successfully reaches the goal without any further test agent intervention. If the system decided to visit I_1 first, the adaptive test agent strategy would have blocked the system from reaching the goal directly from I_1 until it visits I_2. This is an example with a maximum flow of $F = 2$, corresponding to the two unique ways for the system to reach the goal. For an alternative system controller in which the system chooses to approach the goal through I_1, the simulated trace resulting from the test agent strategy is shown in Fig. 4.17b.

Runtimes

Tables 4.5 and 4.6 list the optimization size and runtimes for all the simulated and hardware experiments discussed prior. Table 4.5 corresponds to experiments that identify static and/or reactive obstacles, and Table 4.6 corresponds to experiments in which a test agent strategy is synthesized from the output of the routing optimization. The problem size (e.g., automaton size, graph size) and graph construction times for all experiments are given in Table 4.4. The sizes of automata, transition systems, and product graphs are listed by the tuple $(|V|, |E|)$.

To further study the scalability of the routing optimization, I tabulate the runtimes for randomized gridworld experiments for various specification sub-tasks in Tables 4.7 and 4.8. These computations were conducted on an Apple M2 Pro with 16 GB of RAM using Gurobipy [114]. The construction of DBAs from specifications was implemented using Spot [98]. In the randomized experiments, the Gurobi solver for the MILP has a timeout condition set at 10 minutes to find at least a feasible solution. Once the solver finds a feasible solution, it is given another minute to return a solution with the optimality guarantee. If the solver cannot guarantee opti-

Table 4.4: Automata and graph construction runtimes for simulated and hardware experiments

Experiment	$\lvert\mathcal{B}_\pi\rvert$	$\lvert T\rvert$	$\lvert G\rvert$	G[s]
Example 4.1	(4, 9)	(15, 53)	(27, 96)	0.0270
Refueling	(6, 18)	(265, 1047)	(332, 1346)	0.6655
Mars Exploration	(36, 354)	(376, 1522)	(4073, 17251)	75.8313
Example 4.2	(8, 27)	(6, 17)	(20, 56)	0.0452
Beaver Rescue	(12, 54)	(7, 19)	(15, 39)	0.0470
Motion Primitives	(16, 81)	(15, 42)	(72, 207)	0.4286
Maze 1	(16, 81)	(26, 80)	(196, 604)	1.6226
Patrolling	(6, 18)	(386, 1539)	(210, 831)	0.4573
Maze 2	(8, 27)	(21, 66)	(80, 252)	0.2195

Table 4.5: Routing optimization runtimes for simulated and hardware experiments with static and/or reactive obstacles

Experiment	|BinVars|	|ContVars|	|Constraints|	Opt[s]	Flow	|cuts|
Solving **MILP-STATIC**						
Example 4.1	73	87	540	0.0003	3.0	14
Refueling	1014	1261	19819	0.8682	2.0	199
Mars Exploration	13178	16604	1646480	46.6209	2.0	1641
Solving **MILP-REACTIVE**						
Example 4.2	25	115	409	0.0003	2.0	4
Beaver Rescue	8	154	441	0.0001	2	2
Motion Primitives	106	761	2606	0.0005	3.0	15

Table 4.6: Runtimes for simulated and hardware experiments with dynamic agents

Experiment	BinVars	Opt[s]	Controller[s]	$\lvert \mathtt{C}_{\text{ex}}\rvert$	Flow	|cuts|
Solving **MILP-AGENT**						
Maze 1	355	0.0010	100.0	4	1.0	3
Patrolling	621	6.0535	16.1191	0	1.0	13
Maze 2	176	0.0292	7.151	8	2.0	8

Table 4.7: Run times (with mean and standard deviation) for random grid world experiments solving **MILP-REACTIVE**

Experiment		5 × 5		10 × 10		15 × 15		20 × 20	
$\vert AP\vert$	$\vert\mathcal{B}_\pi\vert$	Optimization[s], Success Rate (%)							
Reachability:									
2	(4, 9)	5.63 ± 13.43	100	64.62 ± 38.75	100	67.38 ± 25.47	100	68.63 ± 31.12	100
3	(8, 27)	23.36 ± 38.15	100	61.68 ± 35.12	100	91.54 ± 31.41	100	117.82 ± 34.89	100
4	(16, 81)	22.49 ± 36.33	100	83.52 ± 29.25	100	171.49 ± 50.72	100	317.62 ± 89.08	100
Reachability & Reaction:									
3	(6, 21)	5.97 ± 13.21	100	61.06 ± 34.67	100	71.64 ± 41.03	100	85.20 ± 19.49	100
5	(20, 155)	17.19 ± 25.51	100	78.44 ± 34.71	100	159.91 ± 76.63	100	279.86 ± 148.23	90
7	(68, 1065)	52.71 ± 41.23	100	331.32 ± 187.28	90	585.21 ± 67.58	15	$\mathbf{600.00 \pm 0.00}$	**0**
Reachability & Safety:									
3	(6, 18)	0.76 ± 1.52	100	70.82 ± 89.70	100	63.68 ± 27.54	100	80.58 ± 20.79	100
4	(6, 18)	0.15 ± 0.29	100	71.47 ± 80.61	100	59.59 ± 38.92	100	76.02 ± 27.11	100
5	(6, 18)	0.12 ± 0.18	100	94.68 ± 88.04	100	71.34 ± 30.89	100	82.54 ± 22.69	100

Table 4.8: Run times (with mean and standard deviation) for random grid world experiments solving **MILP-STATIC**.

Experiment		5 × 5		10 × 10		15 × 15		20 × 20	
$\vert AP\vert$	$\vert\mathcal{B}_\pi\vert$	Optimization [s], Success Rate (%)							
Reachability:									
2	(4, 9)	8.17 ± 13.14	100	54.07 ± 17.98	100	60.17 ± 0.12	100	60.17 ± 0.10	100
3	(8, 27)	27.78 ± 21.71	100	60.17 ± 0.10	100	60.48 ± 0.86	100	74.02 ± 38.70	100
4	(16, 81)	52.60 ± 14.05	100	60.42 ± 0.34	100	82.02 ± 41.26	100	265.41 ± 203.51	80
Reachability & Reaction:									
3	(6, 21)	10.62 ± 14.85	100	60.09 ± 0.06	100	60.23 ± 0.24	100	60.34 ± 0.46	100
5	(20, 155)	20.41 ± 19.21	100	67.77 ± 31.90	100	95.31 ± 116.65	95	268.50 ± 222.14	75
7	(68, 1065)	36.64 ± 23.34	100	110.63 ± 92.81	100	419.77 ± 214.30	55	$\mathbf{556.38 \pm 131.06}$	**10**
Reachability & Safety:									
3	(6, 18)	1.27 ± 1.47	100	60.08 ± 0.06	100	57.27 ± 12.61	100	60.32 ± 0.24	100
4	(6, 18)	0.17 ± 0.23	100	60.06 ± 0.05	100	60.14 ± 0.10	100	60.30 ± 0.19	100
5	(6, 18)	0.11 ± 0.16	100	54.15 ± 17.80	100	60.17 ± 0.09	100	60.29 ± 0.26	100

mality in that time frame, the feasible solution is returned. If the optimizer returns at least a feasible solution, the run is counted as a success. An empirical observation is that Gurobi often finds an optimal solution but takes even longer to produce an optimality guarantee. For the randomized experiments, gridworlds from size 5×5 to 20×20 are considered, and for each gridsize, problem instances are randomly generated. In the allotted time, if the optimization returns that a problem instance is infeasible, then the instance is discarded and a new one is generated in its place.

Tables 4.7 and 4.8 tabulate the optimization runtimes and success rate for solving **MILP-REACTIVE** and **MILP-STATIC**, respectively. The optimization runtime lists the mean and standard deviation for 20 instances. The success rate indicates

Table 4.9: Graph construction runtimes (with mean and standard deviation) for random grid world experiments

Experiment		5×5	10×10	15×15	20×20
$\lvert AP\rvert$	$\lvert\mathcal{B}_\pi\rvert$	Graph Construction [s]			
Reachability:					
2	(4, 9)	0.046 ± 0.001	0.224 ± 0.0056	0.554 ± 0.009	1.078 ± 0.011
3	(8, 27)	0.344 ± 0.007	1.661 ± 0.022	4.004 ± 0.048	7.376 ± 0.061
4	(16, 81)	1.997 ± 0.077	9.895 ± 0.109	23.512 ± 0.179	43.188 ± 0.454
Reachability & Reaction:					
3	(6, 21)	0.090 ± 0.001	0.424 ± 0.016	1.037 ± 0.004	2.044 ± 0.013
5	(20, 155)	1.628 ± 0.087	7.560 ± 0.023	18.019 ± 0.129	33.539 ± 0.144
7	(68, 1065)	44.809 ± 0.996	209.612 ± 1.732	488.611 ± 6.308	869.060 ± 16.870
Reachability & Safety:					
3	(6, 18)	0.102 ± 0.002	0.508 ± 0.010	1.278 ± 0.022	2.557 ± 0.023
4	(6, 18)	0.116 ± 0.002	0.590 ± 0.009	1.485 ± 0.024	2.918 ± 0.046
5	(6,18)	0.179 ± 0.027	0.960 ± 0.037	2.329 ± 0.072	4.482 ± 0.116

the percentage of instances in which at least one feasible solution was returned within the allotted time. In addition to the gridsize, the specification length was also scaled for three classes of system and test objectives: i) reachability, ii) reachability and reaction, and iii) reachability and safety. In the first case with reachability objectives, the system and test specification are $\varphi_{\text{sys}} = \Diamond p_0$ and $\varphi_{\text{test}} = \bigwedge_{i=1}^{n} \Diamond p_i$, and the total number of atomic propositions are $|AP|= |\{p_0, \dots, p_n\}|= n + 1$, scaled upto $n = 3$ (or $|AP|= 4$). For the reachability and reaction objectives, the system objective comprises of a reachability objective and a conjunction of delayed reaction specification pattern: $\varphi_{\text{sys}} = \Diamond p_0 \wedge \bigwedge_{i=1}^{n} \Box(p_i \rightarrow \Diamond q_i)$. The test objective for this case is a conjunction of the triggers corresponding to the system objective: $\varphi_{\text{test}} = \bigwedge_{i=1}^{n} \Diamond p_i$. Therefore, the total number of atomic propositions are $|AP|= |\{p_0, \dots, p_n, q_1, \dots, q_n\}|= 2n + 1$, scaled upto $n = 3$. Finally, in the reaction and safety case, the system objective consists of reachability and safety specifications: $\varphi_{\text{sys}} = \Diamond p_1 \wedge \bigwedge_{i=2}^{n} \Box\neg p_i$ and the test objective is a single reachability specification $\varphi_{\text{test}} = \Diamond p_0$. The total number of atomic propositions are $|AP|= |\{p_0, \dots, p_n\}|= n + 1$, scaled upto $n = 3$. Note that only the length of the system objective changes as the specification size is increased.

Table 4.9 lists the sizes of the specifications as well as runtimes for graph construction (mean and standard deviation across 20 instances). The product graph construction is a basic implementation in Python, and is not optimized for speed. In future work, off-the-shelf symbolic methods can be leveraged to compute the product graphs much more quickly.

4.12 Conclusions and Future Work

This work on flow-based synthesis of test strategies can help test engineers automatically synthesize test environments (e.g., where should obstacles be placed, how should the test agent strategy be implemented) that are guaranteed to meet specified system and test objectives. This chapter simplifies the routing optimization introduced in the previous chapter, and presents MILP formulations for the different types of test environments. Furthermore, this chapter shows how a reactive test strategy can inform the choice of a test agent and also find an agent strategy that implements the reactive test strategy. This is made possible by via GR(1) synthesis and a counter-example guided approach to resolving the MILP to exclude dynamically infeasible test strategies. Another important contribution of this chapter is in establishing the computational complexity of the routing problem, which means that using an MILP for the routing problem is an appropriate choice. Despite the combinatorial nature of the problem, extensive experiments show that it can handle medium-sized problem instances (thousands of integer variables) in a reasonable time. The synthesized test strategies are reactive to system behavior, and route it through the test objective, and if the system demonstrates unsafe behavior, it is a fault in the system design. When the routing problem is solved to optimality, the resulting test strategy is not overly-restrictive, and the is realized with the fewest number of obstacles.

There are several exciting directions for future work. First, this framework can be extended to automatically select from a library of test agents to optimize for testing cost. Secondly, the use of symbolic methods in graph construction to improve the overall runtime of the framework. Thirdly, finding good convex relaxations to the MILP would result in dramatic speed-up since we would only have to solve a linear program. However, a straight-forward convex relaxation on the binary variables does not return meaningful solutions; finding an often-tight convex solution would require more careful study. Fourth, integrating the high-level test synthesis in this work with dynamics from lower levels of the control hierarchy is an important open problem. This effort would include: i) interfacing with falsification tools to automatically synthesis difficult tests, and ii) incorporate timing constraints into system and test objectives. Finally, we must relate the synthesized tests to a notion of coverage, and choose system and test objectives that that maximize the coverage metric. A more comprehensive discussion on future directions is given in Chapter 6.

Chapter 5

ASSUME-GUARANTEE CONTRACTS FOR COMPOSITIONAL TESTING

5.1 Introduction

The previous chapters discuss the synthesis of test strategies from system and test objectives. In this chapter, we will motivate the idea of compositional test plans. It might be desireable to construct a more complex test objective from simpler unit tests, or to break-down a complex test into simpler tests, either by testing on smaller sub-systems or conducting simpler tests. This chapter is a step in the direction towards composable test plans. First, we will introduce a mathematical and algorithmic framework in which simpler test objectives can be merged to form a more complex test objective. Then, we will introduce mathematical frameworks based in assume-guarantee contract operations to formally describe test campaigns, and how tests can be merged or decomposed. For simplicity, we assume that the test environment has already equipped to handle the test objectives.

This work is adapted from:

J.B Graebener*, A. Badithela*, R. M. Murray. (2022). "Towards Better Test Coverage: Merging Unit Tests for Autonomous Systems." In: *2022 NASA Formal Methods (NFM)*, pp. 133–155. DOI: 10.1007/978-3-031-06773-0_7.

A. Badithela*, J.B Graebener*, I. Incer* , R. M. Murray. (2023). "Reasoning over Test Specifications Using Assume-Guarantee Contracts." In: *2023 NASA Formal Methods (NFM)*, pp. 278–294. DOI: 10.1007/978-3-031-33170-1_17.

The contract-based-design framework was first introduced as a design methodology for modular software systems [115–117] and later extended to complex cyber-physical systems [118–120]. Following the definition of assume-guarantee contracts earlier in the chapter, we will now cover background on other contract operators [121]. For ease of reading, we will repeat the definition of assume-guarantee contracts below, and introduce other operators.

Definition 5.1 (Assume-Guarantee Contract)**.** Let $\mathcal{B}$ be a universe of behaviors, then a *component* M is a set of behaviors $M \subseteq \mathcal{B}$. A *contract* is the pair $\mathcal{C} = (A, G)$, where A are the assumptions and G are the guarantees. A component E is an *environment* of the contract $\mathcal{C}$ if $E \models A$. A component M is an *implementation* of the contract, $M \models \mathcal{C}$ if $M \subseteq G \cup \neg A$, meaning the component provides the specified guarantees if it operates in an environment that satisfies its assumptions. There exists a partial order of contracts, we say $\mathcal{C}_1$ is a refinement of $\mathcal{C}_2$, denoted $\mathcal{C}_1 \leq \mathcal{C}_2$, if $(A_2 \leq A_1)$ and $(G_1 \cup \neg A_1 \leq G_2 \cup \neg A_2)$. We say a contract $\mathcal{C} = (A, G)$ is in canonical, or saturated, form if $\neg A \subseteq G$.

$$\begin{tikzcd}
\mathcal{C} & & \mathcal{C}/\mathcal{C}' \\
 & \text{iff} & \\
\mathcal{C}' \parallel \mathcal{C}_1 \arrow[uu] & & \mathcal{C}_1 \arrow[uu]
\end{tikzcd}$$

(a) Composition and quotient.

$$\begin{tikzcd}
 & \mathcal{C}_1 \bullet \mathcal{C}_2 & \\
\mathcal{C}_1 & \mathcal{C}_1 \parallel \mathcal{C}_2 \arrow[u] & \mathcal{C}_2 \\
 & \mathcal{C}_1 \wedge \mathcal{C}_2 \arrow[u] \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

(b) Order of operations.

Figure 5.1: Contract operators and the partial order, defined in relation to the refinement operator, of their resulting objects.

Assume the following contracts are in canonical form. The meet or conjunction of two contracts exists [118] and is given by $\mathcal{C}_1 \wedge \mathcal{C}_2 = (A_1 \cup A_2, G_1 \cap G_2)$. Composition [122] yields the specification of a system given the specifications of the components: $\mathcal{C}_1 \parallel \mathcal{C}_2 = ((A_1 \cap A_2) \cup \neg(G_1 \cap G_2), G_1 \cap G_2)$. Given specifications $\mathcal{C}$ and $\mathcal{C}_1$, the quotient is the largest specification $\mathcal{C}_2$ such that $\mathcal{C}_1 \parallel \mathcal{C}_2 \leq \mathcal{C}$ [123]: $\mathcal{C}/\mathcal{C}_1 = (A \cup G_1, (G \cap A_1) \cup \neg(A \cup G_1))$. Strong merger [124] yields a specification that is satisfied by a system that also satisfies the two given specifications $\mathcal{C}_1$ and $\mathcal{C}_2$: $\mathcal{C}_1 \bullet \mathcal{C}_2 = (A_1 \cap A_2, (G_1 \cap G_2) \cup \neg(A_1 \cap A_2))$. The reciprocal (or mirror) [124, 125] is a unary operation which inverts assumptions and guarantees: $\mathcal{C}^{-1} = (G, A)$.

Remark 5.1. The statement contract $\mathcal{C}_1$ is more refined than contract $\mathcal{C}_2$ should be interpreted as follows. A system implementation for $\mathcal{C}_1$ has fewer assumptions on its environment and must provide more guarantees. In looking at sets of behaviors: the system must handle a larger set of environment behaviors while providing stricter guarantees. Since the guarantees of $\mathcal{C}_1$ are stricter than $\mathcal{C}_2$, the set of system behaviors satisfying guarantees of C_1 is smaller than that of C_2.

5.2 Preliminary Work on Merging Unit Tests

Recall the definition of a discrete-state system introduced in Chapter 4. In the previous chapter, the system specification could not be described as a GR(1) specification since the property that there will always exist a path to satisfying the system goal could not be characterized as a GR(1) formula. This is because the precise environment agents for a scenario were not given to the system; the system is only informed that actions in the test harness can be restricted, but these restrictions will always be such that the system has a feasible path. In this chapter, we assume that such a test environment has been constructed according to the test synthesis framework established in the previous chapter, and it is such that there always exists a path for the system goal. For simplicity, we consider a class of system objectives which can be written in the assume-guarantee form. The assumptions specify the dynamics of the test environment agents that will be used in the test scenario, and tests considered are such that even the worst-case dynamics of the test agents will not prevent the system from satisfying its requirements. Since this system objectives is a subset of the system objective considered in the previous chapter, we will used the term system specification instead. Let T_{sys} be the system transition system.

Definition 5.2 (System Specification)**.** A *system specification* φ_{sys} is the $GR(1)$ formula,

$$\varphi_{\text{sys}} = (\varphi_{\text{test}}^{\text{init}} \wedge \square\varphi_{\text{test}}^{s} \wedge \square \diamondsuit \varphi_{\text{test}}^{f}) \rightarrow (\varphi_{\text{sys}}^{\text{init}} \wedge \square\varphi_{\text{sys}}^{s} \wedge \square \diamondsuit \varphi_{\text{sys}}^{f}) \,, \tag{5.1}$$

where $\varphi_{\text{sys}}^{\text{init}}$ is the initial condition that the system needs to satisfy, φ_{sys}^{s} encode system dynamics and safety requirements on the system, and φ_{sys}^{f} specifies recurrence goals for the system which is defined to be on a sink state of the system. Likewise, $\varphi_{\text{test}}^{\text{init}}$, $\varphi_{\text{test}}^{s}$, and $\varphi_{\text{test}}^{f}$ represent assumptions the system has on the test environment.

