Formal Methods for Test and Evaluation: Reasoning over Tests, Automated Test Synthesis, and System Diagnostics

Thesis by
Josefine Berta Marie Graebener

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

Caltech

CALIFORNIA INSTITUTE OF TECHNOLOGY
Pasadena, California

2024
Defended April 30, 2024

ACKNOWLEDGEMENTS

I am truly grateful to the mentors, collaborators, friends, and family members who have all played an important role in making this work possible. I would like to start by thanking my advisor, Richard Murray, for his mentorship and guidance throughout my journey at Caltech. Being able to benefit from him sharing his time and wisdom has greatly shaped my views on research and allowed me to grow into the researcher that I have become today. I would also like to thank my candidacy and thesis committee members, Daniel Meiron, Soon-Jo Chung, Mani Chandy, and Joel Burdick, for their time and valuable feedback throughout this journey.

I am deeply grateful to my collaborators for sharing their brilliant ideas with me and creating a comfortable and supportive work environment. To Apurva Badithela, Inigo Incer, Wyatt Ubellacker, Denizalp Goktas, Leila Meshkat, Tung Phan-Minh, James Ragan, and Christian Stromberger, I have been lucky to have been able to work with each and every one of you and wish you the best in your adventures to come.

I would also like to thank Michele Judd, and the team from the Keck Institute of Space Studies. Michele, you have been an invaluable mentor and role model to me and I am extremely grateful that I got the opportunity to work with you and learn from you. I would also like to thank the rest of the Caltech Space Challenge team, Liam Heidt, Niyati Desai, Ying Luo, and Brayden Aller. It was truly an incredible experience to plan this successful (and fun) event with all of you.

This list would be incomplete without thanking my undergraduate thesis advisor, Markus Czupalla, for believing in me and putting me on this path to follow his mantra — get out, stand out— and pursue a graduate degree at Caltech. I would not be where I am today without his mentorship in my early days in academia.

To my friends, here at Caltech, and back home in Germany, thank you for being there for me, talking about research and life, and making sure that I always had fun along the way.

I would like to thank my family, my parents, Marion and Theodor, and my sister, Pauline, for all the love and support that they have given me throughout my entire life. Thank you for being my fiercest protectors and loudest cheerleaders.

To my husband, David, thank you for your unwavering love over the past 12 years. Regardless of any physical distance separating us, you have shown me more support and patience than I could have ever wished for and I cannot wait to see what our future holds.

Last but not least, I would like to thank my dog, Lumi, who has been my loyal companion for the past 4 years. She provided me company when I worked long hours and always made sure I took much-needed breaks to play fetch and clear my mind.

ABSTRACT

With the integration of autonomous systems into our everyday lives edging closer to reality, ensuring the safety of these systems is paramount. Part of the safety verification process is a rigorous testing procedure, which currently does not exist for autonomous vehicles. In this thesis, we aim to provide approaches using formal methods to increase the efficiency of testing campaigns. First, we provide a framework based on assume-guarantee contracts to specify tests in the form of a test structure. Using these test structures, we then show how to combine, split, and compare tests. Additionally, we characterize when tests can be combined and when the resulting test requires temporal constraints. Next, we demonstrate the approach on examples and find a strategy for a test agent using winning sets and Monte Carlo tree search.

Second, we present a framework to automatically synthesize a test environment, consisting of static and reactive obstacles, and dynamic test agents. We characterize the desired test behavior in a system and a test objective in the form of a linear temporal logic specification, consisting of sub-tasks commonly used for robotic missions. This test environment must ensure that the test is not impossible (i.e. a correct system can pass the test), but also that every test execution that satisfies the system objective also satisfies the test objective. We use tools from automata theory to construct the virtual product graph that represents all possible test executions, and the virtual system graph, which corresponds to the system's perspective. We formulate this routing problem as a network flow optimization on the virtual product graph in the form of a mixed integer linear program for different test environments. We show that this routing problem is NP-hard. We propose a counterexample-guided search using GR(1) synthesis to find a strategy for a test agent. This framework is demonstrated in several examples in simulation and hardware.

Lastly, we present a framework to diagnose a system-level fault by identifying the component responsible for the failure. We make use of assume-guarantee contracts and Pacti, a tool for compositional system analysis and design, to construct a diagnostics map, which allows us to trace a system-level guarantee to possible causes. We show that this framework can reduce the number of statements that need to be checked in the diagnostics process. We illustrate this framework on several abstract examples and two examples inspired by a real-world autonomous system.

PUBLISHED CONTENT AND CONTRIBUTIONS

[1] Josefine B Graebener, Apurva Badithela, Denizalp Goktas, Wyatt Ubellacker, Eric V Mazumdar, Aaron D Ames, and Richard M Murray. "Flow-Based Synthesis of Reactive Tests for Discrete Decision-Making Systems with Temporal Logic Specifications". In: *arXiv preprint* (2024). arXiv: `2404.09888 [cs.FL]`. URL: `https://arxiv.org/abs/2404.09888`.
J. B. G. participated in developing the problem statement and approach, working out the proofs, coding the implementation, and writing the manuscript.

[2] Apurva Badithela, Josefine B Graebener, Inigo Incer, and Richard M Murray. "Reasoning over Test Specifications using Assume Guarantee Contracts". In: *NASA Formal Methods: 15th International Symposium, NFM 2023, Houston, TX, USA, May 16–18, 2023, Proceedings*. 2023, tbd. URL: `https://link.springer.com/chapter/10.1007/978-3-031-33170-1_17`.
J. B. G. participated in the conception of the project, developing the problem statement and approach, coding the implementation, and writing the manuscript.

[3] Apurva Badithela, Josefine B Graebener, Wyatt Ubellacker, Eric V Mazumdar, Aaron D Ames, and Richard M Murray. "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)*. 2023, pp. 12430–12436. DOI: `10.1109/ICRA48891.2023.10160841`. URL: `https://ieeexplore.ieee.org/abstract/document/10160841`.
J. B. G. participated in developing the problem statement and approach, coding the implementation, and writing the manuscript.

[4] Josefine B Graebener, Apurva Badithela, and Richard M Murray. "Towards Better Test Coverage: Merging Unit Tests for Autonomous Systems". In: *NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, Proceedings*. 2022, pp. 133–155. URL: `https://dl.acm.org/doi/abs/10.1007/978-3-031-06773-0_7`.
J. B. G. participated in the conception of the project, developing the problem statement and approach, coding the implementation, and writing the manuscript.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 Motivation

Autonomous systems play an increasingly important role in today's society, and their impact is only expected to be even more substantial in the future. The development of autonomous systems has attracted significant attention in recent years, as the deployment of autonomy in real-world settings has increased. Self-driving cars are operating in traffic alongside human-operated cars, while robots and drones are used for package delivery and warehouse management. In the future, self-driving cars offer great potential to provide safe and efficient transportation, but the applications of autonomous systems are not solely for transportation needs. For example, autonomous systems could provide life-saving services in the medical field, or be used for the exploration of dangerous and hostile environments such as nuclear reactors or planetary surface exploration with new areas of operation emerging as the technological possibilities evolve. Key benefits of deploying autonomy in the transportation sector are expected to be a reduction in traffic congestion, increased safety on the roads, and providing mobility options for individuals who are unable to drive. With this great potential, designers of autonomous systems have a great responsibility to ensure that the autonomous systems that humans need to trust in critical situations are reliable and safe. More than 90% of traffic accidents are caused by human error [88, 140]. Eliminating the human factor has the potential to mitigate this problem, as humans routinely make errors that autonomy can prevent [8, 52]. However, in recent years, several traffic deaths have occurred due to autonomously driving vehicles [54]. These fatalities were caused by failures in perception or prediction of human behavior [102]. Currently, autonomous systems are not at the point where we can reap the benefits yet, as the risks are still significant. The safety of self-driving cars is a main concern, as these systems face unique challenges from the interaction of the system component, cybersecurity threats, perception issues, and sensor failures among others. Irrespective of the specific purpose and area of deployment, autonomous systems abide by the same core principles. In particular, the systems must be *self-aware*, i.e. be aware of their state and be able to make high-level decisions. Furthermore, the systems must be able to *adapt* to changes in their operating environment and by learning from past behavior, and *acuate* their plans without human intervention [70, 142]. These key characteristics require a significant paradigm shift during the development, verification, and validation process of a product. Among other challenges, the increase in complexity requires novel testing approaches, as the testing effort increases disproportionately with the complexity of a system [145].

Automotive Autonomy

A key industry that will benefit from the deployment of autonomy in their products is the automotive industry. Different levels of driving autonomy have been specified in the *Society of Automotive Engineers (SAE) Levels of Driving Automation* [133] and range from Level 0 (no driving automation) to Level 5 (full autonomy). A Level 0 vehicle can be equipped with driver assistance warning systems, such as blind spot warning or emergency braking. In Level 1, a vehicle can be equipped with lane centering or adaptive cruise control, while a vehicle in Level 2 has both features. Vehicles up to and including Level 2 are characterized as driver support features and need constant supervision, the human is the driver and is responsible for operating the vehicle. Vehicles in Level 3 and Level 4 are in the conditional autonomy class, where the cars can operate autonomously, but only under certain conditions. The autonomy of these vehicles is limited by geographic region or weather conditions, examples of these systems can include traffic jam chauffeurs or local driver-less taxis. A vehicle in Level 3 can still request the human to take over control, while in Level 4 the vehicle does not need to be equipped with a steering wheel or pedals at all. Level 5 is classified as fully autonomous, the vehicle can drive independently everywhere in all conditions. A high-level overview of the different levels is given in Figure 1.1.

Car manufacturers are reaching for autonomy in SAE Levels 4 and 5, but there is still a significant gap to bridge. Recently, Mercedes-Benz became the first car manufacturer that received state approval to sell a SAE Level 3 conditionally automated vehicle, as a standard-production vehicle to U.S. customers. California and Nevada are the first two states that certified the use of this system on major freeways up to a speed of 40 miles per hour during daytime. Apart from Mercedes-Benz, Nuro Inc. and Waymo are also authorized by the Department of Motor Vehicles (DMV) for the deployment of autonomous vehicles in certain regions of California. Currently, six companies hold autonomous vehicle driverless testing permits from the DMV in California. Most of these autonomous driving companies focus on developing their vehicles to provide autonomous ridesharing services as an eco-friendly and

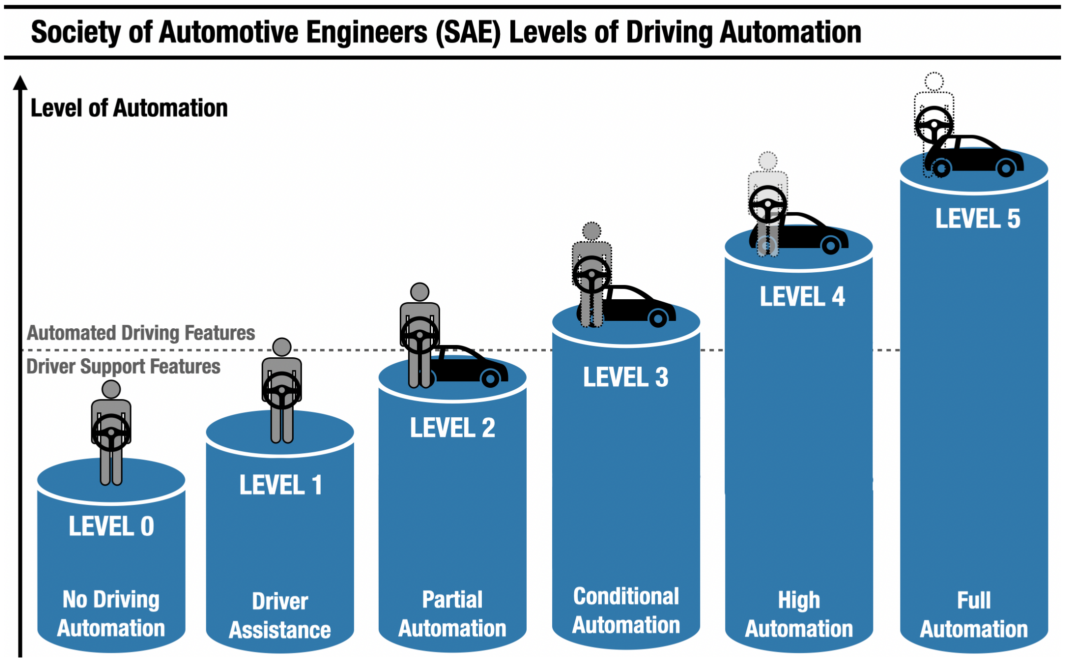


Figure 1.1: SAE Levels of driving automation.

cost-effective alternative.

Safety Standards

The international standard for functional safety for road vehicles (ISO26262) [84], mandates a functional safety development process and corresponding documentation that any car manufacturer and supplier must comply with to certify their product for deployment in a commercial passenger vehicle. It outlines a risk classification system and aims to reduce the hazards caused by malfunctioning electric or electronic systems. A crucial part of the development framework outlined in the functional safety standard is the verification and validation (V&V) processes. Verification pertains to answering the question: ‘Did we build the system correctly?’, while validation answers the question of: ‘Did we build the correct system?’. The process of verification ensures that the product meets the specified requirements and fulfills the intended purpose. Techniques for verification involve reviewing requirements and analyzing the system via simulation, modeling, or formal methods. In addition, verification tests include unit testing, integration testing, and model-based testing. During the validation process, the system is evaluated for its intended purpose in real-world conditions. This can include track testing where scenarios are constructed and executed to test the system in a controlled version environment, or on-the-road testing where the driving performance of the system is monitored in

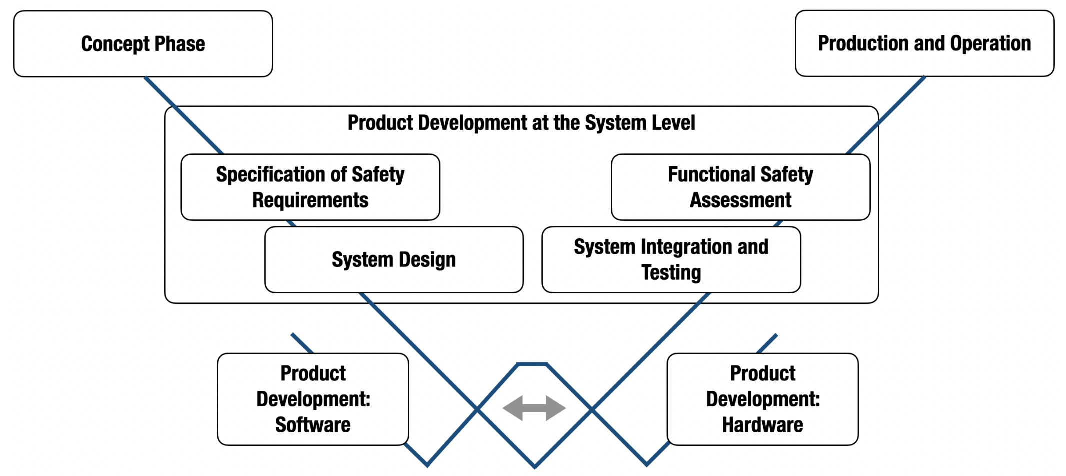

Concept Phase
Production and Operation
Product Development at the System Level
Specification of Safety Requirements
Functional Safety Assessment
System Design
System Integration and Testing
Product Development: Software
Product Development: Hardware

Figure 1.2: V-model according to ISO 26262, functional safety standard in road vehicles.

its actual operating environment. This comprehensive testing process, in addition to the stringent development process, aims to provide confidence in the system's adherence to the safety requirements. In its current form, the functional safety standard only applies to cars that are controlled by a driver, therefore it does not apply to fully autonomous cars yet. However, as the landscape of autonomous cars evolves the standard is expected to evolve to pertain to fully autonomous cars as well. As autonomous vehicles rely on external data provided by non-deterministic algorithms, even without the presence of faults, the system's behavior might be hazardous due to performance limitations [90]. To mitigate this, the safety of the intended functionality (SOTIF), which is defined as the absence of unreasonable risk due to a hazard caused by functional insufficiencies, was introduced in standard ISO PAS 21448 [85]. This must be checked for the entire Operational Design Domain (ODD) of the system. While there are other ongoing efforts on standardization [100, 157], there is no standard development process for autonomous vehicles yet [3, 70, 94, 97, 119]. Thus, verification and validation are especially critical to ensure safety of the system.

1.2 Test and Evaluation

Challenges for Testing Autonomy

The development process of an autonomous system is a very time- and cost-intensive task, which requires a rigorous testing procedure which is designed to verify the claims of the system designer. The conventional testing methods of fault avoidance, fault removal, and fault tolerance are not applicable for an autonomous system, as the system's behavior and the operating environment can change over time [70, 109]. In [97], the authors identified five key challenges for testing autonomous vehicles: driver out of the loop, complex requirements, non-deterministic algorithms, inductive learning algorithms, and fail-operational systems. For a fully autonomous vehicle, the driver cannot be counted on to handle exceptional situations, such as mechanical failures and unforeseen operating conditions. Including this fault and exception handling as a task handled by the computer is likely to dramatically increase the system complexity compared to a system that simply assists a driver but can still fall back on the driver to take over [76, 97]. Furthermore, the system requirements are complex and partially unknown. Along with failure handling, the operational environment in a traffic situation can exhibit hazards including but not limited to inclement weather conditions, animal hazards, and other drivers committing traffic rule violations. These adverse events are numerous, can occur in any combination, and are impossible to capture in a classical written requirement [70, 97]. As an autonomous system's behavior changes, the system's behavior is non-deterministic and likely cannot be reproduced, which poses another significant challenge. This leads to difficulty in testing specific edge cases and presenting exactly the right conditions to reliably prompt the system to exhibit the desired behavior. Due to the same reason, evaluating tests is difficult, as the correct system behavior is not unique and thus requires multiple tests to increase the confidence in the system's performance [70, 97, 142]. Using learning techniques poses other significant challenges for the testing process [28, 102, 130]. Learning relies on data to derive the model, making the quality and diversity of training data critical to the performance of the system [142]. Another significant limitation of machine learning is that the resulting behavior is not understandable to humans, presenting a significant challenge in validating the system's safety [97, 161]. Moreover, an autonomous system is required to be fail-operational, which requires redundancy in the system's design. The structure of the redundancy depends on the fault model and the design approach and can consist of three or more components whose failures need to be independent and fault-free at the start of each mission [68]. One can argue that an autonomous vehicle does not require the same level of redundancy as an aircraft, as the failover mission time is significantly shorter (i.e. come to a stop or pull over instead of flying to the nearest airport, possibly hours away). [76, 97]

From the non-technical viewpoint, several other challenges need to be overcome, such as legal and ethical questions [52, 139, 154], and user concerns [127]. Depending on the proposed solutions to these questions, technical considerations may change or new technical questions might arise that come with additional testing concerns [97].

Current Testing Approaches

Currently, there are several different approaches focused on to testing and evaluation of autonomous systems, each aiming to address different challenges, albeit with their limitations and difficulties. As autonomous systems are inherently hybrid, many testing approaches rely on techniques used for software testing.

One approach is *statistical analysis*, where autonomous vehicles are deployed on the road and data is collected. A key element of this testing procedure is the 'time between interventions', which refers to the interval during which the autonomous car operates without intervention from a safety driver or other external assistance. Human intervention can be required due to safety concerns, technical issues, or other unexpected events where the autonomous car cannot handle the situation independently. Autonomous car companies try to increase the time between interventions, as this is a measure of how well the car can operate independently in real-world conditions. According to a study by the RAND corporation, an autonomous vehicle fleet needs to drive for 275 million miles without any safety concerns or incidents to demonstrate that its maximum failure rate is that of a human driver with 95% confidence, where the human driver benchmark corresponds to a failure rate of 1.09 fatalities per 100 million miles driven [88]. Depending on the degree of confidence and the rate of improvement over the human driver benchmark, these estimates range from hundreds of millions of miles to billions of miles. Thus, reaching this threshold as the only source of confidence in the safety of the vehicle is impractical. In 2023, Waymo's level 4 rider-only operations reached the milestone of one million miles driven and published a report analyzing the failure rate [153]. So far, these on-the-road operations show promise to increase the safety of driving, but more time is required for the sample size to grow to increase confidence in their operations. To alleviate the time and cost issues arising in this approach, virtual testing of the system in simulation can be applied. Another real-world testing approach is *track testing*, where specific test cases are pre-planned and executed on the hardware on a private test track, where the operational environment consists of agents controlled by the test engineers that execute the test plans [156, 162]. Another example of track-testing were the qualification tests for the DARPA Urban Challenge [32].

Any virtual testing requires the use of *simulation techniques*, which are a key component of any test and evaluation process. Simulation allows us to visualize and evaluate a virtual test, and recreate specific test cases, while also reducing the hardware cost and improving testing efficiency. Nevertheless, a simulation does not necessarily align with the real-world operational environment, which makes the accuracy of a simulation vary. Popular tools for the simulation of dynamic systems are MATLAB [77] and Simulink [46], and ROS [128] and Gazebo [96] for the simulation of robotic systems, among others. For autonomous driving applications, simulation tools include Carla [48], SUMO [21], SYNCHRO [75], CarMaker [35], Prescan [141], CarSim [12], and Autoware [89].

Another testing approach is *scenario-based* testing, where a test scenario is specified. A scenario is a pre-defined temporal sequence of events and actions in the ODD of a system that can be executed on the system in simulation or hardware [151]. The level of detail of the description of the scenario can vary, and languages and tools for describing and rendering scenarios in simulation have been developed, including Scenic [60] and OpenSCENARIO [116]. Scenario-based testing has been widely applied to testing automated vehicles, including large research projects [86, 157]. Currently, test scenarios are designed by test engineers, who rely on their product know-how and experience, or by analyzing crash data and recreating traffic scenarios [143]. Finding scenarios is an active area of research and many other testing techniques are used to identify interesting and challenging test scenarios [132]. One approach to try and reduce the number of test scenarios makes use of *combinatorial testing*, where input parameters are combined and adjusted [98]. Combinatorial testing for autonomous vehicles has been studied in [101]. Scenario parameters can also be modified using a *fuzz testing* approach, in use at Waymo [155]. Another related technique is *fault injection*, where faulty parameter values (e.g. inaccurate sensor information) are provided to the system.

A Model-Based Systems Engineering approach can be beneficial during the development process, as it can reduce the number of errors introduced in a product [51, 70]. This leads to *model-based testing* as a valuable tool for quality assurance, where the system's capabilities and limitations are modeled, and this model can be used to generate test cases [91]. The employed models vary and can include modeling languages such as the Unified Modeling Language (UML) [73, 126] and the System Modeling Language (SysML), Petri nets [9, 134], belief-desire-intention (BDI) models [11, 113] models, behavioral models [67], among others. Fault-tree analysis for test generation has been studied in [93]. Coupling model-based testing with *search-based techniques* has been studied in [22, 25, 87]. Other search-based testing techniques include using genetic algorithms [2, 62, 108, 137], or surrogate models such as Gaussian process regression [5, 92, 111], neural networks [49], and Markov Decision Processes and reinforcement learning [40, 55].

Formal methods are mathematically rigorous techniques for the specification and verification of a system. The formal specifications are precise and unambiguous and can be used to prove properties of the system, often with the use of powerful tools such as model checkers and automated theorem provers. After several incidents of system failures, including the Ariane 5 launch vehicle, and Mars Pathfinder, later investigations have revealed that formal verification procedures would have revealed the fault ahead of time [15]. As autonomous systems always include a software component, formal methods hold promise to tackle the aforementioned verification and testing challenges. Formal methods have been used specifically to support software testing [72]. Using a *model-checker* for test generation was proposed for software testing [7, 34, 50] and has become a widely used technique for testing of autonomous systems [59]. Test cases are constructed from counterexamples generated by model-checkers but can be inconclusive if the system behavior deviates from the expected behavior. Linear temporal logic (LTL) model checkers have been used to find test cases in [57, 58, 123, 146]. Approaches using model checkers are commonly *white box*, as they rely on a model of the system being known to the tester. Nevertheless, *black box* approaches combine model checking with other techniques to allow reasoning over a (partially) unknown implementation of a system [120]. To allow test case generation for a system under test that reacts to its environment, adaptive specification-based testing strategies have been employed in [4, 27, 71, 121, 148]. In [27], the authors combine the use of a model checker and reactive synthesis [125] to find an adaptive test strategy from an LTL specification of the system under test and a fault model. *Runtime verification* [18] is another formal verification technique, where the system's behavior is constantly monitored during operation or simulation [17, 106]. Another approach to finding test scenarios uses *falsification*, where the scenario parameters are found such that a metric of mission success is violated [20, 61]. This metric can be in the form of formal temporal logic specifications [1, 10, 47, 53, 61, 149, 83].

Viewing testing as a game between the system under test and the tester, where the system tries to hide faults, and the tester tries to find them, has been explored in [6, 26, 160] and in cooperative game settings in [16, 41].

Rigorous test campaigns have to be designed, implemented, and executed to aid in the certification of safety-critical autonomous systems [138]. The above-mentioned techniques show promising initial results to improve the testing of autonomous systems. Nevertheless, testing complex autonomous systems remains a key challenge that needs to be solved to achieve human confidence in the system's behavior in a real-world setting.

In this thesis, the proposed frameworks and approaches are complementary to scenario generation and techniques such as falsification. We aim to provide a formal description of a test, to allow reasoning over tests, and to automatically synthesize test environments from test descriptions. We want to find these test environments and strategies that ensure that the desired test behavior is observed, while also making sure that the test is not impossible (i.e. a correct system can pass the test), and minimally interfering with the system.

System Diagnostics

Any safety-critical system requires efficient detection of system abnormalities and faults to avoid dangerous situations and identify safety hazards. For this reason system diagnostics have been an active area of research, only becoming more important since autonomous systems have become more prevalent [36]. Faults are defined as a deviation from the correct, expected system behavior, observable in at least one system property or parameter [136]. Therefore, fault diagnosis differs from other system verification techniques described previously, as it considers systems whose behavior deviates from the system model. Fault diagnosis consists of three areas: fault detection, fault isolation, and fault identification [63]. Fault detection refers to identifying when and where a fault occurs, from the observable system output. Fault isolation considers the location of the fault, and fault identification refers to finding the type, shape, and size of the fault. There are four main fault diagnosis methods: model-based, signal-based, knowledge-based, hybrid, and active. In 1971, Beard introduced Model-based fault diagnosis with the intent to replace hardware redundancy by analytical redundancy [19]. Model-based fault diagnosis uses different techniques to monitor the actual system outputs and compare them to the predicted values. Signal-based fault diagnosis uses measured signals instead of input-output models, and extracts features (or patterns) from a signal to make a diagnostic decision [63]. Knowledge-based fault diagnosis consists of a knowledge base and an inference engine [37]. Hybrid approaches are a combination of the above mentioned methods [63]. Active fault detection is concerned with designing auxiliary input

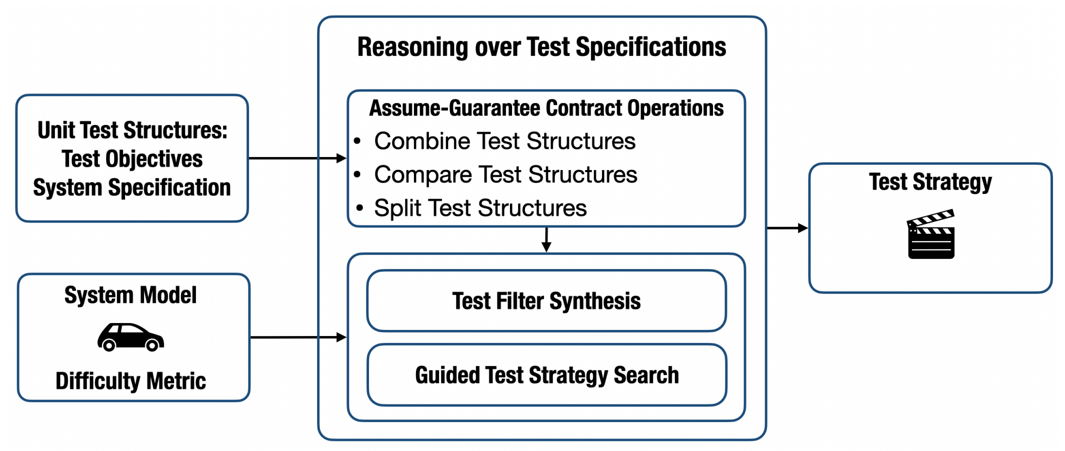

Reasoning over Test Specifications
Unit Test Structures: Test Objectives System Specification
Assume-Guarantee Contract Operations
• Combine Test Structures
• Compare Test Structures
• Split Test Structures
Test Strategy
System Model
Difficulty Metric
Test Filter Synthesis
Guided Test Strategy Search

Figure 1.3: Overview of the framework presented in Chapter 3.

vectors to reveal faults [114]. Diagnostics has been studied extensively in the area of computer science and engineering. Some early and and heavily influential works include [44] and [131]. In [44], a model-based approach to diagnose faults in complex systems by observing symptoms and using reasoning techniques was introduced in 1987 by De Kleer and Williams. In [131], formal logical framework to diagnose faults consisting of three main components: a knowledge base, an observation base, and a set of inference rules was developed by Reiter in 1987. In formal methods, the problem of explaining why for certain robot specifications no implementing control strategy exists has been studied in [129], while 'repairing' specifications has been studied in [29]. Recently, assume-guarantee contract operators have been used for specification repair in [103].

In this thesis we present an approach to system diagnostics using assume-guarantee contract theory and available tools that perform contract operations. The proposed approach aims to increase the efficiency of the diagnostics process by identifying which information is relevant to identify the cause of the fault.

1.3 Summary of Contributions

This thesis aims to provide a framework to systematically construct tests and the corresponding test environments. The contributions of this thesis are presented in the following paragraphs.

In Chapter 3, a framework that allows for reasoning over tests is established. This is done by characterizing a test structure in the form of assume-guarantee contracts. Test structures contain a formal description of the test objective and the system under test, and can be used to combine, or split tests accordingly. In addition, test structures

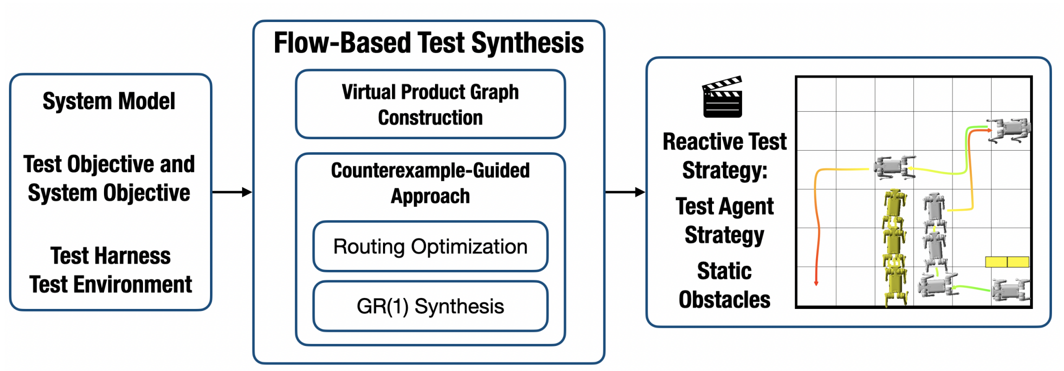


Figure 1.4: Overview of the framework presented in Chapter 4.

and test campaigns can be compared to allow for the selection of the most desirable test. These concepts are illustrated in examples. In practice, executing a test for a combined objective does not necessarily provide the information required to be confident in the system's performance concerning each task individually. To mitigate this, temporal constraints are introduced that ensure that each test objective will be observed individually. An approach to finding a controller for test agents using receding horizon winning set synthesis and Monte Carlo tree search is presented and demonstrated in two examples.

In Chapter 4, a more general approach to test generation is presented. This approach allows for testing objectives characterized by specification sub-tasks in reachability, reaction, and avoidance form. The test environment is characterized as consisting of static obstacles, reactive obstacles, and dynamic test agents. We leverage automata theory to represent the satisfaction of the test objectives in graph form and utilize network flow optimization to find restrictions on system actions for different test environments. The problem of finding these restrictions is shown to be NP-hard. These restrictions can be placed on the system using the test environment. For this, we present an algorithm that couples the optimization with controller synthesis using a counter-example guided search to find the controller for the dynamic test agent from the result of the optimization. Lastly, this approach is demonstrated in several examples including hardware examples.

In Chapter 5, a framework to diagnose faulty components from a system-level violation based on assume-guarantee contracts is proposed. This framework utilized Pacti, a tool for compositional analysis and design, that allows computing contract operations and augmenting it to trace the system-level guarantee to the responsible component. We show how to compose the components to create a system

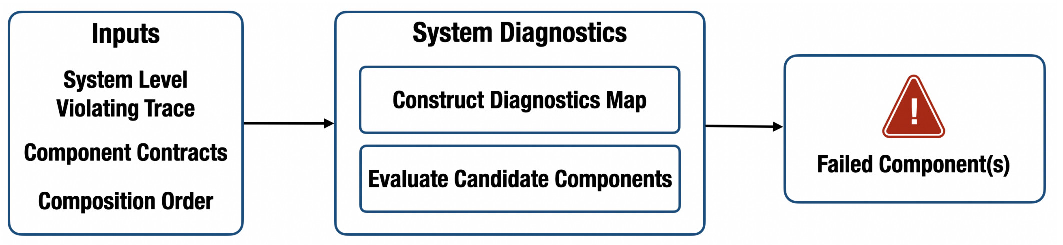

Inputs
System Level Violating Trace
Component Contracts
Composition Order
System Diagnostics
Construct Diagnostics Map
Evaluate Candidate Components
Failed Component(s)

Figure 1.5: Overview of the framework presented in Chapter 5.

and the corresponding system-level contract. During the composition of the system, we identify the information required to map the system-level guarantees to component-level terms. We present an algorithm to systematically trace the system-level violation back to the cause and show that with this procedure the number of statements to be evaluated during the diagnostics process can be reduced. Lastly, we demonstrate this process in several abstract examples and two examples inspired by real-world test cases for an autonomous vehicle.

Chapter 2

MATHEMATICAL BACKGROUND

In this section we will introduce the mathematical background and concepts that the approaches developed in this work build on.

2.1 Assume-Guarantee Contracts

To reason about the specifications, we will make use of the contract-based-design framework first introduced as a design methodology for modular software systems [45, 99, 107] and later extended to complex cyber-physical systems [115, 135]. We will adopt the mathematical framework presented by Benveniste et al. [24] and Passerone et al. [118].

Definition 2.1 (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 $C = (A, G)$, where A are the assumptions and G are the guarantees. A component E is an *environment* of the contract C if $E \models A$. A component M is an *implementation* of the contract, $M \models C$ 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 C_1 is a refinement of C_2, denoted $C_1 \leq C_2$, if $(A_2 \leq A_1)$ and $(G_1 \cup \neg A_1 \leq G_2 \cup \neg A_2)$. We say a contract $C = (A, G)$ is in *canonical*, or *saturated*, form if $\neg A \subseteq G$.

Multiple operations are known for assume guarantee contracts — see [78]. Assume the following contracts are in canonical form. The *meet* or *conjunction* of two contracts exists [23] and is given by $C_1 \wedge C_2 = (A_1 \cup A_2, G_1 \cap G_2)$. Composition [24] yields the specification of a system given the specifications of the components:

$$\begin{tikzcd} C & \text{iff} & C/C' \\ C' \parallel C_1 \arrow[u] & & C_1 \arrow[u] \end{tikzcd}$$

(a) Composition and quotient.

$$\begin{tikzcd} & C_1 \bullet C_2 & \\ C_1 & C_1 \parallel C_2 \arrow[u] & C_2 \\ & C_1 \wedge C_2 \arrow[u] \arrow[ul] \arrow[ur] & \end{tikzcd}$$

(b) Order of operations.

Figure 2.1: Contract operators and the partial order of their resulting objects.

$C_1 \parallel C_2 = ((A_1 \cap A_2) \cup \neg(G_1 \cap G_2), G_1 \cap G_2)$. Given specifications C and C_1, the quotient is the largest specification C_2 such that $C_1 \parallel C_2 \leq C$. It is given by [82]: $C/C_1 = (A \cup G_1, (G \cap A_1) \cup \neg(A \cup G_1))$. Strong merging [118] yields a specification obeyed by a system that obeys two given specifications C_1 and C_2: $C_1 \bullet C_2 = (A_1 \cap A_2, (G_1 \cap G_2) \cup \neg(A_1 \cap A_2))$. The reciprocal (or mirror) [112, 118] is a unary operation which inverts assumptions and guarantees: $C^{-1} = (G, A)$.

2.2 Linear Temporal Logic and Automata Theory

To state requirements of the system and the test in the form of a specification, we will make use of linear temporal logic (LTL). LTL is a temporal logic describing linear-time properties, allowing reasoning over the timing of events, where each point in time has a single successor. The use of LTL for formally verifying properties of computer programs was first introduced by Pnueli in 1977 [124].

Definition 2.2 (Linear Temporal Logic (LTL) [15])**.** The syntax of *linear temporal logic (LTL)* is given as:

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

with $a \in AP$, where AP is the set of atomic propositions, the Boolean connectors conjunction $\wedge$ and negation $\neg$, and the temporal operators 'next' $\bigcirc$ and 'until' $\mathcal{U}$. From conjunction and negation, we can derive the entirety of propositional logic including disjunction $\vee$, implication $\rightarrow$, and equivalence $\leftrightarrow$. The temporal operators 'always' $\square$ and 'eventually' $\diamond$ can be derived from $\mathcal{U}$ as

$$\diamond \varphi = \mathit{True}\, \mathcal{U}\, \varphi, \quad \square\varphi = \neg \diamond \neg\varphi.$$

From these temporal operators, we can derive 'always eventually' $\square\diamond$ and 'eventually always' $\diamond\square$, which specify that a proposition will be true infinitely often (progress) or eventually forever (stability) respectively. Let φ be an LTL formula over AP. The semantics of LTL formula φ are defined over an infinite word $\sigma = s_0 s_1 \cdots$ as follows:

$\sigma \models \mathit{True}$,

For $a \in AP$, $\sigma \models a$ iff $\sigma_0 \models a$,

$\sigma \models \varphi_1 \wedge \varphi_2$ iff $\sigma \models \varphi_1$ and $\sigma \models \varphi_2$,

$\sigma \models \neg\varphi$ iff $\sigma \not\models \varphi$,

$$\sigma \models \bigcirc\varphi \text{ iff } \sigma[1, \cdots] = s_1 s_2 \cdots \models \varphi,$$

$$\sigma \models \varphi_1 \mathcal{U} \varphi_1 \text{ iff } \exists j \geq 0, \sigma[j, \cdots] \models \varphi_2 \text{ and } \sigma[i, \cdots] \models \varphi_1, \text{for all } 0 \leq i < j,$$

where $\sigma[j, \cdots]$ denotes the word fragment $s_j s_{j+1} \cdots$.

The class of generalized reactivity of rank 1 (GR(1)) [122] is a fragment of LTL specifications and can capture safety ($\Box$), liveness ($\Diamond$), and recurrence ($\Box\Diamond$) requirements. A GR(1) formula φ for a robotic system s is given as follows:

$$\varphi = (\varphi_e^{\text{init}} \wedge \Box\varphi_e^s \wedge \Box\Diamond\varphi_e^f) \rightarrow (\varphi_s^{\text{init}} \wedge \Box\varphi_s^s \wedge \Box\Diamond\varphi_s^f), \tag{2.1}$$

where φ_s^{init}, $\Box\varphi_s^s$, and $\Box\Diamond\varphi_s^f$, define the initial, safety and recurrence requirements on the system s respectively. Similarly, φ_e^{init}, $\Box\varphi_e^s$, and $\Box\Diamond\varphi_e^f$, define requirements on the environment e of the system s. The time complexity of GR(1) synthesis is $O(|V|^3)$, where $|V|$ is the size of the state space [122].

Definition 2.3 (Finite Transition System)**.** A *finite transition system* (FTS) is the tuple

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

where S denotes a finite set of states, A is a finite set of actions, $\delta : S \times A \rightarrow S$ the transition relation, S_0 the set of initial states, AP the set of atomic propositions, and $L : S \rightarrow 2^{AP}$ denotes the labeling function.

Definition 2.4 (Deterministic Büchi Automaton)**.** A *non-deterministic Büchi automaton* (NBA) [31] 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, $\delta : Q \times \Omega \rightarrow Q$ 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 $A \in \Omega$.

For every LTL formula φ, there exists an equivalent non-deterministic Büchi automaton that can be converted into a deterministic Büchi automaton [15].

Definition 2.5 (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 $A \in \Omega$, $\delta((q_1, q_2), A) = (q_1', q_2')$ where $\mathcal{B}_1.\delta(q_1, A) = q_1'$ and $\mathcal{B}_2.\delta(q_2, A) = q_2'$.

Definition 2.6 (Product of Büchi Automaton and Finite Transition System)**.** The *product* of a deterministic Büchi automaton $\mathcal{B}$ and a finite transition system T, where the alphabet of $\mathcal{B}$ is the labels of T, is the transition system $T \otimes \mathcal{B} = (S, A, \delta, S_0, AP, L)$, with the set of states $S \coloneqq T.S \times \mathcal{B}.Q$, the initial states $S_0 \coloneqq \{(s_0, q) \mid s_0 \in T.S_0, \exists q_0 \in \mathcal{B}.Q_0 \text{ s.t. } \mathcal{B}.\delta(q_0, T.L(s_0)) = q\}$, the set of actions $A \coloneqq T.A$, the set of atomic propositions $AP \coloneqq \mathcal{B}.Q$, the labeling function L, where for all $(s, q) \in S$ we have $L((s, q)) \coloneqq \{q\}$. The transition relation δ, where for all $s, s' \in T.S$, for all $q, q' \in B.Q$ we say that $\delta((s, q), a) \coloneqq (s', q')$ iff there exists $a \in T.A$ such that $T.\delta(s, a) = s'$ and $\mathcal{B}.\delta(q, T.L(s')) = q'$.

2.3 Flow Networks

In this work we will leverage flow networks, a concept from graph theory, to represent different possible test executions. In particular, flow networks are directed graphs consisting of vertices and edges, each edge with a corresponding capacity and a flow, and a set of source vertices and sink vertices [56]. They can be used to model any kind of material transport, such as traffic on a road network, resource allocation, or electricity in a network of circuits. Flow networks also allow for reasoning over multiple different flows in a multi-commodity flow network, where different commodities compete for edge capacity [74]. For the purpose of this work we will assume unit edge capacity for all networks and a single flow.

Definition 2.7 (Flow Network [39])**.** A *flow network* is a tuple $\mathcal{N} = (V, E, (V_S, V_T))$, where V denotes the set of vertices, $E \subseteq V \times V$ the set of edges, $V_S \subseteq V$ the source vertices, and $V_T \subseteq V$ the sink vertices. On the flow network $\mathcal{N}$, we can define the *flow* vector $\mathbf{f} \in \mathbb{R}^{|E|}_{\geq 0}$ that has to satisfy the standard flow constraints. These are defined as follows: First, the capacity constraint ensures that the flow on an edge will not exceed the capacity,

$$0 \leq f^e \leq 1, \forall e \in E. \tag{2.2}$$

Second, flow conservation is defined for each vertex as

$$\sum_{u \in V} f^{(u,v)} = \sum_{u \in V} f^{(v,u)}, \forall v \in V \setminus \{V_s, V_t\}. \tag{2.3}$$

Third, no flow into the source or out of the sink is permitted:

$$f^{(u,v)} = 0 \text{ if } u \in V_T \text{ or } v \in V_S. \tag{2.4}$$

The flow value on the network $\mathcal{N}$, also denoted as the *total flow*, is defined as

$$F := \sum_{\substack{(u,v)\in E,\\ u\in V_s}} f^{(u,v)}. \tag{2.5}$$

Chapter 3

REASONING OVER TEST SPECIFICATIONS

3.1 Introduction

In this chapter, we focus on developing a formal approach to assist test engineers in reasoning over test campaigns. We propose an approach rooted in assume-guarantee contract algebra to reason over tests as pairs of contracts, a test structure describing the test objective in the form of a specification, and the corresponding system specification. These specifications are given at a high level of abstraction characterizing the desired test behavior. Assuming this high-level test description is provided by a test engineer, we aim to bridge the gap between manual test generation and falsification (starting from an already specified scene) by operating on this high-level test description. From this test structure, we can generate the tester specification which can be used to synthesize the required test environment. Additionally, we can reason over test structures by applying assume-guarantee contract operators to combine, split, and compare tests.

This framework aims to reduce the complexity of test campaigns by providing a systematic approach to reason over test objectives and the corresponding system specifications. The work in this chapter was published in [13, 65], and was done in collaboration with Apurva Badithela and Inigo Incer.

3.2 Characterizing a Test

To define a test, we need information about i) the system under test and its specification to be tested and ii) specifications for the test environment that characterize the desired behavior that should be observed during the test. We assume that the desired test behavior is provided in the form of a specification by the test engineer, while the system specification is provided by the system designer. One key concept of this framework is that the test objective is unknown to the system since doing so would reveal the test strategy to the system and consequently would impair the significance of the test. The system specification states assumptions about the test environment in which the system is expected to operate. From the test objective, together with the system specification, we can generate a specification for the test environment from which we can synthesize the corresponding strategies of the tester agents under certain conditions.

We define the test objective as an assume-guarantee contract describing the desired test behavior as defined below.

Definition 3.1 (Test Objective)**.** A *test objective* $C^{\text{obj}} = (\mathit{True}, G^{\text{obj}})$ is an assume-guarantee contract, where G^{obj} characterizes the set of desired test behaviors, and it contains a formal description of the specific behaviors that the test engineer would like to observe during the test.

An example of a test objective could be waiting for a car to pass during an unprotected left turn, navigating a busy T-intersection, or performing an emergency braking maneuver. While the system is aware of its task (e.g. reach a goal location) it might not be aware of where or when specific events will take place during the test.

Definition 3.2 (System Specification)**.** The *system specification* is an assume-guarantee contract denoted by $C^{\text{sys}} = (A^{\text{sys}}, G^{\text{sys}})$, 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 provide in an environment where A^{sys} is satisfied. In particular, A^{sys} are the assumptions requiring a safe test environment, and $\neg A_i^{\text{sys}} \cup G_i^{\text{sys}}$ are the guarantees on the specific subsystem that will be tested:

$$C^{\text{sys}} = (A^{\text{sys}}, \neg A^{\text{sys}} \cup \bigcap_i (\neg A_i^{\text{sys}} \cup G_i^{\text{sys}})).$$

Considering an example of a self-driving car, A^{sys} would contain the general requirements on the operating environment, ensuring that other agents have to follow the laws of physics and do not have the intention to make a collision unavoidable. More specifically, for a perception subsystem under test, A_i^{sys} and G_i^{sys} could defined such that if a person is unobstructed in the line of sight (assumption), the perception system will detect and classify them correctly (guarantee). From the test objective and the system specification, we can define a test strucuture which comprises of these two contracts as follows.

Definition 3.3 (Test Structure)**.** A *test structure* is a tuple denoted by $\mathfrak{t} = (C^{\text{obj}}, C^{\text{sys}})$, which includes the test objective and the corresponding system specification for the test.

This test structure contains information about the system under test and the specific test objective and allows us to generate the specification for the test environment that is needed to ensure a satisfactory test execution.

Definition 3.4 (Test Satisfaction)**.** A test execution is defined to be *satisfactory* if i) the desired test behavior was observed successfully, or ii) the system has failed to meet its requirements.

For a satisfactory test, we require the cause of failure to be due to the system under test and not due to the test environment, meaning the test either demonstrated that the system can successfully pass this test or revealed a system fault. We want to find a test environment that together with the system will result in a test that satisfies the test objective. Therefore we can derive the specification for the test environment from the test structure by computing the mirror of the system contract, merged with the test objective. This is equivalent to computing the quotient of C^{obj} and C^{sys} [78].

Definition 3.5 (Test Environment Contract)**.** Given a test structure $\mathsf{t} = (C^{\text{obj}}, C^{\text{sys}})$ the *test environment contract* C^{tester} is defined as

$$C^{\text{tester}} = (C^{\text{sys}})^{-1} \bullet C^{\text{obj}} = C^{\text{obj}}/C^{\text{sys}}.$$

The tester contract can therefore directly be computed as

$$C^{\text{tester}} = (G^{\text{sys}}, G^{\text{obj}} \cap A^{\text{sys}} \cup \neg G^{\text{sys}}). \tag{3.1}$$

Intuitively this specification can be seen as the test environment assuming a system that will satisfy its specification, and in turn guaranteeing a safe operating environment for the system, while also ensuring that the test objective is satisfied, or the system does not satisfy its guarantees. Due to the system and the tester being interconnected, we need to break this feedback loop to be able to use this specification to synthesize a test environment. As it is the responsibility of the tester to provide a safe operating environment to the system, a test environment has to satisfy the specification

$$\bigcap_i (\neg A_i^{\text{sys}} \cup G_i^{\text{sys}}) \rightarrow A^{\text{sys}} \cap G^{\text{obj}}. \tag{3.2}$$

That is, assuming that each of the system's subsystems performs its task as specified, the test environment provides a safe environment for the system and will ensure the satisfaction of the test objective.

3.3 Comparing Tests

In this section, we will focus on comparing test structures and test campaigns. We define a test campaign, $\mathsf{TC} = \{\mathsf{t}_i\}_{i=1}^n$, as a finite list of test structures that are specified by the test engineer. Being able to compare test structures is key to selecting which test campaign to actually execute on the system when multiple options are given. A test campaign that is more refined will ensure that the system is tested for a more refined set of test objectives, possibly under a more stringent set of system specifications. To define an ordering of test campaigns, we first show how to generate a single test structure from a test campaign.

Definition 3.6. Given a test campaign $\mathtt{TC} = \{\mathsf{t}_i\}_{i=1}^{n}$, the *test structure generated by this campaign*, denoted $\tau(\mathtt{TC})$, is

$$\tau(\mathtt{TC}) = \mathsf{t}_1 \parallel \ldots \parallel \mathsf{t}_n.$$

To be able to define an order of test campaigns, first, we need to define a notion of order for test structures by making use of contract refinement.

Definition 3.7. We say that the test structure $(C_1^{\text{obj}}, C_1^{\text{sys}})$ *refines* the test structure $(C_2^{\text{obj}}, C_2^{\text{sys}})$, written $(C_1^{\text{obj}}, C_1^{\text{sys}}) \leq (C_2^{\text{obj}}, C_2^{\text{sys}})$, if contract refinement occurs element-wise, i.e., if $C_1^{\text{sys}} \leq C_2^{\text{sys}}$ and $C_1^{\text{obj}} \leq C_2^{\text{obj}}$.

Now that we can define refinement for test structures generated from test campaigns by comparing the test structures element-wise, we can define an ordering on the respective test campaigns.

Definition 3.8. Given two test campaigns $\mathtt{TC}$ and $\mathtt{TC}'$, we say that $\mathtt{TC} \leq \mathtt{TC}'$ if $\tau(\mathtt{TC}) \leq \tau(\mathtt{TC}')$.

This notion of order for test campaigns is extremely useful because it allows us to replace any test campaign with a more refined test campaign, as the more refined test campaign will test the system for more stringent specifications in a more stringent setting. Additionally, we will make use of test campaign refinement for splitting tests. The following sections will describe how assume-guarantee contract operations can be used to perform operations on test structures.

3.4 Combining Tests

We now provide a framework to combine unit test campaigns into a single system-level test structure. Suppose we have test structures $(C_i^{\text{obj}}, C_i^{\text{sys}})$ for $i \in \{1, 2\}$ with test environment (tester) contracts C_i^{tester}. We interpret the specifications C_i^{tester} as viewpoints of the tester that apply to different specifications of the system. When we merge the tester specifications, we obtain a single test structure given as follows:

Proposition 3.1. $C_1^{\text{tester}} \bullet C_2^{\text{tester}} = (C_1^{\text{obj}} \parallel C_2^{\text{obj}})/\left(C_1^{\text{sys}} \parallel C_2^{\text{sys}}\right)$.

Proof. Merging tester contracts yields

$$\begin{aligned}
C_1^{\text{tester}} \bullet C_2^{\text{tester}} =&(C_1^{\text{obj}}/C_1^{\text{sys}}) \bullet (C_2^{\text{obj}}/C_2^{\text{sys}}) \\
=&(C_1^{\text{obj}} \bullet (C_1^{\text{sys}})^{-1}) \bullet (C_2^{\text{obj}} \bullet (C_2^{\text{sys}})^{-1}) && \text{([81], Section 3.1)} \\
=&(C_1^{\text{obj}} \bullet C_2^{\text{obj}}) \bullet \left(((C_1^{\text{sys}})^{-1}) \bullet ((C_2^{\text{sys}})^{-1})\right) \\
=&(C_1^{\text{obj}} \bullet C_2^{\text{obj}}) \bullet \left(C_1^{\text{sys}} \parallel C_2^{\text{sys}}\right)^{-1} && \text{([78], Table 6.1)} \\
=&(C_1^{\text{obj}} \bullet C_2^{\text{obj}})/\left(C_1^{\text{sys}} \parallel C_2^{\text{sys}}\right) \\
=&(C_1^{\text{obj}} \parallel C_2^{\text{obj}})/\left(C_1^{\text{sys}} \parallel C_2^{\text{sys}}\right), && (A_1^{\text{obj}} = A_2^{\text{obj}} = \mathit{True}))
\end{aligned}$$

which is the list $(C_1^{\text{obj}} \parallel C_2^{\text{obj}}, C_1^{\text{sys}} \parallel C_2^{\text{sys}})$. □

The resulting contract is the tester contract for the test structure given by the parallel compositions of the objective contracts and system contracts, separately. As we are defining the system specification as requirements on the subsystem to be tested, the composition of the system specifications represents a system consisting of the individual subsystems. We use Proposition 3.1 to define an operation on test structures directly:

Definition 3.9. Given test structures $\mathfrak{t}_i = (C_i^{\text{obj}}, C_i^{\text{sys}})$ for $i \in \{1, 2\}$, we define their *composition* $\mathfrak{t}_1 \parallel \mathfrak{t}_2$ as

$$(C_1^{\text{obj}}, C_1^{\text{sys}}) \parallel (C_2^{\text{obj}}, C_2^{\text{sys}}) = (C_1^{\text{obj}} \parallel C_2^{\text{obj}}, C_1^{\text{sys}} \parallel C_2^{\text{sys}}).$$

For the composition of the test structures to correspond to a valid test, we require the composed test objective and the resulting tester contract to be satisfiable.

Example 3.1 (Stopping Maneuver)**.** Consider a test setup with a single-lane road and a pedestrian on a crosswalk. The agent under test is an autonomous car, which has to detect the pedestrian and come to a stop in front of the crosswalk under different visibility conditions. These requirements are encoded in the system specification and the test objective. The setup for this test is shown in Figure 3.1. Three unit test objective contracts are specified by the test engineer. The first test objective is as follows:

$$C_1^{\text{obj}} = \big(\mathit{True}, \quad \varphi_{\text{init}}^{\text{car}} \wedge \Box\varphi_{\text{low}}^{\text{vis}} \wedge \Diamond\, \varphi_{\text{cw}}^{\text{ped}} \wedge \varphi_{\text{cw}}^{\text{ped}} \rightarrow \Diamond\, \varphi_{\text{cw}}^{\text{stop}}\big),$$

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 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 least one cell in front of the crosswalk cell C_{cw}. The second test objective is given as

$$C_2^{\text{obj}} = \Big(\mathit{True}, \quad \varphi_{\text{init}}^{\text{car}} \wedge \Box\varphi_{\text{high}}^{\text{vis}} \wedge \Diamond\, \varphi_{\text{cw}}^{\text{ped}} \wedge \varphi_{\text{cw}}^{\text{ped}} \rightarrow \Diamond\, \varphi_{\text{cw}}^{\text{stop}}\Big),$$

where $\varphi_{\text{high}}^{\text{vis}} := \varphi^{\text{vis}} \models \text{high}$ denotes high visibility conditions; and lastly the third test objective is given as

$$C_3^{\text{obj}} = \big(\mathit{True}, \quad \exists k : (v_{\text{car}} = V_{\max} \wedge x_{\text{car}} = C_k) \rightarrow \Diamond\, \varphi_{k+d_{\text{braking}}}^{\text{stop}}\big),$$

where the car has to drive at a specified speed of $V_{\max}$ in an arbitrary cell C_k and stop within the allowed braking distance d_{braking}. This test represents the mechanical requirement of stopping without specifying any interaction with a pedestrian. Note that neither of the test objective contracts holds information about the system's capabilities to detect a pedestrian, only that the system needs to stop in front of a pedestrian.