Once again, the objective is to synthesize a test strategy for the test environment given the test specification. Unlike the system specification, the infinitely often sub-task specification need not be restricted to be satisfied in a terminal state.

Definition 5.3 (Test Specification)**.** A *test specification* φ_{test} is the $GR(1)$ formula,

$$\varphi_{\text{test}} := (\varphi_{\text{sys}}^{\text{init}} \wedge \square\varphi_{\text{sys}}^{s} \wedge \square \diamondsuit \varphi_{\text{sys}}^{f}) \rightarrow (\varphi_{\text{test}}^{\text{init}} \wedge \square\varphi_{\text{test}}^{s} \wedge \square \diamondsuit \varphi_{\text{test}}^{f} \wedge \square\psi_{\text{test}}^{s} \wedge \square \diamondsuit \psi_{\text{test}}^{f}) \,, \tag{5.2}$$

where $\varphi_{\text{sys}}^{\text{init}}$, φ_{sys}^{s} and φ_{sys}^{f}, $\varphi_{\text{test}}^{\text{init}}$, $\varphi_{\text{test}}^{s}$ and $\varphi_{\text{test}}^{f}$ are propositional formulas from equation (5.1). Additionally, $\square\psi_{\text{test}}^{s}$ and $\square \diamondsuit \psi_{\text{test}}^{f}$ describe the safety and recurrence formulas for the test environment in addition to the dynamics of the test environment known to the system. Note that the system is unaware of these additional sub-task specifications (similarly to the previous chapters), and the test environment is such that the system is allowed to satisfy its requirements. Defining the test specification in this manner allows for i) synthesizing a test in which the system, if properly designed, can meet φ_{sys}, and ii) specifying additional requirements on the test environment, unknown to the system at design time. We assume that test specifications are defined *a priori*; we leave automatically finding relevant test specifications to future work.

Having defined the system and test specifications, we define a product transition system that represents the turn-based dynamics of the two players: Let T_{prod} be a turn-based product transition system constructed from T_{sys} and T_{test}, where $T_{\text{prod}}.S := T_{\text{sys}}.S \times T_{\text{test}}.S$, $T_{\text{prod}}.A := T_{\text{sys}}.A \times T_{\text{test}}.A$, and $T_{\text{prod}}.\delta \subseteq T_{\text{prod}}.S \times T_{\text{prod}}.A \times T_{\text{prod}}.S$ denotes the turn-based transition function. In particular, for every transition $(s, a_s, s') \in T_{\text{sys}}.\delta$, we have $((s,t),(a_s,a_t),(s',t)) \in T_{\text{prod}}.\delta$ where $t \in T_{\text{test}}.S$ and $a_t \in T_{\text{test}}.A$. The transitions originating due to test agent actions are constructed similarly. From the product transition system, we can construct a game graph that maintains two copies of each state — one from which the system player acts and the other from which the test environment acts.

Definition 5.4 (Game Graph)**.** Let V_{sys} and V_{test} be copies of the states $T_{\text{prod}}.S$. Let E_{sys} and E_{test} correspond to the transitions in the game graph:

$$\begin{aligned} E_{\text{sys}} =& \{((s,t),(s',t)) \mid \exists a_s \in T_{\text{sys}}.A,\ \forall a_t \in T_{\text{test}}.A,\ ((s,t),(a_s,a_t),(s',t)) \in T_{\text{prod}}.\delta\}, \\ E_{\text{test}} =& \{((s,t),(s,t')) \mid \exists a_t \in T_{\text{test}}.A,\ \forall a_s \in T_{\text{sys}}.A,\ ((s,t),(a_s,a_t),(s,t')) \in T_{\text{prod}}.\delta\}. \end{aligned} \tag{5.3}$$

Then, the *game graph* is a directed graph $G = (V, E)$ is a directed graph with vertices $V := V_{\text{sys}} \cup V_{\text{test}}$ and edges $E := E_{\text{sys}} \cup E_{\text{test}}$.

On the game graph, a player strategy, and the test execution resulting from it are given below.

Definition 5.5 (Strategy)**.** On the game graph G, a policy for the system is a function $\pi_{\text{sys}} : V^* V_{\text{sys}} \to V_{\text{test}}$ such that $(s, \pi_{\text{sys}}(w.s)) \in E_{\text{sys}}$, where $s \in V_{\text{sys}}$ and $w \in V^*$. Similarly defined, π_{test} denotes the test environment policy, where * is the Kleene star operator.

Definition 5.6 (Test Execution)**.** A *test execution* $\sigma = v_0 v_1 v_2 \ldots$ starting from vertex $v_0 \in V$ is an infinite sequence of states on the game graph G. Since G is a turn-based game graph, the states in the test execution alternate between V_{sys} and

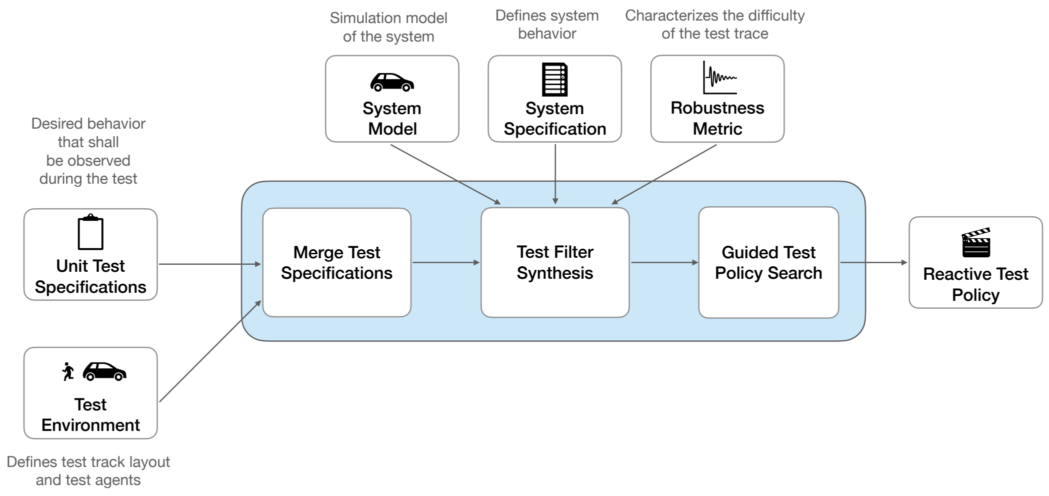

Simulation model of the system
System Model
Defines system behavior
System Specification
Characterizes the difficulty of the test trace
Robustness Metric
Desired behavior that shall be observed during the test
Unit Test Specifications
Merge Test Specifications
Test Filter Synthesis
Guided Test Policy Search
Reactive Test Policy
Test Environment
Defines test track layout and test agents

Figure 5.2: Overview of the merging unit tests.

V_{test}, so if $V_1 \in V_{\text{sys}}$, then $v_{i+1} = \pi_{\text{sys}}(v_0 \ldots V_1)$. Let $\sigma_{s_0}(\pi_{\text{sys}} \times \pi_{\text{test}})$ be the test execution starting from state $s_0 \in V_{\text{sys}}$ for policies π_{sys} and π_{test}. Let Σ denote the set of all possible test executions on G. A robustness metric $\rho : \Sigma \to \mathbb{R}$ is a function evaluated assigning a scalar value to a test execution.

Problem 5.1. Given system and environment transition systems, $\mathcal{T}_{\text{sys}}$ and $\mathcal{T}_{\text{test}}$, two unit test objectives $\varphi_{\text{test},1}$ and $\varphi_{\text{test},2}$, and a robustness metric ρ, find a test strategy π^*_{test}, such that

$$\begin{aligned} \pi^*_{\text{test}} \quad = \quad \arg\max_{\pi_{\text{test}}} \quad & \rho(\sigma(\pi_{\text{sys}} \times \pi_{\text{test}})) \\ \text{s.t.} \quad & \sigma(\pi_{\text{sys}} \times \pi_{\text{test}}) \models (\varphi_{\text{test},1} \wedge \varphi_{\text{test},2}) \,, \quad \forall\, \pi_{\text{sys}} \models \varphi_{\text{sys}}. \end{aligned} \tag{5.4}$$

Example 5.1 (Running Example — Lane Change)**.** Consider the lane change scenario illustrated in Figure 5.3. The system (red car) is required to change lanes into the lower lane before the track ends without colliding with the test environment agents (blue cars). The system liveness requirement is, $\varphi^f_{\text{sys}} := (y_{\text{sys}} = 2)$, and its safety requirement of no collisions is with test agent labeled i, is: $\neg(y_{\text{sys}} = y_{\text{test},i} \wedge x_{\text{sys}} = x_{\text{test},i}) \in \varphi^s_{\text{sys}}$. In the first two panels, we observe the test agent changing lanes in front of and behind a test car, respectively. In the merged test execution of the third panel, we see the test agent change lanes exactly in between the two blue cars.

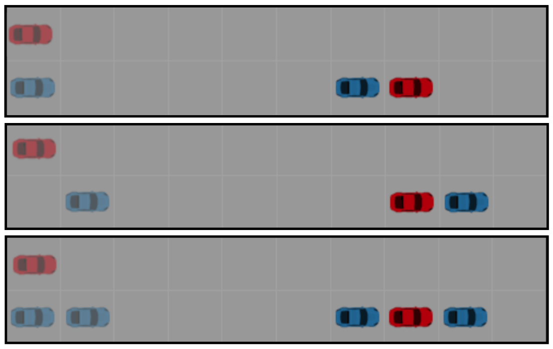


Figure 5.3: Lane change example with initial (left) and final (right) configurations. The x-coordinates are numbered from left to right, and y-coordinates are numbered top to bottom, starting from 1. The system (red) is required to merge into the lower lane without colliding. Merging in front of (top), behind (center), or in between (bottom) tester agents (blue).

5.3 Strong Merge Operator

In this section, we formalize the the construction of a single test specification from unit test specifications using the *strong merge* operator from contract theory. Additionally, we will introduce the notion of adding temporal constraints to the merged test specification to ensure that the resulting test execution reliably satisfies all the unit test specifications. Finally, for the merged test specifications, we use Monte-Carlo Tree Search to find a test strategy on the game graph such that a metric of difficulty is maximized.

The *strong merge operator* defines the merge of two contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ as follows:

$$\begin{aligned}\mathcal{C}_1 \bullet \mathcal{C}_2 =&(a_1 \wedge a_2, (a_1 \wedge a_2) \rightarrow [(a_1 \rightarrow g_1) \wedge (a_2 \rightarrow g_2)]) \\ =&(a_1 \wedge a_2, \neg a_1 \vee \neg a_2 \vee (g_1 \wedge g_2)).\end{aligned} \tag{5.5}$$

Additionally, other operators from assume-guarantee contract theory such as *composition* and *conjunction* [122, 124] will be introduced later in the chapter. Among all these operators, strong merge is the only operator that conjoins assumptions of the individual contracts, and consequently, enforces all unit test specifications to hold true. Thus, we choose the strong merge operator to derive the merged test specification.

Given any two unit test specifications, $\varphi_{\text{test},1}$ and $\varphi_{\text{test},2}$, the corresponding contracts are $\mathcal{C}_1 = (a_1, a_1 \rightarrow g_1)$ and $\mathcal{C}_2 = (a_2, a_2 \rightarrow g_2)$, where $a_i = (\varphi^{\text{init}}_{\text{sys}} \wedge \Box\varphi^{s}_{\text{sys}} \wedge \Box\Diamond\varphi^{f}_{\text{sys}})$ is the assumptions on the system (under test), and $g_i = (\varphi^{\text{init}}_{\text{test},i} \wedge \Box\varphi^{s}_{\text{test},i} \wedge \Box\Diamond\varphi^{f}_{\text{test},i} \wedge \Box\psi^{s}_{\text{test},i} \wedge \Box\Diamond\psi^{f}_{\text{test},i})$ is the guarantees for unit test i. We use the term $g_{t,i} := \psi^{f}_{\text{test},i})$ to refer to the liveness portion of the test objective unknown to the system under test.

Remark 5.2. We make a few simplifying assumptions on the unit test guarantees g_i. First, we assume that the only recurrence requirements in the test specification is $\Box \Diamond \psi^f_{\text{test},i}$, which is not known to the system since it is not a part of the system's assumptions on the environment. Second, we assume that the merged test environment $T_{\text{test},m}$ is a simple Cartesian product of the unit test environments, $T_{\text{test},1}$ and $T_{\text{test},2}$. On the merged test environment, we take the agents from the individual tests: we translate the initial conditions of the agents in the unit tests $\varphi^{\text{init}}_{\text{test},1}$ and $\varphi^{\text{init}}_{\text{test},2}$, and test agent dynamics $\varphi^s_{\text{test},1}$ and $\varphi^s_{\text{test},2}$ are also the same.

Definition 5.7 (Merged Test Specification)**.** From the merged contract $\mathcal{C}_m := \mathcal{C}_1 \bullet \mathcal{C}_2 = (a_m, a_m \to g_m)$, the specification $\varphi_{\text{test},m} = a_m \to g_m$, where $a_m = a_1 \wedge a_2$, and $g_m = [(a_1 \to g_1) \wedge (a_2 \to g_2)]$ is the *merged test objective*. A test environment strategy $\pi_{\text{test},m}$ for merged test objective $\varphi_{\text{test},m}$ results in a test execution $\sigma \models \varphi_{\text{test},m}$.

The following result is taken from [126].

Lemma 5.1. Given unit test specifications $\varphi_{\text{test},1}$ and $\varphi_{\text{test},2}$ such that $\varphi_{\text{test},m} = a_m \to g_m$ is the corresponding merged test specification. Then, for every test execution $\sigma \models \varphi_{\text{test},m}$ such that $\sigma \models a_m$, we also have that $\sigma \models \varphi_{\text{test},1}$ and $\sigma \models \varphi_{\text{test},2}$.

Proof. Suppose C_1 and C_2 are the assume-guarantee contracts corresponding to unit test specifications $\varphi_{\text{test},1}$ and $\varphi_{\text{test},2}$. Applying strong merge operator on contracts C_1 and C_2, we get:

$$\begin{aligned}\mathcal{C}_1 \bullet \mathcal{C}_2 =& (a_1 \wedge a_2, (a_1 \wedge a_2) \to [(a_1 \to g_1) \wedge (a_2 \to g_2)]) \\ =& (a_1 \wedge a_2, \neg a_1 \vee \neg a_2 \vee (g_1 \wedge g_2)).\end{aligned} \tag{5.6}$$

Thus, the merged test specification $\varphi_{\text{test},m} = \neg a_1 \vee \neg a_2 \vee (g_1 \wedge g_2)$ requires either one of the assumptions to not be satisfied, or for both the guarantees hold. Since $\sigma \models a_m = a_1 \wedge a_2$, and $\sigma \models \varphi_{\text{test},m}$, we get that $\sigma \models \varphi_{\text{test},1}$ and $\sigma \models \varphi_{\text{test},2}$. □ □

Guarantees g_1 and g_2 are used guide the choice of a test strategy; strategies that vacuously satisfy the merged test specification by violating the assumptions are not returned. This is necessary in order to give the system an opportunity to satisfy its specification. If the assumptions on the merged test specifications are violated, it would be because of a fault in system design.

Example 5.2 (Lane Change (continued))**.** In the lane change example, the unit test specifications are changing into the lane *behind* a blue car and changing into the lane *in front* of the blue car. For each specification, the saturated assume guarantee contracts are defined as $\mathcal{C}_1 = (a_1, a_1 \rightarrow g_1)$ and $\mathcal{C}_2 = (a_2, a_2 \rightarrow g_2)$ with $a_1 = \varphi_{\text{sys}}^{\text{init}} \wedge \Box\varphi_{\text{sys}}^{s} \wedge \Box\Diamond(y = 2)$ and $g_1 = \Box\Diamond(y = y_1 = 2 \wedge x = x_1 + 1)$, and $a_2 = \varphi_{\text{sys}}^{\text{init}} \wedge \Box\varphi_{\text{sys}}^{s} \wedge \Box\Diamond(y = 2)$ and $g_2 = \Box\Diamond(y = y_2 = 2 \wedge x = x_2 - 1)$ being the assumptions and guarantees of the two individual tests. Thus, applying the strong merge operation to the unit contracts results in the guarantee,

$$g_m = \Box\Diamond(y = y_1 = 2 \wedge x = x_1 + 1) \wedge \Box\Diamond(y = y_2 = 2 \wedge x = x_2 - 1). \quad (5.7)$$

5.4 Temporal Constraints on Merging Tests

Naively merging test objectives might not always result in a merged test execution that checks the constituent unit test objectives. In the running example on lane change, lane change maneuver behind a vehicle in the other lane does not always coincide with a proper lane change in front of another vehicle. That is, there may exist many test executions of changing lanes behind a vehicle, and some of them, but not all, coincide with changing lanes in front of another vehicle. In these scenarios, the test specifications can be merged *in parallel*, without any additional temporal constraints on how agents for each test environment must operate.

However, when all executions resulting from a one of the unit test specification also satisfy the other (as we will see in the unprotected left turn example), the merged test specification alone is not sufficient. We need to add temporal constraints so that there is a time in which each test specification is checked individually.

The following result is taken from [126].

Lemma 5.2. If for two test specifications $\varphi_{\text{test},1}$ and $\varphi_{\text{test},2}$, and the set of all test executions Σ, we have $\sigma \models \varphi_{\text{test},1} \iff \sigma \models \varphi_{\text{test},2} \forall \sigma \in \Sigma$, then these tests cannot be parallel-merged. Instead, the temporal constraint must be enforced on $g_{t,1}$ and $g_{t,2}$.

Proof. We refine the general specification in equation (5.6), which allows any temporal structure, to include the temporal constraints in the guarantees. The temporally constrained merged test specification is thus defined as $\varphi'_{\text{test},m} = a_m \rightarrow g'_m$, with

$$g'_m = \neg a_1 \vee \neg a_2 \vee (\Diamond(g_{t,1} \wedge \neg g_{t,2}) \wedge \Diamond(\neg g_{t,1} \wedge g_{t,2}) \wedge (g_1 \wedge g_2)). \quad (5.8)$$

Because any trace σ satisfying $\varphi'_{\text{test},m}$ will also satisfy $\varphi_{\text{test},m}$, $\sigma \models \varphi'_{\text{test},m} \Rightarrow \sigma \models \varphi_{\text{test},m}$. Any test trace satisfying this specification will consist of at least one occurrence of visiting a state satisfying $g_{t,1}$ and not $g_{t,2}$ and vice versa. Thus the guarantees of the specifications for each unit test, $g_{t,1}$ and $g_{t,2}$ are checked individually during the merged test which satisfies the temporal constraints. $\square$

Receding Horizon Synthesis of Test strategy Filter

Since the test specification characterizes the set of possible test executions, we need a strategy for the test environment that is consistent with the test specification. In this section, we detail the construction of an auxiliary game graph and algorithms for receding horizon synthesis of the test specification on the auxiliary game graph. This filter will then be used to find the test strategy using Monte-Carlo Tree Search.

Auxiliary Game Graph G_{aux}

Assume we are given a game graph $G = (V, E)$ constructed according to Definition 5.4, and a (merged) test specification $\varphi_{\text{test},m}$ in $GR(1)$ form as in equation (5.2). Then, for each recurrence requirement in the test specification, $\square \diamondsuit \psi^f_{\text{test}}$, we can find a set of states $\mathcal{I} = \{i_1, \dots, i_n\} \subseteq V$ that satisfy the propositional formula ψ^f_{test}. For each $i \in \mathcal{I}$, there exists a non-empty subset of vertices $V^s \subseteq V$ that can be partitioned into $\{\mathcal{V}^i_0, \dots, \mathcal{V}^i_n\}$. We follow [18] in partitioning the states; $\mathcal{V}^i_k$ is the set of states in V that is exactly k steps away from the goal state i. From this partition of states, we can construct a partial order, $\mathcal{P}^i = (\{\mathcal{V}^i_0, \dots, \mathcal{V}^i_n\}, \leq)$, such that $\mathcal{V}^i_l \leq \mathcal{V}^i_{l-1}$ for all $l \in \{0, \dots, n\}$. This partial order will be useful in the receding horizon synthesis of the test strategy outlined below [18].