The system capabilities are encoded in the system specifications, which are provided by the system designer. For each test objective, we are given the corresponding system specification, which describes the required capabilities of the system for that test objective (e.g. perception, mechanical requirements, etc.). Each system specification relies on the system being in a safe environment, where the transitions of the environment agents are ensured to be safe. This is denoted as $A^{\text{sys}} = \Box\varphi_{\text{dyn}}^{\text{ped}} \wedge \Box\varphi_{\text{dyn}}^{\text{vis}}$, where $\varphi_{\text{dyn}}^{\text{ped}}$, and $\varphi_{\text{dyn}}^{\text{vis}}$ denote the dynamics of the pedestrian, and the visibility conditions, respectively. We use the same notation for the set and formula A^{sys}, which can be inferred from context. The system contract C_1^{sys} corresponding to the first test objective is given as

$$\begin{aligned} C_1^{\text{sys}} = \Big(A^{\text{sys}}, \quad & \Box\varphi_{\text{dyn}}^{\text{car}} \wedge \Box\,(\varphi_{\text{low}}^{\text{vis}} \rightarrow v \leq V_{\text{low}}) \\ & \wedge\, \Box\,\big(\mathtt{detectable}_{\text{low}}^{\text{ped}} \rightarrow \Diamond\, \varphi_{\text{ped}}^{\text{stop}}\big) \vee \neg A^{\text{sys}}\Big), \end{aligned}$$

where $\varphi_{\text{dyn}}^{\text{car}}$, describes the dynamics of the car. The maximum speed that the car is allowed to drive at in low visibility conditions is V_{low}, and $\mathtt{detectable}_{\text{low}}^{\text{ped}}$ is defined as

$$\texttt{detectable}^{\text{ped}}_{\text{low}} := x_{\text{car}} + dist^{\text{low}}_{\text{min}} \leq x_{\text{ped}} \leq x_{\text{car}} + dist^{\text{low}}_{\text{max}},$$

which describes the pedestrian being in the 'buffer' zone in front of the car, where $dist^{\text{low}}_{\text{min}}$ denotes the minimum distance such that the car can come to a full stop, and $dist^{\text{low}}_{\text{max}}$ denotes the maximum distance at which the car can detect a pedestrian in low visibility conditions. The system specification for the second test objective, the system contract C^{sys}_2, is given as

$$\begin{aligned} C^{\text{sys}}_2 = \Big(A^{\text{sys}},\quad & \Box\varphi^{\text{car}}_{\text{dyn}} \wedge \Box\,(\varphi^{\text{vis}}_{\text{high}} \rightarrow v \leq V_{\text{max}}) \\ & \wedge\ \Box\,(\texttt{detectable}^{\text{ped}}_{\text{high}} \rightarrow \Diamond\,\varphi^{\text{stop}}_{\text{ped}}) \vee \neg A^{\text{sys}}\Big), \end{aligned}$$

describing driving in high visibility conditions with a maximum speed of V_{max} and $\texttt{detectable}^{\text{ped}}_{\text{high}}$ denoting the pedestrian being detectable in the 'buffer' zone for high visibility conditions. The third system specification C^{sys}_3 is given as

$$C^{\text{sys}}_3 = \Big(A^{\text{sys}},\quad \Box\varphi^{\text{car}}_{\text{dyn}} \vee \neg A^{\text{sys}}\Big),$$

with the braking distance as a function of speed being part of the car's dynamics denoted by $\varphi^{\text{car}}_{\text{dyn}}$. For each pair of system specifications and test objectives, we can synthesize the test environment according to equation (3.2). Now we will find combinations of these tester structures $\mathfrak{t}_i = (C^{\text{obj}}_i, C^{\text{sys}}_i)$ that we can use instead of executing all tests individually. We will start by computing the combined test structure $\mathfrak{t} = \mathfrak{t}_2 \parallel \mathfrak{t}_3$. The combined test objective contract C^{obj} is computed as

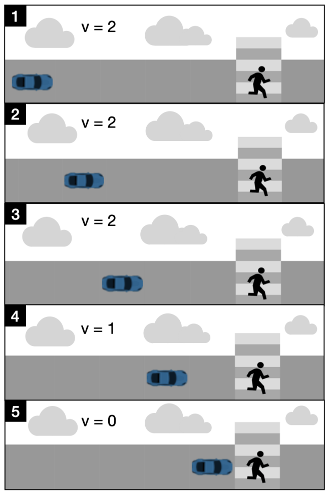
$$\begin{aligned} C^{\text{obj}} = C^{\text{obj}}_2 \parallel C^{\text{obj}}_3 = \big(True,\quad & \varphi^{\text{car}}_{\text{init}} \wedge \Box\varphi^{\text{vis}}_{\text{low}} \wedge \Diamond\,\varphi^{\text{ped}}_{\text{cw}} \wedge \varphi^{\text{ped}}_{\text{cw}} \rightarrow \Diamond\,\varphi^{\text{stop}}_{\text{cw}} \wedge \\ & \exists k : (v_{\text{car}} = V_{\text{max}} \wedge x_{\text{car}} = C_k) \rightarrow \Diamond\,\varphi^{\text{stop}}_{k+d_{\text{braking}}}\big). \end{aligned} \tag{3.3}$$

The combined system contract is computed as

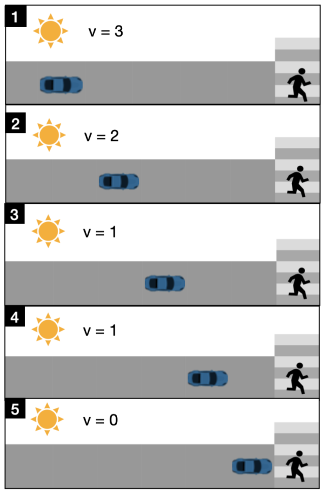
$$C^{\text{sys}} = C^{\text{sys}}_2 \parallel C^{\text{sys}}_3 = \big(A^{\text{sys}} \cup \neg(G^{\text{sys}}_2 \cap G^{\text{sys}}_3),\quad G^{\text{sys}}_2 \cap G^{\text{sys}}_3\big).$$

We will relax this system contract by removing $\neg(G^{\text{sys}}_2 \cap G^{\text{sys}}_3)$ from the assumptions to ensure that the assumptions are in the same form as we require for the system contract in Definition 3.2. Consequently, the tester contract resulting from this system contract is more refined. So the system contract becomes

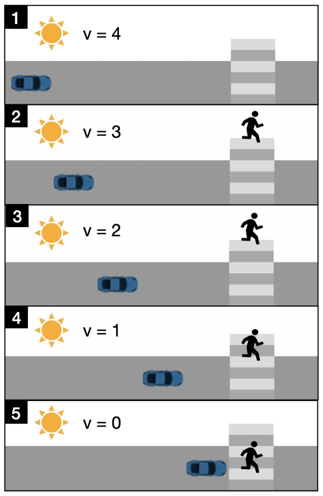
$$\begin{aligned} C^{\text{sys}} = \big(A^{\text{sys}},\quad & \Box\varphi^{\text{car}}_{\text{dyn}} \wedge \Box\,(\varphi^{\text{vis}}_{\text{high}} \rightarrow v \leq V_{\text{max}}) \\ & \wedge\ \Box(\texttt{detectable}^{\text{ped}}_{\text{high}} \rightarrow \Diamond\,\varphi^{\text{stop}}_{\text{ped}}) \vee \neg A^{\text{sys}}\big). \end{aligned} \tag{3.4}$$



(a) Low visibility with a stationary pedestrian.



(b) High visibility with a stationary pedestrian.



(c) High visibility with a reactive pedestrian.

Figure 3.1: Test execution snapshots of the car stopping for a pedestrian. Figure 3.1a shows a test execution satisfying C_1^{tester}, Figure 3.1b satisfies C_2^{tester} and Figure 3.1c satisfies C_2^{tester} and C_3^{tester}.

From equations (3.3) and (3.4), we construct the test structure $\mathfrak{t} = (C^{\text{obj}}, C^{\text{sys}})$, where every implementation that satisfies to equation (3.2) describes a valid test environment for this combined test. This merged tester specification describes a test environment where we will see the car decelerate from V_{max} and stop in front of the crosswalk in high visibility conditions. To ensure that test structures can be combined, we need to check whether the resulting test objective and the corresponding tester contract are satisfiable. We will now explain which combinations of the given test structures cannot be implemented for either of these reasons. Computing the composition $\mathfrak{t}_1 \parallel \mathfrak{t}_2$ is not possible, as the composition of the test objectives $C_1^{\text{obj}} \parallel C_2^{\text{obj}}$ results in a contract with empty guarantees. This is the case, because $\square\varphi_{\text{low}}^{\text{vis}}$ and $\square\varphi_{\text{high}}^{\text{vis}}$ are disjoint, as the visibility conditions cannot be *high* and *low* at the same time. Thus these two test structures are not composable with each other. The composition $\mathfrak{t}_1 \parallel \mathfrak{t}_3$ does not result in a feasible test — the test objective requires a maximum speed of V_{max}, but the system is constrained to a maximum speed of $V_{\text{low}} < V_{\text{max}}$ in low visibility conditions, resulting in $G^{\text{sys}} \cap G^{\text{obj}} = \emptyset$.

Figure 3.1 shows snapshots of manually constructed test executions satisfying the tester contracts corresponding to $\mathfrak{t}_1, \mathfrak{t}_2$, and $\mathfrak{t}_2 \parallel \mathfrak{t}_3$. The simulation is in a grid world setting, where the car will move one cell forward if it has a positive speed v, and can accelerate or decelerate by one unit during every time step, meaning if the car is driving at a higher speed, it will take more cells to come to a stop. 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. So in Figure 3.1a it 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 3.1b we can see that the pedestrian is detected and the car slows down gradually until it reaches the cell in front of the crosswalk. Figure 3.1c shows a test for the tester contract corresponding to $\mathfrak{t}_2 \parallel \mathfrak{t}_3$, 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 exactly $d_{\text{braking}} = 4$ cells away from the pedestrian. 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. ■

Remark: Sometimes in addition to the combined test contract, the test executions must satisfy further constraints, informed by domain knowledge, to provide useful information to the test engineer. In the case of combining tests, a metric can be useful in determining whether we get the desired information from the execution of the combined test. Later in this chapter, we will show how temporal constraints are added to refine the combined test objective and provide an example. Instead of refining the test structure, such additional constraints can also be handled during test environment synthesis. This can be helpful in determining if and how tests can be combined for a given available environment and the desired test information.

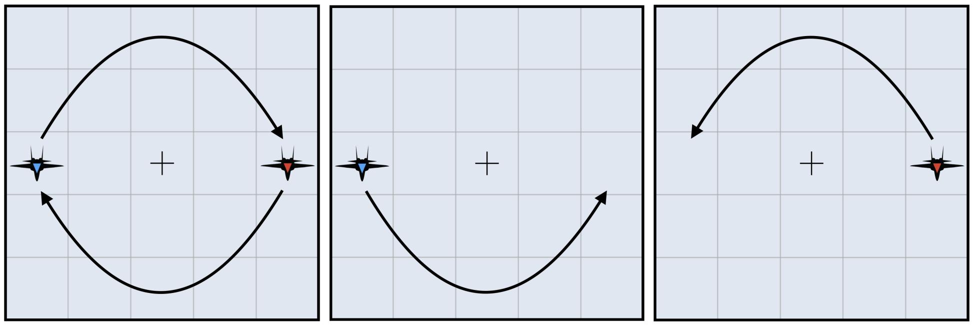
3.5 Splitting Tests

Now that we have a formal framework for how to combine tests, a natural next question to ask is whether it is possible to split a test into multiple tests, to allow executing them individually. Such a situation might be more beneficial in some cases, for example, if a test was already run previously, or if a test resulted in a system failure requiring executing the test in smaller increments for diagnostics purposes. In this section, we will focus on how to split a test structure from another given test structure.

Proposition 3.2. Let $\mathfrak{t} = (C^{\text{obj}}, C^{\text{sys}})$ and $\mathfrak{t}_1 = (C_1^{\text{obj}}, C_1^{\text{sys}})$ be two test structures and let $\mathfrak{t}_q = (C^{\text{obj}}/C_1^{\text{obj}}, C^{\text{sys}}/C_1^{\text{sys}})$. For any test structure $\mathfrak{t}_2 = (C_2^{\text{obj}}, C_2^{\text{sys}})$, 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 $\mathfrak{t}/\mathfrak{t}_1$.



(a) Left: Original test. Center: Given unit test. Right: Split test.

Figure 3.2: Front view of tests satisfying the original test structure (left), the unit test structure (center) and the split test structure (right) for Example 3.2 Case I.

Proof. $\mathfrak{t}_2 \leq \mathfrak{t}_q \quad \Leftrightarrow \quad C_2^{\text{sys}} \leq C^{\text{sys}}/C_1^{\text{sys}}$ and $C_2^{\text{obj}} \leq C^{\text{obj}}/C_1^{\text{obj}} \Leftrightarrow \quad (C_2^{\text{obj}} \parallel C_1^{\text{obj}}, C_2^{\text{sys}} \parallel C_1^{\text{sys}}) \leq (C^{\text{obj}}, C^{\text{sys}}) \quad \Leftrightarrow \quad \mathfrak{t}_2 \parallel \mathfrak{t}_1 \leq \mathfrak{t}$. □

In Proposition 3.2 we define the contract that is split from $\mathfrak{t}$ as $\mathfrak{t}_1 = (C_1^{\text{obj}}, C_1^{\text{sys}})$, which corresponds to splitting out the subsystem C_1^{sys} and its corresponding test objective C_1^{obj} from the original test structure $\mathfrak{t}$. This might not always be the intended action, as for some applications, removing a task from the test objective or removing a subsystem without modifying the respectively other part of the test structure is beneficial. We can define additional quotients as i) $\mathfrak{t}_q = (C^{\text{obj}}, C^{\text{sys}}/C_1^{\text{sys}})$, which still requires satisfaction of the test objective; or ii) $\mathfrak{t}_q = (C^{\text{obj}}/C_1^{\text{obj}}, C^{\text{sys}})$, which splits out a test objective, while keeping the system intact. Using a quotient of type (i) could be helpful if parts of the system can be simulated and replaced by a test harness while testing for the same test objective. This might facilitate the placement of additional sensors at the test harness-system interfaces to monitor the subsystem's behavior during the test execution. On the other hand, a quotient of type (ii) could be useful for testing the full system for smaller test objectives independently, for example, to allow parallelization of tests in simulation.

Example 3.2 (Aircraft Formation Maneuvers)**.** Imagine a scenario where two aircraft, a_1 and a_2 are expected to execute a formation flying maneuver during a test. The maneuver involves flying along a straight path while swapping positions in a spiral motion, either clockwise or counterclockwise. For this test, our attention will be on the longitudinal swapping motion, assuming that the two aircraft will hold a constant speed allows us to disregard the forward component in this simplified example. In this example, the tester has the capacity to send directives to the two aircraft, these directives contain the swapping command and the direction that it should be executed. A clockwise swap directive sent to aircraft a_i is denoted by $\texttt{directive}^{\text{cw}}_{\text{swap}}(a_i)$, and a counterclockwise directive is denoted by $\texttt{directive}^{\text{ccw}}_{\text{swap}}(a_i)$, where the two positions are denoted x_L and x_R for the left and right position in the formation, respectively. To ensure the safety of the aircraft in the formation, we need to enforce that the directives are i) not conflicting (i.e. both aircraft receive either clockwise or counterclockwise directives), and ii) not issued until the current directive was successfully executed. These constraints on the directives are captured in $G^{\texttt{dir}}_{\text{safe}}$ and $G^{\texttt{dir}}_{\text{limit}}$. Let the dynamics for aircraft a_i be described in G^{dyn}_i, and the no-collision requirement be captured in G^{safe}. The direction of the swapping motion is encoded as G^{cw} and G^{ccw}, for clockwise and counterclockwise motions. In the event that a swap directive is issued to aircraft a_i, it is required to execute this directive. This is captured in G^{swap}_i as follows:

$$\begin{aligned} G^{\text{swap}}_i =& \Box\big(\texttt{directive}^{\text{cw}}_{\text{swap}}(a_i) \rightarrow \texttt{execute}^{\text{cw}}_{\text{swap}}(a_i)\big) \quad \wedge \\ & \Box\big(\texttt{directive}^{\text{ccw}}_{\text{swap}}(a_i) \rightarrow \texttt{execute}^{\text{ccw}}_{\text{swap}}(a_i)\big), \end{aligned} \tag{3.5}$$

where the $\texttt{execute}^{\text{ccw}}_{\text{swap}}(a_i)$ command is defined in Table 3.1. Now we can define the requirements of the system under test as the following system contract:

$$C^{\text{sys}} = (A^{\text{sys}}, G^{\text{sys}}) = (G^{\texttt{dir}}_{\text{limit}} \wedge G^{\texttt{dir}}_{\text{safe}}, G^{\text{safe}} \wedge G^{\text{swap}} \wedge G^{\text{dyn}}), \tag{3.6}$$

where $G^{\text{swap}} := G^{\text{swap}}_1 \wedge G^{\text{swap}}_2$, and $G^{\text{dyn}} := G^{\text{dyn}}_1 \wedge G^{\text{dyn}}_2$, are the dynamics and swapping guarantees of both aircraft.

Table 3.1: Subformulae used in the construction of G^{sys} and G^{obj}.

Formulae
$\texttt{reach\_goal}_{\text{swap}}(a_i) := (x_i = x_R) \rightarrow \big(\Diamond(x_i = x_L) \wedge \Box(x_i = x_L \rightarrow \bigcirc(x_i = x_L))\big)$ $\wedge(x_i = x_L) \rightarrow \big(\Diamond(x_i = x_R) \wedge \Box(x_i = x_R \rightarrow \bigcirc(x_i = x_R))\big)$
$\texttt{execute}^{\text{cw}}_{\text{swap}}(a_i) := \texttt{reach\_goal}_{\text{swap}}(a_i) \wedge \Box G^{\text{cw}}_i$
$\texttt{execute}_{\text{swap}}(a_i)^{\text{ccw}}(a_i) := \texttt{reach\_goal}_{\text{swap}} \wedge \Box G^{\text{ccw}}_i$
$\varphi^{\text{cw}} := \Diamond\, \texttt{directive}^{\text{cw}}_{\text{swap}}(a_1) \wedge \Diamond\, \texttt{directive}^{\text{cw}}_{\text{swap}}(a_2)$
$\varphi^{\text{ccw}} := \Diamond\, \texttt{directive}^{\text{ccw}}_{\text{swap}}(a_1) \wedge \Diamond\, \texttt{directive}^{\text{ccw}}_{\text{swap}}(a_2)$

With the contract for the system under test constructed, we can now focus on the test objective.

Case I: Splitting off a subsystem and the objective. Given the test objective as a clockwise swapping maneuver defined as $C^{\text{obj}} = (\mathit{True}, G^{\text{obj}})$, where $G^{\text{obj}} := \varphi^{\text{cw}}$, assume that aircraft a_1 was already certified for this maneuver. The test structure corresponding to aircraft a_1 and the corresponding swapping test objective is given as $\mathfrak{t}_1 = (C_1^{\text{obj}}, C_1^{\text{sys}})$, with $C^{\text{obj}} = (\textit{True}, \Diamond\, \texttt{directive}_{\text{swap}}^{\text{cw}}(a_1))$ and $C_1^{\text{sys}} := (G_{\text{limit}}^{\texttt{dir}} \wedge G_{\text{safe}}^{\texttt{dir}}, G_1^{\text{swap}} \wedge G^{\text{dyn}_1}),$. We can now compute the test structure resulting from the split of the overall test structure and the unit test structure for aircraft a_1. First, we can compute the resulting test objective by applying the quotient and substituting *True* for the assumptions. In the last step, we refine the resulting contract by setting the assumptions to *True*.

$$\begin{aligned} C^{\text{obj}}/C_1^{\text{obj}} &= (A^{\text{obj}} \cap G_1^{\text{obj}}, G^{\text{obj}} \cap A_1^{\text{obj}} \cup \neg(A^{\text{obj}} \cap G_1^{\text{obj}})) \\ &= (G_1^{\text{obj}}, G^{\text{obj}} \cup \neg G_1^{\text{obj}}) \geq \ (\textit{True}, G^{\text{obj}} \cup \neg G_1^{\text{obj}}). \end{aligned}$$

This contract can now be further refined in the context of the test objectives. To refine the guarantees, we can shrink the guarantee set as follows.

$$\begin{aligned} C^{\text{obj}}/C_1^{\text{obj}} &= \texttt{directive}_{\text{swap}}^{\text{cw}}(a_1) \wedge \texttt{directive}_{\text{swap}}^{\text{cw}}(a_2) \vee \neg\texttt{directive}_{\text{swap}}^{\text{cw}}(a_1) \\ &= \texttt{directive}_{\text{swap}}^{\text{cw}}(a_2) \vee \neg\texttt{directive}_{\text{swap}}^{\text{cw}}(a_1) \\ &\geq \texttt{directive}_{\text{swap}}^{\text{cw}}(a_2) \end{aligned}$$

In this example it is clear to see that domain knowledge can be very helpful as designer input is required to ensure that the refinement results in a useful contract.

The quotient of the system contracts can be computed by

$$\begin{aligned} C^{\text{sys}}/C_1^{\text{sys}} =& \big(A^{\text{sys}} \cap G_1^{\text{sys}}, G^{\text{sys}} \cap A_1^{\text{sys}} \cup \neg(A^{\text{sys}} \cap G_1^{\text{sys}})\big) \\ =& \big(G_{\text{limit}}^{\text{dir}} \wedge G_{\text{safe}}^{\text{dir}} \wedge G_1^{\text{swap}} \wedge G_1^{\text{dyn}}, (G^{\text{safe}} \wedge G_2^{\text{swap}} \wedge G_2^{\text{dyn}}) \\ & \vee \neg(G_1^{\text{swap}} \wedge G_1^{\text{dyn}} \wedge G_{\text{limit}}^{\text{dir}} \wedge G_{\text{safe}}^{\text{dir}})\big) \\ =& \big(G_{\text{limit}}^{\text{dir}} \wedge G_{\text{safe}}^{\text{dir}} \wedge G_1^{\text{swap}} \wedge G_1^{\text{dyn}}, (G^{\text{safe}} \wedge G_2^{\text{swap}} \wedge G_2^{\text{dyn}})\big). \end{aligned} \tag{3.7}$$

The last step in the simplification is due to the saturation of the contract. In this resulting system specification, we can see that the requirements on aircraft a_1 are in the assumptions. Aircraft a_1 is now no longer a part of the system, but it is controlled by the tester. This now gives the test engineers the choice, of whether to replace aircraft a_1 with a 'virtual' aircraft and feed its simulated position to aircraft a_2's sensors during the test or if it is necessary to deploy the aircraft.

Case II: Splitting off a part of the objective. Consider the same test setup consisting of the two aircraft. Assume the test objective is given as

$$G^{\text{obj}} = \varphi^{\text{cw}} \wedge \varphi^{\text{ccw}},$$

where the test engineers want to see a clockwise swap and a counterclockwise swap. Assuming we know that both aircraft have already successfully demonstrated the clockwise swap, we can split this requirement from the overall test objective. After computing the quotient and refining the resulting contract, we end up with $G_q^{\text{obj}} = \varphi^{\text{ccw}}$ for the desired test objective. In this case, we are splitting the test objectives into different instances, while the system contract is kept intact. That is this test requires an execution where both aircraft are deployed for a counterclockwise swapping maneuver.

Case III: Splitting off a part of the system. In this case, the test engineers might deem it beneficial to split off one aircraft from the other, for example in the case of two identical aircraft it might not be necessary to test both. For the case of splitting off aircraft a_1, only the system contract quotient is computed according to equation (3.7). The resulting test structure now contains a_1 in the test harness, controlled by the tester and such it can be replaced by artificial sensor inputs to a_1.

Remark 3.1. In Example 3.2 it becomes apparent how domain knowledge is crucial to refine the contracts in a meaningful way. Additionally, it is important to note that this approach can only reason over what is specified. That is, in the case of this example, we can only reason over the directives that are sent. If there are any interactions between the components that are not captured in the specifications, such as possible wake turbulence for this example, they need to be explicitly contained in the specification to be taken into account with this approach.

3.6 Finding a Test Strategy

In this section, we will illustrate on two examples how to find a test strategy for a combined test that maximizes a difficulty metric. We will focus on autonomous car examples, where we can find the strategy for the test agents using the test objective generated from combining the unit test objectives in the form of reachability requirements.

Table 3.2: Subformulae describing the system requirements for Example 3.3.

Formulae
$\varphi_{\text{sys}}^{\text{init}} := x_{\text{sys}} = 0 \wedge y_{\text{sys}} = 1$
$\varphi_{\text{sys}}^{\text{dyn}} := \square\big((x_{\text{sys}} = x_i \wedge y_{\text{sys}} = y_j) \rightarrow \bigcirc\big((x_{\text{sys}} = x_i \wedge y_{\text{sys}} = y_j) \vee (x_{\text{sys}} = x_i + 1 \wedge y_{\text{sys}} = y_j) \vee (x_{\text{sys}} = x_i + 1 \wedge y_{\text{sys}} = y_j + 1)\big)\big)$ for $1 \leq i \leq 10, \quad 1 \leq j \leq 2$
$\varphi_{\text{sys}}^{\text{safe}} := \square\neg(y_{\text{sys}} = 2 \wedge x_{\text{sys}} = x_{\text{test},k})$, for $1 \leq k \leq 2$
$\varphi_{\text{sys}}^{\text{prog}} := \diamond(x_{\text{sys}} = 2)$

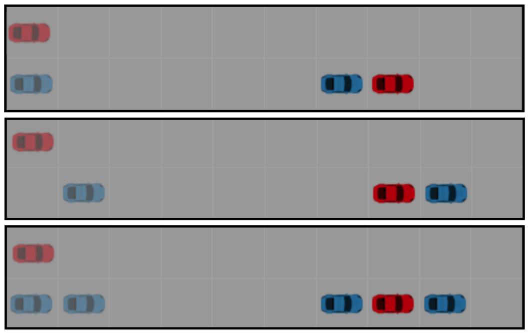


Figure 3.3: Initial system and test agent configurations (red and blue, respectively) shown on the left, with final configurations on the right for unit tests (top and center) and combined test (bottom) for Example 3.3.

Table 3.3: Subformulae describing the tester requirements for Example 3.3.

Formulae
$\varphi_{\text{test}}^{\text{init}} := x_{\text{test},1} = 0 \wedge x_{\text{test},2} = 1$
$\varphi_{\text{test}}^{\text{dyn}} := \square\big((x_{\text{test},k} = x_i) \rightarrow \bigcirc\big((x_{\text{test},k} = x_i \vee x_{\text{test},k} = x_i + 1\big)\big)$ for $1 \leq i \leq 10, \quad 1 \leq k \leq 2$
$\varphi_{\text{test}}^{\text{safe}} := \square\neg(y_{\text{sys}} = 2 \wedge x_{\text{sys}} = x_{\text{test},k})$, for $1 \leq k \leq 2$
$\varphi_{\text{test},1}^{\text{prog}} := \diamond(y_{\text{sys}} = 2 \wedge x_{\text{sys}} = x_{\text{test},1} + 1)$
$\varphi_{\text{test},2}^{\text{prog}} := \diamond(y_{\text{sys}} = 2 \wedge x_{\text{sys}} = x_{\text{test},2} - 1)$

Example 3.3 (Lane Change)**.** Consider the following two unit tests for an autonomous car, the system under test. The road layout consists of two lanes with a length of 10 grid cells each. The x-coordinate increases along the road from left to right, ranging from 1 to 10, and the y-coordinate corresponds to the top and bottom lanes, designated as 1 and 2 respectively. In each of the tests, the car starts in the top lane ($y_{\text{sys}} = 1$) and wants to merge into the lower lane ($y_{\text{sys}} = 2$). The dynamics of the car are such that it is allowed to move forward one grid cell, stay in the same grid cell, or merge diagonally, designated by $\varphi_{\text{sys}}^{\text{dyn}}$. Furthermore, its safety condition is that it is not allowed to collide with a tester agent. The system specification is therefore given as

$$C^{\text{sys}} = (\varphi_{\text{test}}^{\text{dyn}} \wedge \varphi_{\text{test}}^{\text{safe}}, \varphi_{\text{sys}}^{\text{dyn}} \wedge \varphi_{\text{sys}}^{\text{safe}} \wedge \varphi_{\text{sys}}^{\text{prog}}), \tag{3.8}$$

where the individual subformulas are given in Table 3.2 and Table 3.3.

The test specifications are given as merge *behind* another car, and merge *in front* of another car, respectively. Formally the test objectives are given as $C_1^{\text{obj}} = (\textit{True}, G_1^{\text{obj}})$ and $C_2^{\text{obj}} = (\textit{True}, G_2^{\text{obj}})$, where $G_1^{\text{obj}} := \varphi_{\text{test},1}$ and $G_2^{\text{obj}} := \varphi_{\text{test},2}$

are the progress requirements described in Table 3.3.

We can now compute the combined test objective by taking the composition of the test objectives. Note that due to the fact that the assumptions in the test objective contracts are simply *True*, and the resulting contract will be refined by setting the assumptions to *True*, the result is equivalent to the merge of the two contracts. The refined combined test objective is as follows:

$$C_1^{\text{obj}} \parallel C_2^{\text{obj}} \geq (\mathit{True}, G_1^{\text{obj}} \wedge G_2^{\text{obj}}) = (\mathit{True}, \varphi_{\text{test},1}^{\text{prog}} \wedge \varphi_{\text{test},2}^{\text{prog}}). \tag{3.9}$$

As both unit tests are designed to test the same system, the corresponding combined test structure is the following:

$$\mathfrak{t} = (C^{\text{obj}}, C^{\text{sys}}), \tag{3.10}$$

for the combined test objective contract $C^{\text{obj}} := (\mathit{True}, \varphi_{\text{test},1}^{\text{prog}} \wedge \varphi_{\text{test},2}^{\text{prog}})$.

From a test structure, we can generate the tester contract to enable us to generate the strategy for the two tester agents according to equation (3.2). We want the tester agent strategy to ensure that every test execution will satisfy the test objective. Using this specification to find a tester strategy comes with a caveat that is important to highlight. This framework allows for the synthesis of the tester from the corresponding specification only if the tester has the authority to ensure that all test executions can reach a state where the accepting sets of the system and the test specifications coincide. Furthermore, the tester should not be allowed to violate the system's assumptions to trivially satisfy the formula. In this section, we will illustrate how we can find such a strategy for Example 3.3 using the Temporal Logic and Planning Toolbox (TuLiP) [159] and Monte Carlo tree search (MCTS). Assume that we have access to a model of the system in the form of the system's transition system and its system contract.

Definition 3.10 (Transition System)**.** A *transition system* is a tuple $\mathcal{T} := (S, \rightarrow)$, where S is a set of states and $\rightarrow \subseteq S \times S$ is a transition relation. If $\exists$ a transition from $s \in S$to $s' \in S$, we write $s \rightarrow s'$.

Definition 3.11 (System Transition System)**.** Let **VarSys** be the set of system variables, and let S_{sys} be all possible valuations of **VarSys**. The *transition system corresponding to the system under test* is the tuple $\mathcal{T}_{\text{sys}} = (S_{\text{sys}}, \rightarrow_{\text{sys}})$, where the transition relation $\rightarrow_{\text{sys}}$ corresponds to the dynamics of the system.

For implementation purposes, we assume that the system controller is synthesized from a GR(1) specification, that can be found from a system contract if the subformulae of the system contract are in GR(1) form.

Definition 3.12 (System Specification)**.** A *system specification* Φ_{sys} is the $GR(1)$ formula,

$$\Phi_{\text{sys}} = (\varphi_{\text{test}}^{\text{init}} \wedge \square\varphi_{\text{test}}^{\text{safe}}) \rightarrow (\varphi_{\text{sys}}^{\text{init}} \wedge \square\varphi_{\text{sys}}^{\text{safe}} \wedge \square\diamond\varphi_{\text{sys}}^{\text{prog}})\,, \tag{3.11}$$

where $\varphi_{\text{sys}}^{\text{init}}$ is the initial condition, $\varphi_{\text{sys}}^{\text{safe}}$ encode system dynamics and safety requirements, and $\varphi_{\text{test}}^{\text{prog}}$ specifies recurrence goals contained in the guarantees of the system contract C_{sys}. Likewise, $\varphi_{\text{test}}^{\text{init}}$ and $\varphi_{\text{test}}^{\text{safe}}$ represent assumptions of the system contract on its environment.

Additionally, the test agents are modeled as a transition system that captures their dynamics.

Definition 3.13 (Test Agent Transition System)**.** Let **VarTest** be the set of test agent variables, and let S_{test} be all possible valuations of **VarTest**. The *test agent transition system* is the tuple $\mathcal{T}_{\text{test}} = (S_{\text{test}}, \rightarrow_{\text{test}})$, where the transition relation corresponds to the dynamics of the tester agents.

We will construct a turn-based product transition system of the system and test agent transition systems as follows.