We construct an auxiliary game graph $G_{\text{aux}} = (V_{\text{aux}}, E_{\text{aux}})$ (illustrated in Figure 5.4) to accommodate any temporal constraints on the merged test specification before proceeding to synthesize a filter for the test strategy. Without loss of generality, we elaborate on the auxiliary graph construction in the case of one recurrence requirement in each unit specification, but this approach can be easily extended to multiple progress requirements. An illustration of the auxiliary graph is given in Figure 5.4. Let $\varphi_{\text{test},1}$ and $\varphi_{\text{test},2}$ be the two unit test specifications, with $\psi^f_{\text{test},1}$ and $\varphi^f_{\text{test},2}$, respectively. First, we make three copies of the game graph $G = (V, E)$ — $G_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}} = (V_{1\vee 2}, E_{1\vee 2})$, $G_{\varphi_{\text{test},1}} = (V_1, E_1)$, and $G_{\varphi_{\text{test},2}} = (V_2, E_2)$. Note that, $V_{1\vee 2}$, V_1 and V_2 are all copies of V, but are denoted differently for differentiating between the vertices that constitute G_{aux}, and a similar argument applies to edges of these subgraphs. Let $\mathcal{V}^i_0 = \bigcup \mathcal{V}^{i_j}_0 \subseteq V_{1\vee 2}$ be the set of states in $G_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}}$ that satisfy propositional formula $\psi^f_{\text{test},1}$. Likewise, the set of states $\mathcal{V}^k_0 \subseteq V_{1\vee 2}$ satisfy

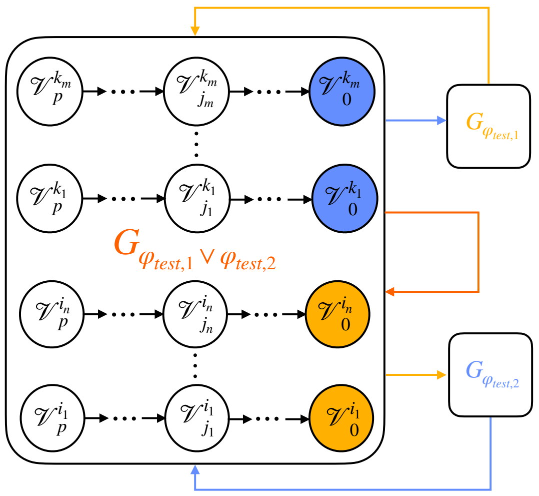


Figure 5.4: Auxiliary game graph construction for the merged test specification of unit test specifications $\varphi_{\text{test},1}$ and $\varphi_{\text{test},2}$. Subgraphs $G_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}}$, $G_{\varphi_{\text{test},1}}$ and $G_{\varphi_{\text{test},2}}$ are copies of the game graph G constructed per Definition 5.4. In $G_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}}$, the sets of states at which the progress propositional formulas of test specifications, $\varphi_{\text{test},1}$ and $\varphi_{\text{test},2}$, are satisfied are shaded yellow and blue, respectively.

the propositional formula $\psi^f_{\text{test},2}$.

Now, we connect the various subgraphs through the vertices in $\mathcal{V}^i_0$ and $\mathcal{V}^k_0$. Let (v^k_0, u) be an outgoing edge from a node $v^k_0 \in \mathcal{V}^k_0$, and let u_1 be the vertex in subgraph $G_{\text{test},1}$ that corresponds to vertex u in $G_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}}$. Remove edge (v^k_0, u) and add the edge (v^k_0, u_1). Likewise, every outgoing edge from $\mathcal{V}^i_0 \cup \mathcal{V}^k_0$ in $G_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}}$ is replaced by adding edges to $G_{\varphi_{\text{test},1}}$ and $G_{\varphi_{\text{test},2}}$. On subgraphs $G_{\varphi_{\text{test},1}}$ and $G_{\varphi_{\text{test},2}}$, vertices are partitioned and partial orders are constructed once again for $\psi^f_{\text{test},1}$ and $\psi^f_{\text{test},2}$, respectively. From $\mathcal{V}^i_0$ defined on the nodes of the graph $G_{\varphi_{\text{test},1}}$, every outgoing edge is replaced by a corresponding edge to $G_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}}$. Subgraph $G_{\varphi_{\text{test},2}}$ is connected back to $G_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}}$ in a similar manner. The construction of the auxiliary graph G_{aux} and partial order $\mathcal{P}^i$ is summarized in Algorithm 7. Our choice of constructing the auxiliary graph in this manner is amenable to constructing a simple partial order as outlined below.

Assumption 5.1. For unit test specifications $\varphi_{\text{test},1}$ and $\varphi_{\text{test},2}$ with recurrence specifications φ^p_1 and φ^p_2, respectively, such that $\varphi^p_1 = \square\lozenge\, \psi^f_{\text{test},1}$ and $\varphi^p_2 = \square\lozenge\, \psi^f_{\text{test},2}$. Suppose there exist partial orders $\mathcal{P}^i = (\{\mathcal{V}^i_n, \ldots, \mathcal{V}^i_0\}, \leq)$ and $\mathcal{P}^k = (\{\mathcal{V}^k_m, \ldots, \mathcal{V}^k_0\}, \leq)$ on G corresponding to $\psi^f_{\text{test},1}$ and $\psi^f_{\text{test},2}$, respectively. Assume that at least one of the following is true: (a) there exists an edge (u_1, v_2) where $u_1 \in \mathcal{V}^i_0$ and $v_2 \in \mathcal{V}^k_j$ for some $j \in \{1, \ldots, m\}$, (b) there exists an edge (u_2, v_1) where $u_2 \in \mathcal{V}^k_0$ and $v_1 \in \mathcal{V}^i_j$ for some $j \in \{1, \ldots, n\}$.

The following Lemma is taken from [126].

Lemma 5.3. If Assumption 5.1 holds, there exists a partial order on G_{aux} for the merged recurrence propositional formula, $\psi^f_{\text{test},m}$, where $\psi^f_{\text{test},m}$ is the propositional

Algorithm 7: Construction of Partial Order and Auxiliary Graph

```
1: procedure GAUX($(G, \psi^f_{\text{test},1}, \psi^f_{\text{test},2})$)
     Input: Game graph $G = (V, E)$, propositional formulas $\psi^f_{\text{test},1}$ and $\psi^f_{\text{test},2}$
   constituting the progress requirements of unit test specifications
     Output: Auxiliary game graph $G_{\text{aux}}$
2:
3:     $G_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}} := (V, E) \leftarrow G$ Initialize subgraph
4:     $G_{\varphi_{\text{test},1}} := (V_1, E_1) \leftarrow G$ Initialize subgraph
5:     $G_{\varphi_{\text{test},2}} := (V_2, E_2) \leftarrow G$ Initialize subgraph
6:     $[\mathcal{P}^i_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}}, \mathcal{P}^k_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}}] \leftarrow$ Partial order$(G_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}}, [\psi^f_{\text{test},1}, \psi^f_{\text{test},2}])$
7:     $\mathcal{P}^i_{\varphi_{\text{test},1}} \leftarrow$ Partial order$(G_{\varphi_{\text{test},1}}, \psi^f_{\text{test},1})$
8:     $\mathcal{P}^k_{\varphi_{\text{test},2}} \leftarrow$ Partial order$(G_{\varphi_{\text{test},2}}, \psi^f_{\text{test},2})$
9:     $E^r_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}} \subseteq E$ Deleting outgoing edges from $\mathcal{V}^i_0 \cup \mathcal{V}^k_0 \subseteq V$ within
   $G_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}}$
10:    $E^a_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}}$ Adding edges from $\mathcal{V}^i_0 \cup \mathcal{V}^k_0 \subseteq V$ to subgraphs $G_{\varphi_{\text{test},1}}$ and $G_{\varphi_{\text{test},2}}$
11:    $E^r_{\varphi_{\text{test},1}} \subseteq E_1$ Deleting outgoing edges from $\mathcal{V}^i_0 \subseteq V_1$ within $G_{\varphi_{\text{test},1}}$
12:    $E^a_{\varphi_{\text{test},1}}$ Adding edges from $\mathcal{V}^i_0 \subseteq V_1$ to subgraph $G_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}}$
13:    $E^r_{\varphi_{\text{test},2}} \subseteq E_2$ Deleting outgoing edges from $\mathcal{V}^k_0 \subseteq V_2$ within $G_{\varphi_{\text{test},2}}$
14:    $E^a_{\varphi_{\text{test},2}}$ Adding edges from $\mathcal{V}^k_0 \subseteq V_2$ to subgraph $G_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}}$
15:    $V_{\text{aux}} = V \cup V_1 \cup V_2$
16:    $E_{\text{aux}} = (E \setminus E^r_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}}) \cup (E_1 \setminus E^r_{\varphi_{\text{test},2}}) \cup (E_2 \setminus E^r_{\varphi_{\text{test},2}}) \cup E^a_{\varphi_{\text{test},2}} \cup E^a_{\varphi_{\text{test},1}} \cup$
   $E^a_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}}$
17:    $G_{\text{aux}} = (V_{\text{aux}}, E_{\text{aux}})$
18:    return $G_{\text{aux}}, \mathcal{P}^i_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}}, \mathcal{P}^k_{\varphi_{\text{test},1} \vee \varphi_{\text{test},2}}, \mathcal{P}^i_{\varphi_{\text{test},1}}, \mathcal{P}^k_{\varphi_{\text{test},2}}$
```

formula that evaluates to true at: (i) all $v \in V_{1 \vee 2}$ such that $v \models \psi^f_{\text{test},1} \wedge \psi^f_{\text{test},2}$, (ii) all $v \in V_1$ such that $v \models \psi^f_{\text{test},1}$, and (iii) all $v \in V_2$ such that $v \models \psi^f_{\text{test},2}$.

Proof. Let $\mathcal{V}^m_0 \subseteq V_{\text{aux}}$ denote the non-empty set of states at which $\psi^f_{\text{test},m}$ evaluates to true. Then, let $\mathcal{V}^m_j \subseteq V_{\text{aux}}$ be the subset of states that is at least j steps away from a vertex in $\mathcal{V}^m_0$. Then, construct the partial order $\mathcal{P}^m = (\{\mathcal{V}^m_l, \ldots, \mathcal{V}^m_0\}, \leq)$, where l is the distance of the farthest vertex connected to $\mathcal{V}^m_0$. The subset of vertices $\bigcup_j \mathcal{V}^m_j \subseteq V_{\text{aux}}$ is non-empty because $\mathcal{V}^m_0$ is non-empty. Furthermore, from Assumption 5.1, if (a) holds, there exists a $j \in \{1, \ldots, l\}$ such that $\mathcal{V}^m_j \cap \mathcal{V}^i_0$ is non-empty. Likewise, if (b) holds, there exists a $j \in \{1, \ldots, l\}$ such that $\mathcal{V}^m_j \cap \mathcal{V}^k_0$ is non-empty. Therefore, for some $j \in \{1, \ldots, l\}$ there exists a test execution σ over the game graph G_{aux} such that $\sigma \models \Box \Diamond \psi^f_{\text{test},m}$. □

Remark 5.3. If Assumption 5.1 is not true, the unit tests corresponding to test objectives $\varphi_{\text{test},1}$ and $\varphi_{\text{test},2}$ cannot be merged.

Receding Horizon Synthesis on G_{aux}

We use receeding horizon synthesis for a more scalable construction of the winning set $\mathcal{W}^{\mathcal{I}}$ — the set of states from which the test environment can still satisfy the test objective. This winning set will then serve as a safety filter during Monte Carlo Tree Search to exclude trajectories that do not satisfy the test objectives. Further details on receeding horizon temporal logic planning can be found in [18].

For a test objective $\varphi_{\text{test},1}$ with progress propositional formula $\psi^f_{\text{test},1}$, let $\mathcal{I}$ be the set of states on G_{aux} at which $\psi^f_{\text{test},1}$ evaluates to true. Suppose the product state of the system and environment is some j steps away from a state $i \in \mathcal{I}$: $v \in \mathcal{V}^i_{j+1}$. If we want the test environment to guide the execution to two steps ahead to $\mathcal{V}^i_{j-1}$, the intermediate specification for the test environment is as follows.

$$\psi^i_j = (v \in \mathcal{V}^i_{j+1} \wedge \Phi \wedge \square\varphi^s_{\text{sys}} \wedge \square\diamondsuit\,\varphi^f_{\text{sys}}) \rightarrow (\square\diamondsuit(\mu^i_{\text{visited},j-1}) \wedge \square\varphi^s_{\text{test}} \wedge \square\psi^s_{\text{test}} \wedge \square\Phi), \tag{5.9}$$

where Φ is the invariant condition that ensures that ψ^i_j is realizable, and $\mu^i_{\text{visited},j-1}$ is an auxiliary variable which becomes true (and remains true) once the product state v has reached a state $j-1$ steps away from i: $v \in \mathcal{V}^i_{j-1}$. The construction of the invariant set Φ is given in [18]. It is sufficient to for the test environment to guide the execution to at least one node $i \in \mathcal{I}$, which can be formally stated as,

$$\Psi^{\mathcal{I}}_j = \vee_{i\in\mathcal{I}}\, \psi^i_j\,. \tag{5.10}$$

The set of states of G_{aux} from which the test environment has a strategy to satisfy equation (5.10) is denoted by $\mathcal{W}^{\mathcal{I}}_j$. This set serves as a short-horizon filter to guide the test strategy from j steps away to the goal set $\mathcal{I}$.

Consider the set of shortest paths: $\{Path(v,i) | v \in V,\, i \in \mathcal{I}\}$. Let j_{max} denote the length of the longest path in this set. The overall winning set filter is the union of individual winning sets:

$$\mathcal{W}^{\mathcal{I}} = \bigcup_{j=1}^{j_{\text{max}}} \mathcal{W}^{\mathcal{I}}_j\,. \tag{5.11}$$

Construction of $\mathcal{W}^{\mathcal{I}}$, and its use as a safety filter for finding test strategies using MCTS is outlined in Algorithm 8. For the merged test objective, $\mathcal{W}^{\mathcal{I}}$ is generated on G_{aux} where $\mathcal{I}$ is the set of states corresponding to $\psi^f_{\text{test},m}$. We will need the following notation to denote the graph induced by the set $\mathcal{W}^{\mathcal{I}}$. Let $G_{\mathcal{W}^{\mathcal{I}}} = (V_{\mathcal{W}}, E_{\mathcal{W}})$ be the subgraph of G_{aux} induced by $\mathcal{W}^{\mathcal{I}}$ such that $V_{\mathcal{W}} = \mathcal{W}^{\mathcal{I}} \subseteq V_{\text{aux}}$ and $E_{\mathcal{W}} = \{(u, v) \in E_{\text{aux}} \,|\, u \in \mathcal{W}^{\mathcal{I}} \wedge v \in \mathcal{W}^{\mathcal{I}}\}$.

On $\mathcal{W}^{\mathcal{I}}$ as a test strategy filter

Inspired by work on shield synthesis [127], we use the winning set $\mathcal{W}^{\mathcal{I}}$ as a filter to guide rollouts in the Monte Carlo Tree Search sub-routine for finding the test strategy. Since $\Psi^{\mathcal{I}}_j$ is a disjunction of short-horizon $GR(1)$ specifications, it is possible that an execution always satisfies $\Psi^{\mathcal{I}}_j$ without ever satisfying the progress requirement $\square \diamondsuit \psi^f_{\text{test}}$. This happens when the test execution makes progress towards some $i \in \mathcal{I}$ but never actually reaches a goal in $\mathcal{I}$, resulting in a live lock. Further details addressing this are given in the Appendix. We *assume* that the graph is constructed such that there are no such cycles. In addition to using $W^{\mathcal{I}}$ to ensure that $\Psi^{\mathcal{I}}_j$ will always be satisfied, we enforce progress by only allowing the search procedure to take actions that will lead to a state which is closer to one of the goals $i \in \mathcal{I}$. Thus, the search procedure will ensure that for every state $v_l \in \mathcal{V}^i_j$, the control strategy for the next horizon will end in $v_{l'} \in \mathcal{V}^i_k$, such that $k \leq l$ for at least one goal $i \in \mathcal{I}$.

The following theorem and proof is taken from [126].

Theorem 5.1. Receding horizon synthesis of test filter $\mathcal{W}^{\mathcal{I}}$ is such that any test execution σ on $G_{\mathcal{W}^{\mathcal{I}}}$ starting from an initial state in $V_{\mathcal{W}} \cap V$ satisfies the test specification in equation (5.2).

Proof. For the recurrence formula of the merged test specification, $\square \diamondsuit \psi^f_{\text{test},m}$, suppose there exists a single vertex on G_{aux} that satisfies $\psi^f_{\text{test},m}$. Then, it is shown in [18] that if there exists a partial order $(\{\mathcal{V}^i_p, \ldots, \mathcal{V}^i_0\}, \leq)$ on G_{aux}, we can find a set of vertices $\mathcal{W}^i \subseteq V_{\text{aux}}$, such that every test execution σ that remains in $\mathcal{W}^i$, will satisfy the safety requirements $\square \varphi^s_{\text{test}}$ and $\square \psi^s_{\text{test}}$, and the invariant Φ. Furthermore, given the partial order $(\{\mathcal{V}^i_p, \ldots, \mathcal{V}^i_0\}, \leq)$, one can find a test policy to ensure that the σ makes progress along the partial order such that for some $t > 0$, $\sigma_t \in \mathcal{V}^i_0$. However, in case of multiple vertices in G_{aux} that satisfy $\psi^f_{\text{test},m}$, we need to extend the receding horizon synthesis to specification $\Psi^{\mathcal{I}}_j$. We construct the filter $\mathcal{W}^{\mathcal{I}}$ and also check that for every test execution σ, there exists $i \in \mathcal{I}$ such that for every $k \geq 0$, $\sigma_k \in \mathcal{V}^i_j$ and $\sigma_{k+1} \in \mathcal{V}^i_{j'}$. Therefore, because the auxiliary game graph is assumed to not have cycles, the test execution makes progress on the partial order of at least one $i \in \mathcal{I}$ at each timestep, thus eventually satisfying $\psi^f_{\text{test},m}$. Thus every execution of our algorithm will satisfy equation (5.2). $\square$

Algorithm 8: Merge Unit Tests $(\varphi_{\text{test},1}, \varphi_{\text{test},2}, \varphi_{\text{sys}}, \mathcal{T}_{\text{sys}}, \mathcal{T}_{\text{test},1}, \mathcal{T}_{\text{test},2}, \rho)$

```
1: procedure MergeUnitTests($(\varphi_{\text{test},1}, \varphi_{\text{test},2}, \varphi_{\text{sys}}, \mathcal{T}_{\text{sys}}, \mathcal{T}_{\text{test},1}, \mathcal{T}_{\text{test},2}, \rho)$)
     Input: Unit test specifications $\varphi_{\text{test},1}$ and $\varphi_{\text{test},2}$, system specification $\varphi_{\text{sys}}$,
   System $\mathcal{T}_{\text{sys}}$, unit test environments, $\mathcal{T}_{\text{test},1}$ and $\mathcal{T}_{\text{test},2}$, and quantitative metric of
   robustness $\rho$,
     Output: Merged test specification $\varphi_{\text{test},m}$, Merged test environment $\mathcal{T}_{\text{test},m}$,
   Merged test policy $\pi_{\text{test},m}$
2:     $\mathcal{C}_1, \mathcal{C}_2 \leftarrow$ Construct contracts for $\varphi_{\text{test},1}$ and $\varphi_{\text{test},2}$
3:     $\mathcal{T}_{\text{test}} \leftarrow \mathcal{T}_{\text{test},1} \times \mathcal{T}_{\text{test},2}$ Merged test environment
4:     $\mathcal{T}_{\text{prod}} \leftarrow \mathcal{T}_{\text{sys}} \times \mathcal{T}_{\text{test}}$ Product transition system
5:     $G \leftarrow$ Game graph from product transition system $\mathcal{T}_{prod}$
6:     $\mathcal{C}_m := (a_m, a_m \rightarrow g_m) \leftarrow$ strong merge($\mathcal{C}_1, \mathcal{C}_2$) Constructing the merged
   specification
7:     $\varphi_{\text{test},m} \leftarrow a_m \rightarrow g_m$ Merged test specification
8:     $G_{\text{aux}} \leftarrow$ Auxiliary game graph.
9:     $\mathcal{I} = \{s \in \mathcal{V}_{\text{aux}} | s \models \psi^f_{\text{test},m}\}$ Defining goal states and partial orders
10:    for $i \in \mathcal{I}$ do
11:        $\mathcal{P}^i := \{(\mathcal{V}^i_p, \dots, \mathcal{V}^i_0)\} \leftarrow$ Partial order for goal $i$
12:        $\psi^i_j \leftarrow$ Receding horizon specification for goal $i$ at distance $j$
13:    $\mathcal{W}^{\mathcal{I}} := \bigcup\{\mathcal{W}^i_j\} \leftarrow$ Test policy filter for goal $i$ at a distance of $j$
14:    $\pi_{\text{test},m} \leftarrow$ Searching for test policy guided by $\mathcal{W}^{\mathcal{I}}$
15:    return $\varphi_{\text{test},m}, \mathcal{T}_{\text{test},m}, \pi_{\text{test},m}$
```

Test Strategy Synthesis: Monte Carlo Tree Search is used to sample trajectories on G_{aux} after applying the safety filter $\mathcal{W}^{\mathcal{I}}$ to find a reactive test strategy $\pi_{\text{test},m}$ that satisfies the merged test objective. This procedure allows for optimizing for a metric of difficulty while also ensuring that all test strategies do not construct impossible tests for the system. Using MCTS with an upper confidence bound (UCB) was introduced in [128] as the upper confidence bound for trees (UCT) algorithm, which guarantees that given enough time and memory, the tree search converges to the optimal solution. We use MCTS to find $\pi^*_{\text{test},m}$, the optimal solution to Problem 5.1 for the merged test objective.

The following theorem and proof are taken from [126].

Theorem 5.2. Algorithm 1 is sound.

Proof. This follows by construction of the algorithm and the use of MCTS with UCB. Given a test policy π_{test} and a system policy π_{sys}, for every resulting execution $\sigma_{\pi_{\text{sys}} \times \pi_{\text{test}}}$ starting from an initial state in $\mathcal{W}^{\mathcal{I}}$, it is guaranteed that $\sigma \models \varphi_{\text{test},m}$

by Theorem 5.1. This is because for any action chosen by the test environment according to the policy π_{test} found by MCTS, we are guaranteed to remain in $\mathcal{W}^{\mathcal{I}}$ for any valid system policy π_{sys}. If $\mathcal{W}^{\mathcal{I}} = \emptyset$ or the initial state is not in $\mathcal{W}^{\mathcal{I}}$, the algorithm will terminate before any rollout is attempted and no policy is returned. It can be shown that the probability of selecting the optimal action converges to 1 as the limit of the number of rollouts is taken to infinity. For convergence analysis of MCTS, please refer to [128]. $\square$

Complexity: The time complexity of $GR(1)$ synthesis is in the order of $O(|N|^3)$, where N is the number of states needed to define the GR(1) formula. To improve scalability, our algorithm uses a receding horizon approach to synthesize the winning sets, which further reduces the time complexity. The upper confidence tree algorithm of MCTS is given as $O(ijkl)$ with j the number of rollouts, k the branching factor of the tree, l the depth of the tree, and i the number of iterations.

Simulation Experiments

This framework is illustrated on discrete gridworld examples where the system controller is non-deterministic and the test agents behave according to the synthesized test strategy. The Temporal Logic and Planning Toolbox (TuLiP) [109] is used for constructing winning sets [108], and an open-source script[1] for the online MCTS algorithm to find the test strategy. Simulation videos of at the linked GitHub repository[2].

Lane Change

For the lane change example, we define $\rho(\sigma)$ as the x-value of the cell in which the system finished its lane change maneuver.The test strategy is found to be consistent with the test objective in equation (5.7) while also maximizing by maximizing $\rho(\sigma)$. The metric ρ is the chosen metric of difficulty; the closer to the end of the lane, the fewer attempts the system will have for a successful lane change. Snapshots of the resulting test execution are depicted in Figure 5.5.

Unprotected left turn

In this example, the test environment consists of a pedestrian and a blue car, and the system is the red car, as illustrated in Figure 5.6. The unit tests correspond to

[1]https://gist.github.com/qpwo/c538c6f73727e254fdc7fab81024f6e1
[2]https://github.com/jgraeb/MergeUnitTests

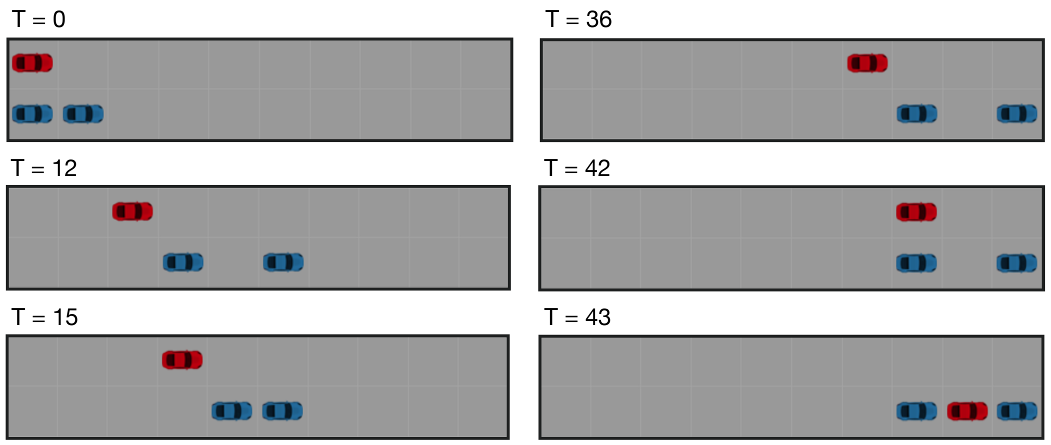


Figure 5.5: Snapshots during the execution of the test generated by our framework. The system under test (red car) needs to merge onto the lower lane between the two test agents (blue cars).

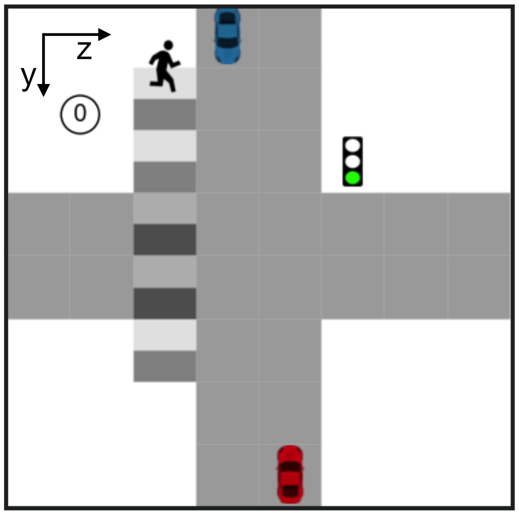


Figure 5.6: Layout of the unprotected left turn at intersection example. The system starts in cell (7,4) and wants to reach the goal cell (0,3), while the initial positions of the test agents are at the beginning of the road and crosswalk.

waiting for an oncoming car to pass the intersection, and waiting for a pedestrian to pass before taking a left turn.

The system requirement is to safely take an unprotected left turn. The unit specifications for waiting for the pedestrian are defined according to equation (5.2):

$$\varphi_{sys}^{\text{init}} = (\mathbf{x}_S \in I_S), \quad \varphi_{sys}^{f} = (\mathbf{x}_S \in \mathcal{S}_G), \quad \psi_{\text{test},ped}^{f} = (\mathbf{x}_S \in \mathcal{S}_P \wedge \mathbf{x}_P \in T_P)\,, \tag{5.12}$$

where $\mathbf{x}_S$ is the system state, I_S is the initial state of the system, $\mathcal{S}_G$ is the set of goal state following the left turn, $\mathbf{x}_P$ is the pedestrian state, and $\mathcal{S}_P$ are the states in which the car must wait for the pedestrian if the pedestrian state is in T_P. Similarly, the unit test objective for waiting for the test car is given as follows:

$$\varphi_{\text{sys}}^{\text{init}} = (\mathbf{x}_S \in I_S), \quad \varphi_{\text{sys}}^{f} = (\mathbf{x}_S \in \mathcal{S}_G), \quad \psi_{\text{test},car}^{f} = (\mathbf{x}_S \in \mathcal{S}_C \wedge \mathbf{x}_C \in T_C)\,, \tag{5.13}$$

where the C denotes the test agent car in blue. The coordinate system has origin in the upper left corner with cell $(y, z) = (0, 0)$, with the y-axis facing south and the z-axis facing east. The crosswalk locations are numbered from north to south, starting at 0.

The initial states of the test agents are $\mathbf{x}_C = (0, 3)$ and $\mathbf{x}_P = 0$, and the initial state of the system is $\mathbf{x}_S = (7, 4)$. The goal state for the system is $\mathbf{x}_G = (0, 3)$. In this example, $\mathbf{x}_G$ is the only element in $\mathcal{S}_G$. The state in which the system needs to wait for the pedestrian and the car, $\mathcal{S}_C$ and $\mathcal{S}_P$, respectively, are both $\mathbf{x} = (4, 4)$. When the test agent has not yet approached the intersection or has just approached the intersection, the system must wait. These states of the test agent are $\mathcal{T}_C = \{(0, 3), (1, 3), (2, 3), (3, 3)\}$. Similarly, the states of the pedestrian for which the system has to wait are $\mathcal{S}_P = \{1, 2, 3, 4, 5\}$, which represent the cells on the crosswalk, that map to grid coordinates. Note that if the pedestrian is in cell 0, the system is not required to wait for the pedestrian, as she is too far away from the road. The traffic light sequence is predetermined, the light will be green for a fixed number of time steps t_g, followed by t_y time steps of yellow and red for t_r time steps. We are assuming that the system designer supplied the robustness metric as the time until the traffic light turns red, resulting in a harder test the closer the light is to red once the system successfully takes the turn.

The robustness metric at a state is defined to be the time left until the traffic light changes to red, starting at the moment the system enters the intersection. The robustness over the entire trajectory is the minimum value of the robustness of all states in the trajectory. The smaller the value of this robustness, the more difficult the test for the reason that the system has fewer opportunities to successfully complete its task.

Additionally, this is an example in which all trajectories of the car taking a left turn while waiting for the pedestrian will also satisfy the condition of waiting for the test car and vice-versa. That is, $\sigma \models \Diamond\, \psi_{\text{test},ped}^{f} \iff \sigma \models \Diamond\, \psi_{\text{test},car}^{f}$. As a result, for this example, we add temporal constraints to the merged test objective to ensure that the two events do not entirely coincide.

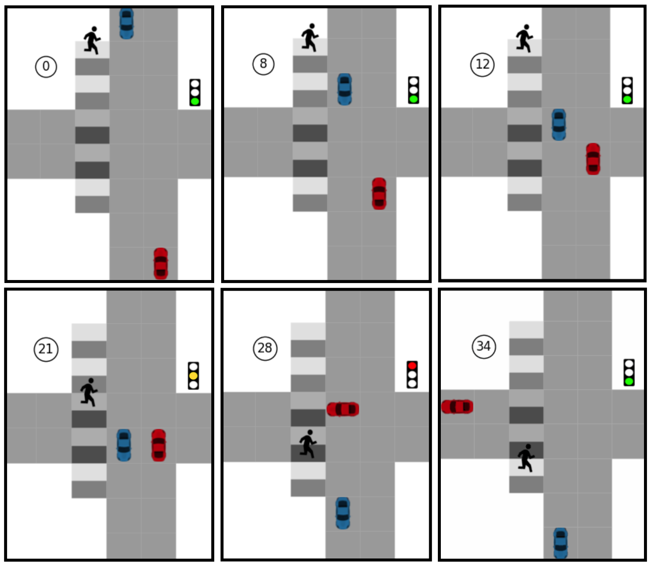


Figure 5.7: Snapshots during the execution of the unprotected left turn test generated by our framework. The autonomous vehicle (AV) under test (red) should take an unprotected left turn and wait for the pedestrian and the car (blue) individually, which are agents of the test environment. In the snapshots at time steps 8 and 12, the AV waits just for the car, and in time step 21 it waits just for the pedestrian.

The resulting test execution is shown in Figure 5.7. As expected, we see the system first waiting for the test car to pass the intersection. Even after the tester car passes, the pedestrian is still traversing the crosswalk, causing the system to wait for the pedestrian, satisfying the temporally constrained merged test objective.

5.5 Contract Theory for Formalizing Compositional Testing

So far, we have seen the use of the strong merge operator in constructing a single test from unit tests. In this part of the chapter, we explore the use of assume-guarantee contracts not only to combine tests, but also split complex tests into simpler unit tests on the overall system or on subsystems. We further explore the algebra of assume-guarantee contracts, and leverage contract operators to formalize this reasoning over test objectives. Finally, we illustrate test executions corresponding to the combined and split test structures in a discrete autonomous driving example and an aircraft formation-flying example. This work is a step towards formal methods to construct test campaigns from unit tests.

To apply concepts from this formalism, we introduce the test structure — a tuple that carries i) the formal specifications of the system under test, and ii) the test objective, which is specified by a test engineer. We build on test structures to define test campaigns and specifications for the tester. We address the following questions using the formalism of assume-guarantee contracts:

(Q1) *Comparing Tests:* Is it possible to define an ordering of tests? When is one test considered a refinement of another? See Section 5.8.

(Q2) *Combining Tests:* Can multiple unit test objectives be checked in a single test execution? See Section 5.7.

(Q3) *Splitting Tests:* From a complex test objective, can we split into component-level tests or split the test objective into simpler objectives? See Section 5.9.

5.6 Test Structures and Tester Specifications

For conducting a test, we need i) the system under test and its specification to be tested and ii) specifications for the test environment that ensure that a set of behaviors (specified by the test engineer) can be observed during the test. These sets of desired test behaviors are characterized by the test engineer in the form of a specification. The system specifications make some assumptions about the test environment. The test objective, together with the system specification, is used to synthesize a test environment and corresponding strategies of the tester agents. As a result, the test objective is not made known to the system since doing so would reveal the test strategy to the system. These concepts are formally defined below.

Definition 5.8. The *system specification* is the assume-guarantee contract denoted by $\mathcal{C}^{\mathrm{sys}} = (A^{\mathrm{sys}}, G^{\mathrm{sys}})$, where A^{sys} are the assumptions that the system makes on its operating environment, and G^{sys} denotes the guarantees that it is expected to satisfy if A^{sys} evaluates to $\top$. In particular, A^{sys} are the assumptions requiring a safe test environment, and $\neg A_i^{\mathrm{sys}} \cup G_i^{\mathrm{sys}}$ are the guarantees on the specific subsystem that will be tested.

$$\mathcal{C}^{\mathrm{sys}} = (A^{\mathrm{sys}}, \neg A^{\mathrm{sys}} \cup \bigcap_i (\neg A_i^{\mathrm{sys}} \cup G_i^{\mathrm{sys}})).$$

Definition 5.9. A *test objective* $\mathcal{C}^{\mathrm{obj}} = (\top, G^{\mathrm{obj}})$, where G^{obj} characterizes the set of desired test behaviors, is a formal description of the specific behaviors that the test engineer would like to observe during the test.

These contracts can be refined or relaxed using domain knowledge. Using definitions (5.8) and (5.9), we define a *test structure*, which is the unitary object that we use to establish our framework and for the analysis in the rest of the chapter.

Definition 5.10. A *test structure* is the tuple $\mathfrak{t} = (\mathcal{C}^{\mathrm{obj}}, \mathcal{C}^{\mathrm{sys}})$ comprising of the test objective and the system requirements for the test.

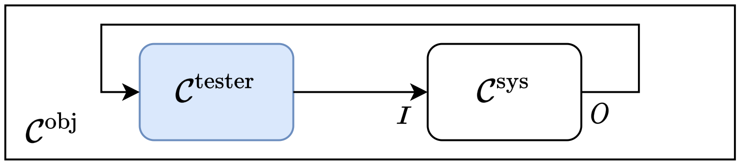

$\mathcal{C}^{\text{tester}}$
$\mathcal{C}^{\text{sys}}$
$\mathcal{C}^{\text{obj}}$
I
O

Figure 5.8: Block diagram showing contracts specifying the system specification $\mathcal{C}^{\text{sys}}$, the test objective $\mathcal{C}^{\text{obj}}$, and the test environment $\mathcal{C}^{\text{tester}}$.

Given the system specification and the test objective, we need to determine the specification for a valid test environment, which will ensure that if the system meets its specification, the desired test behavior will be observed. The resulting test execution will then enable reasoning about the capabilities of the system. If the test is executed successfully, the system passed the test, and conversely, if the test is failed, it is because the system violated its specification and not due to an erroneous test environment.

Now we need to find the specification of the test environment, the tester contract $\mathcal{C}^{\text{tester}}$, in which the system can operate and will satisfy the test objective according to Figure 5.8, with I, O denoting the inputs and outputs of the system contract. This contract can be computed as the mirror of the system contract, merged with the test objective, which is equivalent to computing the quotient of $\mathcal{C}^{\text{obj}}$ and $\mathcal{C}^{\text{sys}}$ [121]:

$$\mathcal{C}^{\text{tester}} = (\mathcal{C}^{\text{sys}})^{-1} \bullet \mathcal{C}^{\text{obj}} = \mathcal{C}^{\text{obj}} / \mathcal{C}^{\text{sys}}.$$

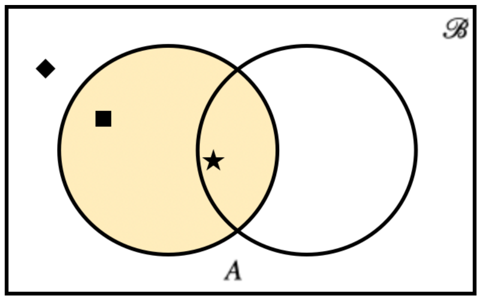
The tester contract can therefore directly be computed as

$$\mathcal{C}^{\text{tester}} = (G^{\text{sys}}, G^{\text{obj}} \cap A^{\text{sys}} \cup \neg G^{\text{sys}}). \tag{5.14}$$

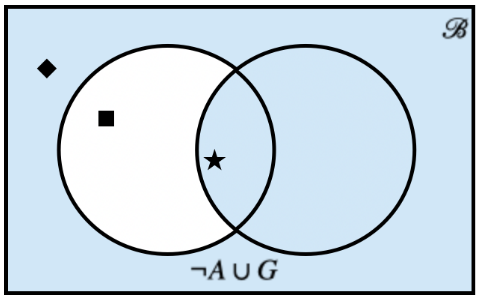
Remark: Since it is the tester's responsibility to ensure a safe test environment, A^{sys}, a test is synthesized with respect to the following specification,

$$\bigcap_i (\neg A_i^{\text{sys}} \cup G_i^{\text{sys}}) \to A^{\text{sys}} \cap G^{\text{obj}}. \tag{5.15}$$

A successful test execution lies in the set of behaviors $A^{\text{sys}} \cap G^{\text{sys}} \cap G^{\text{obj}}$, and an unsuccessful test execution is the sole responsibility of the system being unable to satisfy its specification. Thus, any implementation of $\mathcal{C}^{\text{tester}}$ will be an environment in which the system can operate and satisfy $\mathcal{C}^{\text{obj}}$ if the system satisfies its specification, a geometric interpretation is shown in Figure 5.9.



(a) Assumptions A of the contract.



(b) Guarantees $\neg A \cup G$ of the contract.

Figure 5.9: Geometric interpretation of an assume-guarantee contract (A, G) as a pair of sets of behaviors. The first element of the pair describes the set of behaviors for which the assumptions A hold, and the second element describes the set of behaviors for which G holds or A does not hold. The tester failing to provide the guarantees G (square) does not satisfy the contract. The set of desired test executions is in the intersection of the assumptions and guarantees (star), and the set of test executions that fall outside the assumptions (diamond) are because the system under test failed to satisfy its requirements.

5.7 Combining Tests

Earlier in the chapter, the strong merge operator was used to merge unit test objectives into a single objective. However, it required careful specification of assumptions and guarantees in a single GR(1) specification. Using the test structures introduced in the previous section, combining unit test contracts via the strong merge operator is equivalent to merging unit test specifications. The advantage of the new formalism is that it allows us to easily compose system and test objectives separately, without manual checking. The strong merge of test contracts is defined as follows.