Definition 3.14 (Game Graph)**.** The *product* of $\mathcal{T}_{\text{sys}}$ and $\mathcal{T}_{\text{test}}$ is the transition system $\mathcal{T}_{\text{prod}} = (S_{\text{prod}}, \rightarrow_{\text{prod}})$, where $S_{\text{prod}} := S_{\text{sys}} \times S_{\text{test}}$, and $\rightarrow_{\text{prod}} \subseteq S_{\text{prod}} \times S_{\text{prod}}$. The transition relation $\rightarrow_{\text{prod}}$ captures the turn-based dynamics where the two players take turns executing their move and remain stationary while the other agent moves. That is, for $s, s' \in S_{\text{sys}}$ and $t, t' \in S_{\text{test}}$, we have $((s,t),(s',t')) \in \rightarrow_{\text{prod}}$ if $(t,t') \in \rightarrow_{\text{test}}$ and $s = s'$ or if $(s,s') \in \rightarrow_{\text{sys}}$ and $t = t'$.

Next, we will define a game graph that allows us to keep track of from which state the system will choose the next action, and from which state the tester will choose the next action.

Definition 3.15 (Game Graph)**.** Let V_{sys} and V_{test} be copies of S_{prod} and let $E_{\text{sys}} := \{(s,t),(s',t) \in \rightarrow_{\text{prod}} | (s,t) \in V_{\text{sys}}, (s',t) \in V_{\text{test}}\}$ and $E_{\text{test}} := \{(s,t),(s,t') \in \rightarrow_{\text{prod}} | (s,t) \in V_{\text{test}}, (s,t') \in V_{\text{sys}}\}$. The *game graph* $G = (V, E)$, is a directed graph with vertices $V := V_{\text{sys}} \cup V_{\text{test}}$ and edges $E := E_{\text{sys}} \cup E_{\text{test}}$.

Definition 3.16 (Strategy)**.** On the game graph G, a strategy for the system is a function $\pi_{\text{sys}} : V^* V_{\text{sys}} \longrightarrow V_{\text{test}}$ such that $(s, \pi_{\text{sys}}(w \cdot s)) \in E_{\text{sys}}$, where $s \in V_{\text{sys}}$ and $w \in V^*$. The tester strategy $\pi_{\text{test}} : V^* V_{\text{test}} \longrightarrow V_{\text{sys}}$ is similarly defined.

Definition 3.17 (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 turn-based game graph G. The next state is found by applying the system or tester strategy as follows. For $v_i \in V_{\text{sys}}$, then $v_{i+1} = \pi_{\text{sys}}(v_0 \ldots v_i) \in V_{\text{test}}$ and $v_{i+2} = \pi_{\text{test}}(v_0 \ldots v_i v_{i+1}) \in V_{\text{sys}}$, continuing this alternating pattern. We will denote the test execution starting from state $s_0 \in V_{\text{sys}}$ for strategies π_{sys} and π_{test} as $\sigma_{\pi_{\text{sys}} \times \pi_{\text{test}}}(s_0)$.

In addition to a correct tester strategy, we also want to ensure that the resulting test execution is difficult for the system. To capture the notion of difficulty we introduce the following difficulty metric.

Definition 3.18 (Difficulty Metric)**.** Let Σ denote the set of all possible test executions on G. A difficulty metric $\rho : \Sigma \longrightarrow \mathbb{R}$ is a function assigning a scalar value to a test execution $\sigma \in \Sigma$.

Enforcing Temporal Constraints in a Combined Test. When combining unit tests, a situation might arise such that when test objectives are combined it could become unclear whether the information from the resulting test is sufficient to make a claim whether the unit tests would have been successfully passed individually. In this section, we will outline under which conditions the combined test objective requires a more constrained temporal structure. To ensure that the test execution will provide the desired information, we need to make certain that each test specification is sufficiently checked.

Definition 3.19 (Temporally constrained tests)**.** For a test trace σ, let σ_t be the suffix of the trace, starting at time t. Let t_{S1}, t_{S2} be times such that $\sigma_{t_{S1}} \models \varphi^{\text{prog}}_{\text{test},1}$ and $\sigma_{t_{S2}} \models \varphi^{\text{prog}}_{\text{test},2}$, and assume there exists a time t_{F1} such that $t_{F1} = \mathtt{min}(t)$ for all t, $t > t_{S1}$ such that $\sigma_{t_F 1} \not\models \varphi^{\text{prog}}_{\text{test},1}$ and assume that there exists a time t_{F2} such that $t_{F2} = \mathtt{min}(t)$ for all t, $t > t_{S2}$ such that $\sigma_{t_{F2}} \not\models \varphi^{\text{prog}}_{\text{test},2}$. Then if $t_{S1} = t_{S2} = t_1$ and $t_{F1} = t_{F2} = t_2$ the tests are *parallel* in the interval $t \in [t_1, t_2]$. If $t_{S1} < t_{S2}$ and $t_{F1} < t_{F2}$, or $t_{S1} > t_{S2}$ and $t_{F1} > t_{F2}$, the tests are *temporally constrained.*

For example, consider the lane change example. There exist many executions in which one of the unit tests is satisfied (i.e. the car merges in front of a vehicle), but it is not guaranteed that the other specification is satisfied as well. Therefore these two tests can be parallel. In contrast to this there exist test specifications where satisfying one will trivially satisfy the other. Then we are not able to distinguish which specification was checked; thus these unit tests should not be parallel to ensure that during the test there is a point in time where each test specification is satisfied individually.

Proposition 3.3. Let the test objective contract be $C^{\text{obj}} = (\mathit{True}, \Diamond p_1 \wedge \Diamond p_2)$, and the set of all test executions be Σ. If for a parallel test, we have that $\sigma \models \Diamond p_1 \iff \sigma \models \Diamond p_2 \; \forall \, \sigma \in \Sigma$, then the temporal constraint must be enforced by refining the test objective contract as follows:

$$C^{\text{obj}} = (\mathit{True}, \Diamond p_1 \wedge \Diamond p_2) \geq (\mathit{True}, \Diamond(p_1 \wedge \neg p_2) \wedge \Diamond(\neg p_1 \wedge p_2)). \tag{3.12}$$

Example 3.4 (Unprotected Left Turn)**.** Consider the unprotected left turn example illustrated in Figure 3.4. The system under test (red car) is tasked with taking a left turn at the intersection. The test agents are a car approaching from the opposite direction (depicted in blue) and a pedestrian on the crosswalk. The two test objectives are seeing the system stop at the intersection to wait for a car and wait for a pedestrian. If these two test executions were parallel, seeing the car successfully stop does not provide enough information to ensure that the car would also successfully stop if just one of the test agents is present. For this example, we need to enforce the temporal constraint on the combined test objective to check that the unit test objectives will also be satisfied individually.

Constructing the Partial Order. To be able to find a test strategy, we need to reason over the progress toward satisfying the test objective while assuming the system will satisfy its contract. In this section, we will outline how we can use the game graph G to find a test policy filter that ensures that every available action satisfies these requirements. As the dynamics of the system and the test agents are already contained in the initial state and the transitions on the game graph, we can focus on the progress requirements of the system and the combined test objective contract. First, assume that we are given a game graph G and a combined test objective guarantee $\varphi_{\text{test}} = \Diamond \varphi_{\text{test},1} \wedge \Diamond \varphi_{\text{test},2}$, that coincides with the system progress requirement $\varphi_{\text{sys}}^{\text{prog}}$ (e.g. a state that satisfies the test objective is also a goal position for the system). Furthermore, we assume that the test is terminated at that position, making these states sink states on the game graph, where the only outgoing transitions are a self-loop between the corresponding state in V_{sys} and in V_{test}. Then

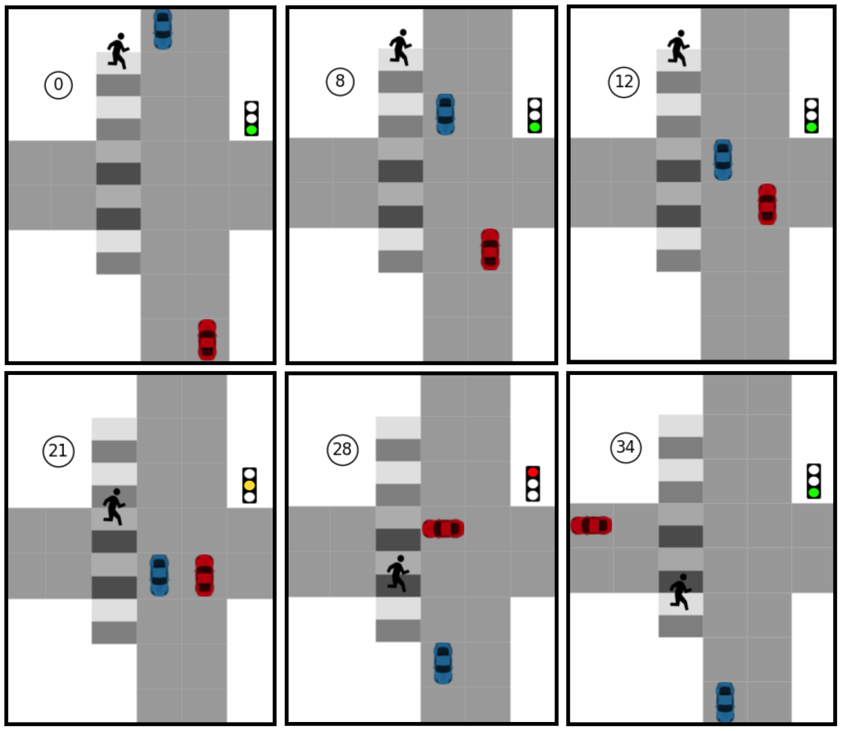


Figure 3.4: Snapshots during the test execution for Example 3.4 generated by our framework. We observe that the system under test (red car) has to wait for the blue car first, and then for the pedestrian on the crosswalk. Passing this test showed that it correctly detected both agents.

we can identify the set of states $\mathcal{I} = \{i_0, \ldots, i_n\} \subseteq V$, that satisfy the propositional formula φ_{test}, we will also refer to $\mathcal{I}$ as the goal states. For each state $i \in \mathcal{I}$, we can find the backward reachable set of states $\mathcal{V}^i \subseteq V$, that we can partition into $\{\mathcal{V}^i_0, \ldots \mathcal{V}^i_m\}$, where $\mathcal{V}^i_k$ is the set of states in V from which goal i can be reached in k steps. From this, the partial order, $\mathcal{P}^i = (\{\mathcal{V}^i_0, \ldots, \mathcal{V}^i_m\}, \leq)$, is derived such that $\mathcal{V}^i_l \leq \mathcal{V}^i_{l-1}$ for all steps $0 \leq l \leq m$, for all goals i in $\mathcal{I}$.

Example 3.3 (Lane Change - continued)**.** In the lane change example, the states in $\mathcal{I}$ correspond to the different ways in which the system can merge *in between* the two test agents. For the given track length of 10 cells, there are eight different possible positions that the cars can be in after a successful merge in between maneuvers.

Now consider a system progress requirement and test objectives that do not coincide at a single state. To be able to construct a partial order towards a desired goal state while ensuring that any temporal constraints are satisfied, we need a way to remember the history during the test execution. We will outline the construction of an *auxiliary game graph* $G_{\text{aux}} = (V_{\text{aux}}, E_{\text{aux}})$, on which we can then construct a partial order towards a desired set of states $\mathcal{I} \subseteq V_{\text{aux}}$. We will outline this procedure for two progress requirements in the test objective, but this procedure can be extended to multiple progress requirements without loss of generality. Assume we are given a combined test objective guarantee $\varphi_{\text{test}} = \Diamond p_1 \wedge \Diamond p_2$, with p_1, and p_2 propositional formulas, and the corresponding game graph G. The system progress specification corresponds to reaching the goal labeled g, given as $\varphi_{\text{sys}}^{\text{prog}} = \Diamond g$ with g propositional formulas. We start by making four copies of the game graph $G = (V, E)$ — $G_0 = (V_0, E_0)$, $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, and $G_3 = (V_3, E_3)$. Each set of states V_0, V_1, V_2, and V_3, along with sets of edges E_0, E_1, E_2, and E_3 is a copy of V and E, respectively. Let $\mathcal{I}_{0,1}$ be the states in V_0 that satisfy the propositional formula p_1, and let $\mathcal{I}_{0,2}$ be the states in V_0 that satisfy the propositional formula p_2. In addition, let $\mathcal{I}_{1,2}$ be the states in V_1 that satisfy the propositional formula p_2, and let $\mathcal{I}_{2,1}$ be the states in V_2 that satisfy the propositional formula p_1. Next, we will connect the subgraphs as follows. Let $(v_0, u) \in V_0$ be an outgoing edge from a vertex $v_0 \in \mathcal{I}_{0,1}$, and let u_1 be the vertex in subgraph G_1 that corresponds to vertex u in G_0. Remove edge (v_0, u) from E_0 and add the edge (v_0, u_1) to the set E_{connect}. Similarly let $(v_0, u) \in V_0$ be an outgoing edge from a vertex $v_0 \in \mathcal{I}_{0,2}$, and let u_2 be the vertex in subgraph G_2 that corresponds to vertex u in G_0 and remove (v_0, u) from E_0 and add (v_0, u_2) to E_{connect}. Let $(v_1, u) \in V_1$ be an outgoing edge from a vertex $v_1 \in \mathcal{I}_{1,2}$, and let u_3 be the vertex in subgraph G_3 that corresponds to vertex u in G_3 and remove (v_1, u) from E_1 and add (v_1, u_3) to E_{connect}. In the same way we connect the nodes in $u_2 \in \mathcal{I}_{2,1}$ to their corresponding node in $v_3 \in V_3$ by removing the edges $(u_2, v) \in E_3$ and adding an edge (u_2, v_3) to E_{connect}. We can then create the auxiliary game graph $G_{\text{aux}} := (V_{\text{aux}}, E_{\text{aux}})$, where $V_{\text{aux}} := V_0 \cup V_1 \cup V_2 \cup V_3$, and $E_{\text{aux}} := E_0 \cup E_1 \cup E_2 \cup E_3 \cup E_{\text{connect}}$. The goal vertices $v \in V_{\text{aux}}$ are such that $v \in V_3$, and the states v satisfy the propositional formula g, and are denoted as $\mathcal{I}_g \subseteq V_{\text{aux}}$. The resulting graph is illustrated in Figure 3.5. The states in which p_1 is satisfied are highlighted in blue, the states where p_2 is satisfied are highlighted in yellow, and the goal state $s \in \mathcal{I}_g$ is highlighted in orange. The different paths via the layered graphs correspond to the different orderings in which states that satisfy p_1 and p_2 can be visited during the test. For each goal $i \in \mathcal{I}_g$, we can construct the partial order $\mathcal{P}^i = (\{\mathcal{V}_0^i, \ldots, \mathcal{V}_n^i\}, \leq)$ from the backward reachable set $\mathcal{V}^i \in V_{\text{aux}}$.

Lemma 3.4. Given an auxiliary game graph G_{aux}, an initial state $v_0 \in G_{\text{aux}}$, and goal states $\mathcal{I}_g \subseteq G_{\text{aux}}$, if $v_0 \notin \bigcup_{i \in \mathcal{I}} \mathcal{P}^i$, then there does not exist a successful strategy for the test agents.

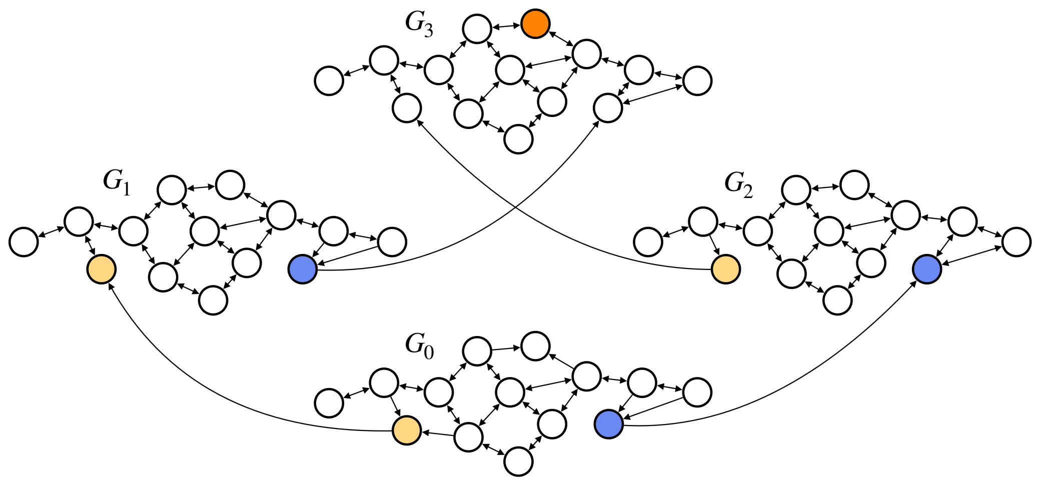


Figure 3.5: Auxiliary game graph G_{aux} illustration encoding temporal constraints for a single goal state (orange).

Example 3.4 (Unprotected Left Turn - continued)**.** In the case of the unprotected left turn example, we require enforcing the temporal constraints. We construct G_{aux}, where the goal states $\mathcal{I}_g$ correspond to the different orders in which the system car could have waited for the approaching car and the pedestrian individually. We find the winning set using the partial orders $\mathcal{P}^i$ for all $i \in \mathcal{I}_g$ on G_{aux}.

Since the winning set is a disjunction of winning sets for multiple system goals, a strategy that always makes progress towards any goal in $\mathcal{I}_g$ does not necessarily have to reach the goal set $\mathcal{I}$. For two goals $i, j \in \mathcal{I}_g$, a policy could result in a live lock by alternating between making progress towards i, while increasing the steps required to reach j, and vice-versa. To mitigate this, we assume that partial orders do not contain such cycles.

Receding Horizon Synthesis as Test Policy Filter. We leverage receding horizon synthesis presented in [158] to scalably compute the set of states $\mathcal{W}$ from which the test environment can realize the test specification on the system in a test execution. Note that we are not synthesizing a test strategy using the receding horizon approach, but are instead using $\mathcal{W}$ as a filter on a search algorithm (MCTS) that finds an optimal (in this case, most difficult) test policy. Further details on applying receding horizon strategies for temporal logic planning can be found in [158]. A distinction in our work is that there can be multiple states in a graph G that satisfy a progress requirement on the test specification. Let $\mathcal{I}_g$ be the set of goal states on G (or G_{aux}). Specifically, for some goal $i \in \mathcal{I}_g$, if the product state starts at j steps from i (i.e. $v \in \mathcal{V}^i_{j+1}$), the test agent strategy is required to guide the product state to $\mathcal{V}^i_{j-1}$. The corresponding GR(1) specification for the test agents is

$$\psi^i_j = (v \in \mathcal{V}^i_{j+1} \wedge \Phi \wedge \square \varphi^{\text{safe}}_{\text{sys}} \wedge \square \diamond \varphi^{\text{prog}}_{\text{sys}}) \rightarrow (\square \diamond (v \in \mathcal{V}^i_{j-1}) \wedge \square \varphi^{\text{safe}}_{\text{test}} \wedge \square \Phi), \quad (3.13)$$

where Φ is the invariant condition that ensures that ψ^i_j is realizable. See [158] for further details on how this invariant can be constructed. Since there are $|\mathcal{I}_g|$ different ways to satisfy the goal requirement p, and the test specification requires that we satisfy p for at least one $i \in \mathcal{I}_g$. To capture this in the receding horizon framework, the test execution must progress toward at least one $i \in \mathcal{I}_g$. For j steps away from the goal set $\mathcal{I}_g$, this is formally stated as,

$$\Psi^{\mathcal{I}}_j = \vee_{i \in \mathcal{I}} \psi^i_j \,. \quad (3.14)$$

The set of states from which the test environment has a strategy that satisfies the specification in equation (3.14) is the short horizon filter, denoted by $\mathcal{W}^{\mathcal{I}}_j$. Let $j_{\max}$ denote the supremum of all shortest paths from a vertex $v \in V$ to some $i \in \mathcal{I}_g$. Then, the overall test policy filter is the union of short-horizon test policy filters,

$$\mathcal{W}^{\mathcal{I}} = \bigcup_{j=1}^{j_{\max}} \mathcal{W}^{\mathcal{I}}_j \,. \quad (3.15)$$

Finding the Test Agent Policy. Ensuring that any test agent strategy remains in the winning set $\mathcal{W}^{\mathcal{I}}$ enforces that the resulting test execution σ will satisfy the corresponding test objective guarantee φ_{test}. However, in addition, we require the resulting test execution to also be a difficult test, characterized by a high difficulty score $\rho(\sigma)$. We will apply the winning set $\mathcal{W}^{\mathcal{I}}$ as a filter to guide the rollouts in Monte Carlo tree search (MCTS) to ensure that every available action remains in the winning set $\mathcal{W}^{\mathcal{I}}$. MCTS is a heuristic search algorithm that combines the benefits of random sampling with tree search procedures. MCTS had a significant impact on the artificial intelligence community and has achieved great success in decision-making for complex problems that can be represented as a tree [30]. Using MCTS with an upper confidence bound (UCB) was introduced in [95] as upper confidence bound for trees (UCT) which guarantees that given enough time and memory, the result converges to the optimal solution. We use MCTS to find the test agent strategy π^*_{test}, such that the resulting test execution $\sigma_{\pi_{\text{sys}} \times \pi_{\text{test}}}$ maximizes the difficulty metric ρ. The complexity of this approach is driven by the complexity of the construction of the winning set for the GR(1) specification, which is $O(|V|^3)$, where V is the size of the state space. In this case, the state space is all valuations of the system and tester variables used in the GR(1) specification. In particular, to improve the scalability of this framework, we are using a receding horizon approach to find the winning sets, for details on the time complexity please refer to [158]. The complexity for MCTS is exponential in the depth of the tree [144]. The number of rollouts and iterations are design variables, that can be chosen to ensure convergence for the particular application and chosen search policy.

Algorithm 3.1 Merge Unit Tests ($\mathfrak{t}_{\text{test},1}, \mathfrak{t}_{\text{test},2}, \varphi_{\text{sys}}, \mathcal{T}_{\text{sys}}, \mathcal{T}_{\text{test},1}, \mathcal{T}_{\text{test},2}, \rho$)

```
 1: procedure CombinedTestStrategy($\mathfrak{t}_{\text{test},1}, \mathfrak{t}_{\text{test},2}, \mathcal{T}_{\text{sys}}, \mathcal{T}_{\text{test},1}, \mathcal{T}_{\text{test},2}, \rho$)
Input: Unit test structures $\mathfrak{t}_{\text{test},1} = (\mathcal{C}_{\text{sys},1}, \mathcal{C}_{\text{obj},1})$ and $\mathfrak{t}_{\text{test},2} = (\mathcal{C}_{\text{sys},2}, \mathcal{C}_{\text{obj},2})$, System
    $\mathcal{T}_{\text{sys}}$, unit test environments $\mathcal{T}_{\text{test},1}$ and $\mathcal{T}_{\text{test},2}$, and difficulty metric $\rho$,
Output: Merged test strategy $\pi_{\text{test},m}$
 2:     $\mathcal{T}_{\text{test}} \leftarrow \mathcal{T}_{\text{test},1} \times \mathcal{T}_{\text{test},2}$      ▷ Combine unit test environments
 3:     $\mathcal{T}_{\text{prod}} \leftarrow \mathcal{T}_{\text{sys}} \times \mathcal{T}_{\text{test}}$      ▷ Product transition system
 4:     $G \leftarrow$ Game graph from product transition system $\mathcal{T}_{\text{prod}}$
 5:     $\mathfrak{t} \leftarrow \mathfrak{t}_1 \parallel \mathfrak{t}_2$      ▷ Combine test structure
 6:     $\varphi_{\text{obj}} \leftarrow G_{\text{obj},1} \wedge G_{\text{obj},2}$      ▷ Refined combined test objective
 7:     $G_{\text{aux}} \leftarrow$ Construct auxiliary game graph
 8:     $\mathcal{I} = \{s \in \mathcal{V}_{\text{aux}} \mid s \models \psi^f_{\text{test},m}\}$      ▷ Define goal states
 9:     for $i \in \mathcal{I}$ do
10:         $\mathcal{P}^i \leftarrow \{(\mathcal{V}^i_p, \dots, \mathcal{V}^i_0)\}$      ▷ Partial order on $G_{\text{aux}}$ for goal $i$
11:         $\psi^i_j \leftarrow$ Equation 3.13      ▷ Specification for goal $i$ at distance $j$
12:     for $j \in [0, \dots, j_{\max}]$ do      ▷ Maximum number of steps to the goal $j_{\max}$
13:         $\Psi^{\mathcal{I}}_j \leftarrow$ Equation 3.14      ▷ Receding horizon specification for distance $j$
14:     $\mathcal{W}^{\mathcal{I}} \leftarrow$ Equation 3.15      ▷ Test policy filter for goal set $\mathcal{I}$
15:     $\pi_{\text{test}} \leftarrow$ Search for test policy that maximizes $\rho$ guided by $\mathcal{W}^{\mathcal{I}}$
16:     return $\pi_{\text{test}}$
```

Example 3.3 (Lane Change - continued)**.** Returning to the lane change example, we define the difficult metric as the x-coordinate of the cell in which the finishes the lane change maneuver. The system model is such that it will change lanes if the diagonal grid cell is not occupied by a test agent, otherwise, it will randomly decide to move forward or stay in the same cell. We search for the strategy for the test agents using the filtered MCTS procedure outlined above, where we simulate the system and the test agents during each rollout. Figure 3.6 shows the resulting test execution, where the system merges in between the two tester agents in cell 9, which corresponds to the most difficult test.

Example 3.4 (Unprotected Left Turn - continued)**.** The test execution found using this framework is depicted in Figure 3.4. The system under test (red) takes an unprotected left turn and waits for the pedestrian and the car (blue) individually. In

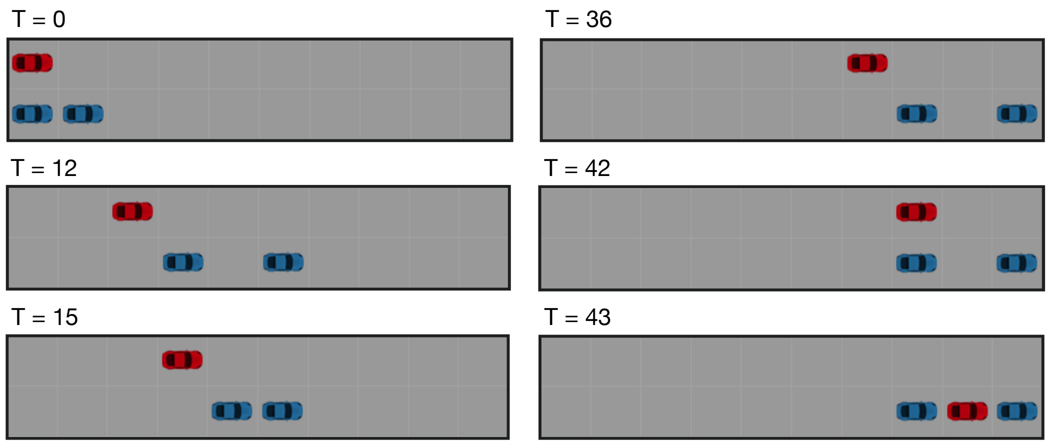


Figure 3.6: Snapshots during the execution of the test generated by this framework for different timesteps T. The system under test (red car) merges into the lower lane between the two test agents (blue cars) at $T = 43$.

the snapshots at time steps 8 and 12, the system waits just for the approaching car, and in time step 21, it waits just for the pedestrian.

3.7 Conclusion

In this chapter, we presented a formal framework to characterize tests and allow reasoning over the test structures. We defined how to compare test campaigns, combine and split tests, and illustrated the procedure on examples. Furthermore, we showed how to find a difficult test policy from a specification in the case of a combined system and tester goal by receding horizon winning set synthesis and a search procedure. Additionally, we showed how to find the test agent policy for a combined test under the addition of temporal constraints by manually constructing an auxiliary game graph. In the following chapter, we will investigate how to generalize and automate the test environment generation and test agent policy synthesis.

Chapter 4

SYNTHESIZING REACTIVE TEST ENVIRONMENTS

4.1 Introduction

One major challenge to automating testing lies in creating the environment for a test, which, in the case of autonomous vehicles, involves placing any obstacles or finding the strategy for agents controlled by the test engineer. Every test must strike a delicate balance between ensuring that the system is challenged sufficiently, while also allowing that a correctly functioning system can successfully pass the test. In the previous chapter, we presented an approach to find the strategy for a combined test utilizing winning sets and Monte-Carlo Tree Search (MCTS). The limitations of the winning-set and MCTS-based approach become evident due to the required assumptions as outlined in Section 3.6. Specifically, the test agent accepting states were required to be a subset of the system's accepting states. Consequently, the test execution could only be routed towards the system's acceptance states that aligned with the test objective acceptance states. This limits the possible test cases significantly.

Instead, we desire to automatically construct a test strategy for independent system and test objectives. Specifically, the system must be unaware of the specific test objective, because disclosing the entire test to the system—and in turn the system designer—would defeat the purpose of testing. Moreover, states in which the test agents can violate the system's assumptions are part of the winning set for the test objective. However, entering these states needs to be avoided as it would create an impossible test. Hence, we need to reason about the system's perspective as well as the progress toward satisfying the test objective separately. However, we do not want to constrain the system more than necessary. Preventing an incorrect system from making wrong decisions would be detrimental as the entire premise of testing is about observing whether a system makes the correct choice given alternative options.