Proposition 5.1. $\mathcal{C}_1^{\text{tester}} \bullet \mathcal{C}_2^{\text{tester}} = (\mathcal{C}_1^{\text{obj}} \parallel \mathcal{C}_2^{\text{obj}})/\left(\mathcal{C}_1^{\text{sys}} \parallel \mathcal{C}_2^{\text{sys}}\right).$

Proof. Merging tester contracts yields

$$\begin{aligned}
\mathcal{C}_1^{\text{tester}} \bullet \mathcal{C}_2^{\text{tester}} &= (\mathcal{C}_1^{\text{obj}}/\mathcal{C}_1^{\text{sys}}) \bullet (\mathcal{C}_2^{\text{obj}}/\mathcal{C}_2^{\text{sys}}) \\
&= (\mathcal{C}_1^{\text{obj}} \bullet (\mathcal{C}_1^{\text{sys}})^{-1}) \bullet (\mathcal{C}_2^{\text{obj}} \bullet (\mathcal{C}_2^{\text{sys}})^{-1}) && \text{([129], Section 3.1)} \\
&= (\mathcal{C}_1^{\text{obj}} \bullet \mathcal{C}_2^{\text{obj}}) \bullet \left((\mathcal{C}_1^{\text{sys}})^{-1}) \bullet ((\mathcal{C}_2^{\text{sys}})^{-1})\right) \\
&= (\mathcal{C}_1^{\text{obj}} \bullet \mathcal{C}_2^{\text{obj}}) \bullet \left(\mathcal{C}_1^{\text{sys}} \parallel \mathcal{C}_2^{\text{sys}}\right)^{-1} && \text{([121], Table 6.1)} \\
&= (\mathcal{C}_1^{\text{obj}} \bullet \mathcal{C}_2^{\text{obj}})/\left(\mathcal{C}_1^{\text{sys}} \parallel \mathcal{C}_2^{\text{sys}}\right) \\
&= (\mathcal{C}_1^{\text{obj}} \parallel \mathcal{C}_2^{\text{obj}})/\left(\mathcal{C}_1^{\text{sys}} \parallel \mathcal{C}_2^{\text{sys}}\right), && (A_1^{\text{obj}} = A_2^{\text{obj}} = \top))
\end{aligned}$$

which is the list $(\mathcal{C}_1^{\text{obj}} \parallel \mathcal{C}_2^{\text{obj}}, \mathcal{C}_1^{\text{sys}} \parallel \mathcal{C}_2^{\text{sys}})$. □

The merged test constract is constructed from parallel compositions of the objective contracts and system contracts, separately. Composition of the system contracts should be interpreted as specifications on the subsystems. The composition of test structures is defined as:

Definition 5.11. Given test structures $\mathfrak{t}_i = (\mathcal{C}_i^{\text{obj}}, \mathcal{C}_i^{\text{sys}})$ for $i \in \{1, 2\}$, we define their *composition* $\mathfrak{t}_1 \parallel \mathfrak{t}_2$ as

$$(\mathcal{C}_1^{\text{obj}}, \mathcal{C}_1^{\text{sys}}) \parallel (\mathcal{C}_2^{\text{obj}}, \mathcal{C}_2^{\text{sys}}) = (\mathcal{C}_1^{\text{obj}} \parallel \mathcal{C}_2^{\text{obj}}, \mathcal{C}_1^{\text{sys}} \parallel \mathcal{C}_2^{\text{sys}}).$$

Example 5.3 (Car Pedestrian)**.** Recall the car-pedestrian example from Chapter 2, which we will adopt with slight modifications. Consider a test environment shown in Figure 5.10 consisting of a single lane road, a crosswalk with a pedestrian, and different visibility conditions. The system under test is an autonomous car driving on the road which must stop for the pedestrian at the crosswalk no matter the visibility conditions. The first test objective under low visibility is formalized by a test engineer as:

$$\mathcal{C}_1^{\text{obj}} = (\top, \quad \varphi_{\text{init}}^{\text{car}} \wedge \Box \varphi_{\text{low}}^{\text{vis}} \wedge \Diamond \varphi_{\text{cw}}^{\text{ped}} \wedge (\varphi_{\text{cw}}^{\text{ped}} \rightarrow \Diamond \varphi_{\text{cw}}^{\text{stop}})),$$

where $\varphi_{\text{low}}^{\text{vis}} := \varphi^{\text{vis}} \models \text{low}$, denotes low visibility conditions, $\varphi_{\text{init}}^{\text{car}}$ the initial conditions of the car (position x_{car} and velocity v_{car}), $\varphi_{\text{cw}}^{\text{ped}}$ denotes the pedestrian being on the crosswalk, and $\varphi_{\text{cw}}^{\text{stop}} := x_{\text{car}} \leq C_{\text{cw}-1} \wedge v_{\text{car}} = 0$ the stopping maneuver at one cell before the crosswalk cell C_{cw}. Similarly, the test objective contract under high visibility is also given as:

$$\mathcal{C}_2^{\text{obj}} = \Big(\top, \quad \varphi_{\text{init}}^{\text{car}} \wedge \Box \varphi_{\text{high}}^{\text{vis}} \wedge \Diamond \varphi_{\text{cw}}^{\text{ped}} \wedge (\varphi_{\text{cw}}^{\text{ped}} \rightarrow \Diamond \varphi_{\text{cw}}^{\text{stop}})\Big),$$

where $\varphi_{\text{high}}^{\text{vis}} := \varphi^{\text{vis}} \models \text{high}$ represents high visibility test environment. Finally, the dynamics of braking when a pedestrian is detected is given by the contract,

$$\mathcal{C}_3^{\text{obj}} = (\top, \quad \exists k : (v_{\text{car}} = V_{\text{max}} \wedge x_{\text{car}} = C_k) \rightarrow \Diamond \varphi_{k+d}^{\text{stop}}),$$

where the car is required to drive at specified speed V_{max} in an arbitrary cell C_k, and then eventually stop within the stopping distance d specified by the user; $\varphi_{k+d}^{\text{stop}}$ specifies that the car must stop at or before cell C_{k+d}. Note that we assume that the stopping distance d is large enough such that the car moving at maximum speed can come to a stop within d steps. This test objective specifies the stopping requirement on the car irrespective of the environment. Note that none of the test objective contracts reason over the system's capabilities to detect a pedestrian, only requiring that the system needs to stop at the crosswalk if a pedestrian is in it. This is important since we do not want the test objective contract to have guarantees that depend on the performance of individual components (e.g., perception) of the system.

Requirements on the system are provided by the system designers and test engineers. Each of the following system contracts assume that the environment is safe (e.g., the environment agents will not adversarially crash into the car). This is denoted as $A^{\text{sys}} = \square\varphi_{\text{dyn}}^{\text{ped}} \wedge \square\varphi_{\text{dyn}}^{\text{vis}}$, where $\varphi_{\text{dyn}}^{\text{ped}}$, and $\varphi_{\text{dyn}}^{\text{vis}}$ denote the dynamics of the pedestrian, and the visibility conditions, respectively.

$$\begin{aligned}\mathcal{C}_1^{\text{sys}} = \Big(A^{\text{sys}},\quad & \square\varphi_{\text{dyn}}^{\text{car}} \wedge \square\left(\varphi_{\text{low}}^{\text{vis}} \rightarrow v \leq V_{\text{low}}\right) \wedge \\ & \square\left(\texttt{detectable}_{\text{low}}^{\text{ped}} \rightarrow \diamond\, \varphi_{\text{ped}}^{\text{stop}}\right) \vee \neg A^{\text{sys}}\Big),\end{aligned}$$

where $\varphi_{\text{dyn}}^{\text{car}}$, describes the dynamics of the car. V_{low} is the maximum permissible speed of the car under low-visibility conditions. The expression $\texttt{detectable}_{\text{low}}^{\text{ped}}$ describes the pedestrian being in a buffer zone in front of the car, and is formally defined as,

$$\texttt{detectable}_{\text{low}}^{\text{ped}} := x_{\text{car}} + dist_{\text{min}}^{\text{low}} \leq x_{\text{ped}} \leq x_{\text{car}} + dist_{\text{max}}^{\text{low}},$$

where $dist_{\text{min}}^{\text{low}}$ is the minimum distance for the car to reach a full stop, and $dist_{\text{max}}^{\text{low}}$ is the maximum distance at which the car can detect a pedestrian in low visibility conditions. The second system objective contract describes driving in high visibility conditions:

$$\begin{aligned}\mathcal{C}_2^{\text{sys}} = \Big(A^{\text{sys}},\quad & \square\varphi_{\text{dyn}}^{\text{car}} \wedge \square\left(\varphi_{\text{high}}^{\text{vis}} \rightarrow v \leq V_{\text{max}}\right) \wedge \\ & \square\left(\texttt{detectable}_{\text{high}}^{\text{ped}} \rightarrow \diamond\, \varphi_{\text{ped}}^{\text{stop}}\right) \vee \neg A^{\text{sys}}\Big),\end{aligned}$$

where V_{max} is the maximum speed, and the expression $\texttt{detectable}_{\text{high}}^{\text{ped}}$, defined similarly to $\texttt{detectable}_{\text{low}}^{\text{ped}}$, denotes the pedestrian being detectable in the 'buffer' zone for high visibility conditions. The third system objective contract for the dynamics of the car,

$$\mathcal{C}_3^{\text{sys}} = \Big(A^{\text{sys}},\quad \square\varphi_{\text{dyn}}^{\text{car}} \vee \neg A^{\text{sys}}\Big),$$

where $\varphi_{\text{dyn}}^{\text{car}}$ describes the dynamics of the car, including the distance to come to a full stop as a function of car speed. For each pair of system and test objectives, a test can synthesized for the specification constructed by equation (5.15). We can find combinations of test structures $\mathfrak{t}_i = (\mathcal{C}_i^{\text{obj}}, \mathcal{C}_i^{\text{sys}})$ that can be executed instead of individual tests. Consider the combined test structure $\mathfrak{t} = \mathfrak{t}_2 \parallel \mathfrak{t}_3$. The corresponding combined test objective contract $\mathcal{C}^{\text{obj}}$ is:

$$\begin{aligned}\mathcal{C}^{\text{obj}} = \mathcal{C}_2^{\text{obj}} \parallel \mathcal{C}_3^{\text{obj}} = (\top, \quad &\varphi_{\text{init}}^{\text{car}} \wedge \Box\varphi_{\text{high}}^{\text{vis}} \wedge \Diamond\, \varphi_{\text{cw}}^{\text{ped}} \wedge \varphi_{\text{cw}}^{\text{ped}} \rightarrow \Diamond\, \varphi_{\text{cw}}^{\text{stop}} \wedge \\ &\exists k : (v_{\text{car}} = V_{\text{max}} \wedge x_{\text{car}} = C_k) \rightarrow \Diamond\, \varphi_{k+d}^{\text{stop}}).\end{aligned} \tag{5.16}$$

Likewise, the combined system objective contract is:

$$\mathcal{C}^{\text{sys}} = \mathcal{C}_2^{\text{sys}} \parallel \mathcal{C}_3^{\text{sys}} = (A^{\text{sys}} \cup \neg(G_2^{\text{sys}} \cap G_3^{\text{sys}}), \quad G_2^{\text{sys}} \cap G_3^{\text{sys}}).$$

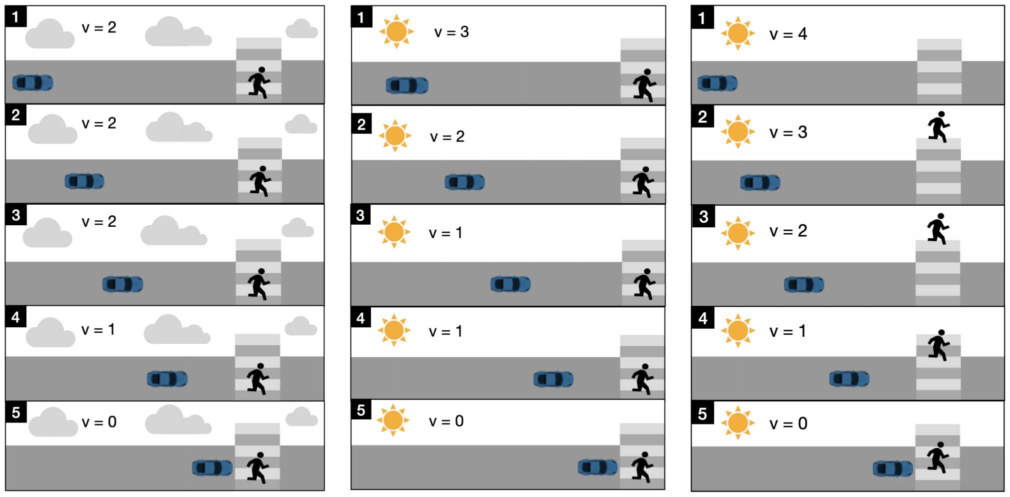
The term $\neg(G_2^{\text{sys}} \cap G_3^{\text{sys}})$ can be removed from the assumptions of $\mathcal{C}^{\text{sys}}$ to relax the system objective contract for ensuring that the assumptions conform to those required by Definition 5.8. Therefore, the system objective contract becomes:

$$\begin{aligned}\mathcal{C}^{\text{sys}} =(A^{\text{sys}}, \quad &\Box\varphi_{\text{dyn}}^{\text{car}} \wedge \Box\, (\varphi_{\text{high}}^{\text{vis}} \rightarrow v \leq V_{\text{max}}) \wedge \\ &\Box(\texttt{detectable}_{\text{high}}^{\text{ped}} \rightarrow \Diamond\, \varphi_{\text{ped}}^{\text{stop}}) \vee \neg A^{\text{sys}}).\end{aligned} \tag{5.17}$$

Equations (5.16) and (5.17) result in a test structure $\mathfrak{t} = (\mathcal{C}^{\text{obj}}, \mathcal{C}^{\text{sys}})$, and we can implement test environments to satisfy equation (5.15) with respect to the test structure $\mathfrak{t}$. The combined test structure $\mathfrak{t} = (\mathcal{C}^{\text{obj}}, \mathcal{C}^{\text{sys}}$ results in a test which requires the car to decelerate from V_{max} in high visibility conditions and come to a stop before the crosswalk.

To determine when two test structures can be combined, we need to check if the combined test objective and the combined test structure are satisifiable. Therefore, two test structures cannot be combined if either of these conditions is untrue. For example, the combination $\mathfrak{t}_1 \parallel \mathfrak{t}_2$ is not possible because composition of the constituent test objectives $\mathcal{C}_1^{\text{obj}} \parallel \mathcal{C}_2^{\text{obj}}$ has an empty set of guarantees. This is because $\Box\varphi_{\text{low}}^{\text{vis}}$ and $\Box\varphi_{\text{high}}^{\text{vis}}$ is disjoint since visibility cannot be both *high* and *low* at the same time. Now consider test structures $\mathfrak{t}_1$ and $\mathfrak{t}_3$; while these test structures can be composed to have a non-empty set of guarantees, the resulting test structure is not realizable by any test environment. The composition $\mathfrak{t}_1 \parallel \mathfrak{t}_3$ results in a test structure with the test objective requiring a maximum speed of V_{max}, but with the system constrained to a maximum speed of $V_{\text{low}} < V_{\text{max}}$ in low visibility conditions. Therefore, $G^{\text{sys}} \cap G^{\text{obj}} = \emptyset$, and both the system and test objectives cannot be satisfied in a single trace of the system.

Figure 5.10 illustrates manually constructed test executions that satisfy test contracts corresponding to $\mathfrak{t}_1, \mathfrak{t}_2$, and $\mathfrak{t}_2 \parallel \mathfrak{t}_3$, respectively. The car controller is implemented on a discrete grid world; at some positive speed v the car moves forward by



(a) Low visibility with a stationary pedestrian. (b) High visibility with a stationary pedestrian. (c) High visibility with a reactive pedestrian.

Figure 5.10: Test execution snapshots of the car stopping for a pedestrian. Figure 5.10a shows a test execution satisfying $\mathcal{C}_1^{\text{tester}}$, Figure 5.10b satisfies $\mathcal{C}_2^{\text{tester}}$ and Figure 5.10c satisfies $\mathcal{C}_2^{\text{tester}}$ and $\mathcal{C}_3^{\text{tester}}$.

v cells. At each time step, the car can choose to continue at the same speed or to accelerate or decelerate.

In the low visibility setting, the car can drive at a maximum speed of $v = 2$ and it can detect a pedestrian up to two cells away as illustrated in Figure 5.10a. The car is able to detect the pedestrian and come to a full stop in front of the crosswalk. In a high visibility setting, the car can drive at a maximum speed of $v_{\text{max}} = 4$, and it can detect the pedestrian up to 5 cells ahead. In Figure 5.10b, we can see that the pedestrian is detected and the car slows down gradually until is reaches the cell in front of the crosswalk. The test for the combined test structure $\mathfrak{t} = \mathfrak{t}_2 \parallel \mathfrak{t}_3$ is shown in Figure 5.10c, where we see the pedestrian entering the crosswalk in high visibility conditions when the car is driving at its maximum speed of $v = 4$ and is 10 cells away from the crosswalk. This test execution now checks the test objective of detecting a pedestrian in high visibility conditions and executing the braking maneuver with the desired constant deceleration from its maximum speed down to zero. ■

5.8 Comparing Tests

A test campaign is a set of tests, each characterized by a test structure. Choosing a test campaign out of several possibilities requires a principled approach to comparing test campaigns. A more refined test campaign is preferable since the system will be tested for a more refined set of test objectives and possibly for a more stringent set of system specifications. Let $\mathfrak{t}_i = (\mathcal{C}_i^{\text{obj}}, \mathcal{C}_i^{\text{sys}})$ be test structures for $1 \leq i \leq n$. When generating tests for $\mathfrak{t}_i$, we want to ensure that our test execution satisfies the constraints set out by $\mathcal{C}_i^{\text{obj}}$ in the context of system behaviors defined by $\mathcal{C}_i^{\text{sys}}$. As seen in Section 5.6, the tester contract can be computed using the quotient operator. We characterize a test campaign, $\mathtt{TC} = \{\mathfrak{t}_i\}_{i=1}^n$, as a finite list of test structures specified by the test engineer. Definition 5.12 allows us to generate a single test structure from a test campaign.

Definition 5.12. Given a test campaign $\mathtt{TC} = \{\mathfrak{t}_i\}_{i=1}^n$, the *test structure generated by this campaign*, denoted $\tau(\mathtt{TC})$, is

$$\tau(\mathtt{TC}) = \mathfrak{t}_1 \parallel \ldots \parallel \mathfrak{t}_n.$$

A notion of ordering between test structures is necessary for establishing an ordering of test campaigns. This order is also important for defining the split of a test into unit tests, as we shall see later.

Definition 5.13. The test structure $(\mathcal{C}_1^{\text{obj}}, \mathcal{C}_1^{\text{sys}})$ *refines* the structure $(\mathcal{C}_2^{\text{obj}}, \mathcal{C}_2^{\text{sys}})$, written $(\mathcal{C}_1^{\text{obj}}, \mathcal{C}_1^{\text{sys}}) \leq (\mathcal{C}_2^{\text{obj}}, \mathcal{C}_2^{\text{sys}})$, if contract refinement occurs element-wise, i.e., if $\mathcal{C}_1^{\text{sys}} \leq \mathcal{C}_2^{\text{sys}}$ and $\mathcal{C}_1^{\text{obj}} \leq \mathcal{C}_2^{\text{obj}}$.

Finally, the order of refinement between test campaigns can be defined as follows. In a refined test campaign $\mathtt{TC}$ of $\mathtt{TC}'$, the system and test objective contracts of the test structure corresponding to $\mathtt{TC}$ are more refined. That is, the test objective handles a larger set of system behaviors with stricter requirements (i.e., more constraints) on what the desired test execution should look like. In addition, the system might potentially be required to satisfy stricter guarantees on its behavior under a larger set of assumptions. For these reasons, it is preferable to choose a refined test campaign.

Definition 5.14. Given two test campaigns $\mathtt{TC}$ and $\mathtt{TC}'$, we say that $\mathtt{TC} \leq \mathtt{TC}'$ if $\tau(\mathtt{TC}) \leq \tau(\mathtt{TC}')$.

5.9 Splitting Tests

In this section, we explore the notion of splitting test structures. One of our motivations for doing this is failure diagnostics, in which we wish to look for root causes of a system-level test failure. To split test structures, we look for the existence of a quotient — see [129]. Suppose there exists a test structure $\mathfrak{t}$ that we want to split, and suppose one of the pieces of this decomposition, $\mathfrak{t}_1$, is given to us. Our objective is to find $\mathfrak{t}_2$ such that $\mathfrak{t}_1 \parallel \mathfrak{t}_2 \leq \mathfrak{t}$. The following result tells how to compute the optimum $\mathfrak{t}_2$. This optimum receives the name *quotient of test structures*.

Proposition 5.2. Let $\mathfrak{t} = (\mathcal{C}^{\text{obj}}, \mathcal{C}^{\text{sys}})$ and $\mathfrak{t}_1 = (\mathcal{C}_1^{\text{obj}}, \mathcal{C}_1^{\text{sys}})$ be two test structures and let $\mathfrak{t}_q = (\mathcal{C}^{\text{obj}}/\mathcal{C}_1^{\text{obj}}, \mathcal{C}^{\text{sys}}/\mathcal{C}_1^{\text{sys}})$. For any test structure $\mathfrak{t}_2 = (\mathcal{C}_2^{\text{obj}}, \mathcal{C}_2^{\text{sys}})$, we have

$$\mathfrak{t}_2 \parallel \mathfrak{t}_1 \leq \mathfrak{t} \quad \text{if and only if} \quad \mathfrak{t}_2 \leq \mathfrak{t}_q.$$

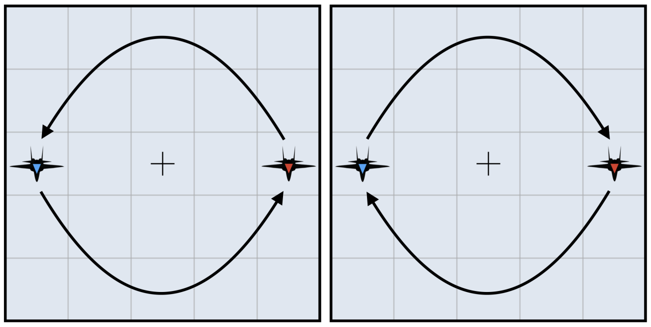
We say that $\mathfrak{t}_q$ is the quotient of $\mathfrak{t}$ by $\mathfrak{t}_1$, and we denote it as $\mathfrak{t}/\mathfrak{t}_1$.

Proof. $\mathfrak{t}_2 \leq \mathfrak{t}_q \quad \Leftrightarrow \quad \mathcal{C}_2^{\text{sys}} \leq \mathcal{C}^{\text{sys}}/\mathcal{C}_1^{\text{sys}}$ and $\mathcal{C}_2^{\text{obj}} \leq \mathcal{C}^{\text{obj}}/\mathcal{C}_1^{\text{obj}} \Leftrightarrow \quad (\mathcal{C}_2^{\text{obj}} \parallel \mathcal{C}_1^{\text{obj}}, \mathcal{C}_2^{\text{sys}} \parallel \mathcal{C}_1^{\text{sys}}) \leq (\mathcal{C}^{\text{obj}}, \mathcal{C}^{\text{sys}}) \quad \Leftrightarrow \quad \mathfrak{t}_2 \parallel \mathfrak{t}_1 \leq \mathfrak{t}$. $\square$

Remark: The method of constructing the quotient test structure in Proposition 5.2 involves taking the quotient of the system contracts as well as the test objectives, meaning that we remove a subsystem from the overall system, and remove a part of the test objective. Depending on the use case, we can consider two further situations, where we can define the test structure $\mathfrak{t}_1$ such that: i) only removing a subsystem from the overall system, which gives the quotient $\mathfrak{t}_q = (\mathcal{C}^{\text{obj}}, \mathcal{C}^{\text{sys}}/\mathcal{C}_1^{\text{sys}})$; and ii) only separating a part of the test objective: $\mathfrak{t}_q = (\mathcal{C}^{\text{obj}}/\mathcal{C}_1^{\text{obj}}, \mathcal{C}^{\text{sys}})$. The quotient test structures of type (i) could be useful in adding further test harnesses to monitor sub-systems under for the same test objective, and test structures of type (ii) could be useful in monitoring overall system behavior under a more unit test objective. In future work, we will study automatically choosing the relevant quotient test structure for specific use cases.

Label	Formula
φ_{setgoal}	$\square(x_{init,i} = R_1 \rightarrow x_{g,i} = R_2) \wedge \square(x_{init,i} = R_2 \rightarrow x_{g,i} = R_1)$
$\texttt{execute}_{\text{swap}}^{\text{ccw}}(a_i)$	$\Diamond(x_i = g_i) \wedge \square(x_i = g_i \rightarrow \bigcirc(x_i = g_i)) \wedge \square\varphi_{\text{traj,i}}^{\text{ccw}}$
$\texttt{execute}_{\text{swap}}^{\text{cw}}(a_i)$	$\Diamond(x_i = g_i) \wedge \square(x_i = g_i \rightarrow \bigcirc(x_i = g_i)) \wedge \square\varphi_{\text{traj,i}}^{\text{cw}}$
$\varphi_{\text{swap,i}}^{\text{cw}}$	$\square(\texttt{directive}_{\text{swap}}^{\text{cw}}(a_i) \rightarrow \Diamond(x_i = g_i))$
$\varphi_{\text{swap,i}}^{\text{ccw}}$	$\square(\texttt{directive}_{\text{swap}}^{\text{ccw}}(a_i) \rightarrow \Diamond(x_i = g_i))$
φ^{cw}	$\Diamond\, \texttt{directive}_{\text{swap}}^{\text{cw}}(a_1) \wedge \Diamond\, \texttt{directive}_{\text{swap}}^{\text{cw}}(a_2)$
φ^{ccw}	$\Diamond\, \texttt{directive}_{\text{swap}}^{\text{ccw}}(a_1) \wedge \Diamond\, \texttt{directive}_{\text{swap}}^{\text{ccw}}(a_2)$

Table 5.1: Subformulas for constructing G^{sys} and G^{obj}.


(a) Executions satisfying the original test structure.

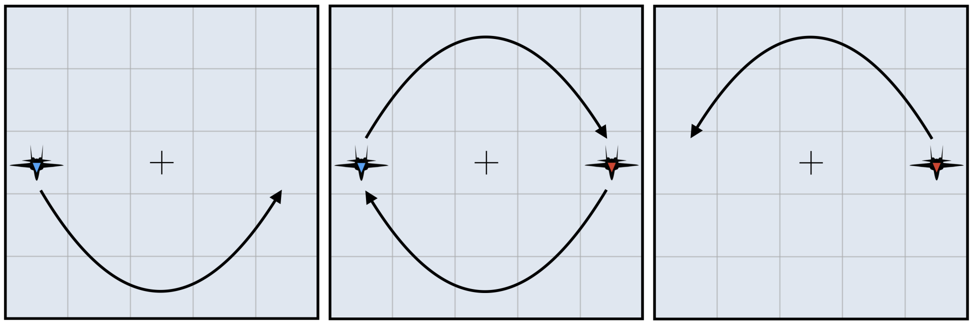

(b) Left: Given unit test. Center and right: Possible executions for the split test.

Figure 5.11: Front view of test executions satisfying the original test structure and the split test structure.

Example 5.4. Consider two aircraft, a_1 and a_2, flying parallel to each other undergoing a formation flying test shown in Figure 5.11a where two aircraft need to swap positions longitudinally in a clockwise or counterclockwise spiral motion. Assume that during this test execution a system-level failure has been observed, but it is unknown which aircraft is responsible for the failure during which stage of the maneuver. We will make use of our framework to split test structures to help identify the subsystem responsible for the failure.

The aircraft communicate with a centralized computer that issues waypoint directives to each aircraft in a manner consistent to the directives issued to other aircraft to ensure that there are no collisions. The dynamics of aircraft a_i on the gridworld is specified by G_i^{dyn}, and the safety or no collision requirement on all aircraft is given in G^{safe}. The swap requirement, G_i^{swap}, specifies the maneuver that each aircraft must take in the event that a directive is issued.

$$\begin{aligned} G_i^{\text{swap}} =& \square(\texttt{directive}^{\text{cw}}_{\text{swap}}(a_i) \rightarrow \texttt{execute}^{\text{cw}}_{\text{swap}}(a_i)) \wedge \\ & \square(\texttt{directive}^{\text{ccw}}_{\text{swap}}(a_i) \rightarrow \texttt{execute}^{\text{ccw}}_{\text{swap}}(a_i)). \end{aligned}$$

For example, in the case of a counter-clockwise swap directive issued to aircraft a_1 starting in region R_1, the aircraft must eventually reach the counter-clockwise swap goal, R_2, by traveling in the counter-clockwise direction, and upon reaching the goal must stay there as long as no new directive is issued. These maneuvers are specified in the `execute` subformulas in Table 5.1. The swap goals, g_i, for the aircraft are determined by their respective positions, $x_{\text{init,i}}$, when the directives are issued (see Table 5.1).

In this example, the tester fills the role of the supervisor. If the tester decides on all aircraft swapping clockwise, then the clockwise directives to each aircraft will be issued: $\varphi^{\text{cw}} = \Diamond\, \texttt{directive}^{\text{cw}}_{\text{swap}}(a_1) \wedge \Diamond\, \texttt{directive}^{\text{cw}}_{\text{swap}}(a_2)$. Similarly, φ^{ccw} denotes the eventual issue of counter-clockwise swap directives to both aircraft. All the temporal logic formulas required to construct the test structure associated with this example are summarized in Table 5.1. Moreover, no new directives are issued until all current directives are issued and all aircraft have completed the swap executions corresponding to the current directives (labeled as $G^{\texttt{dir}}_{\text{limit}}$). Finally, the aircraft are never issued conflicting swap directions — all aircraft are instructed to go clockwise or counterclockwise (labeled as $G^{\texttt{dir}}_{\text{safe}}$). For simplicity, we choose not to write out $G^{\texttt{dir}}_{\text{limit}}$ and $G^{\texttt{dir}}_{\text{safe}}$ in their extensive forms. Thus, the requirements for the system under test are as follows:

$$\mathcal{C}^{\text{sys}} = (A^{\text{sys}}, G^{\text{sys}}) = (G^{\texttt{dir}}_{\text{limit}} \wedge G^{\texttt{dir}}_{\text{safe}},\, G^{\text{safe}} \wedge \bigwedge_i G^{\text{swap}}_i \wedge G^{\text{dyn}}_i). \tag{5.18}$$

That is, assuming that the supervisor issues consistent directives, and issues new directives only when all aircraft have completed the executions corresponding to the current round of directives, the aircraft system is required to guarantee safety and successful execution of the swap maneuver corresponding to the current directive. If we were to write the system requirements for a single aircraft, the corresponding contract would be similar:

$$\mathcal{C}^{\text{sys}}_i = (A^{\text{sys}}_i, G^{\text{sys}}_i) = (G^{\texttt{dir}}_{\text{limit}} \wedge G^{\texttt{dir}}_{\text{safe}},\, G^{\text{swap}}_i \wedge G^{\text{dyn}}_i). \tag{5.19}$$

$$\begin{aligned} \mathcal{C}^{\text{obj}} &= (\top, G^{\text{obj}}), \\ G^{\text{obj}} &= \Box((\texttt{directive}^{cw}_{\text{swap}}(a_1) \wedge \texttt{directive}^{cw}_{\text{swap}}(a_2)) \vee (\texttt{directive}^{ccw}_{\text{swap}}(a_2) \\ &\quad \wedge \texttt{directive}^{ccw}_{\text{swap}}(a_1)) \rightarrow \Diamond(x_1 = R_2 \wedge x_2 = R_1)). \end{aligned} \tag{5.20}$$

Observe that G^{obj} represents the tester issuing either clockwise or counter-clockwise swap directives. One of the unit tests is to the have the aircraft a_1 starting at

$x_{init,1} = R_1$ (and as a result, $x_g = R_2$) get the counter-clockwise swap directive to reach $x_g = R_2$. The corresponding unit test structure $\mathfrak{t}_1 = (\mathcal{C}_1^{\text{obj}}, \mathcal{C}_1^{\text{sys}})$ can be written as follows:

$$\mathcal{C}_1^{\text{obj}} = (\top, G_1^{\text{obj}}) = (\top, \Diamond\, \texttt{directive}_{\text{swap}}^{\text{ccw}}(a_1)) \tag{5.21}$$

$$\mathcal{C}_1^{\text{sys}} = (A_1^{\text{sys}}, G_1^{\text{sys}}) = (G_{\text{limit}}^{\texttt{dir}} \wedge G_{\text{safe}}^{\texttt{dir}},\, G_1^{\text{swap}} \wedge G_1^{\text{dyn}}). \tag{5.22}$$

Following Proposition 5.2, the second unit test structure can be derived by separately applying the quotient operator on the test objectives and the system contract. Applying the quotient on the test objective, we substitute $\top$ for the assumptions to simplify, and we refine the quotient contract $\mathcal{C}^{\text{obj}}/\mathcal{C}_1^{\text{obj}}$ by replacing its assumptions with $\top$:

$$\begin{aligned}\mathcal{C}^{\text{obj}}/\mathcal{C}_1^{\text{obj}} &= (A \cap G_1^{\text{obj}}, G \cap A_1^{\text{obj}} \cup \neg(A \cap G_1^{\text{obj}}))\\ &= (G_1^{\text{obj}}, G \cup \neg G_1^{\text{obj}}) \geq (\top, G^{\text{obj}} \cup \neg G_1^{\text{obj}}).\end{aligned}$$

Designer input is important for refining this contract resulting from applying the quotient; a similar observation has been documented for quotient operators in previous work [123]. Domain knowledge can be helpful in refining the contracts. Using $\neg G_1^{\text{obj}}$ as context, the contract $(\top, G^{\text{obj}} \cup \neg G_1^{\text{obj}})$ can be simplified to $(\top, \neg G_1^{\text{obj}} \vee \varphi_1 \vee \varphi_2)$, where $\varphi_1 = (\Diamond\, \texttt{directive}_{\text{swap}}^{\text{ccw}}(a_2) \wedge \neg\varphi^{\text{cw}})$ and $\varphi_2 = \varphi^{\text{cw}} \wedge \neg\varphi^{\text{ccw}}$. Then, $\neg G_1^{\text{obj}}$ is discarded and the test objective of the second unit test can be defined as a refinement of this simplified contract arising from the quotient:

$$\mathcal{C}_{a_2}^{\text{obj}} = (\top, \varphi_1 \vee \varphi_2) \leq (\top, \neg G_1^{\text{obj}} \vee \varphi_1 \vee \varphi_2). \tag{5.23}$$

In equation (5.23), there are two types of test executions that would be the unit contract obtained by applying the quotient operator: i) A counter-clockwise directive is issued to aircraft a_2 and no clockwise directives are issued to either aircraft, or ii) Both aircraft are issued clockwise directives and no counter-clockwise directives. Note that φ_1 and φ_2 cannot be implemented in the same test by construction. Finally, the unit system contract can also by found by applying the quotient operator:

$$\begin{aligned}\mathcal{C}^{\text{sys}}/\mathcal{C}_1^{\text{sys}} =&(A^{\text{sys}} \cap G_1^{\text{sys}}, G^{\text{sys}} \cap A_1^{\text{sys}} \cup \neg(A^{\text{sys}} \cap G_1^{\text{sys}}))\\ =&(G_{\text{limit}}^{\text{dir}} \wedge G_{\text{safe}}^{\text{dir}} \wedge G_1^{\text{swap}} \wedge G_1^{\text{dyn}}, (G^{\text{safe}} \wedge G_2^{\text{swap}} \wedge G_2^{\text{dyn}})\\ &\vee \neg(G_1^{\text{swap}} \wedge G_1^{dyn} \wedge G_{\text{limit}}^{\text{dir}} \wedge G_{\text{safe}}^{\text{dir}}))\\ =&(G_{\text{limit}}^{\text{dir}} \wedge G_{\text{safe}}^{\text{dir}} \wedge G_1^{\text{swap}} \wedge G_1^{\text{dyn}}, (G^{\text{safe}} \wedge G_2^{\text{swap}} \wedge G_2^{\text{dyn}})).\end{aligned} \tag{5.24}$$

We refine the quotient contract by keeping the assumptions to be *true*.

$$\mathcal{C}_{a_2}^{\text{sys}} = (\top, \neg(G_1^{\text{comm}} \wedge G_1^{\text{dyn}}) \vee (G^{\text{safe}} \wedge \bigwedge_i G_i^{\text{comm}} \wedge G_i^{\text{dyn}})) \tag{5.25}$$

$$= (\top, \neg(G_1^{\text{comm}} \wedge G_1^{\text{dyn}}) \vee (G^{\text{safe}} \wedge G_2^{\text{dyn}} \wedge G_2^{\text{dyn}})). \tag{5.26}$$

Remark: Observe that equation (5.24) carries the swap and dynamics requirements of aircraft a_1 in its assumptions. Since we choose to separate aircraft a_1 from the overall aircraft system, this quotient contract can be satisfied by making aircraft a_1 a part of the tester. For a test execution of $\mathfrak{t}_2$, the tester can choose to keep aircraft a_1 as a part of the test harness for the operational test involving aircraft a_2, or choose to not deploy a_1 during the test execution. Assuming that aircraft a_1 satisfies its swap requirements, and that the supervisor satisfies the requirements on the directives, $G_{\text{limit}}^{\text{dir}}$ and $G_{\text{safe}}^{\text{dir}}$, then this unit system contract guarantees that the aircraft a_2 satisfies its swap requirements, and all the aircraft together satisfy the safety requirements.

The system requirement $\mathcal{C}_2^{\text{sys}} = \mathcal{C}^{\text{sys}} / \mathcal{C}_1^{\text{sys}}$ and the test objective together result in the following possible tester specifications,

$$\begin{aligned}\mathcal{C}_{\varphi_1}^{\text{tester}} = \Big(& G^{\text{safe}} \wedge G_2^{\text{swap}} \wedge G_2^{\text{dyn}},\ G_{\text{limit}}^{\text{dir}} \wedge G_{\text{safe}}^{\text{dir}} \wedge G_1^{\text{swap}} \wedge G_1^{\text{dyn}} \\ & \wedge \Diamond\, \texttt{directive}_{\text{swap}}^{\text{ccw}}(a_2) \wedge \neg\varphi^{\text{cw}} \Big).\end{aligned} \tag{5.27}$$

$$\begin{aligned}\mathcal{C}_{\varphi_2}^{\text{tester}} = \Big(& G^{\text{safe}} \wedge G_2^{\text{swap}} \wedge G_2^{\text{dyn}},\ G_{\text{limit}}^{\text{dir}} \wedge G_{\text{safe}}^{\text{dir}} \wedge G_1^{\text{swap}} \wedge G_1^{\text{dyn}} \\ & \wedge \Diamond\, \texttt{directive}_{\text{swap}}^{\text{cw}}(a_1) \wedge \Diamond\, \texttt{directive}_{\text{swap}}^{\text{cw}}(a_2) \wedge \neg\varphi^{\text{ccw}} \Big).\end{aligned} \tag{5.28}$$

From equation (5.27), we see that the tester does not require aircraft a_1 for any dynamic maneuvers, so it need not be deployed. In equation (5.28), even though aircraft a_1 would be a part of the test harness, it needs to be deployed for the tester contract, $\mathcal{C}_{\varphi_2}^{\text{tester}}$, to be satisfied. These tests resulting from the quotient test structure will help with determining the source of the failure that arose in the more complex test. ■

5.10 Conclusions and Future Work

In this chapter, we covered how assume-guarantee contract operations can aid in merging, comparing, and splitting specifications that define individual tests. While the previous chapter synthesized a test environment and strategy from the system abstraction, here we assume that such an environment is already synthesized. The ideas in this chapter are preliminary, and further research is needed for practical and large-scale construction of test campaigns while exploiting notions of compositionality. Yet, our framework based on the mathematical foundations of assume-guarantee contracts provides a useful formalism to reason about sets of behaviors that are covered by a test objective. An interesting direction of future work is to investigate how *coverage* arguments can be built from synthezing tests in this manner. Given a set of behaviors covered by a test structure, one could optimize for the worst-case test strategy using a robustness metric, preliminary versions of which were illustrated earlier in the chapter. This can be significantly expanded to systems with dynamics and specifications with timing constraints. Additionally, we would need to derive a guarantee that evaluations and conclusions from running the most difficult test for a test contract determines with high probability the success of possible other test executions in the same test contract.