We desire to systematically find a test strategy that strikes a balance between allowing the system the freedom to make decisions, even if they are erroneous, while minimally interfering with the system. In this chapter, we show that this routing problem is NP-hard, and we present a framework that reasons over flows in a net-

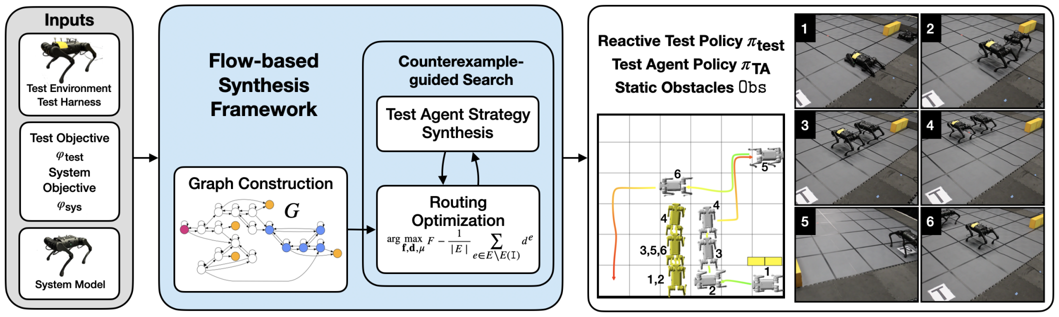

Inputs
Test Environment
Test Harness
Test Objective
φ_{test}
System Objective
φ_{sys}
System Model
Flow-based Synthesis Framework
Graph Construction
G
Counterexample-guided Search
Test Agent Strategy Synthesis
Routing Optimization
$\arg\max_{\mathbf{f},\mathbf{d},\mu} F - \frac{1}{|E|}\sum_{e\in E\setminus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

Figure 4.1: Overview of the flow-based synthesis framework.

work instead of paths, which allows for tractable implementations of this inherently combinatorial problem.

We present a framework that characterizes the test behavior in the form of a system objective and a test objective consisting of reachability, reaction, and safety tasks; and finds a reactive test strategy for a given system model that is guaranteed to restrict the system as minimally as possible, while ensuring that a correctly designed system will be able to satisfy the system and test objective.

In this chapter, we present a flow-based test synthesis approach that allows us to find static and reactive environments, as well as strategies for dynamic test agents. Specifically, we:

1. Present an approach to construct graphs that allows us to reason over the test executions from both the test and the system's perspectives.
2. Frame the routing problem as a network flow optimization for different available test environments.
3. Present a procedure for synthesizing the test agent strategy corresponding to the result of the optimization from a GR(1) specification.
4. Employ a counter-example guided search to find a realizable test agent strategy.
5. Establish that the complexity of the routing problem is NP-hard.
6. Demonstrate the framework on several examples in simulation and hardware experiments.

The work in this chapter appeared in [64] and a preliminary version appeared in in [14]. This work was done jointly with Apurva Badithela.

4.2 System Under Test and Test Environment

In this section, we will focus on the models used for the system under test and the test environment, which consists of agents and other obstacles that the test engineer can control. The system under test, also referred to as the system, is defined as follows.

Definition 4.1 (System)**.** The system under test is modeled as a finite transition system T (according to Definition 2.3) with a single initial state, that is, $|T.S_0| = 1$. For every state of the system, there exists a self-loop transition, which corresponds to a stay-in-place action. We require that there exists at least one terminal state for the system, such that once the system arrives in that state its only available action is to remain there.

For ease of notation, we denote the set of transitions in T as $T.E$, where for all $s \in T.S$ and for all $s' \in T.S$, the transition $(s, s') \in T.S$ if there exists $a \in T.A$ such that $T.\delta(s, a) = s'$.

For a simpler presentation, we assume that all system transitions (except for sink states) are bidirectional. That is, for all $(s, s') \in T.E$, if s' is not a terminal state, we also have $(s', s) \in T.E$. This assumption can be relaxed as explained in Remark 4.3.

An *execution* σ is an infinite sequence $\sigma = s_0 s_1 \ldots$, where $s_0 \in S_0$ and $s_k \in T.S$ 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* is a function $\pi : (T.S)^* T.S \rightarrow T.A$.

We define a test harness that defines which state-action (s, a) pairs of the system can be restricted during the test, such that the system is prevented from taking action a from state $s \in T.S$.

Definition 4.2 (Test Harness)**.** Let the set of actions $A_H \subseteq T.A$ denote the subset of system actions that can be restricted by the test harness. The *test harness* $H : T.S \rightarrow 2^{A_H}$ maps states of the transition system to actions that can be restricted from that state.

The test harness corresponds to the authority of the test designer over the system under test. In our examples, we assume that we can only externally influence the system, and not force it to take certain actions. Therefore, A_H does not contain any self-loop actions. The actions in the test harness can be restricted during the test by any agent that is controllable by the test engineer, including any obstacles that

Table 4.1: Sub-task specification patterns.

Name	Formula	
Visit	$\bigwedge_{i=1}^{m} \Diamond p^i$	(4.1)
Sequenced Visit	$\Diamond(p^0 \wedge (\Diamond(p^1 \wedge \ldots \Diamond p^m)))$	(4.2)
Safety	$\Box \neg p$	(4.3)
Instantaneous Reaction	$\Box(p \rightarrow q)$	(4.4)
Delayed Reaction	$\Box(p \rightarrow \Diamond q)$	(4.5)

are placed by the test engineer. The test environment therefore corresponds to the realization of the transition restrictions in the test harness in the following ways.

Definition 4.3 (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 $(s, s') \in T.E$ is a restriction on the system transition (s, s') that is placed at the beginning of a test and remains in place for the entire duration of the test. A *reactive obstacle* on $(s, s') \in T.E$ is a restriction on the system transition (s, s') that can be enabled and disabled over the course of the test, making it a temporary restriction for the system. A *dynamic test agent* is modeled by the transition system T_{tester}, and can occupy states in $T.S$, thus restricting the system from entering the state that the test agent occupies.

Similarly to Chapter 3, we will characterize the desired test behavior using a specification. To this end, we will introduce the system objective and the test objective that will contain specification sub-tasks as described in Table 4.1. These sub-tasks are defined over atomic propositions AP and can be evaluated over system states $T.S$. When both the system and the test objective are satisfied together, they describe the desired test behavior. In particular, it is reasonable to assume that for a realistic test, some aspects of the test will not be revealed to the system beforehand. These aspects that the system is not aware of are captured in the test objective.

Definition 4.4 (Test Objective)**.** The *test objective* φ_{test} contains at least one visit or sequenced visit sub-task or a conjunction of these sub-tasks.

In contrast to the test objective, some aspects of the test need to be known to the system. For example, for a test that ensures that a system does not enter unsafe regions, the system needs to be aware of which regions are considered unsafe, and that it needs to avoid unsafe areas. To ensure that a test execution satisfies a reaction task, the system is required to be aware of the correct reaction it is expected to perform.

Definition 4.5 (System Objective)**.** The *system objective* φ_{sys} contains at least one visit or sequenced visit sub-task. In addition, it can also contain a conjunction of safety, instantaneous and/or delayed reaction, and visit and/or sequenced visit sub-tasks.

To ensure that we can route a test execution through a desired behavior to a final goal position, we require the test objective and system objective to at least consist of a single reachability task, respectively. From this requirement, it follows that one of the reachability tasks in the system objective needs to correspond to a sink state on the transition system T. Furthermore, encoding a reaction task requires including the expected reaction in the system objective, and ensuring that the reaction is triggered by including the trigger as a reachability specification in the test objective. These definitions allow us to construct two Büchi automata that encode these specifications. The Büchi automaton $\mathcal{B}_{\text{test}}$ corresponds to the test objective φ_{test}. The Büchi automaton $\mathcal{B}_{\text{sys}}$ corresponds to the system objective φ_{sys}. We say that the *system reaches its goal* if the system trace is accepted by $\mathcal{B}_{\text{sys}}$, and thus satisfies the system objective. To keep track of the event-based history of the test execution concerning the system objective and the test objective we define the specification product making use of Definition 2.5.

Definition 4.6 (Specification Product)**.** The *specification product* is the product of two deterministic Büchi automata $\mathcal{B}_\pi := \mathcal{B}_{\text{sys}} \otimes \mathcal{B}_{\text{test}}$. The states $(q_{\text{sys}}, q_{\text{test}}) \in \mathcal{B}_\pi.Q$, where $q_{\text{sys}} \in \mathcal{B}_{\text{sys}}.Q$ and $q_{\text{test}} \in \mathcal{B}_{\text{test}}.Q$, capture the event-based progression of the test and are referred to as the history variables.

Each state of the specification product $q \in \mathcal{B}_\pi.Q$ is a tuple of the corresponding states in $\mathcal{B}_{\text{sys}}.Q$ and $\mathcal{B}_{\text{test}}.Q$, thus remembering the progress of the test concerning the system and test objective, respectively.

In addition to satisfying the system objective, a system under test also needs to be able to operate safely in its environment, which includes not colliding with any test agent or obstacles, and ensuring that any low-level controller implementation can be modeled by T.

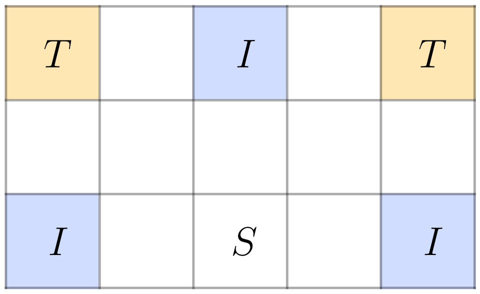
Definition 4.7 (System Guarantees)**.** The system *guarantees* are a conjunction of the system objective, its initial condition, safe interaction with the test environment, and the system's dynamics given by T.

To enable the system to satisfy its guarantees, it is necessary to define its operating environment, captured by the system assumptions.

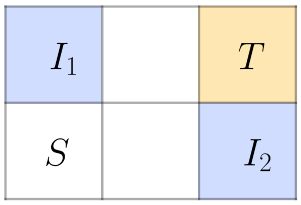
Definition 4.8 (System Assumptions)**.** The *assumptions* that the system makes on its environment are:
A1. The test environment can consist of static obstacles, reactive obstacles, and test agents whose dynamics are known to the system.
A2. The test environment will not take any action that will inevitably lead to unsafe behavior (e.g. the test agent will not collide into the system).
A3. At any time during the test execution, there will always be a path for the system to reach its goal.
A4. The system can choose to break any livelock and make progress toward its goal.

A *correct system strategy* is a strategy that satisfies the system's guarantees given that the system's assumptions are satisfied. The system's assumptions require a path to exist at any time during the test execution. However, this path is allowed to change over time. This corresponds to the system assuming that the test agent will not block it forever without specifically stating any restrictions on the test agent's behavior. This specification cannot be expressed as an LTL formula as the test agents would be assumed to behave in the worst-case manner. Thus the test agents are neither adversarial nor fully cooperative and in many cases, this cannot be captured by an LTL specification.

In real-world applications, it is likely that tests have a defined start and a defined endpoint. In our framework, we will rely on LTL specifications, which are evaluated over infinite traces. Thus, we need to bridge the gap between the finiteness of a real-world test and the infinite traces required to evaluate an LTL formula. To achieve this, upon termination of the test execution in state s_n, we supplement the trace by an infinite suffix s_n^ω, effectively extending the finite trace σ_n to the infinite trace $\sigma = \sigma_n s_n^\omega$. Constructing this infinite trace allows us to benefit from the tools available for LTL. Interpreting LTL over finite traces has been an active area of research in [42, 43, 69, 110]. This approach of supplementing the trace with the infinite suffix raises the question of when a test should be terminated. The decision of when to terminate the test will dictate whether this test trace will hold



(a) Example 4.1.



(b) Example 4.2.

Figure 4.2: Grid world layouts for examples.

the information required to evaluate the success of the test. Tests that are terminated prematurely might result in inconclusive results [18]. As previously described, we require the system transition system to contain a terminal state that corresponds to a reachability task in the system objective; this corresponds to the system goal state, which would ensure that the test was successfully completed. We assume that the test engineer will allow the system enough time to reach this goal state. This means that for a system that enters an unsafe state, or for any reason stops making progress towards the goal for an unreasonably long time, the test engineer can decide to terminate the test. In that case, the system did not pass the test and this unexpected behavior needs to be analyzed in detail.

In this chapter, we will describe the flow-based synthesis framework and illustrate the key concepts on two running examples.

Example 4.1 (Static Grid World)**.** The system under test is an agent on the grid world shown in Figure 4.2a. It can transition from one cell to any adjacent cell, in either direction (N-E-S-W); we will refer to this as a standard grid world. The test environment consists of only static obstacles. The system's initial position is designated by S, and the goal state is designated T, where the system objective is given as $\varphi_{\text{sys}} = \Diamond T$. The system Büchi automaton $\mathcal{B}_{\text{sys}}$ is shown in Figure 4.3a with the acceptance states highlighted in yellow. The test objective is to route the test execution through one of the states labeled I, encoded in the test objective, $\varphi_{\text{test}} = \Diamond I$. The corresponding test objective $\mathcal{B}_{\text{test}}$ automaton is illustrated in Figure 4.3b, with the acceptance states highlighted in blue. The specification product $\mathcal{B}_{\text{prod}}$ with the acceptance states corresponding to the system and test objective is shown in Figure 4.3c.

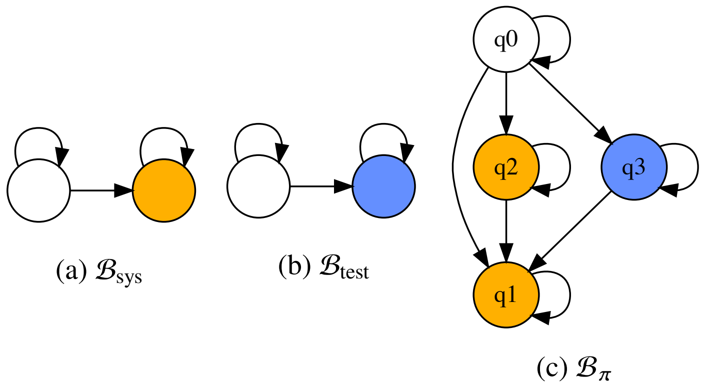

q0
q2
q3
q1
(a) $\mathcal{B}_{\text{sys}}$
(b) $\mathcal{B}_{\text{test}}$
(c) $\mathcal{B}_{\pi}$

Figure 4.3: Automata for Example 4.1.

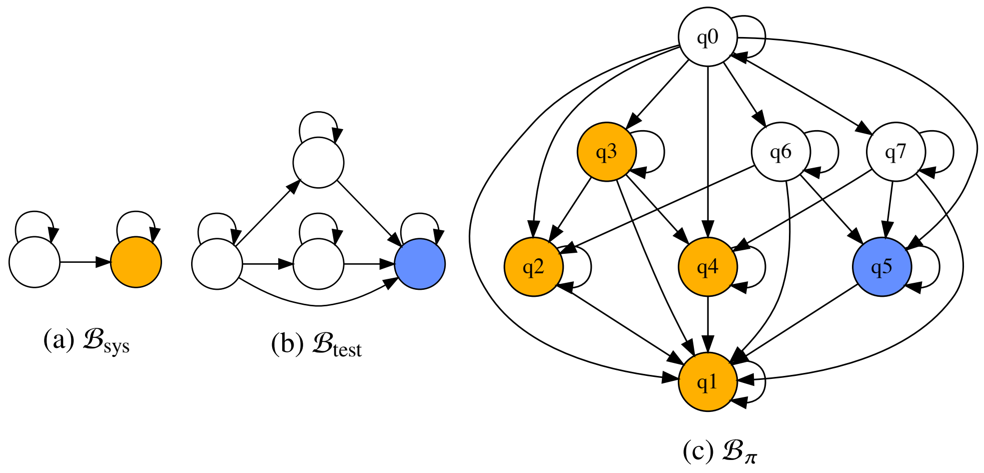

q0
q3
q6
q7
q2
q4
q5
q1
(a) $\mathcal{B}_{\text{sys}}$
(b) $\mathcal{B}_{\text{test}}$
(c) $\mathcal{B}_{\pi}$

Figure 4.4: Automata for Example 4.2.

Example 4.2 (Reactive Grid World)**.** In this example, the system can transition on a standard grid world with cells denoted as I_1, I_2, and T, illustrated in Figure 4.2b. The test environment for this example consists of reactive obstacles. The system's initial position is labeled S, and its goal is reaching the cell labeled T. Thus, the system objective is $\varphi_{\text{sys}} = \Diamond T$. The test objective requires the system to visit both the cell labeled I_1 and the cell labeled I_2 in any order, making the test objective $\varphi_{\text{test}} = \Diamond I_1 \wedge \Diamond I_2$. The corresponding automata for this example are illustrated in Figure 4.4, where the states shaded in yellow represent the system objective acceptance states, and the states highlighted in blue represent the test objective acceptance states.

4.3 Reactive Test Strategy Problem

In this section, we will state the test environment synthesis problem. We assume that the test engineer provides a system objective and a test objective, which describes the desired test behavior. Then, we find a reactive test strategy for which every successful test execution will satisfy the test objective and the system objective.

Definition 4.9 (Reactive Test Strategy)**.** A reactive test strategy $\pi_{\text{test}} : (T.S)^*T.S \rightarrow 2^{A_H}$ defines the set of restricted system actions at each state during its execution σ. For some finite prefix $s_0 \dots s_i$ of execution σ, $\pi_{\text{test}}(s_0 \dots s_i) \subseteq H(s_i)$ is the set of actions that the system cannot take from state s_i. A test environment can realize a reactive test strategy π_{test} if it can restrict system actions according to π_{test}.

Let $\Sigma_{\text{fin}} := (T.S)^*T.S$ be the set of all finite prefixes of system traces. At each time step $k \geq 0$, a correct system strategy $\pi_{\text{sys}} : \Sigma_{\text{fin}} \rightarrow T.A \setminus \pi_{\text{test}}(\Sigma_{\text{fin}})$ must pick from available actions at state s_k. The resulting system execution is denoted as $\sigma(\pi_{\text{sys}} \times \pi_{\text{test}})$.

Remark 4.1. Note that the test environment externally blocks system transitions and, as a consequence, restricts corresponding actions that the system can safely take. When actions are restricted by the test environment, the system strategy π_{sys} should select from the available actions at each state. Since these restrictions can be placed during the test execution, the system might have to re-plan and choose a different action than originally planned.

Definition 4.10. Given a test environment, system T, system and test objectives, φ_{sys} and φ_{test}, a reactive test strategy π_{test} is said to be *feasible* if and only if: i) the test environment can realize π_{test}, ii) there exists a correct system strategy π_{sys}, and iii) any system execution corresponding to a correct π_{sys} satisfies the system and test objectives: $\sigma(\pi_{\text{sys}} \times \pi_{\text{test}}) \models \varphi_{\text{test}} \wedge \varphi_{\text{sys}}$.

Note that the test strategy is not aiding the system in achieving the system objective; it only restricts system actions such that the test objective is realized. That is, the system is free to choose an incorrect strategy, in which case there are no guarantees.

Furthermore, the test strategy should allow the system to make multiple decisions at each step of the execution, if possible, as opposed to leaving a single allowed action. For any system execution $\sigma = s_0 s_1 \dots$, every finite prefix of σ maps to a history variable $q \in \mathcal{B}_\pi.Q$. For each σ, we can define a corresponding state-history trace $\vartheta = (s, q)_0, (s, q)_1, \dots$, where history variable q at time step i corresponds to the prefix of $s_0 \dots s_i$ of σ.

Definition 4.11 (Restrictiveness of a Test Strategy)**.** State-history traces ϑ_1 and ϑ_2 are *unique* if they do not share consecutive state-history pairs. 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 *least restrictive* if the cardinality of Θ_u is maximized.

While the set of all state-history traces Θ can be infinite, the set of unique traces Θ_u is finite. This is because the system and the specification product each have a finite number of states and history variables, respectively. As every trace in the set Θ_u is unique, this results in a finite set 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 least restrictive.

The restrictions on system actions placed by the test strategy can be realized in several ways in the test environment. For example, a dynamic test agent, together with any static obstacles, can be used to enforce the test strategy. This leads to the second problem of synthesizing a reactive strategy for a test agent to realize the test strategy. That is, at each time step of the test execution, the test environment consisting of an agent and static obstacles restricts the system actions according to π_{test}.

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_{tester}, find the test agent strategy π_{tester} and the set of static obstacles `Obs` that: i) satisfy the system's assumptions on its environment, and ii) realize a reactive test strategy π_{test} that is least-restrictive and feasible.

4.4 Graph Construction

This section focuses on the construction of the graphs required to reason over different possible test executions and the corresponding progress concerning the system and test objectives. We will make use of the synchronous product of a Büchi automaton and a finite transition system from Definition 2.6 to construct the following graphs.

Definition 4.12 (Virtual Product Graph)**.** A *virtual product graph* is the product $G := T \otimes \mathcal{B}_\pi$.

The virtual product graph captures the system trace on the transition system and the progress of the test execution concerning the system and the test objective. To capture the system's perspective, we define the system product graph as follows.

Definition 4.13 (System Product Graph)**.** A *system product graph* is defined as $G_{\text{sys}} := T \otimes \mathcal{B}_{\text{sys}}$.

The system product graph only contains information concerning the system objective and is agnostic to the test objective. This corresponds to the system model that we were provided and the aspects of the tests that were revealed to the system in the system objective. This graph is crucial to ensure that any test that is generated is not impossible from the system's perspective.

Lemma 4.1. For every path $\vartheta_n^{\text{sys}} = (s, q_{\text{sys}})_0 \dots (s, q_{\text{sys}})_n$ on G_{sys}, there exists a corresponding path on G.

Proof. For a given path $\vartheta_n^{\text{sys}} = (s, q_{\text{sys}})_0 \dots (s, q_{\text{sys}})_n$ on G_{sys}, there exists a sequence $q_{\text{test}\,0}, \dots, q_{\text{test}\,n} \in \mathcal{B}_{\text{test}}.Q$ corresponding to the trace $\sigma = s_0, \dots s_n$. By construction of G, the sequence $(s, (q_{\text{sys}}, q_{\text{test}}))_0, \dots, (s, (q_{\text{sys}}, q_{\text{test}}))_n$ is a path on G. $\square$

On the virtual product graph G we will identify nodes that correspond to the acceptance conditions of the system objective and the test objective and denote them especially.

Definition 4.14 (Source, Intermediate, and Target Nodes)**.** We define the *source nodes* $\mathtt{S}$, the *intermediate nodes* $\mathtt{I}$, and the *target nodes* $\mathtt{T}$ as follows:

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

The source node set $\mathtt{S}$ corresponds to the initial condition of the system and is a singleton. The intermediate nodes $\mathtt{I}$ correspond to system states in which the test objective is satisfied. The target nodes $\mathtt{T}$ represent the system states for which the acceptance condition for the system objective is met. Again, to capture the system's perspective we define the system target nodes on G_{sys}.

Definition 4.15 (System Target Nodes)**.** We define the *system target nodes* $\mathtt{S}_{\text{sys}}$ on G_{sys} as

$$\mathtt{T}_{\text{sys}} := \{(s, q) \in G_{\text{sys}}.S \mid q \in \mathcal{B}_{\text{sys}}.F\}. \tag{4.6}$$

Lemma 4.2. For a path $\vartheta_n = (s, q)_0, \ldots, (s, q)_n$ on G, where $(s, q)_0 \in \mathtt{S}$, the corresponding system trace $\sigma_n = s_0 \ldots s_n$ satisfies the system objective, $\sigma \models \varphi_{\text{sys}}$ iff $(s, q)_n \in \mathtt{T}$. Furthermore, if $\sigma \models \varphi_{\text{test}}$, then for some $0 \leq i \leq n$, the path ϑ_n contains a state-history pair $(s, q)_i \in \mathtt{I}$.

Proof. Every path on G corresponds to a possible trace $\sigma = s_0 \ldots s_n$ on $T.S$. By construction of G, the target nodes $\mathtt{T}$ are defined such that if $(s, q)_n \in \mathtt{T}$, then the corresponding infinite trace $\sigma \models \varphi_{\text{sys}}$. Similarly for the set of intermediate nodes $\mathtt{I}$, if ϑ_n contains a state-history pair $(s, q)_i \in \mathtt{I}$ at some time step $0 \leq i \leq n$, then the corresponding infinite trace $\sigma \models \varphi_{\text{test}}$. □

To be able to determine how system action restrictions found on the virtual product graph G map to restrictions on the system product graph G_{sys}, we define the following projection $\mathcal{P}_{G \to G_{\text{sys}}} : G.S \to G_{\text{sys}}.S$ as

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

To denote the state $s \in T.S$ that corresponds to the state $u = (s, (q_{\text{sys}}, q_{\text{test}})) \in G.S$ we write $u.s$. Analogously, we refer to the state $s \in T.S$ that corresponds to $v = (s, q_{\text{sys}}) \in G_{\text{sys}}.S$ by $v.s$.

4.5 Network Flow Optimization

In this section, we will describe how we set up the optimization for the different test environments, explain the purpose of each constraint, and characterize properties of the solution.

Optimization Setup

We define the flow network $\mathcal{G} := (V, E, (\mathtt{S}, \mathtt{T}))$, where $V := G.S$, $E := G.E$, source and target nodes correspond to $\mathtt{S}$ and $\mathtt{T}$, with the corresponding flow $\mathbf{f} \in \mathbb{R}^{|E|}$. For simplicity, we use the same notation to refer to nodes and edges on the graph and the corresponding flow network. The Boolean edge cut vector $\mathbf{d} \in \mathbb{B}^{|E|}$ represents whether edges are cut or not. That is, $d^e = 1$ refers to edge $e \in E$ being cut, and $d^e = 0$ implies that edge e is not cut. This is defined as follows:

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

where E_H corresponds to the edges that cannot be cut by the test harness, defined as

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

The edges into and out of the intermediate $\mathtt{I}$ nodes are denoted as $E(\mathtt{I}) := \{(u,v) \in E \mid u \in \mathtt{I} \text{ or } v \in \mathtt{I}\}$. To solve Problem 4.1, we formulate a mixed-integer linear program (MILP).

Objective. To find the least restrictive test, we want to maximize the system's freedom in satisfying the test objective. To capture this, we optimize for edge cuts that maximize the flow value on $\mathcal{G}$. However, a realization of maximum flow on a network is not unique. To ensure that we do not cut any edges unnecessarily, we subtract the sum of the edge cuts from the flow value:

$$\sum_{\substack{(u,v)\in E, \\ u \in \mathtt{S}}} f^{(u,v)} - \frac{1}{|E|} \sum_{e \in E} d^e. \tag{4.9}$$

The regularizer $\frac{1}{|E|}$ on the sum of edge cuts is chosen such that it will not compete with the maximum flow value on the network. The weighted sum $\frac{1}{|E|}\sum_{e\in E} d^e$ is always between 0 and 1, and binary edge cuts always result in integer flow values. Thus, the optimization will always favor increasing the maximum flow value rather than reducing edge cuts.

Network flow constraints. First, the network flow optimization is subject to the following standard constraints on flow $\mathbf{f}$:

$$\text{Flow constraints (2.2), (2.3), and (2.4) on flow network } \mathcal{G}. \tag{c2}$$

An edge that is cut restricts flow completely, while an edge that is not cut may or may not have flow,

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

Partition constraints. The following constraints ensure that all flow across the network will be routed through $\mathtt{I}$ by partitioning the network. To accomplish this, we adapt the partitioning conditions given in [152] as follows. Except for the $\mathtt{I}$ nodes, we divide the remaining nodes into two groups defined by the partition variable $\boldsymbol{\mu} \in \mathbb{R}^{|V \setminus \mathtt{I}|}$, and ensure that the nodes $\mathtt{S}$ belong to one group, and $\mathtt{T}$ belong to the other:

$$0 \le \mu^v \le 1, \quad \mu^{\mathtt{S}} - \mu^{\mathtt{T}} \ge 1, \forall v \in V \setminus \mathtt{I}. \tag{c4}$$

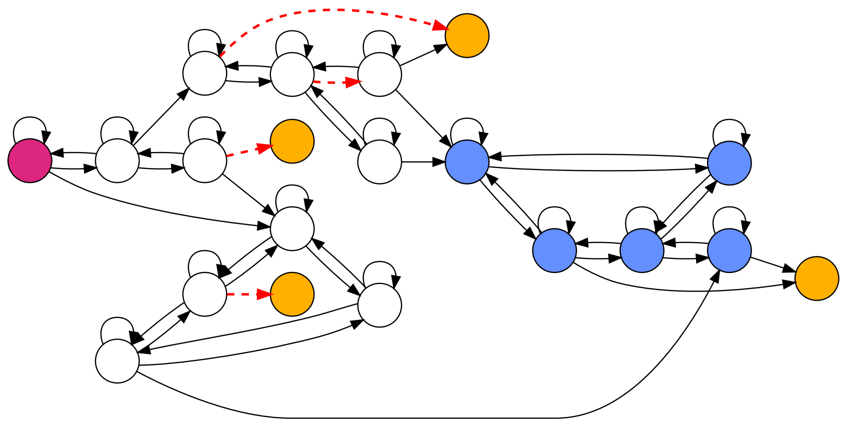


Figure 4.5: Virtual product graph G for Example 4.2 with cuts found from **MILP-REACTIVE** highlighted as red dashed edges.