Chapter 6

CONCLUDING REMARKS

6.1 Thesis Contributions

This thesis covered the following two themes: i) evaluating perception models using metrics that are relevant to system-level specifications as well as the downstream planning logic, and ii) synthesis of reactive test environments and strategies.

Task-Relevant Evaluation of Perception: In this thesis, we considered the problem of evaluating the object detection and classification task of perception given system-level specifications. We identified confusion matrices as an appropriate model of sensor error for the object detection and classification task, and using principles of automata theory and probabilistic model checking, we formally defined probability functions to relate the confusion matrix to a probabilistic model of system state evolution. Confusion matrices are a popular choice for comparing and evaluating detection models in computer vision. This work is the first to formally establish a link between confusion matrices and system-level probability of satisfying a temporal logic specification. Qualitatively, our theoretical approach matches empirical observations in experimental work conducted in industry [13]. Furthermore, our approach lends a quantitative framework for designers to choose appropriate detection models based on their specifications. For instance, the precision-recall tradeoff which is well-known in detection tasks, is manifested in the system-level performance, and is quantified in the form of probabilistic guarantees. Due to this, we can compute desired lower bounds on detection performance (e.g., lower bounds on precision, false negative rate etc.) from desired quantitative system guarantee. We did this as a case study using the system design optimization tool, Pacti [68].

Based on these theoretical fundamentals, the second contribution in this direction is proposing new metrics for evaluating detection models that are more relevant to the system-level specification and the downstream controller. For this, we introduced proposition-labeled confusion matrices, in which traditional class labels are replaced by propositional formulas that are relevant to controller design. Furthermore, evaluations can be grouped at the same level of abstraction as the downstream controller that receives these detections as input. We evaluated a pre-trained

Pointpillars model that detects objects based on LiDaR data on the entire nuScenes dataset, and illustrate the result for a car-pedestrian example.

Reactive Test Synthesis: Automated test synthesis is a technical challenge motivated by the need for certification of safety-critical autonomous systems. These systems are expected to reason over both discrete and continuous states and inputs. This thesis studies synthesis of reactive test plans for high-level decision-making over discrete states and inputs. Specifications of the system are encoded in the system objective. In addition, user-defined specification of desired test behavior are encoded in the test objective, which is not revealed to the system under test. In this thesis, test objectives are manually specified. However, there is potential for automating this process as discussed in the future directions section later in the chapter.

In this thesis, we covered a test synthesis framework to restrict system actions reactively via a test harness. These restrictions can be implemented by the test environment using static and/or reactive obstacles, and dynamic test agents. First, we construct a product graph that tracks the system dynamics, and realization of the system and test objectives. Effectively, a path on the product graph represents a test execution. The routing problem is formulated as an optimization in which the test execution to realize the test objective without making it impossible for the system to realize the system objective. Via a reduction from 3-SAT, the computational complexity of this routing problem is shown to be NP-hard. This thesis covers two main approaches to solve the routing problem: Stackelberg game with coupled constraints and a mixed-integer linear program. The mixed-integer linear program can be solved more efficiently, and with guarantees that a feasible solution is a feasible test strategy. For different test environment types, the mixed-integer program can be easily modified by adding linear constraints to account for different environment types and to exclude any solutions. Static obstacles can be implemented as a special case of the reactive setting by adding constraints to enforce the non-reactivity of these restrictions. Furthermore, the dynamic agent strategy is synthesized to realize the reactive test strategy by matching the optimization solution.

Finally, we conducted hardware experiments using a pair of quadrupedal robots. The framework is agnostic to the specific controllers at the lower levels of the control stack, thus illustrating that the high-level tests synthesized by this framework can be effectively translated to hardware with test environments comprising of static and reactive obstacles, and dynamic test agents.

6.2 Future Directions

There are several exciting future directions for research on specification, testing, and verification of autonomous cyber-physical systems, guided by compelling demonstrations in hardware and simulation.

Layered, hierarchical test synthesis

Oftentimes, system-level failures in complex systems emerge from poor interfaces and interactions between subsystems. Current approaches to identifying failures with respect to specifications relies on black-box optimization methods, which are typically limited to identifying input signals in the continuous domain. While there is some work on identifying discrete-valued test inputs, it is often limited to variables that remain constant throughout the test (e.g., color of objects, the decision to place static barriers in the scene).

Figure 6.1 illustrates the vertical stack of the planning and control modules. The high-level planner, which operates at a slower timescale, is responsible for long-horizon decision-making which involves reasoning over fundamentally discrete variables. At the mid-level, trajectories with waypoints are planned for the robotic system. Finally the low-level controller, operating at faster speeds, executes the mid-level plan. In this thesis, we studied test environment synthesis for the high-level planner. The falsification approaches to identifying test cases are traditionally used to find failures at the mid-level and low-level. Furthermore, falsification algorithms often output open-loop trajectories, instead of reactive test strategies.

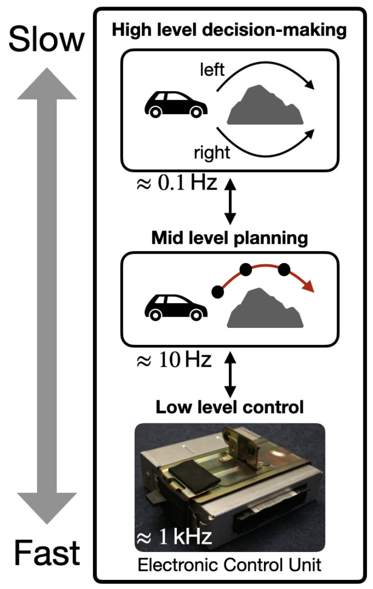


Figure 6.1: Overview of the planning and control software stack.

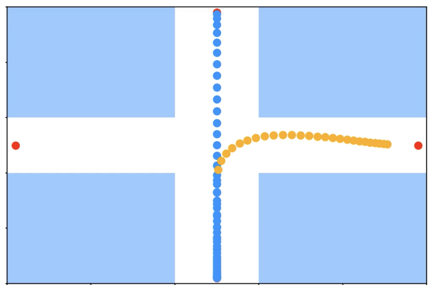
An open question is to identify falsifying instances resulting from a *combination* of poor *high-level decision making* together with continuous *nonlinear dynamics* at the low-level (e.g., incorrectly switching to a different dynamical mode, causing the system to violate safety or progress specifications). This is non-trivial even in simple hybrid system examples, especially when the system architecture and control design are black-box to the tester, and attempts to identify these failures by jointly searching over discrete and continuous parameters will not scale. To address this challenge, we would need to i) infer how high-level commands affect continuous dynamics, and ii) infer when switches to different high-level modes result in dynamically unsafe trajectories. In addition to falsifying components at different levels of the control stack, we would also need to falsify the interfaces that map between them.

As an example, consider a simple switched system shown in Figure 6.2. The system in this example is a point-mass which must avoid the unsafe regions shaded in blue, and has two operating modes: a north-south mode, and an east-west mode. The tester has access to a switch command: when a switch command is issued, the system must eventually switch to the other operating mode. The system is a 2-dimensional double integrator, and has a simple model-predictive controller with a quadratic cost consisting of: i) position error with respect to the north-south and east-west axes, ii) control effort, and iii) terminal cost defined on the system state. The terminal cost function is a 2-norm of the distance to the goal (north/south/east/west) with zero velocity along either axis. The optimization includes constraints on control effort that can be exerted in each direction.

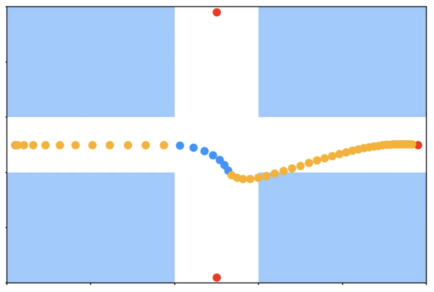
With no knowledge of the control design, and a couple of trials of commanding a single switch, the trajectories of this system would indicate a safe implementation. For example, see Figure 6.2a, in which the system safely avoids the unsafe region in executing the switch command. However, two switch commands in quick succession result in a falsifying trajectory of the system, highlighting the flaw in controller. In this example, the decision to switch twice is a fundamentally discrete choice, which current falsification algorithms typically cannot handle. The role of reactivity and layered architecture also becomes evident: i) the second switch is command in reaction to the system response to the first switch, and ii) the dynamical behavior (low-level behavior) of the system in response to the switch command combined with its decision to switch modes immediately without regard for safety (high-level decision) is what ultimately results in the failing trace.

Incorporating low-level dynamics

A challenge in hierarchical test and evaluation is in identifying the appropriate surrogate model of the system on which to test. As discussed previously, suppose the system-under-test is black-box to the test engineer, that is, the planning and control architecture and implementation is unknown to the tester. This would also imply



(a) Point mass (system) starting from the north position and in north-south mode (in blue), and commanded to switch to east-west mode (in orange).



(b) Point mass (system) starting from the west position and in east-west mode (in orange), and then commanded to switch twice in quick succession. The blue portion of the trajectory indicates the system switching to north-south mode before reverting to the east-west mode as shown in orange.

Figure 6.2: Simple switched system example, with position of the system shown at discrete time intervals.

that the system models used by the designers is also unknown to the test engineer. Therefore, the only entities known to the test engineer are the system specifications, the operational domain, and a black-box simulator or the physical system.

Though specifications are defined on the overall system, the subsystems responsible for the satisfaction of these specifications depends partly on the system implementation. For example, the requirement that the robot must remain safe likely requires multiple subsystems working together to satisfy safety. However, the requirement that the system exhibit certain motion primitives, e.g., quadruped must walk at a certain speed, might be the responsibility of a low-level controller. Depending on the scenario, certain specifications become prerequisites for other specifications. For example, the safety specification of the quadruped requires to evade an adversary might require it to consistently execute the walk motion primitive at a certain speed. For a broadly defined scenario such as evasion, if we are able to automatically order specifications according to this notion of pre-requisites, we can use the pre-requisite specifications to construct a model of high-level behavior of the system. In the literature on hybrid cyber-physical systems, this is related to work on constructing high-level system abstractions from low-level controllers using tools including reachability analysis. A concrete first step would be to review the literature on abstraction for control synthesis [130–133], and leveraging these methods to build a similar paradigm for abstraction, and discretization for testing.

The previous subsection is largely concerned with the case in which an abstraction of the system is assumed, and we need an efficient approach to testing that includes reasoning over both discrete and continuous variables. In this subsection, we are concerned with finding abstractions suitable for testing. There are two high-level directions for future work. First, how can we use prerequisite specifications to gather data from the system, and construct abstractions to model high-level behavior? These abstractions can include quantitative metrics of difficulty of executing lower level motion primitives. Second, what are the fundamental limitations of constructing these abstractions given the black-box nature of the system? Constructing these abstractions would potentially require a combination of model-based and data-driven methods — model-based in the sense that the physical dynamics, but not the control implementation, of the system might be known to the tester, and data-driven to get statistical data on the specific system implementation.

Criticality, coverage, and compositionality of test plans

Identifying critical test objectives is ill-posed when the operational design domain (ODD) is vast and difficult to characterize. While it might be hard to define a critical scenario in general, can we comparatively evaluate the criticality of two test scenarios? We can begin by studying the criticality of scenarios from a controls perspective (assuming perfect perception) before expanding to criticality from the system perspective (including all tasks pertaining to perception, reasoning, and control). There are two contending perspectives. On the one hand, criticality of a scenario will depend partly on the system controller — a scenario that is challenging (e.g., in terms of control effort, optimality, robustness) for one controller need not be equally challenging for another. Yet, at the same time, some scenarios seem universally less critical than others. For example, a safe unprotected left turn at a T-intersection amidst busy two-way traffic is more compelling than taking the same unprotected left turn without traffic. When are scenarios comparable? How do we quantify comparative criticality, and can we identify a class of controllers for which one scenario is more critical than the other?

The aforementioned question also relates to coverage. Ideally, successfully passing a more critical test should imply high confidence that the system would pass the less critical test. We would need to define a coverage metric that is consistent with this notion of criticality while also capturing the diversity of possible scenarios that are not easily comparable. For example, would reactive test scenarios (e.g., test agents moving in an office space) cover "open-loop" tests in which the test environment is completely static (e.g., static yet cluttered office environment)?

Another direction to tackle the coverage question is to decompose it to the subsystem level as opposed to the scenario level. Since the ODD is often vast and difficult to define, we can focus on coverage for inputs to various subsystems in the control stack, which can be relatively low-dimensional in comparison to multi-modal sensor data received by the perception system. Can we then *compose* coverage guarantees from testing the individual subsystems during development to infer coverage at the system-level? Can we rely on this analysis to identify operational tests that check multiple unit-level tests at once?

Task-relevant metrics for perception

While this thesis identifies task-relevant metrics for object detection and classification tasks, corresponding metrics for other perception tasks such as tracking and behavior prediction still remain to be studied. Secondly, it is not clear which metrics offer the tightest system-level guarantees. This thesis offers preliminary results — confusion matrices chosen based on system-level specifications and downstream control logic result in less conservative evaluations overall. However, further research on investigating the tightness of these guarantees needs to be studied, along with hardware validation of the derived system-level guarantees.

Thirdly, one can extend these principles to perception-planning-control architectures where the interfaces between modules are less distinct. The confusion matrix only accounts for the final layer of the model's neural network, which outputs a scalar value to classify the object. However, higher dimensional learned features of the model could contain rich information on the model's performance which can be exploited. In which system-level architectures is the confusion matrix a sufficient metric of detection error? Which learned features of the detection models are a better representation of model performance with respect to the system-level task? Finally, these task-relevant evaluation criteria are often meant for offline evaluations of perception models, and the resulting system-level guarantees are limited to scenarios from the distribution used for model evaluations. Future work should study the how these guarantees can be updated via runtime monitoring or how they can degrade if a scenario is out-of-distribution.

Bibliography

[1] Zoox, "Putting Zoox to the test: preparing for the challenges of the road," 2021. https://zoox.com/journal/structured-testing/, Last accessed on 2024-04-11.

[2] Waymo, "A blueprint for av safety: Waymo's toolkit for building a credible safety case," 2020. https://waymo.com/blog/2023/03/a-blueprint-for-av-safety-waymos/#:~:text=A%20safety%20case%20for%20fully,evidence%20to%20support%20that%20determination., Last accessed on 2024-05-05.

[3] F. Favarò, L. Fraade-Blanar, S. Schnelle, T. Victor, M. Peña, J. Engstrom, J. Scanlon, K. Kusano, and D. Smith, "Building a credible case for safety: Waymo's approach for the determination of absence of unreasonable risk," 2023. www.waymo.com/safety.

[4] N. Kalra and S. M. Paddock, "Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability?," *Transportation Research Part A: Policy and Practice*, vol. 94, pp. 182–193, 2016.

[5] N. Webb, D. Smith, C. Ludwick, T. Victor, Q. Hommes, F. Favaro, G. Ivanov, and T. Daniel, "Waymo's safety methodologies and safety readiness determinations," 2020.

[6] I. S. Organization, "Road vehicles: Safety of the intended functionality (ISO Standard No. 21448:2022)," 2022. https://www.iso.org/standard/77490.html, Last accessed on 2024-04-11.

[7] L. Li, W.-L. Huang, Y. Liu, N.-N. Zheng, and F.-Y. Wang, "Intelligence testing for autonomous vehicles: A new approach," *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 2, pp. 158–166, 2016.

[8] H. Winner, K. Lemmer, T. Form, and J. Mazzega, "Pegasus—first steps for the safe introduction of automated driving," in *Road Vehicle Automation 5*, pp. 185–195, Springer, 2019.

[9] "DARPA Urban Challenge." https://www.darpa.mil/about-us/timeline/darpa-urban-challenge.

[10] "Technical Evaluation Criteria." https://archive.darpa.mil/grandchallenge/rules.html.

[11] P. Koopman and M. Wagner, "Challenges in autonomous vehicle testing and validation," *SAE International Journal of Transportation Safety*, vol. 4, no. 1, pp. 15–24, 2016.

[12] J. Eskenazi and W. Jarett, "Explore: See the 55 reports — so far — of robot cars interfering with SF fire dept.," 2023. `https://missionlocal.org/2023/08/cruise-waymo-autonomous-vehicle-robot-taxi-driverless-car-reports-san-francisco/`, Last accessed on 2024-04-11.

[13] H. Zhao, S. K. Sastry Hari, T. Tsai, M. B. Sullivan, S. W. Keckler, and J. Zhao, "Suraksha: A framework to analyze the safety implications of perception design choices in avs," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pp. 434–445, 2021.

[14] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Temporal-logic-based reactive mission and motion planning," *IEEE Transactions on Robotics*, vol. 25, no. 6, pp. 1370–1381, 2009.

[15] M. Kloetzer and C. Belta, "A fully automated framework for control of linear systems from temporal logic specifications," *IEEE Transactions on Automatic Control*, vol. 53, no. 1, pp. 287–297, 2008.

[16] M. Lahijanian, S. B. Andersson, and C. Belta, "A probabilistic approach for control of a stochastic system from LTL specifications," in *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*, pp. 2236–2241, IEEE, 2009.

[17] V. Raman, A. Donzé, M. Maasoumy, R. M. Murray, A. Sangiovanni-Vincentelli, and S. A. Seshia, "Model predictive control with signal temporal logic specifications," in *53rd IEEE Conference on Decision and Control*, pp. 81–87, IEEE, 2014.

[18] T. Wongpiromsarn, U. Topcu, and R. M. Murray, "Receding horizon temporal logic planning," *IEEE Transactions on Automatic Control*, vol. 57, no. 11, pp. 2817–2830, 2012.

[19] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," in *International Conference on Computer Aided Verification*, pp. 97–117, Springer, 2017.

[20] M. Fazlyab, M. Morari, and G. J. Pappas, "Probabilistic verification and reachability analysis of neural networks via semidefinite programming," in *2019 IEEE 58th Conference on Decision and Control (CDC)*, pp. 2726–2731, IEEE, 2019.

[21] M. Fazlyab, M. Morari, and G. J. Pappas, "Safety verification and robustness analysis of neural networks via quadratic constraints and semidefinite programming," *IEEE Transactions on Automatic Control*, 2020.

[22] H.-D. Tran, X. Yang, D. M. Lopez, P. Musau, L. V. Nguyen, W. Xiang, S. Bak, and T. T. Johnson, "NNV: The neural network verification tool for

deep neural networks and learning-enabled cyber-physical systems," in *International Conference on Computer Aided Verification*, pp. 3–17, Springer, 2020.

[23] T. Dreossi, S. Jha, and S. A. Seshia, "Semantic adversarial deep learning," in *International Conference on Computer Aided Verification*, pp. 3–26, Springer, 2018.

[24] S. A. Seshia, A. Desai, T. Dreossi, D. J. Fremont, S. Ghosh, E. Kim, S. Shivakumar, M. Vazquez-Chanlatte, and X. Yue, "Formal specification for deep neural networks," in *International Symposium on Automated Technology for Verification and Analysis*, pp. 20–34, Springer, 2018.

[25] T. Dreossi, A. Donzé, and S. A. Seshia, "Compositional falsification of cyber-physical systems with machine learning components," *Journal of Automated Reasoning*, vol. 63, no. 4, pp. 1031–1053, 2019.

[26] S. Topan, K. Leung, Y. Chen, P. Tupekar, E. Schmerling, J. Nilsson, M. Cox, and M. Pavone, "Interaction-dynamics-aware perception zones for obstacle detection safety evaluation," in *2022 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1201–1210, IEEE, 2022.

[27] K. Chakraborty and S. Bansal, "Discovering closed-loop failures of vision-based controllers via reachability analysis," *IEEE Robotics and Automation Letters*, vol. 8, no. 5, pp. 2692–2699, 2023.

[28] A. Dokhanchi, H. B. Amor, J. V. Deshmukh, and G. Fainekos, "Evaluating perception systems for autonomous vehicles using quality temporal logic," in *International Conference on Runtime Verification*, pp. 409–416, Springer, 2018.

[29] A. Balakrishnan, A. G. Puranic, X. Qin, A. Dokhanchi, J. V. Deshmukh, H. B. Amor, and G. Fainekos, "Specifying and evaluating quality metrics for vision-based perception systems," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1433–1438, IEEE, 2019.

[30] B. Bauchwitz and M. Cummings, "Evaluating the reliability of Tesla model 3 driver assist functions," 2020.

[31] H. Kress-Gazit, D. C. Conner, H. Choset, A. A. Rizzi, and G. J. Pappas, "Courteous cars," *IEEE Robotics & Automation Magazine*, vol. 15, no. 1, pp. 30–38, 2008.

[32] H. Kress-Gazit and G. J. Pappas, "Automatically synthesizing a planning and control subsystem for the DARPA Urban Challenge," in *2008 IEEE International Conference on Automation Science and Engineering*, pp. 766–771, IEEE, 2008.

[33] T. Wongpiromsarn, S. Karaman, and E. Frazzoli, “Synthesis of provably correct controllers for autonomous vehicles in urban environments,” in *2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, pp. 1168–1173, IEEE, 2011.

[34] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, “CARLA: An open urban driving simulator,” in *Conference on Robot Learning*, pp. 1–16, PMLR, 2017.

[35] D. J. Fremont, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli, and S. A. Seshia, “Scenic: a language for scenario specification and scene generation,” in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 63–78, 2019.

[36] Y. Annpureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan, “S-taliro: A tool for temporal logic falsification for hybrid systems,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 254–257, Springer, 2011.

[37] G. E. Fainekos and G. J. Pappas, “Robustness of temporal logic specifications for continuous-time signals,” *Theoretical Computer Science*, vol. 410, no. 42, pp. 4262–4291, 2009.

[38] G. E. Fainekos, S. Sankaranarayanan, K. Ueda, and H. Yazarel, “Verification of automotive control applications using s-taliro,” in *2012 American Control Conference (ACC)*, pp. 3567–3572, IEEE, 2012.

[39] S. Sankaranarayanan and G. Fainekos, “Falsification of temporal properties of hybrid systems using the cross-entropy method,” in *Proceedings of the 15th ACM international conference on Hybrid Systems: Computation and Control*, pp. 125–134, 2012.

[40] S. Bak, S. Bogomolov, A. Hekal, N. Kochdumper, E. Lew, A. Mata, and A. Rahmati, “Falsification using reachability of surrogate koopman models,” in *Proceedings of the 27th ACM International Conference on Hybrid Systems: Computation and Control*, HSCC ’24, (New York, NY, USA), Association for Computing Machinery, 2024.

[41] A. Donzé, “Breach, a toolbox for verification and parameter synthesis of hybrid systems,” in *International Conference on Computer Aided Verification*, pp. 167–170, Springer, 2010.

[42] C. E. Tuncali, G. Fainekos, H. Ito, and J. Kapinski, “Simulation-based adversarial test generation for autonomous vehicles with machine learning components,” in *2018 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1555–1562, IEEE, 2018.

[43] C. Menghi, P. Arcaini, W. Baptista, G. Ernst, G. Fainekos, F. Formica, S. Gon, T. Khandait, A. Kundu, G. Pedrielli, *et al.*, "Arch-comp 2023 category report: Falsification," in *10th International Workshop on Applied Verification of Continuous and Hybrid Systems. ARCH23*, vol. 96, pp. 151–169, 2023.

[44] T. Dreossi, D. J. Fremont, S. Ghosh, E. Kim, H. Ravanbakhsh, M. Vazquez-Chanlatte, and S. A. Seshia, "Verifai: A toolkit for the formal design and analysis of artificial intelligence-based systems," in *International Conference on Computer Aided Verification*, pp. 432–442, Springer, 2019.

[45] A. Corso, P. Du, K. Driggs-Campbell, and M. J. Kochenderfer, "Adaptive stress testing with reward augmentation for autonomous vehicle validatio," in *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pp. 163–168, IEEE, 2019.

[46] S. Feng, H. Sun, X. Yan, H. Zhu, Z. Zou, S. Shen, and H. X. Liu, "Dense reinforcement learning for safety validation of autonomous vehicles," *Nature*, vol. 615, no. 7953, pp. 620–627, 2023.

[47] X. Qin, N. Arechiga, J. Deshmukh, and A. Best, "Robust testing for cyber-physical systems using reinforcement learning," in *Proceedings of the 21st ACM-IEEE International Conference on Formal Methods and Models for System Design*, MEMOCODE '23, (New York, NY, USA), p. 36–46, Association for Computing Machinery, 2023.

[48] S. A. Seshia, D. Sadigh, and S. S. Sastry, "Toward verified artificial intelligence," *Commun. ACM*, vol. 65, p. 46–55, jun 2022.

[49] B. Johnson and H. Kress-Gazit, "Probabilistic analysis of correctness of high-level robot behavior with sensor error," 2011.

[50] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2019.

[51] X. Wang, R. Li, B. Yan, and O. Koyejo, "Consistent classification with generalized metrics," 2019.

[52] P. Antonante, H. Nilsen, and L. Carlone, "Monitoring of perception systems: Deterministic, probabilistic, and learning-based fault detection and identification," *arXiv preprint arXiv:2205.10906*, 2022.

[53] M. Hekmatnejad, S. Yaghoubi, A. Dokhanchi, H. B. Amor, A. Shrivastava, L. Karam, and G. Fainekos, "Encoding and monitoring responsibility sensitive safety rules for automated vehicles in signal temporal logic," in *Proceedings of the 17th ACM-IEEE International Conference on Formal Methods and Models for System Design*, pp. 1–11, 2019.

[54] T. Wongpiromsarn and E. Frazzoli, "Control of probabilistic systems under dynamic, partially known environments with temporal logic specifications," in *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, pp. 7644–7651, 2012.

[55] A. Badithela, T. Wongpiromsarn, and R. M. Murray, "Leveraging classification metrics for quantitative system-level analysis with temporal logic specifications," in *2021 60th IEEE Conference on Decision and Control (CDC)*, (Austin, TX, USA (virtual)), pp. 564–571, IEEE, 2021.

[56] C. S. Pasareanu, R. Mangal, D. Gopinath, S. G. Yaman, C. Imrie, R. Calinescu, and H. Yu, "Closed-loop analysis of vision-based autonomous systems: A case study," *arXiv preprint arXiv:2302.04634*, 2023.

[57] S. Beland, I. Chang, A. Chen, M. Moser, J. Paunicka, D. Stuart, J. Vian, C. Westover, and H. Yu, "Towards assurance evaluation of autonomous systems," in *Proceedings of the 39th International Conference on Computer-Aided Design*, pp. 1–6, 2020.

[58] Y. V. Pant, H. Abbas, K. Mohta, R. A. Quaye, T. X. Nghiem, J. Devietti, and R. Mangharam, "Anytime computation and control for autonomous systems," *IEEE Transactions on Control Systems Technology*, vol. 29, no. 2, pp. 768–779, 2021.

[59] P. Karkus, B. Ivanovic, S. Mannor, and M. Pavone, "Diffstack: A differentiable and modular control stack for autonomous vehicles," in *Proceedings of The 6th Conference on Robot Learning* (K. Liu, D. Kulic, and J. Ichnowski, eds.), vol. 205 of *Proceedings of Machine Learning Research*, pp. 2170–2180, PMLR, 14–18 Dec 2023.

[60] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[61] O. Koyejo, N. Natarajan, P. Ravikumar, and I. S. Dhillon, "Consistent multilabel classification.," in *NeurIPS*, vol. 29, (Palais des Congrès de Montréal, Montréal CANADA), pp. 3321–3329, Advances in Neural Information Processing Systems, 2015.

[62] M. Kwiatkowska, G. Norman, and D. Parker, "Prism 4.0: Verification of probabilistic real-time systems," in *International conference on computer aided verification*, pp. 585–591, Springer, 2011.

[63] C. Dehnert, S. Junges, J.-P. Katoen, and M. Volk, "A Storm is coming: A modern probabilistic model checker," in *International Conference on Computer Aided Verification*, pp. 592–600, Springer, 2017.

[64] H. Caesar, V. Bankiti, A. H. Lang, S. Vora, V. E. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, and O. Beijbom, "nuscenes: A multimodal dataset for autonomous driving," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11621–11631, 2020.

[65] A. H. Lang, S. Vora, H. Caesar, L. Zhou, J. Yang, and O. Beijbom, "Pointpillars: Fast encoders for object detection from point clouds," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, (Los Alamitos, CA, USA), pp. 12689–12697, IEEE Computer Society, jun 2019.

[66] M. Contributors, "MMDetection3D: OpenMMLab next-generation platform for general 3D object detection." `https://github.com/open-mmlab/mmdetection3d`, 2020.

[67] S. Gupta, J. Kanjani, M. Li, F. Ferroni, J. Hays, D. Ramanan, and S. Kong, "Far3det: Towards far-field 3d detection," in *2023 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, (Los Alamitos, CA, USA), pp. 692–701, IEEE Computer Society, jan 2023.

[68] I. Incer, A. Badithela, J. Graebener, P. Mallozzi, A. Pandey, S.-J. Yu, A. Benveniste, B. Caillaud, R. M. Murray, A. Sangiovanni-Vincentelli, *et al.*, "Pacti: Scaling assume-guarantee reasoning for system analysis and design," *arXiv preprint arXiv:2303.17751*, 2023.

[69] A. Badithela, T. Wongpiromsarn, and R. M. Murray, "Evaluation metrics of object detection for quantitative system-level analysis of safety-critical autonomous systems," in *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, (Detroit, MI, USA), p. To Appear., IEEE, 2023.

[70] A. Donzé and O. Maler, "Robust satisfaction of temporal logic over real-valued signals," in *International Conference on Formal Modeling and Analysis of Timed Systems*, pp. 92–106, Springer, 2010.

[71] E. Plaku, L. E. Kavraki, and M. Y. Vardi, "Falsification of ltl safety properties in hybrid systems," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 4, pp. 305–320, 2013.

[72] G. Chou, Y. E. Sahin, L. Yang, K. J. Rutledge, P. Nilsson, and N. Ozay, "Using control synthesis to generate corner cases: A case study on autonomous driving," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2906–2917, 2018.

[73] T. Wongpiromsarn, M. Ghasemi, M. Cubuktepe, G. Bakirtzis, S. Carr, M. O. Karabag, C. Neary, P. Gohari, and U. Topcu, "Formal methods for autonomous systems," *arXiv preprint arXiv:2311.01258*, 2023.

[74] G. Fainekos, H. Kress-Gazit, and G. Pappas, "Hybrid controllers for path planning: A temporal logic approach," in *Proceedings of the 44th IEEE Conference on Decision and Control*, pp. 4885–4890, 2005.

[75] R. Majumdar, A. Mathur, M. Pirron, L. Stegner, and D. Zufferey, “Paracosm: A language and tool for testing autonomous driving systems,” *arXiv preprint arXiv:1902.01084*, 2019.

[76] L. Tan, O. Sokolsky, and I. Lee, “Specification-based testing with linear temporal logic,” in *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, 2004. IRI 2004.*, pp. 493–498, IEEE, 2004.

[77] G. Fraser and F. Wotawa, “Using LTL rewriting to improve the performance of model-checker based test-case generation,” in *Proceedings of the 3rd International Workshop on Advances in Model-Based Testing*, pp. 64–74, 2007.

[78] G. Fraser and P. Ammann, “Reachability and propagation for LTL requirements testing,” in *2008 The Eighth International Conference on Quality Software*, pp. 189–198, IEEE, 2008.

[79] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[80] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[81] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger, “Specification patterns for robotic missions,” *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2208–2224, 2019.

[82] R. Bloem, G. Fey, F. Greif, R. Könighofer, I. Pill, H. Riener, and F. Röck, “Synthesizing adaptive test strategies from temporal logic specifications,” *Formal methods in system design*, vol. 55, no. 2, pp. 103–135, 2019.

[83] J. Tretmans, “Conformance testing with labelled transition systems: Implementation relations and test generation,” *Computer Networks and ISDN Systems*, vol. 29, no. 1, pp. 49–79, 1996.

[84] B. K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran, “Killing strategies for model-based mutation testing,” *Software Testing, Verification and Reliability*, vol. 25, no. 8, pp. 716–748, 2015.

[85] R. Hierons, “Applying adaptive test cases to nondeterministic implementations,” *Information Processing Letters*, vol. 98, no. 2, pp. 56–60, 2006.

[86] A. Petrenko and N. Yevtushenko, “Adaptive testing of nondeterministic systems with FSM,” in *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pp. 224–228, IEEE, 2014.

[87] A. Pnueli and R. Rosner, “On the synthesis of a reactive module,” in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 179–190, 1989.

[88] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive (1) designs," *Journal of Computer and System Sciences*, vol. 78, no. 3, pp. 911–938, 2012.

[89] M. Yannakakis, "Testing, optimization, and games," in *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.*, pp. 78–88, IEEE, 2004.

[90] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp, "Optimal strategies for testing nondeterministic systems," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 55–64, 2004.

[91] A. David, K. G. Larsen, S. Li, and B. Nielsen, "Cooperative testing of timed systems," *Electronic Notes in Theoretical Computer Science*, vol. 220, no. 1, pp. 79–92, 2008.

[92] E. Bartocci, R. Bloem, B. Maderbacher, N. Manjunath, and D. Ničković, "Adaptive testing for specification coverage in CPS models," *IFAC-PapersOnLine*, vol. 54, no. 5, pp. 229–234, 2021.

[93] T. Marcucci, J. Umenberger, P. Parrilo, and R. Tedrake, "Shortest paths in graphs of convex sets," *SIAM Journal on Optimization*, vol. 34, no. 1, pp. 507–532, 2024.

[94] T. Marcucci, M. Petersen, D. von Wrangel, and R. Tedrake, "Motion planning around obstacles with convex optimization," *Science Robotics*, vol. 8, no. 84, p. eadf7843, 2023.

[95] H. Zhang, M. Fontaine, A. Hoover, J. Togelius, B. Dilkina, and S. Nikolaidis, "Video game level repair via mixed integer linear programming," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 16, pp. 151–158, 2020.

[96] M. Fontaine, Y.-C. Hsu, Y. Zhang, B. Tjanaka, and S. Nikolaidis, "On the Importance of Environments in Human-Robot Coordination," in *Proceedings of Robotics: Science and Systems*, (Virtual), July 2021.

[97] J. R. Büchi, *On a Decision Method in Restricted Second Order Arithmetic*, pp. 425–435. New York, NY: Springer New York, 1990.

[98] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, É. Renault, and L. Xu, "Spot 2.0 — a framework for ltl and omega-automata manipulation," in *Automated Technology for Verification and Analysis* (C. Artho, A. Legay, and D. Peled, eds.), (Cham), pp. 122–129, Springer International Publishing, 2016.

[99] F. Fuggitti, "Ltlf2dfa," June 2020.

[100] S. Bansal, Y. Li, L. Tabajara, and M. Vardi, "Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 9766–9774, Apr. 2020.

[101] N. Klarlund and A. Møller, *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, University of Aarhus, January 2001. Notes Series NS-01-1. Available from `http://www.brics.dk/mona/`.

[102] D. Goktas and A. Greenwald, "Convex-concave min-max Stackelberg games," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[103] I. Tsaknakis, M. Hong, and S. Zhang, "Minimax problems with coupled linear constraints: computational complexity, duality and solution methods," *arXiv preprint arXiv:2110.11210*, 2021.

[104] M. L. Bynum, G. A. Hackebeil, W. E. Hart, C. D. Laird, B. L. Nicholson, J. D. Siirola, J.-P. Watson, and D. L. Woodruff, *Pyomo–optimization modeling in python*, vol. 67. Springer Science & Business Media, third ed., 2021.

[105] V. V. Vazirani, *Approximation algorithms*, vol. 1. Springer, 2001.

[106] M. Fischetti and M. Monaci, "A branch-and-cut algorithm for mixed-integer bilinear programming," *European Journal of Operational Research*, vol. 282, no. 2, pp. 506–514, 2020.

[107] J. B. Graebener, A. S. Badithela, D. Goktas, W. Ubellacker, E. V. Mazumdar, A. D. Ames, and R. M. Murray, "Flow-based synthesis of reactive tests for discrete decision-making systems with temporal logic specifications," *arXiv preprint arXiv:2404.09888*, 2024.

[108] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, and R. M. Murray, "Tulip: a software toolbox for receding horizon temporal logic planning," in *Proceedings of the 14th international conference on Hybrid systems: computation and control*, pp. 313–314, 2011.

[109] I. Filippidis, S. Dathathri, S. C. Livingston, N. Ozay, and R. M. Murray, "Control design for hybrid systems with tulip: The temporal logic planning toolbox," in *2016 IEEE Conference on Control Applications (CCA)*, pp. 1030–1041, IEEE, 2016.

[110] 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*, pp. 96–106, 2015.

[111] S. A. Cook, "The complexity of theorem-proving procedures," in *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*, pp. 143–152, 2023.

[112] C. H. Papadimitriou, *Computational complexity*, p. 260–265. GBR: John Wiley and Sons Ltd., 2003.

[113] W. Ubellacker and A. D. Ames, "Robust locomotion on legged robots through planning on motion primitive graphs," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 12142–12148, 2023.

[114] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2023.

[115] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, 1975.

[116] L. Lamport, "win and sin: Predicate transformers for concurrency," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 396–428, 1990.

[117] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[118] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, "Multiple viewpoint contract-based specification and design," in *Formal Methods for Components and Objects: 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures* (F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, eds.), (Berlin, Heidelberg), pp. 200–225, Springer Berlin Heidelberg, 2008.

[119] A. L. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-based design for cyber-physical systems," *Eur. J. Control*, vol. 18, no. 3, pp. 217–238, 2012.

[120] P. Nuzzo, A. L. Sangiovanni-Vincentelli, D. Bresolin, L. Geretti, and T. Villa, "A platform-based design methodology with contracts and related tools for the design of cyber-physical systems," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2104–2132, 2015.

[121] I. Incer, *The Algebra of Contracts*. PhD thesis, EECS Department, University of California, Berkeley, May 2022.

[122] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. L. Sangiovanni-Vincentelli, W. Damm, T. A. Henzinger, K. G. Larsen, *et al.*, "Contracts for system design," *Foundations and Trends in Electronic Design Automation*, vol. 12, no. 2-3, pp. 124–400, 2018.

[123] I. Incer, A. L. Sangiovanni-Vincentelli, C.-W. Lin, and E. Kang, "Quotient for assume-guarantee contracts," in *16th ACM-IEEE International Conference on Formal Methods and Models for System Design*, MEMOCODE'18, pp. 67–77, October 2018.

[124] R. Passerone, Í. Íncer Romeo, and A. L. Sangiovanni-Vincentelli, "Coherent extension, composition, and merging operators in contract models for system design," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–23, 2019.

[125] R. Negulescu, "Process spaces," in *CONCUR 2000 — Concurrency Theory* (C. Palamidessi, ed.), (Berlin, Heidelberg), pp. 199–213, Springer Berlin Heidelberg, 2000.

[126] J. B. Graebener^*, A. Badithela^*, and R. M. Murray, "Towards better test coverage: Merging unit tests for autonomous systems," in *NASA Formal Methods* (J. V. Deshmukh, K. Havelund, and I. Perez, eds.), (Cham), pp. 133–155, Springer International Publishing, 2022. A. Badithela and J.B. Graebener contributed equally to this work.

[127] R. Bloem, B. Könighofer, R. Könighofer, and C. Wang, "Shield synthesis," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 533–548, Springer, 2015.

[128] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European conference on machine learning*, pp. 282–293, Springer, 2006.

[129] I. Incer, L. Mangeruca, T. Villa, and A. Sangiovanni-Vincentelli, "The quotient in preorder theories," *arXiv:2009.10886*, 2020.

[130] O. Hussien, A. Ames, and P. Tabuada, "Abstracting partially feedback linearizable systems compositionally," *IEEE Control Systems Letters*, vol. 1, no. 2, pp. 227–232, 2017.

[131] P. Tabuada, G. J. Pappas, and P. Lima, "Composing abstractions of hybrid systems," in *International Workshop on Hybrid Systems: Computation and Control*, pp. 436–450, Springer, 2002.

[132] S. Coogan and M. Arcak, "Efficient finite abstraction of mixed monotone systems," in *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, HSCC '15, (New York, NY, USA), p. 58–67, Association for Computing Machinery, 2015.

[133] J. Liu and N. Ozay, "Abstraction, discretization, and robustness in temporal logic control of dynamical systems," in *Proceedings of the 17th international conference on Hybrid systems: computation and control*, pp. 293–302, 2014.