The two groups are partitioned by the edge cut vector $\mathbf{d}$, where this constraint is only defined over the edges that do not go into or out of nodes in $\mathtt{I}$,

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

Feasibility constraints. To ensure that the test is not impossible from the system's perspective, we map restrictions found on G to G_{sys} via the following feasibility constraints. For each history variable $q \in \mathcal{B}_\pi.Q$, we define the set of state-history pairs that captures the possible first observations of the history variable in a test execution via the function $\mathsf{S}_G : \mathcal{B}_\pi.Q \to G.S$ defined as follows:

$$\begin{aligned} \mathsf{S}_G(q) :=& \{(s,q) \in G.S \mid \\ & \forall((\bar{s},\bar{q}),(s,q)) \in G.E, \ \bar{q} \neq q\}. \end{aligned} \tag{4.10}$$

These sets of states are mapped to G_{sys} as follows:

$$\begin{aligned} \mathsf{S}_{G_{\text{sys}}}(q) :=& \{u \in G_{\text{sys}}.S \mid u = \mathcal{P}_{G \to G_{\text{sys}}}(v), \\ & v \in \mathsf{S}_G(q), \text{ and } \exists \, \text{path}(u, \mathtt{T}_{\text{sys}})\}, \end{aligned} \tag{4.11}$$

where this set is empty if no path from the node u to $\mathtt{T}_{\text{sys}}$ exists on G_{sys}. For each $q \in \mathcal{B}_\pi.Q$ and for each source in $\mathsf{s} \in \mathsf{S}_{G_{\text{sys}}}(q)$, we define a flow network $\mathcal{G}_{\text{sys}}^{(q,\mathsf{s})} := (V_{\text{sys}}, E_{\text{sys}}, (\mathsf{s}, \mathtt{T}_{\text{sys}}))$, where $V_{\text{sys}} := G_{\text{sys}}.S$, and $E_{\text{sys}} := G_{\text{sys}}.E$, with the corresponding flow variable $\mathbf{f}_{\text{sys}}^{(q,\mathsf{s})}$. For each of these flow networks, we define a flow subject to the standard flow constraints:

$$\begin{aligned} & \forall q \in \mathcal{B}_\pi.Q, \forall \mathsf{s} \in \mathsf{S}_{G_{\text{sys}}}(q), \\ & \text{Flow constraints (2.2), (2.3), and (2.4) on network } \mathcal{G}_{\text{sys}}^{(q,\mathsf{s})}. \end{aligned} \tag{c6}$$

For each $\mathcal{G}_{\text{sys}}^{(q,\mathtt{s})}$, we map the edge cuts $\mathbf{d}$ and check that there is still a path from $\mathtt{s}$ to some node in $\mathtt{T}_{\text{sys}}$. This ensures that reactively placing restrictions on system actions does not make it impossible for a correct system strategy to make progress toward its goal. Intuitively, the edge cuts are grouped by the history variable q and checked to ensure that the system has a feasible path when these restrictions are placed on system actions. The edges are grouped by their history variable using the mapping $\mathtt{Gr} : \mathcal{B}_\pi.Q \to 2^{G.E}$, defined as follows:

$$\mathtt{Gr}(q) \coloneqq \{((s,q),(s',q')) \in G.E\}. \tag{4.12}$$

The edge cuts are mapped onto the corresponding $\mathcal{G}_{\text{sys}}^{(q,\mathtt{s})}$ to cut the corresponding flow $\mathbf{f}_{\text{sys}}^{(q,\mathtt{s})}$ as follows:

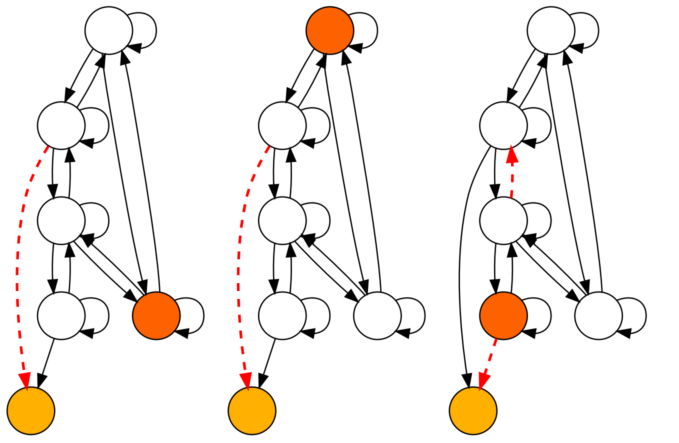
$$\begin{aligned} &\forall q \in \mathcal{B}_\pi.Q, \forall \mathtt{s} \in \mathtt{S}_{G_{\text{sys}}}(q), \forall (u,v) \in \mathtt{Gr}(q), \forall (u',v') \in E_{\text{sys}}, \\ &d^{(u,v)} + f_{\text{sys}}^{(q,\mathtt{s})^{(u',v')}} \leq 1, \;\text{ if } u'.s = u.s \text{ and } v'.s = v.s. \end{aligned}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

Since we are agnostic to the system controller, we need to ensure that a path to the system's goal exists at all times during the test execution. To enforce this, we require a flow of at least 1 on each system flow network $\mathcal{G}_{\text{sys}}^{(q,\mathtt{s})}$,

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

Example 4.2 (Static Grid World - continued)**.** Fig. 4.5 shows the virtual product graph G, with the source $\mathtt{S}$ (magenta), the intermediate nodes $\mathtt{I}$ (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. 4.6a-4.6c show the copies of G_{sys} with their source (orange) and target node (yellow). The graphs in Figs. 4.6a-4.6c correspond to the copies of G_{sys} for the history variables q_0, q_6, and q_7 from $\mathcal{B}_\pi$ shown in Fig. 4.4c. 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.

These feasibility cuts correspond to the reactive constraint setting since edge cuts are placed on $\mathcal{G}$ and depend on the history variable q. Finally, the optimization to identify edge cuts for the reactive test strategy is characterized by the following mixed-integer linear program (MILP) with the cuts $\mathbf{d}$ as the integer variables, and the flow and partition variables taking continuous values.



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

Figure 4.6: System product graphs for Example 4.2 with the cuts found from **MILP-REACTIVE** projected for each history variable highlighted as red dashed edges.

MILP-REACTIVE:

$$\max_{\substack{\mathbf{f},\mathbf{d},\boldsymbol{\mu},\\ \mathbf{f}_{\text{sys}}^{(q,\text{s})}\ \forall q\in\mathcal{B}_\pi.Q\ \forall \text{s}\in \text{S}_{\mathcal{G}_{\text{sys}}}(q)}} \quad F - \frac{1}{|E|}\sum_{e\in E} d^e \qquad \text{s.t.} \quad \text{(c1)-(c3), (c4)-(c5), (c6)-(c8).} \tag{4.13}$$

Static Constraints. We can simplify the feasibility constraints in the case of static obstacles. This corresponds to the requirement that any transition that is restricted will remain restricted for the entire duration of the test. From the system's perspective, the restrictions will not change depending on the history variable q. That is, edges in G corresponding to the same transition in $T.E$ are grouped and share the same cut value:

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

Similarly, the optimization to find edge cuts in a static setting is as follows.

MILP-STATIC:

$$\max_{\mathbf{f},\mathbf{d},\boldsymbol{\mu}} F - \frac{1}{|E|}\sum_{e\in E} d^e \qquad \text{s.t.} \quad \text{(c1)-(c3), (c4)-(c5), (c9).} \tag{4.14}$$

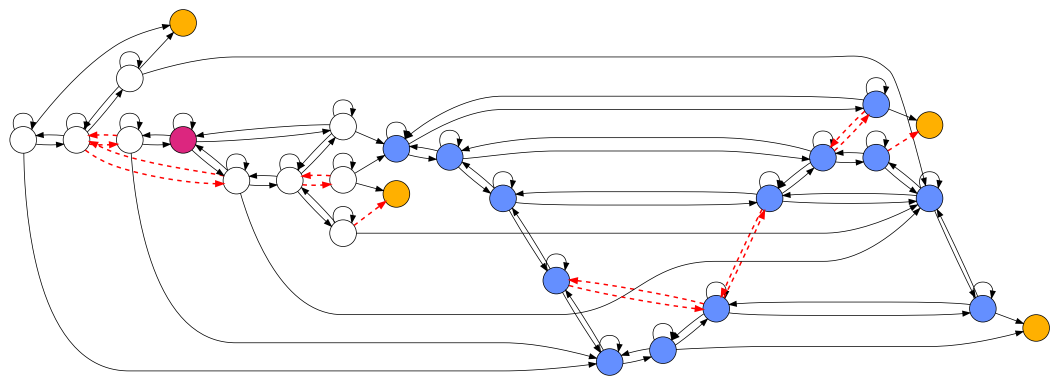


Figure 4.7: Virtual product graph G for Example 4.1 with cuts found using the **MILP-static** highlighted as red dashed edges.

Lemma 4.3. 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 $\mathcal{G}_{\text{sys}} = (V_{\text{sys}}, E_{\text{sys}}, c, (\mathtt{S}_{\text{sys}}, \mathtt{T}_{\text{sys}}))$, where $\mathtt{S}_{\text{sys}} := G_{\text{sys}}.I$. Equation (c8) being satisfied on $\mathcal{G}_{\text{sys}}$ implies that there is a path on G from $\mathtt{S}$ to $\mathtt{T}$ via Lemma 4.1. Additionally, if there is a path on G from $\mathtt{S}$ to $\mathtt{T}$ with the static constraints (c9), then it must be that there exists a path from $\mathtt{S}_{\text{sys}}$ to $\mathtt{T}_{\text{sys}}$ on G_{sys}. □

Remark 4.2. For the reactive constraint setting, we can replace the feasibility constraints (c6)-(c8) by several static constraints. That is, we introduce a copy of $\mathcal{G}$ for each history variable $q \in \mathcal{B}_\pi.Q$ and each source $\mathtt{s} \in \mathtt{S}_G(q)$, denoted $\mathcal{G}^{(q,\mathtt{s})} = (V, E, c, \mathtt{s}, \mathtt{T})$, and require a path from $\mathtt{s}$ to T to exist under a static mapping of the edges in the group $\texttt{Gr}(q)$ by constraint (c9). We choose the former since it reduces the number of variables and constraints in the optimization.

Mixed Constraints. In some cases, it might be desirable to define specific transitions $T.E_{\text{static}} \subseteq T.E$ which require static constraints. The mixed setting of reactive and static transition restrictions can be implemented by enforcing the feasibility constraints (c6)-(c8), and the static constraints (c9) on edges $(u, v) \in E$, where the corresponding transition $(u.s, v.s) \in T.E_{\text{static}}$. Finally, the optimization for the mixed constraint setting is as follows.

MILP-MIXED:

$$\max_{\substack{\mathbf{f},\mathbf{d},\boldsymbol{\mu},\\ \mathbf{f}_{\text{sys}}^{(q,\mathtt{s})}\ \forall q\in\mathcal{B}_\pi.Q\ \forall \mathtt{s}\in\mathtt{S}_{\mathcal{G}_{\text{sys}}}(q)}} \quad F-\frac{1}{|E|}\sum_{e\in E} d^e \tag{4.15}$$
$$\text{s.t.}\quad \text{(c1)-(c3), (c4)-(c5), (c6)-(c8), (c9).}$$

Auxiliary Constraints. Additional constraints can be added to the optimization depending on the test harness or the desired test setup. For example, it might be required to enforce that if an edge is cut, the transition will be blocked in both directions. This can be enforced as follows:

$$\begin{aligned} d^{(u,v)} = d^{(u',v')},\ \forall (u,v),\ (u',v') \in E,\\ \text{if } u.s = v'.s \text{ and } v.s = u'.s. \end{aligned} \tag{c14}$$

Characterizing the Optimization

The optimization described in Section 4.5 returns an edge cut vector $\mathbf{d}^*$ that maximizes the objective in equation (4.9). In this section, we will characterize important properties of the solution. The solution we desire ensures that there is no path from S to T that does not pass through I, and that there exists at least one path from S to T that visits a node in I. As the edge cut vector consists of binary variables, the flow value will be integer-valued. Thus for a strictly positive flow, we are looking for a solution with flow value $F \geq 1$. First, we characterize under which conditions a solution cannot be found. We say that the problem data are *inconsistent* in two scenarios:

i) Given a virtual product graph G there does not exist a path from S to T before placing any edge cuts. Thus the maximum flow value on this network is 0.

ii) Given a virtual product graph G on which there exists a path from S to T, but no path from S to I or from I to T. In this case, the optimization will try to maximize the flow value from S to T, but by the partition constraints, it will be required to place edge cuts to cut all flow that does not pass through the intermediate. On the other hand, the feasibility constraints require there to exist a path to T at any time during the test execution. This is a contradiction that results in the optimization being infeasible.

To facilitate characterizing the optimization solution, we will introduce two networks. These networks will allow us to analyze how the edge cuts will influence any flow on G, the set of cut edges C is defined as

$$C := \{(u,v) \in E \setminus E(\mathtt{I}) \mid d^{*(u,v)} = 1\}. \tag{4.16}$$

First we define the network with cuts $G_{\text{cut}} := (V, E \setminus C, \mathtt{S}, \mathtt{T})$. Then we define the bypass network $G_{\text{by}} := (V \setminus \mathtt{I}, E \setminus (\mathtt{I} \cup C), \mathtt{S}, \mathtt{T})$. As the $\mathtt{I}$ nodes are removed, this network can capture if there exists any flow from $\mathtt{S}$ to $\mathtt{T}$ on G that does not pass through $\mathtt{I}$ after placing the edge cuts, corresponding to a strictly positive flow value on G_{by}, denoted as the *bypass flow value*.

Theorem 4.4. *Given consistent problem data, for each MILP (*(4.13),(4.14),(4.15)*), the optimal cuts C result in a bypass flow value of* 0*.*

Proof. By constraint c4 each node $v \in V \setminus \mathtt{I}$ has an associated potential μ^v, with the potential difference between nodes in $\mathtt{S}$ and nodes in $\mathtt{T}$ being 1. This corresponds to every node in $s \in \mathtt{S}$ having potential $\mu^s = 1$, and every node $t \in \mathtt{T}$ having potential $\mu^t = 0$. On any path $v_0 \dots v_k$ on $\mathcal{G}_{\text{by}}$, where $v_0 \in \mathtt{S}$ and $v_k \in \mathtt{T}$, then we can express the difference in potential values on the path as a telescoping sum: $\mu^{\mathtt{S}} - \mu^{\mathtt{T}} = \sum_{i=0}^{k-1} (\mu^i - \mu^{i+1})$. By constraint (c5), we know that for any edge $(u,v) \in E$, we have that $d^{(u,v)} \geq \mu^u - \mu^v$. 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, due to the edge cut values being binary, we know that for at least one edge (v_i, v_{i+1}) on the path, where $0 \leq i \leq k-1$, the corresponding edge cut value is $d^{(v_i, v_{i+1})} = 1$. By equation (4.16), edges with an edge cut value of 1 belong to the set of cut edges C. Thus, the flow value on $\mathcal{G}_{\text{by}}$, the bypass flow value, is zero. □

Theorem 4.5. *Given consistent problem data, for each MILP (*(4.13),(4.14),(4.15)*), the optimal cuts C are such that for all $q \in \mathcal{B}_\pi.Q$, for all $\mathtt{s} \in \mathtt{S}_{sys}(q)$, there exists a path from $\mathtt{s}$ to $\mathtt{T}_{sys}$ for the system.*

Proof. First, consider the MILP in the reactive setting (4.13). The optimal cuts C correspond to an edge cut vector $\mathbf{d}$ that satisfies the feasibility constraints (c6), (c7), and (c8). The edge cut vales are grouped by their history variable q (see equation (4.12)) and constraint (c7) maps the edge cuts in each group to the corresponding $\mathcal{G}_{\text{sys}}^{(q,\mathtt{s})}$ networks for all $\mathtt{s} \in \mathtt{S}_{\text{sys}}(q)$. This means that all cuts in $\mathtt{Gr}(q)$ are applied to the networks at once, corresponding to the maximum number of obstacles that could be present for that history variable q. By constraint (c8) we require a flow of at least 1 from $\mathtt{s}$ to $\mathtt{T}_{\text{sys}}$ on each network $\mathcal{G}_{\text{sys}}^{(q,\mathtt{s})}$. As the set of edge cuts C is found from the edge cut vector $\mathbf{d}$ that satisfies these constraints, we will have a flow of at least 1, and thus a path to T_{sys} will exist for all $q \in \mathcal{B}_\pi.Q$ after placing the edge cuts. In the static setting (4.14), constraint (c9) ensures that if an edge is cut, it will be cut for all $q \in \mathcal{B}_\pi.Q$. This corresponds to a permanently restricted edge in $\mathcal{G}_{\text{sys}}$. As the optimal edge cuts for consistent problem data ensure that the flow is at least 1 on $\mathcal{G}$, there will be a path to $\mathtt{T}_{\text{sys}}$ on $\mathcal{G}_{\text{sys}}$. The mixed setting (4.15), is subject to the reactive feasibility constraints (c6), (c7), and (c8), with the addition of the static cut constraint (c9). These constraints require a path to exist from $\mathtt{s}$ to $\mathtt{T}_{\text{sys}}$ for every $q \in \mathcal{B}_\pi.Q$, for every $\mathtt{s} \in \mathtt{S}(q)$. □

Theorem 4.5 shows that at any time during the test execution, there exists a path for the system from all sources of that history variable to the system goal $\mathtt{T}_{\text{sys}}$. Under the assumption of bidirectional system transitions, we can show that any node that is reachable during a test execution, (i.e. has a path from the current source $\mathtt{s}$), and had a path to $\mathtt{T}_{\text{sys}}$ before placing any cuts, will still have a path to $\mathtt{T}_{\text{sys}}$ after placing the cuts. This corresponds to the requirement that we do not introduce any dead-ends to the network that did not exist previously.

Remark 4.3. The assumption on bidirectional edges in T can be relaxed by ensuring that the set of edge cuts C does not introduce any deadlocks (i.e. states that are reachable for the system, but no longer have a path to the goal). For this, we can check for every cut in C that the state that the system can reach (the outgoing node of the cut), whether a path to $\mathtt{T}$ still exists. If that is not the case for all cuts in C, then we exclude this solution and resolve the MILP (by adding a constraint analogous to constraint c15 as described in Section 4.7).

Corollary 4.6. By Theorem 4.5 and the assumption of bidirectional system transitions, the set of edge cuts is such that for each $q \in \mathcal{B}_\pi.Q$, for each $\mathtt{s} \in \mathtt{S}(q)$, there will be no nodes in each of the networks $\mathcal{G}_{\text{sys}}^{(q,\mathtt{s})}$ that are reachable from the source $\mathtt{s}$, do not have a path to $\mathtt{T}_{\text{sys}}$, but originally had a path to $\mathtt{T}_{\text{sys}}$ before finding any edge cuts.

Proof. Bidirectional system transitions allow bidirectional transitions within the nodes $(s, q_{\text{sys}}) \in G_{\text{sys}}$ that share the same history variable $q_{\text{sys}} \in \mathcal{B}_{\text{sys}}.Q$ on G_{sys}. Introducing a new dead-end corresponds to creating a set of nodes from which the goal cannot be reached anymore but allowing the system to enter this set. This corresponds to allowing a transition into this set but restricting any transition out of this set. As the system can be inside or outside of this set, this cut configuration cannot be the minimum. However, optimal edge cuts C are the minimum set of edge cuts for that flow value, no unnecessary edges will be cut and the optimization will not return such a cut configuration. □

Lemma 4.7. For each MILP ((4.13),(4.14),(4.15)), the optimal cuts C correspond to maximizing the cardinality of the set of unique state-history traces Θ_u.

Proof. A realization of the flow $\mathbf{f}$ on the virtual product graph $\mathcal{G}$ corresponds to a set of unique state-history traces Θ_u, where the flow value $F = |\Theta_u|$. The MILP objective maximizes the flow value F, and therefore maximizes $|\Theta_u|$. □

4.6 Realizing a Test Strategy

For each setting (static, reactive, and mixed), the optimal cuts from solving the corresponding MILP are used to realize a test strategy with static and/or reactive obstacles. The optimal cuts C for each MILP are parsed into a reactive map $C : \mathcal{B}_\pi.Q \to T.E$, where

$$C(q) \coloneqq \{(s, s') \in T.E \mid ((s, q), (s', q')) \in C^*\}. \tag{4.17}$$

The set $C(q)$ captures cuts that will be used to restrict the system when the state of the test execution is at the history variable q. When the test execution ϑ reaches a state-history pair (s, q) at time step $k \geq 0$, and $C(q)$ contains a system transition $(s, s') \in T.E$, then the reactive test strategy π_{test} will restrict the system action corresponding to this transition. That is, the set of restrictions on the system is given by

$$\begin{aligned} \pi_{\text{test}}(\sigma_k) \coloneqq \{&a \in T.A \mid \\ &s' \in T.\delta(s, a) \text{ and } (s, s') \in C(q)\}. \end{aligned} \tag{4.18}$$

In practice, the reactive test strategy can be realized by the test environment by placing obstacles during the test execution. The set of *active obstacles* $\texttt{Obs}(\sigma_k)$ at time step $k \geq 0$ is defined as the set of all state-action restrictions at time k. The test environment uses the test strategy π_{test} to determine $\texttt{Obs}$ in the following settings.

Instantaneous: In this setting, the test environment *instantaneously* places obstacles for the current history variable q. For any $k \geq 0$, let (s, q) be the state-history pair at time step k of the test execution. Therefore, the set of active obstacles at σ_k

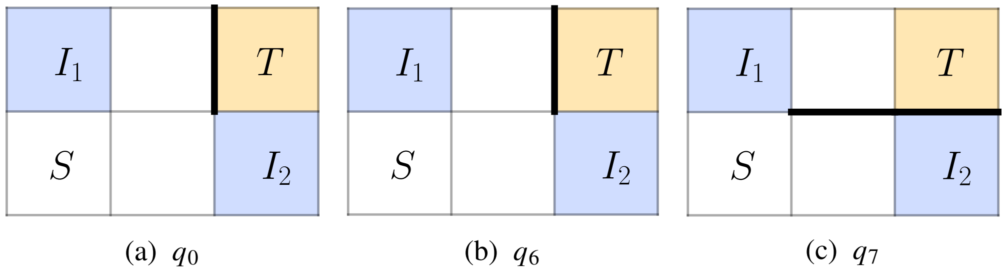


Figure 4.8: Test environment implementation of a reactive test strategy for Example 4.2.

is given as, $\texttt{Obs}(\sigma_k) = \{(s', a) \mid \forall s'' \in T.\delta(s', a) \text{ and } (s', s'') \in C(q)\}$.

Accumulative: In this setting, the test environment *accumulates* obstacles according to the system state during the test execution. For any $k \geq 0$, let $(\bar{s}, \bar{q})$ and (s, q) be the state-history pairs at time steps $k-1$ and k of the test execution, respectively. If $\bar{q} \neq q$, we set active obstacles to be $\texttt{Obs}(\sigma_k) = \{(s, a) \mid \forall a \in \pi_{\text{test}}(\sigma_k)\}$. As the test execution progresses to state-history pair (s', q) at time step $l > k$, any transition restricted by the test strategy is added to the set of active obstacles $\texttt{Obs}(\sigma_l) = \bigcup_{i=k}^{l} \texttt{Obs}(\sigma_i)$ and is restricted by the test environment. These obstacles remain in place until the test execution reaches a state history pair (s'', q') at time step $m > k$, where $q \neq q'$, at which point the test environment resets the set of active obstacles to be $\texttt{Obs}(\sigma_m) = \{(s'', a) \mid \forall a \in \pi_{\text{test}}(\sigma_m)\}$ and restrictions are accumulated until a different history variable is reached.

Lemma 4.8. Regardless of the realization of the active obstacles, as long as no new restrictions that are not in $C(q)$ are introduced, the flow value F remains the same.

Example 4.2 (Small Reactive (continued))**.** Fig. 4.8 illustrates the test environment implementing a reactive test strategy. The reactive test strategy is constructed from the optimal cuts (as depicted in Fig. 4.5) on $\mathcal{G}$ found by solving **MILP (REACTIVE)**. The test starts in history variable q_0 and the system transitions are restricted according to Fig. 4.8a. If the system decides to visit $\texttt{I}_1$ first, the test execution moves to history variable q_6 shown in Fig. 4.8b, whereas if the system decides to visit $\texttt{I}_2$ first, the test execution moves to q_7, as depicted in Fig. 4.8c. This test environment can be implemented in either the instantaneous or the accumulative setting.

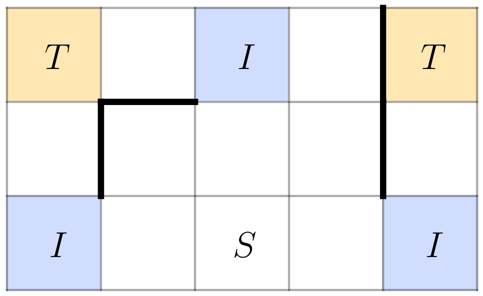


Figure 4.9: Static test environment implementation of the reactive test strategy for Example 4.1.

Static and Mixed Test Environments

The cuts found from **MILP-STATIC** result in a reactive map C in which $C(q) = C(q')$ for all $q, q' \in \mathcal{B}_\pi.Q$. That is, restrictions on system actions remain in place for the entire duration of the test, and do not change depending on the history variable q. In this fully static setting, every edge is in the static area, that is $T.E_{\text{static}} = T.E$. Therefore, the test environment realizes the test strategy by restricting all system actions corresponding to any cut in $C(q)$ for all $q \in \mathcal{B}_\pi.Q$ with static obstacles simultaneously,

$$\texttt{Obs} := \{(u.s, v.s) \in T.E_{\text{static}} \mid (u, v) \in C\}. \tag{4.19}$$

In the mixed setting of static and reactive obstacles, the test strategy resulting from **MILP-MIXED** is implemented similarly to the reactive setting, except for system transitions in $T.E_{\text{static}}$ that are blocked by static obstacles.

Example 4.1 (continued)**.** For the grid world example, Fig. 4.9 illustrates the static test on the grid world, and Fig. 4.7 shows the corresponding cuts C on the virtual product network $\mathcal{G}$. Here, the 14 cuts on $\mathcal{G}$ map to 4 static obstacles since multiple edges on $\mathcal{G}$ correspond to the same transition in T. The optimal flow value is $F^* = 3$ and there is no bypass flow. Thus, as the system navigates from source `S` to target `T`, it must visit at least one of the intermediate nodes `I`.

Remark 4.4. The instantaneous and accumulative implementations of the test environment guide when the obstacles are placed by the test environment. However, this does not have to be the same as when the system senses or observes these restrictions on its actions. We assume that the system can observe all restricted actions from its current state before it commits to an action.

The graph construction, network flow optimization, and finding the reactive test strategy are summarized in Algorithm 4.1.

Algorithm 4.1 Finding the test strategy π_{test}

```
1: procedure FindTestStrategy($T, H, \varphi_{\text{sys}}, \varphi_{\text{test}}$)
      Input: transition system $T$, 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:     $G_{\text{sys}} \leftarrow T \otimes \mathcal{B}_{\text{sys}}$                        ▷ System product
6:     $G \leftarrow T \otimes \mathcal{B}_{\pi}$                            ▷ Virtual Product Graph
7:     $\mathtt{S}, \mathtt{I}, \mathtt{T} \leftarrow \text{IdentifyNodes}(G, \mathcal{B}_{\text{sys}}, \mathcal{B}_{\text{test}})$
8:     $\mathcal{G} \leftarrow \text{DefineNetwork}\ (G, \mathtt{S}, \mathtt{T})$
9:     $\mathfrak{G} \leftarrow \mathtt{set}()$                            ▷ System Perspective Graphs
10:    for $q \in \mathcal{B}_{\pi}.Q$ do
11:        for $\mathtt{s} \in \mathtt{S}_{G_{\text{sys}}}(q)$ do
12:            $\mathcal{G}_{\text{sys}}^{(\mathtt{s},q)} \leftarrow \text{DefineNetwork}(G_{\text{sys}}, \mathtt{s}, \mathtt{T}_{\text{sys}})$
13:            $\mathfrak{G} \leftarrow \mathfrak{G} \cup \mathcal{G}_{\text{sys}}^{(\mathtt{s},q)}$
14:    $\mathbf{d}^* \leftarrow \text{MILP}(\mathcal{G}, T, \mathfrak{G}, \mathtt{I}, H)$          ▷ 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 (4.18)
17:    return $\pi_{\text{test}}$
```

Theorem 4.9. *If the problem data are not inconsistent, the reactive test strategy π_{test} found by Algorithm 4.1 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 $\mathtt{Path}(\mathtt{S}_{\text{sys}}, \mathtt{T}_{\text{sys}})$. By Theorem 4.5 and Corollary 4.6, 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 4.4, 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 Lemma 4.7, π_{test} is least-restrictive. Thus, Problem 4.1 is solved. □

This framework results in a test that is not impossible (with respect to the system objective) for a correctly implemented system. On the other hand, a poorly designed system can still fail. That is, the system is not aided in satisfying the system guarantees.

4.7 Synthesizing a Dynamic Test Agent Strategy

In addition to reactive and dynamic obstacles, the test environment can also contain dynamic test agents, which can restrict system actions by blocking a state that the system now cannot transition into. In this section, we will describe the procedure to set up the MILP for a dynamic test agent and how to map the reactive cuts C to a strategy for the test agent that satisfies the system's assumptions on its environment and preserves the least-restrictiveness of the test. The procedure consists of adapting the **MILP-MIXED** to include the information about the test agent's dynamics and synthesizing a strategy using the Temporal Logic and Planning Toolbox (TuLiP) [159] from the optimal solution of the MILP. In the case that the test agent cannot realize the cuts found by the optimization, we exclude this solution from the MILP and repeat the process until a realizable solution is found, or the MILP becomes infeasible.

Let the test agent be given as a transition system T_{tester}. We require that there exists a state $s \in T.S \cap T_{\text{tester}}.S$ and a state $s' \in T_{\text{tester}}.S$, where $s' \notin T.S$. This corresponds to the requirement that the test agent can block at least one state that is in the system $T.S$, but also has at least one state that the system cannot occupy, we will refer to this state as the *park* state. For the test environment consisting of the test agent and static obstacles, we can define the edges in $T.E$ that the test agent cannot occupy as the static area,

$$T.E_{\text{static}} := \{(u, v) \in T.E \mid v \notin T_{\text{tester}}.S\}, \tag{4.20}$$

which corresponds to the tester not being able to occupy the node corresponding to the incoming edge.

Setting up the MILP for a dynamic agent

In this section, we adapt the **MILP-MIXED** using information about the test agent. First, we augment the objective to incentivize solutions that cut edges into the same state, instead of edges into separate states, if possible. Intuitively this can be helpful to find the test agent strategy as multiple edge cuts can be realized by the agent occupying the same state. For this we introduce the variable $\mathbf{d}_{\text{state}} \in \mathbb{R}_{+}^{|V|}$ that represents whether any of the incoming edges into a node are cut or not, corresponding to a $d^v_{\text{state}} \geq 1$ or a $d^v_{\text{state}} = 0$, respectively. To capture this, we add the following constraint to the optimization:

$$\forall (u, v) \in E, \quad d^{(u,v)} \leq d^v_{\text{state}}. \tag{c10}$$

Next, we modify the objective to incentivize solutions that require fewer nodes to be blocked. With this change, we can add an additional penalty on the number nodes with incoming edge cuts, subject to the minimum number of edge cuts. This ensures that the order of priority remains minimizing the number of edge cuts, and subject to that, choosing the solution that minimizes the number of states required to be blocked to realize these cuts. This is captured in the following objective,

$$F - \frac{1}{1 + |E|} \sum_{e \in E} d^e - \frac{1}{(1 + |E|)|V|} \sum_{v \in V} d^v_{\text{state}}. \tag{4.21}$$

The choice of regularization parameters for the second and third term of the objective results in them together remaining between 0 and 1. This ensures that they do not compete with the flow value F as previously discussed for the original objective. Additionally, the second and third term no not compete with each other, maintaining the desired order of priority. This change in the objective does not remove any valid solutions from the set of possible edge cuts but only determines their ordering. This change is optional, but depending on the desired application it can result in a significant reduction of the required counterexamples.

Lemma 4.10. The objective in equation (4.21) results in a solution that maximizes the flow value F, then minimizes the number of edge cuts, and then minimizes the number of nodes with incoming edge cuts, in this order or priority.

Proof. The sum of the second and third terms together is upper bounded by 1. This maximum value is reached when all edges are cut such that $\frac{1}{1+|E|}|E| + \frac{1}{(1+|E|)|V|}|V| = 1$. The lower bound of 0 is achieved when no cut edges are cut. These bounds ensure that the second and third terms do not compete with the flow value F. Additionally, the second and third terms do not compete with each other, as the third term is upper bounded by $\frac{1}{1+|E|}$. This maximum value corresponds to all states having at least one incoming cut and it has the same penalty as cutting one additional edge in the second term. Therefore, the order of priority is maintained. □

The solution of this MILP returns the optimal set of edge cuts C^*, from which we find the set of static obstacles according to equation (4.19). Any edge cuts in C^* that are not in the static area $T.E_{\text{static}}$ need to be realized by the test agent, the reactive cut map $\mathcal{R} : \mathcal{B}_\pi \to T.E$ is then defined as

$$\begin{aligned}\mathcal{R}(q) := \{(s,s') \in T.E \mid (s,s') \notin T.E_{\text{static}} \text{ and}\\ ((s,q),(s',q')) \in C^*\}.\end{aligned} \tag{4.22}$$

In the next section, we will describe how to set up the GR(1) specification to synthesize a strategy for the test agent that maps the cuts in the reactive cut map to states that the test agent must occupy during the test execution. If such a strategy cannot be found, we use a counterexample-guided approach to exclude the solution C^*, resolve the MILP, and repeat the process.

Strategy Synthesis

To find a strategy for the test agent that realizes the cuts in $\mathcal{R}$, we set up a GR(1) specification for the test agent and synthesize a strategy using TuLiP. The system's state is captured by the variables $\mathtt{x}_{\text{sys}} \in T.S$ and $\mathtt{q}_{\text{hist}} \in \mathcal{B}_\pi.Q$, where $\mathtt{q}_{\text{hist}}$ corresponds to the history variable during the test execution. The test agent's state is captured by the variable $\mathtt{x}_{\text{tester}} \in T_{\text{tester}}.S$. First, we characterize the assumptions of the test agent on the system under test. We start by defining the initial conditions as

$$(\mathtt{x}_{\text{sys}} = s_0 \wedge \mathtt{q}_{\text{hist}} = q_0), \quad s_0 \in T.S_0,\ q_0 \in \mathcal{B}_\pi.Q_0. \tag{a1}$$

For each state $u \in G.S$, we denote the successors as $\mathtt{succ}(u)$, where $v \in \mathtt{succ}(u)$ if there exist $(u,v) \in G.E$. The system's dynamics and the evolution of the history variable are defined for each $(s,q) \in G.S$ as

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

In this framework, we choose a turn-based setting for the system and test agent. We define the variable $\mathtt{turn} \in \mathbb{B}$, where $\mathtt{turn} = 1$ corresponds to the test agent's turn, and $\mathtt{turn} = 0$ corresponds to the system's turn. We encode this assumption on the system under test in the following formula

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

where the system has to remain in place if $\mathtt{turn} = 1$. The turn-based setting is chosen for ease of implementation, a simultaneous setting would result in having to characterize the system and test agent as a Mealy and a Moore machine, respectively [27]. Next, we characterize the system objective as follows:

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

where x_{goal} is the goal state of the system, corresponding to a terminal state in $T.S$, and a reachability objective specified in φ_{sys}. The remainder of the system objective is encoded in φ_{aux}, where φ_{aux} contains the reachability and safety tasks from φ_{sys} and the reaction tasks in their equivalent GR(1) form [104]. Additionally, the system is expected to operate safely with regard to the test agent. That is, it will not take an action that will result in an immediate collision with the test agent. We denote the states that both agents can occupy—the states in which a collision can happen—as $S_\cap := T.S \cap T_{\text{tester}}.S$. Then the following assumption on the system captures this requirement:

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

The test agent's assumptions on the system model are given in formulae (a1)–(a5). The test agent on the other hand is required to satisfy the following guarantees. First, we define the initial conditions for the test agent as

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

Next, the test agent dynamics are defined from T_{tester}, where for every state $s \in T_{\text{tester}}.S$, we have

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

The test agent's dynamics are also constrained by the turn-based setting, enforced by the following guarantee

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

In our implementation, `turn` is a variable whose value is chosen by the test agent. Note that `turn` can be thought of as the environment that both agents operate in as its value is fully specified by the following formula:

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

The test agent is required to safely operate in the system's presence, i.e. not take any action that leads to an immediate collision, as assumed by the system in (**A2** in Def. 4.8), enforced by

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

Next, we will map the cuts in the reactive cut map $\mathcal{R}$ to states that the test agent must occupy during the test execution. For this, we parse the cuts in $\mathcal{R}$ as follows:

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

This formula ensures that when the test execution is in history variable q and the system is in state s, the test agent must occupy state s' when it is the system's turn (i.e. $\mathtt{turn} = 0$) if the cut $(s, q), (s', q')$ is in $\mathcal{R}(q)$. To ensure that the test agent strategy does not block any system actions that are not found by the optimization, we require the test agent strategy to be such that the test agent will occupy a state that is adjacent to the system if this does not correspond to a cut in the reactive cut map $\mathcal{R}$,

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

This requirement ensures that the test agent will not introduce any additional restrictions on system actions not in $\mathcal{R}$. Nonetheless, the test agent moves on the same states as the system and thus might transiently block all paths for the system during the test execution. This might lead to the system not making progress toward its goal while the tester occupies these states, resulting in a livelock. To solve this problem, we introduce a tie-breaking condition, which can depend on the specific example. In this analysis, we will describe a specific requirement for the test agent to only remain in a blocking state for a single time step, thus only transiently blocking the system. This requires that from each blocking state, the test agent can transition into a non-blocking state. We identify the set of blocking states and denote them as $T.S_{\text{block}}$. This tie-breaker condition is captured in

$$\bigwedge_{s \in T.S_{\text{block}}} \square\Big(\mathtt{x}_{\text{tester}} = s \rightarrow \bigcirc\neg\Big(\bigvee_{s' \in T.S_{\text{block}}} \mathtt{x}_{\text{tester}} = s'\Big)\Big). \tag{g8}$$

This condition can be modified for the specific application, where for example it might be necessary to allow the system to spend multiple concurrent time steps in a set of blocking states. From the assumptions (a1)–(a5) and the guarantees (g1)–(g8) we can now synthesize a test agent strategy π_{tester} using TuLiP.

Counterexample-Guided Search

The solution that is returned by the MILP might not result in a GR(1) specification that is realizable for the available test agent. We aim to find the optimal test strategy that can be realized by the test environment. Therefore we exclude the unrealizable solutions by adding the following constraint to the MILP:

$$\sum_{e \in E} d^e - \sum_{e \in C} d^e \geq 1, \; \forall C \in \mathsf{C}_{\text{ex}}, \tag{c15}$$

where C_{ex} is the set of unrealizable solutions. This constraint ensures that the updated solution will not allow all edges in an excluded solution to be cut at the same time and for these edges to be the only edges that are cut, effectively excluding only that solution. However, we allow these edge cuts to be part of a larger set of cut edges. The MILP for a test environment containing a dynamic test agent is thus given as

---

MILP-AGENT:

$$\begin{aligned} \max_{\substack{\mathbf{f}, \mathbf{d}, \mathbf{d}_{\text{state}}, \boldsymbol{\mu}, \\ \mathbf{f}_{\text{sys}}^{(q,\mathsf{s})} \, \forall q \in \mathcal{B}_\pi.Q, \\ \forall \mathsf{s} \in \mathsf{S}_{\mathcal{G}_{\text{sys}}}(q).}} \quad & F - \frac{1}{1+|E|} \sum_{e \in E} d^e - \frac{1}{(1+|E|)|V|} \sum_{v \in V} d^v_{\text{state}} \\ \text{s.t.} \quad & \text{(c1)-(c9), (c10), (c15).} \end{aligned} \tag{4.23}$$

To find a strategy for the dynamic test agent, we solve the **MILP-AGENT** with an empty set of excluded solutions C_{ex}. From the optimal solution $\mathbf{d}^*$ we formulate the GR(1) formula and synthesize the strategy for the test agent. If the specification is unrealizable, we add the edge cuts in the solution to C_{ex} and resolve the **MILP-AGENT**. This is repeated until either a strategy is found, or the **MILP-AGENT** becomes infeasible. In the latter case, an infeasible MILP, this framework cannot find a cut configuration such that the test agent under consideration cannot realize the test restrictions required for the test. In this case the test designers should consider a different test environment, consisting of a different test agent or static and reactive obstacles (e.g. physically using walls or gates, or virtually by adding a software harness that restricts system actions).

The procedure to synthesize a strategy for a dynamic test agent from a given system and test objective, and system model is outlined in Algorithm 4.2.

Lemma 4.11. Let $\mathbf{d}^*$ be the optimal solution to **MILP-AGENT**, and let C be the corresponding set of cuts. From $\mathbf{d}^*$, find `Obs`, the set of static obstacles, and π_{tester}, the test agent strategy, synthesized from the GR(1) formula with assumptions (a1)–(a5) and guarantees (g1)–(g8). From $\mathbf{d}^*$ also find π_{test}, the reactive test strategy

Algorithm 4.2 Finding the strategy for the test agent and static obstacles

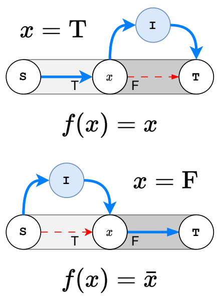
```
1:  procedure TestAgentStrategy($T, T_{\text{tester}}, H, \varphi_{\text{sys}}, \varphi_{\text{test}}$)
      Input: system $T$, test agent $T_{\text{tester}}$, test harness $H$, system objective $\varphi_{\text{sys}}$, test objective $\varphi_{\text{test}}$
      Output: test agent strategy $\pi_{\text{tester}}$
2:      $T.E_{\text{static}} \leftarrow$ Define from $T$, $T_{\text{tester}}$          ▷ Static area (Eq. (4.20)
3:      $\mathcal{G}, \mathfrak{G}, \mathtt{I}, G \leftarrow$ Setup arguments          ▷ Lines 2-13 in Alg. 4.1
4:      $\mathsf{C}_{\text{ex}} \leftarrow \emptyset$          ▷ Initialize empty set of excluded solutions
5:      while True do
6:          $\mathbf{d}^* \leftarrow$Solve MILP-agent($\mathcal{G}, \mathfrak{G}, \mathtt{I}, T, H, \mathsf{C}_{\text{ex}}$)
7:          if Status(MILP) = infeasible then
8:              return infeasible
9:          $C \leftarrow \{(u,v) \in G.E \mid \mathbf{d}^{*(u,v)} = 1\}$          ▷ Cuts on $G$
10:         Obs $\leftarrow$ Define from $C$          ▷ Static Obstacles (Eq. (4.19))
11:         $\mathcal{R} \leftarrow$Define from $C$          ▷ Reactive map (Eq. (4.22))
12:         $\mathbf{A} \leftarrow$ Assumptions (a1)–(a5) from $T$, $T_{\text{tester}}$, $G$, $\varphi_{\text{sys}}$
13:         $\mathbf{G} \leftarrow$ Guarantees (g1)–(g7) from $T$, $T_{\text{tester}}$, $\mathcal{R}$
14:         $\varphi \leftarrow (\mathbf{A} \rightarrow \mathbf{G})$          ▷ Construct GR(1) formula
15:         if Realizable($\varphi$) then
16:             $\pi_{\text{tester}} \leftarrow$ Synthesize for $\varphi$ with TuLiP
17:             return $\pi_{\text{tester}}$, Obs
18:         $\mathsf{C}_{\text{ex}} \leftarrow \mathsf{C}_{\text{ex}} \cup C$
```

corresponding to the optimal cuts C according to Algorithm 4.2. Then π_{tester} and `Obs` realize π_{test}.

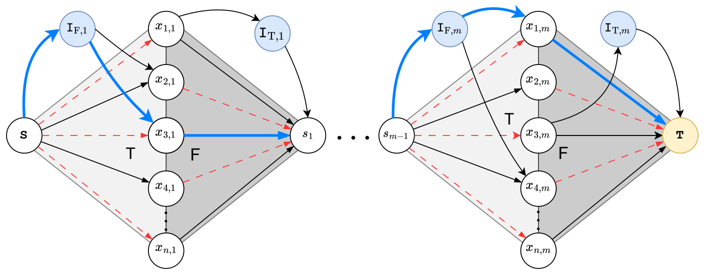
Proof. From $\mathbf{d}^*$ we can find the reactive cut map $C(q)$, which is realized by the reactive test strategy π_{test}. We know that $C(q) = \mathcal{R}(q) \cup \mathtt{Obs}$ by equations (4.17), (4.19), (4.22) for all $q \in \mathcal{B}_{\pi}.Q$. The test agent is required to realize the cuts in $\mathcal{R}(q)$ by guarantee (g6), but it is prohibited from restricting any system transitions not in $\mathcal{R}(q)$ by guarantee (g7). Thus the test agent realizes $\mathcal{R}(q)$ and the test agent strategy π_{tester} together with the set of static obstacles `Obs` realize the reactive test strategy π_{test}. □

Theorem 4.12. *If Algorithm 4.2 returns a feasible test agent strategy π_{tester} and a set of static obstacles* `Obs`*, then this solves Problem 4.2.*

Proof. The test agent strategy π_{tester} and the set of static obstacles `Obs` satisfy the system's assumptions. The dynamics of the tester are specified in (g1)-(g4) and satisfy**A1**. The safety assumption is guaranteed by (g5). By Lemma 4.11, the guarantees (g6) and (g7) realize the optimal cuts from **MILP-agent**. These



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



(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 3-SAT formula, where a truth assignment for the variables can be found using the network flow approach for static obstacles.

cuts are found such that there always exists a path on G_{sys} by constraint (c8). By adding guarantee (g8), the test agent strategy is guaranteed to satisfy the system's assumptions **A3** and **A4**.

By Lemma 4.11, π_{tester} is a realization of a least-restrictive feasible π_{test}. □

4.8 Complexity

The framework comprises three parts: i) graph construction, ii) optimization, and iii) reactive synthesis. For graph construction, we first need to construct Büchi automata from specifications. This has doubly-exponential complexity, $2^{2^{|\phi|}}$, in the length of the formula ϕ in the worst-case [15]. Then, the construction of products is a Cartesian product of two graphs T and $\mathcal{B}_\pi$, and has a time complexity of $O(|T.S|^2 \cdot |\mathcal{B}_\pi.Q|^2)$. GR(1) synthesis is known to have a complexity of $O(|N|)^3$, where N is the number of states required to define the formula. In this section, we will establish the computational complexity of the optimization.

To prove the computational complexity of finding the cuts on the graph, we first prove the computational complexity in the special case of static obstacles. As defined in Sections 4.4 and 4.5, the problem data is a graph $G = (V, E)$ with node groups $\mathtt{S}$, $\mathtt{I}$, $\mathtt{T}$, and the corresponding flow network $\mathcal{G}$. For some edge $e \in E \setminus E(\mathtt{I})$, the binary variable d^e indicates whether the edge is cut: $d^e = 1$. The set C represents the set of edges with $d_e = 1$. For static obstacles, the edges are grouped by the corresponding transition in T. The grouping $\mathtt{Gr}_{\text{static}} : T.E \to G.E$, and defined as

$$\mathtt{Gr}_{\text{static}}((s, s')) := \{(u, v) \in G.E \mid u.s = s, v.s = s'\}. \tag{4.24}$$

For some $(s, s') \in T.E$, all edges $e \in \mathtt{Gr}_{\text{static}}((s, s'))$ have the same d^e value, i.e., if

$d_e = 1$ for some edge e in the group, then all edges in this group will have d_e set to 1. A bypass path on G is some $Path(\mathtt{S}, \mathtt{T})$ which does not visit the intermediate $\mathtt{I}$. The flow value F on $\mathcal{G}$ is defined from the source $\mathtt{S}$ to target $\mathtt{T}$, with each edge having unit capacity. A valid set of edge cuts C is such that i) there does not exist a bypass path, ii) there exists a path from $\mathtt{S}$ to $\mathtt{T}$, and iii) edges respect the grouping $\mathtt{Gr}_{\text{static}}$.

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

This corresponds to finding the valid set of edge cuts C that as first priority, maximizes the flow F, and subsequently chooses the set of edge cuts C with the smallest cardinality $|C|$ (i.e. breaking ties between all valid edge cuts that realize F). For static obstacles, Problem 4.3 corresponds to the following decision problem.

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

Lemma 4.13. Any solution to Problem 4.3 can be used to construct a solution for Problem 4.4 in polynomial time.

Lemma 4.13 implies that if there exists a polynomial-time algorithm to compute a solution to Problem 4.3, then there also exists a polynomial-time algorithm to solve Problem 4.4. Thus, if we can show that Problem 4.4 belongs to the class of NP-hard problems (i.e., there exists a polynomial-time reduction from a problem in NP to Problem 4.4), that would imply that there exists a polynomial-time algorithm to solve Problem 4.4 only if $P = NP$. This in turn would support the MILP approach we provide to solve Problem 4.3. To show that Problem 4.4 is NP-hard, we construct a polynomial-time reduction from 3-SAT to Problem 4.4. This polynomial-time reduction maps any instance of 3-SAT to an instance of Problem 4.4 such that the solution of the constructed instance of Problem 4.4 corresponds to a solution of the instance of the 3-SAT problem.

Definition 4.16 (3-SAT [38])**.** Let $f(x_1, \ldots, x_n) := \bigwedge_{j=1}^{m} c_j$ be a propositional logic formula over Boolean propositions $x_1, \ldots, x_n$ in conjunctive normal form (CNF) in which each clause c_j is a disjunction of three Boolean propositions or their negations. The 3-SAT problem returns *True* if there exists a satisfying Boolean assignment to $f(x_1, \ldots, x_n)$ and *False* otherwise.

Construction 4.1 (Clause to Sub-graph)**.** Given a 3-SAT clause c_j, we can construct a sub-graph representing this clause as follows. For each clause c_j, we introduce nodes s_{j-1} and s_j. Then, we add the nodes $x_{1,j}, \ldots x_{n,j}$ corresponding to variables $x_1, \ldots, x_n$ in the 3-SAT formula. We add the following directed edges for each $x_{i,j}$ node—an incoming edge from node s_{j-1} to $x_{i,j}$, and an outgoing edge from $x_{i,j}$ node to node s_j. Then we add two nodes, denoted by $\mathtt{I}_{\mathrm{T},j}$ and $\mathtt{I}_{\mathrm{F},j}$, to this sub-graph. If x_i appears in the clause c_j, then we connect the $\mathtt{I}_{\mathrm{T},j}$ node by bypassing the edge from $x_{i,j}$ to x_j, and if $\bar{x}_i$ appears in c_j, then we connect $\mathtt{I}_{\mathrm{F},j}$ to bypass the edge from s_{j-1} to $x_{i,j}$ (as shown in Fig. 4.10a).

Constructing a sub-graph for a clause c_j via Construction 4.1 allows us to relate the edge cuts to the Boolean assignment for the variables $x_0, \ldots, x_n$. If the incoming edge into $x_{i,j}$ is cut, then the corresponding Boolean assignment to x_i is *False*, and if the outgoing edge from $x_{i,j}$ is cut, then the corresponding Boolean assignment to x_i is *True*. This ensures that a satisfying assignment for the clause corresponds to edge cuts such that all $Paths(s_{j-1}, s_j)$ are routed through intermediate nodes $\{\mathtt{I}_{\mathrm{T},j}, \mathtt{I}_{\mathrm{F},j}\}$. An assignment that evaluates the clause c_j to *False* corresponds to edge cuts in the sub-graph such that there is no path from s_{j-1} to s_j.

Construction 4.2 (Reduction of 3-SAT to Problem 4.4)**.** Suppose we have an instance of the 3-SAT problem with n variables $x_1, \ldots, x_n$ and m clauses $c_1, \ldots c_m$. First, we construct the sub-graphs for each clause according to Construction 4.1. Let $M := m \times n$. We denote the node s_0 as the source S, and s_m as the sink $\mathtt{T}$. The resulting graph is a series of sub-graphs representing each clause c_j of the 3-SAT formula. For every variable x_i in the formula, we maintain two groups of edges: i) incoming edges $\{(s_{j-1}, x_{i,j}) \mid 1 \leq j \leq m\}$, and ii) outgoing edges $\{(x_{i,j}, s_j) \mid 1 \leq j \leq m\}$. All edges in a group share the same cut value, corresponding to $\mathtt{Gr}_{\text{static}}$. This ensures that every variable has the same Boolean assignment across clauses.

This allows us to construct a graph corresponding to a 3-SAT formula in polynomial time via the procedure outlined in Construction 4.2, also illustrated in Fig. 4.10.

Theorem 4.14. *Problem 4.4 is NP-complete.*

Proof. We will show that Problem 4.4 is NP-hard by showing that Construction 4.2 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 4.2 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 $x_1, \ldots, x_n$. 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 if and only if the corresponding assignment to variables $x_1, \ldots, x_n$ is a satisfying witness for the 3-SAT formula.

First, consider a satisfying witness for Problem 4.4. By Construction 4.2, 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 4.1, a strictly positive flow on the sub-graph of clause c_j implies that c_j is satisfied. By Construction 4.2, 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 4.1, a satisfying witness corresponds to a Path(s_{j-1}, s_j) on the sub-graph for each clause c_j. By Construction 4.2, 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 4.15. Problem 4.3 is NP-hard [117].

Proof. By Theorem 4.14, Problem 4.4 is NP-complete, and therefore by Lemma 4.13, Problem 4.3 is NP-hard. □

Finally, we can make use of the insight gained in this chapter to show the overall complexity of this problem. Additionally, we can identify the computational complexity for the reactive setting. For the reactive setting, the optimization problem (in the sense of computational complexity) can be stated similarly to the static setting in Problem 4.3. For the reactive setting, a valid set of edge cuts is similar to the static setting, except in how edges are grouped, which is discussed in Remark 4.2. The optimization problem and its corresponding decision problem can be stated as follows.

Problem 4.5 (Reactive Obstacles Optimization Problem)**.** Given a graph G, identify a set of cuts C such that the flow F is maximized under the following conditions: i) there does not exist a bypass path, ii) C is the minimum set of edge-cuts for the flow value F, and iii) edges of the same history variable q are grouped in $\mathtt{Gr}(q)$ (Eq. (4.12)), and are then statically mapped to a copy of the product graph G. For each source $\mathtt{s} \in \mathtt{S}_G(q)$ we require a flow of at least one on the corresponding copy $\mathcal{G}q, \mathtt{s})$.

Problem 4.6 (Reactive Obstacles Decision Problem)**.** Given a graph G, identify a valid set of edge cuts C such that the resulting flow F is maximized over the set of all possible edge cuts, and such that $|C|$ is minimized for the flow F.

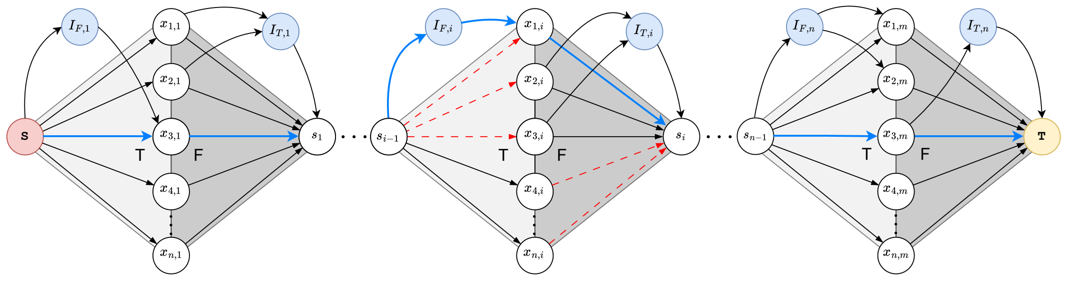
Note that a *valid* set of edge cuts for the reactive problem is different from a valid set of edge cuts for the static problem.

Once again, we prove a reduction from 3-SAT, but to an instance of Problem 4.6 with a single history variable q. Given a 3-SAT formula, the construction of the graph follows from the static setting, but with a few key differences.

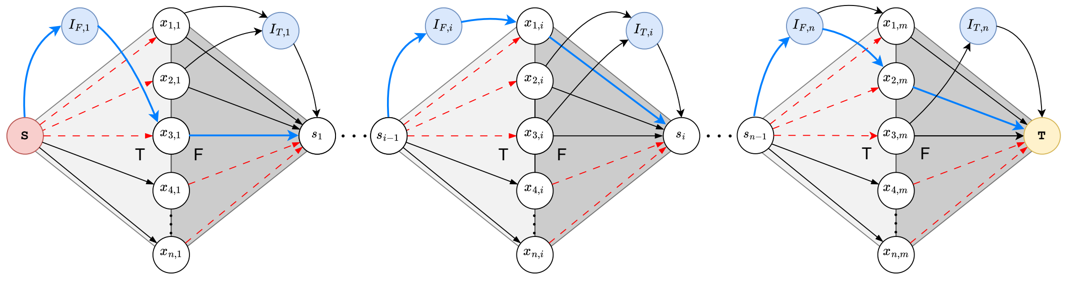
Construction 4.3 (Reduction from 3-SAT to Problem 4.6 with single history variable q)**.** Suppose we have an instance of the 3-SAT problem with n variables $x_1, \dots, x_n$ and m clauses $c_1, \dots c_m$. Let $M := n$. Using Construction 4.2, setup two graphs: G and G_q. The key difference is that G_q follows Construction 4.2 exactly, while in G, edges in a group need not have the same cut value. Furthermore, for each group in G_q, the cut value is set to the maximum edge-cut value in the corresponding group in G.

Theorem 4.16. *Problem 4.5 is NP-complete and Problem 4.6 is NP-hard.*

Proof. The proof follows similarly from Theorem 4.14. In this setting, a witness for Problem 4.6 comprises the maximum edge cut value of each group in G. Construction 4.3 relates edge cuts on G and $G^{(q,\mathtt{S})}$. This implies that edge cuts on G are



(a) Graph G.



(b) Copy of G for history variable q, denoted G_q.

Figure 4.11: Graphs constructed as reduction of 3-SAT to Problem 4.6. Note that edges in G are not grouped, but edges in G_q are grouped and cut according to the maximum cut value in G.

found under the condition that there is a strictly positive flow on G_q under a static mapping of edges in $G^{(q,\mathrm{S})}$. 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 $G^{(q,\mathrm{S})}$ results in a Boolean assignment 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 4.14.

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 $G^{(q,\mathrm{S})}$. 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

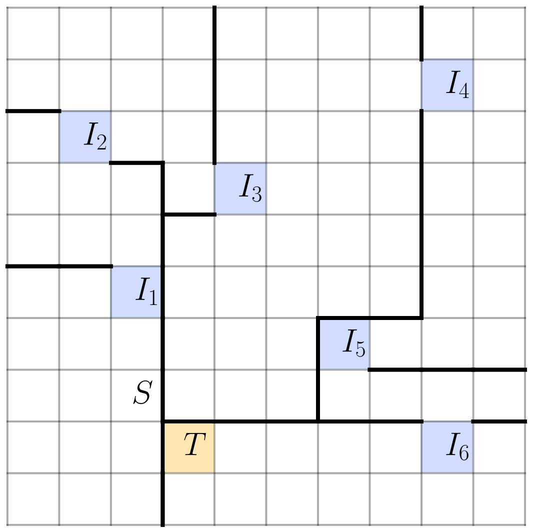


Figure 4.12: Static test environment for the simulated sequencing Experiment 4.1.

a result, Problem 4.6 is NP-complete and following similarly to Corollary (4.15), Problem 4.5 is NP-hard. □

4.9 Experimental Results

The proposed framework was applied to several examples for different test environments and different specification tasks in simulation and actual hardware implementations. In the hardware experiments, we are testing a quadrupedal robot, the Unitree A1. The low level is controlled using a motion primitive layer with behaviors for lying down, standing, walking, and jumping. Individual motion primitives are implemented within a C++ motion primitive framework, and control laws, sensing, and estimation are executed at 1kHz on an Intel NUC with an i7-10710U CPU and 16GB of RAM. Communication to the A1's actuators and sensors is done via UDP. Detailed information on the low-level controller can be found in [150]. The high-level controller was synthesized using TuLiP from a high-level abstraction of the quadruped's transition system consisting of states corresponding to the grid world locations and the available motion primitives. In the simulated examples, the system under test is either a TuLiP-generated grid world controller that can move between grid cells or the simulated Unitree A1 quadruped. Detailed information on the sizes of the graphs and run times for the experiments can be found in Appendix A.1.

Static Test Environments

In this section, we will illustrate different experiments for static test environments. The test strategy was found using Algorithm 4.1 and **MILP-STATIC**.

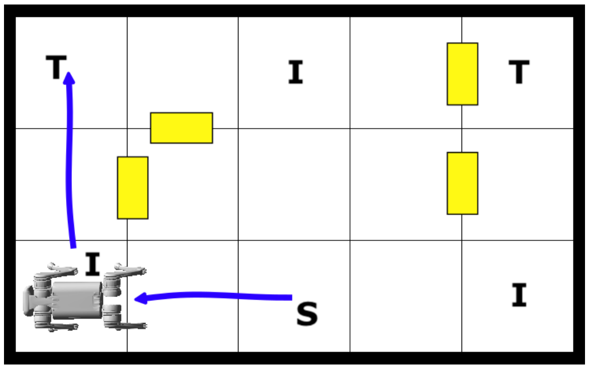


Figure 4.13: Trace for Experiment 4.2.

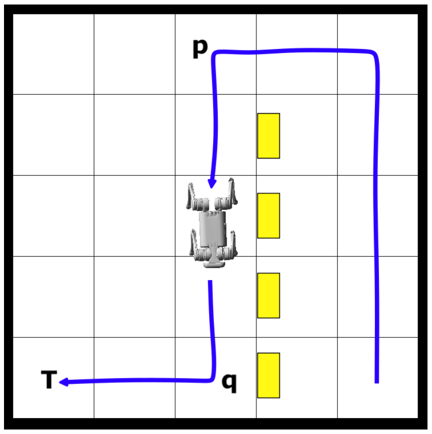


Figure 4.14: Trace for Experiment 4.3.

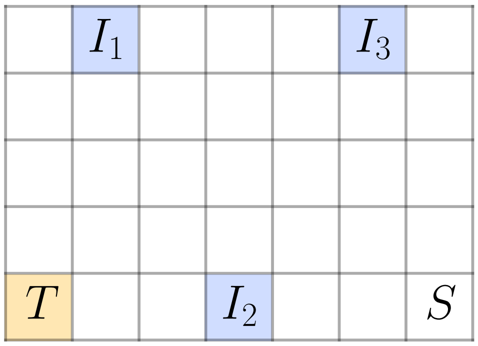
Experiment 4.1 (Sequencing)**.** In this experiment, the system is tested for a sequenced visit task on a standard grid world, where it is required to visit six intermediate locations in a given sequence. The test environment consists of static obstacles. The system objective is given as $\varphi_{\text{sys}} = \Diamond T$ and the tester objective is the sequenced visit task

$$\varphi_{\text{test}} = \Diamond(I_1 \wedge \Diamond(I_2 \wedge \Diamond(I_3 \wedge \Diamond(I_4 \wedge \Diamond(I_5 \wedge \Diamond I_6))))).$$

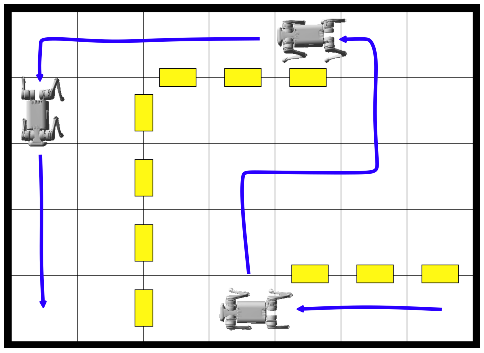
The resulting test environment is shown in Fig. 4.12. We observe that any system that navigates on this grid world would have to visit the intermediate locations in the given order, before arriving at the goal state T. For this experiment, the MILP finds a feasible solution with a flow of 1 and then terminates after the objective does not improve for 5 minutes (explained in Appendix A.1). Thus, this solution might be sub-optimal, making it feasible but not necessarily least-restrictive.

Hardware Experiments

Experiment 4.2 (Example 4.1)**.** This experiment is the hardware implementation of Example 4.1. Figure 4.13 shows the trace of the hardware test execution according to the static obstacle map shown in Figure 4.9. The yellow boxes in Figure 4.13 correspond to the static obstacles found by the flow-based synthesis framework. We observe the system navigate through one of the cells labeled I and then successfully reach the goal state T. For this example, the maximum flow on the virtual game graph is 3 and in this case, this translates to three unique ways for the system to reach one of the goal positions.



(a) Grid world layout.



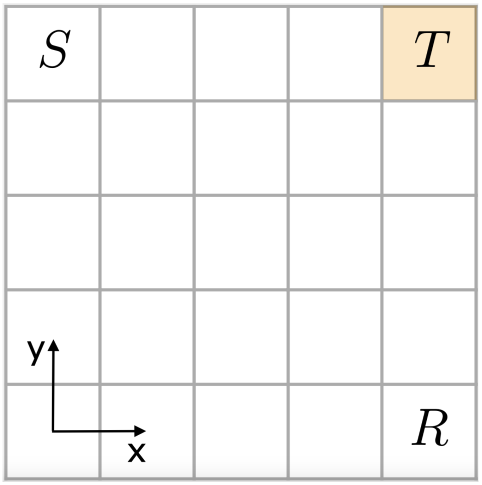
(b) Hardware trace.

Figure 4.15: Grid world layout and test execution trace for Experiment 4.4.

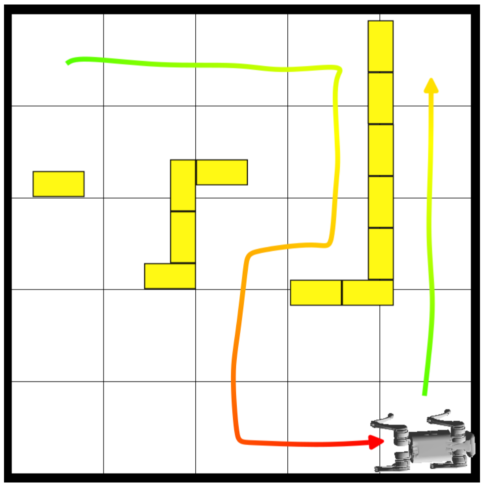
Experiment 4.3 (Triggered Reaction)**.** In this experiment, we show how to implement a triggered reaction for the grid world shown in Figure 4.14. The system objective is given as follows $\varphi_{\text{sys}} = \Diamond T \wedge \Box(p \rightarrow \Diamond q)$, and the test objective consists of the trigger, $\varphi_{\text{test}} = \Diamond p$, where the propositions T, p, and q correspond to the labeled cells on the grid. The flow-based synthesis framework finds static obstacles that are shown as the yellow boxes in Figure 4.14, which are placed such that on the way to T, the system is routed through p. After passing through p a correct system is required to also visit q (as shown in the trace in Figure 4.14). However, the test environment does not force the system to visit q as reaching T without reaching q would result in a failed test.

Experiment 4.4 (Multiple Visits)**.** In this experiment, we find the static test environment for a test objective that contains multiple visit tasks in an arbitrary order, given as $\varphi_{\text{sys}} = \Diamond I_1 \wedge \Diamond I_2 \wedge \Diamond I_3$, where I_1, I_2, and I_3 refers to the locations labeled on the grid shown in Figure 4.15a. The system objective is given as $\varphi_{\text{sys}} = \Diamond T$, which corresponds to reaching the goal location in the bottom left corner of the grid. The flow-based synthesis frameworks finds 148 cuts on G that correspond to 10 static obstacles placed on the grid as shown by the yellow boxes in Figure 4.15b. Due to this obstacle configuration the quadruped is forced to visit I_1, I_2, and I_3 on its way from S to T.

Experiment 4.5 (Refueling)**.** In this example the quadruped state is $\mathbf{x} = (x, y, f)$, where x and y correspond to the coordinates, and f represents the current fuel level. The grid world layout is shown in Figure 4.16a. The quadruped is tasked with reaching its goal location in the top right corner of the grid, labeled T, and not running out of fuel. The fuel capacity is set at 10 units, and each transition between



(a) Grid world layout.



(b) Experiment trace.

Figure 4.16: Grid world layout and experiment trace for Experiment 4.5.

two adjacent grid cells results in the fuel level decreasing by one unit. The refueling location is located in the bottom right corner of the grid, and visiting this cell resets the quadruped's fuel level to its maximum capacity. The system objective is given as

$$\varphi_{\text{sys}} = \Diamond(x = 5 \wedge y = 5) \wedge \Box\neg(f = 0).$$

The desired test behavior is observing that the quadruped's fuel level drops such that it needs to decide to visit the refueling station to successfully make it to its goal. In particular, in this example, we want to see the quadruped be in the lower three rows of the grid with a fuel level lower than 2. The test specification is thus given as

$$\varphi_{\text{test}} = \Diamond(y < 4 \wedge f < 2).$$

In this example, the intermediate locations do not correspond directly to states on the grid, but the transition system contains the information about the fuel level, allowing us to use the flow-based synthesis approach to find the static test environment. Another important aspect of this example is that some of the intermediate states correspond to a fuel level of 0, making them unsafe states for the system. The framework still allows synthesis for this test objective as some states that satisfy φ_{test} are not in conflict with φ_{sys} and these are the states that the test execution is routed through. Figure 4.16b shows the resulting static test environment and the trace of the quadruped. The trace is colored according to the current fuel level of the quadruped. We can see that the quadruped is required to visit the refuel location (pictured in the bottom right of the grid) to not run out of fuel. In addition, observe

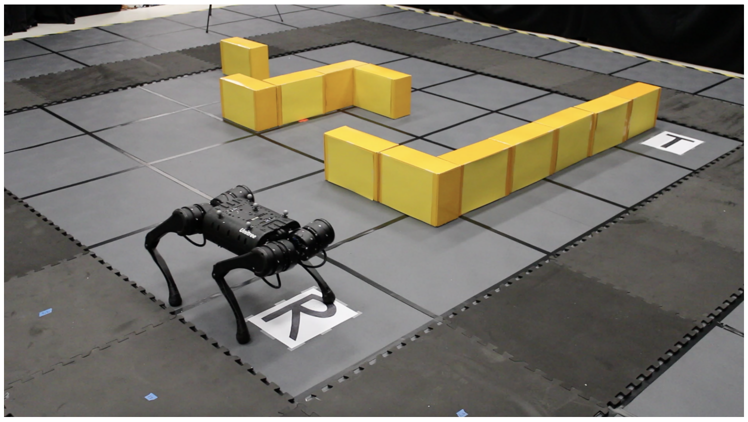


Figure 4.17: Snapshot of the hardware test execution for the refueling Experiment 4.5.

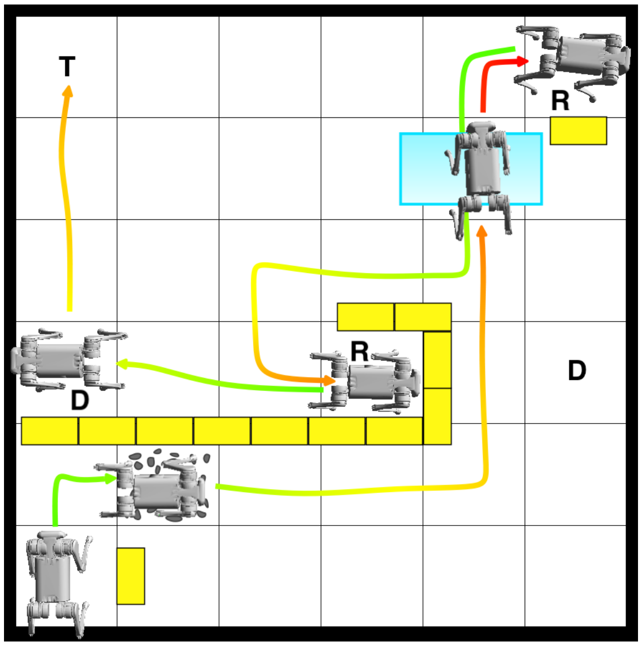


Figure 4.18: Trace for the Mars exploration Experiment 4.6.

that the quadruped could have decided to bypass the refuel station, but this would have resulted in an empty fuel tank and, in turn, it would have failed the test. A snapshot of the hardware test execution is shown in Figure 4.17.

Experiment 4.6 (Mars Exploration)**.** This hardware example is inspired by a planetary exploration mission. The quadruped we are testing represents a rover that has to navigate on the Martian surface and in the case that it finds an interesting sample, it has to take it to a specified drop-off location. Additionally, it has to keep track of its fuel level which should never run out. Its state is thus again given as $\mathbf{x} = (x, y, f)$, and the fuel level is reset to maximum capacity if it visits a refueling location. The 'drop-off' locations are annotated with a 'D' and the refueling locations with a 'R' on the grid in Figure 4.18. The quadruped wants to reach its goal location, $\mathbf{x}_{\text{goal}} = (1, 6)$, marked by a 'T'. The locations of interesting samples are labeled as 'rock' and 'ice' and can also be seen in Figure 4.18. The system objective is given as

$$\varphi_{\text{sys}} = \diamond T \wedge \square(f > 0) \wedge \square(\text{ice} \vee \text{rock} \rightarrow \diamond \text{drop-off}),$$

where ice, rock, and drop-off are the labels of the corresponding grid cells. The test objective corresponds to forcing the quadruped to make the choice to refuel and collect an ice and a rock sample. This is encoded as follows:

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

where $d = |\mathbf{x}_{\text{goal}} - (x, y)|$ is the distance to the goal position. The static test environment is shown in Figure 4.18, where the yellow boxes correspond to the static obstacles that were found using the flow-based synthesis framework. The quadruped trace is shaded in a color gradient according to the current fuel level. First, the quadruped is routed through the rocks, bypasses the first drop-off location, and then visits the ice location on its way to the refueling station. It is important to note that for a successful test execution, it could not have bypassed the ice location as it did not have enough fuel to do so. It is then required to refuel again before it can drop off the samples. We observe that the quadruped is forced to refuel twice to successfully complete its mission. Note that in this experiment, the implemented solution has a flow of $F = 1$. This is a feasible solution to the MILP, but it is sub-optimal. This illustrates the anytime characteristic of our approach, where any solution to the MILP with a flow greater or equal to 1 is a valid test environment. Nonetheless, a least-restrictive test environment corresponds to the optimal solution. In Table A.2 the run time to find the optimal solution is shown.

Reactive Test Environments

In this section, we will illustrate the framework for reactive test environments. The reactive test strategy was generated using Algorithm 4.1 and **MILP-REACTIVE**. The following two reactive examples are inspired by search and rescue missions, where the reactive obstacles correspond to locked or unlocked doors. These examples were simulated and also executed in hardware.

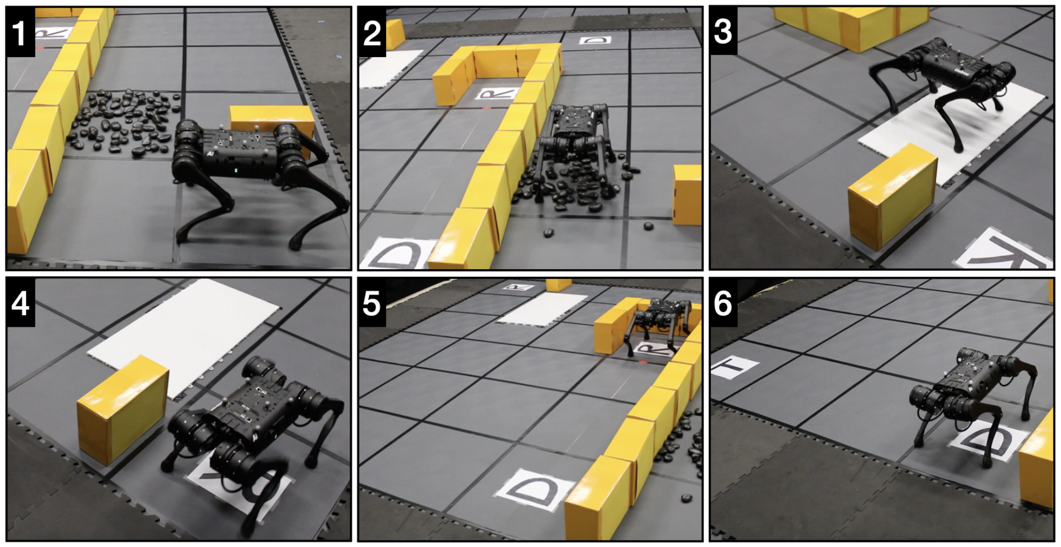


Figure 4.19: Snapshots from the hardware test execution for the Mars exploration Experiment 4.6.

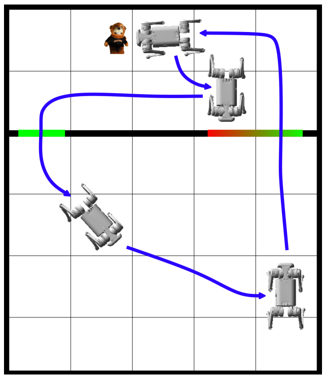
Experiment 4.7 (Beaver Rescue)**.** In this example, the system under test is the quadruped, and its task is to reach the beaver located in the hallway and bring it back into the lab. The quadruped position is denoted as $\mathbf{x}$, its goal location is denoted as 'goal' and the beaver location is denoted as 'beaver'. Then, the system objective is given as the sequence

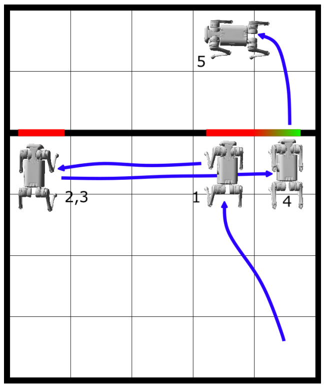
$$\varphi_{\text{sys}} = \Diamond(\text{beaver} \wedge \Diamond\, \text{goal}),$$

which corresponds to the quadruped reaching the beaver's position and thereafter making its way back into the lab. The test objective is to observe that the system uses both lab doors at least once, ideally one on the way to the beaver, and the other one on the way back. Which door the quadruped decides to use first is arbitrary and the test objective is the following the multiple visit specification:

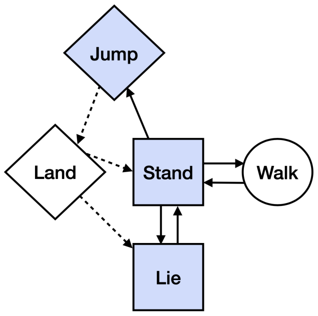
$$\varphi_{\text{test}} = \Diamond\, \text{door}_1 \wedge \Diamond\, \text{door}_2,$$

where door_1 and door_2 correspond to the positions on either side of the corresponding door. The reactive test strategy found by the flow-based synthesis framework corresponds to locking and unlocking the doors of the lab according to the state of the test execution. In the simulated test execution shown in Figure 4.20a, we can see that the quadruped uses door_1 on its way to pick up the beaver, then tries to pass through the same door again, but the reactive test strategy is such that this door is now locked. Then the quadruped is forced to door_2 to successfully bring the beaver back to its goal location in the lab. Snapshots and the trace from the hardware execution


(a) Simulation trace for Experiment 4.7.



(b) Simulation trace for Experiment 4.8.



(c) Motion primitive graph.

Figure 4.20: Unitree A1 motion primitive graph and simulation traces for reactive experiments.

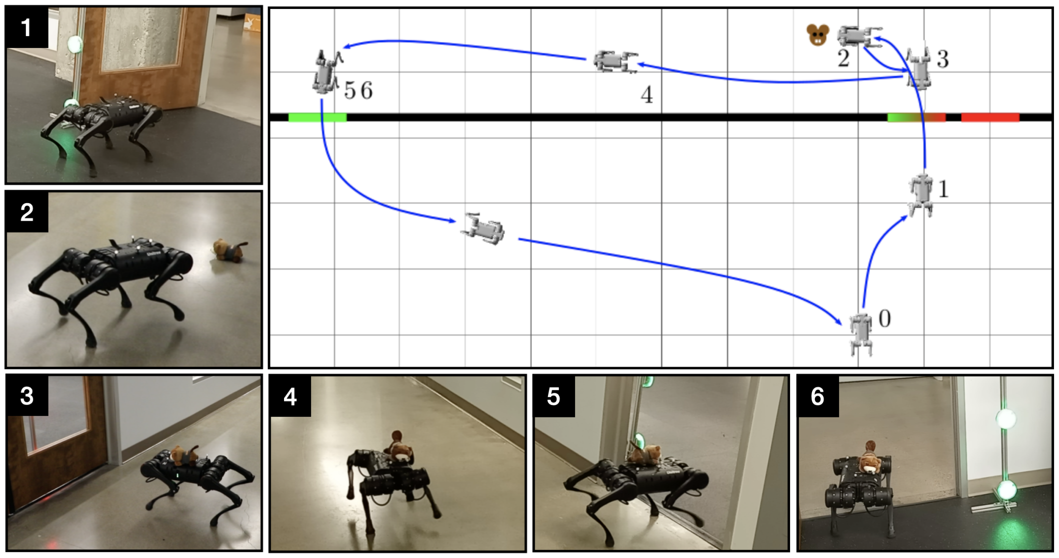


Figure 4.21: Beaver rescue hardware demonstration.

of this experiment are shown in Figure 4.21. Note that this hardware experiment was executed using a bi-level flow optimization, but the flow-based synthesis framework using the MILP resulted in the same reactive test strategy where the time required to solve the optimization was reduced by three orders of magnitude.

Experiment 4.8 (Reactive Motion Primitive Test)**.** In this experiment, the quadruped is required to reach its goal location in the hallway, denoted as ‘goal’. The system objective is $\varphi_{\text{sys}} = \Diamond\,\text{goal}$. The quadruped’s motion primitive graph is shown in Figure 4.20c, and the test objective is to observe all motion primitives. We will disregard ‘walk’ and ‘land’ as these motion primitives are automatically required to

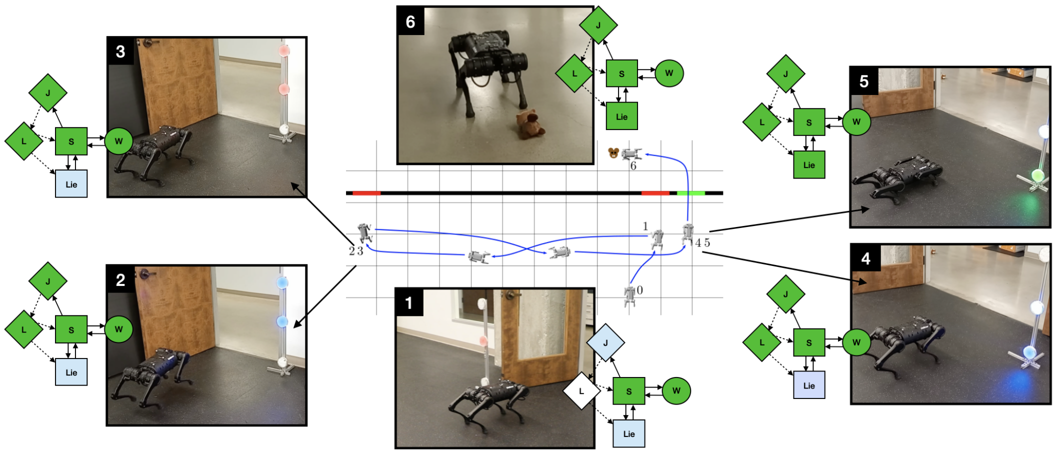


Figure 4.22: Snapshots of the hardware test execution on the Unitree A1 quadruped for Experiment 4.8

transition from one grid cell to another and to execute the ‘jump’ motion primitive, respectively. Thus, the test objective is given as

$$\varphi_{\text{test}} = \Diamond\, \text{jump} \wedge \Diamond\, \text{lie} \wedge \Diamond\, \text{stand}, \tag{4.25}$$

where ‘jump’, ‘lie’, and ‘stand’, correspond to the state of the respective motion primitive in the transition system. Initially, the quadruped is located in the lab, and it has to pass through one of three doors to exit the lab. Each of the three doors can be unlocked by executing specific motion primitives. Locking and unlocking the doors corresponds to the reactive test strategy. In the simulation shown in Figure 4.20b, we observe that the quadruped unsuccessfully tries to unlock the center door first by executing the ‘stand’ motion primitive (annotated by a 1). Then it executes the ‘jump’ and ‘stand’ motion primitives (2 and 3, respectively) at the left door, but the reactive test strategy locks this door as well. Finally, it executes the ‘lie’ motion primitive at the right door (shown by 4) and subsequently this door is unlocked by the reactive test strategy allowing the quadruped to enter the hallway and successfully complete the test (5). Figure 4.22 shows the trace and snapshots of the hardware execution for this experiment. Note that the reactive test strategy for the hardware experiment was found using the bi-level optimization, but the MILP resulted in the same reactive test strategy with a run time reduction by three orders of magnitude.

Reactive Test Environments with Dynamic Test Agents

In the following examples, the test environment consists of a dynamic test agent and if required static obstacles. The strategy for the dynamic test agent and the set of

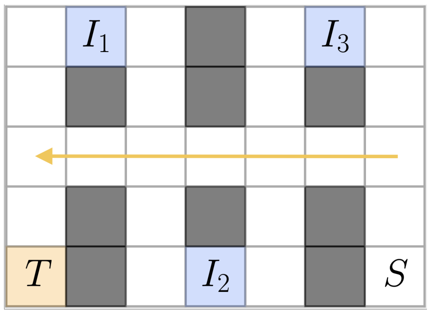
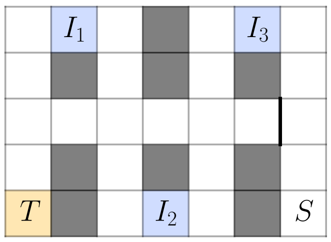


Figure 4.23: Grid world layout for Experiment 4.9.

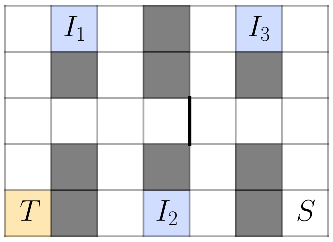
static obstacles are found using the flow-based synthesis procedure as outlined in Algorithm 4.2 and **MILP-agent**.

Simulated Experiments

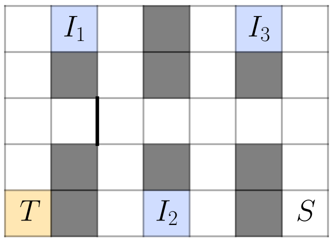
Experiment 4.9 (Grid World 1)**.** In this experiment, the system under test is the quadruped. The grid world layout is shown in Figure 4.23. The quadruped starts in the bottom right corner of the grid world, denoted S, and its task is reaching the goal location T in the bottom left corner. The environment already contains obstacles, marked as the dark gray cells in Figure 4.23. The test agent is another quadruped, which can transition from right to left along the middle row of the grid. It is also able to leave the grid on either side, these states are the designated *park* states that the system quadruped cannot enter. The system quadruped is shown as the gray quadruped in Figure 4.25 and the test agent is shown as the yellow quadruped. The system objective is given as $\varphi_{\text{sys}} = \Diamond T$. The test objective consists of multiple visit tasks in arbitrary order, given as $\varphi_{\text{test}} = \Diamond I_1 \wedge \Diamond I_2 \wedge \Diamond I_3$, where I_1, I_2, and I_3 correspond to the designated locations on the grid. In this example, the static area is designated as any transition into a cell that is not in the center row that the reactive agent can occupy. Note that for this example, the test agent could block the system transiently from making any progress towards the goal by occupying the grid cells in the center column that are outside of the vertical obstacle configuration. To prevent any livelocks, the tester quadruped is allowed to only occupy these states for a single time step, corresponding to guarantee (g8). We find the graphs and solve the **MILP-agent** to find the reactive cuts and the test agent strategy using the counterexample-guided search. The resulting cuts are shown in Figure 4.24 grouped by their history variables. Note that for simplicity we show the cut as a black line, but the transition is only restricted from right to left. The resulting test execution is shown in Figure 4.25. We observe that the test agent quadruped is



(a) Cuts found in q_0.



(b) Cuts found in q_{15}.



(c) Cuts found in q_{12}.

Figure 4.24: Reactive cuts found from **MILP-AGENT** for Experiment 4.9.

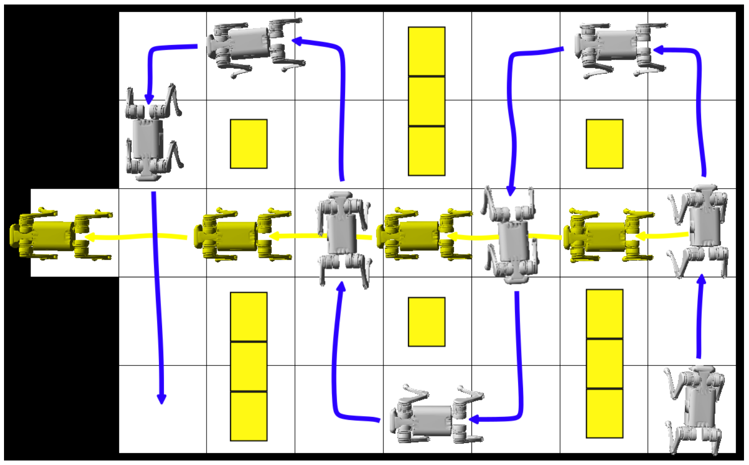


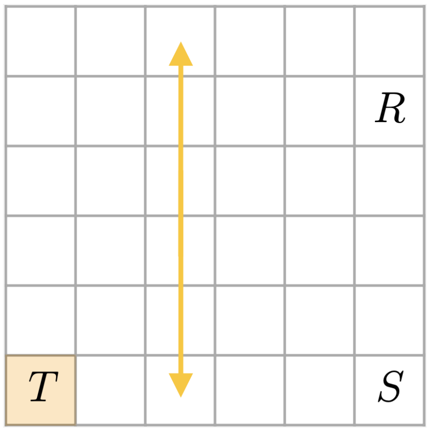
Figure 4.25: Simulated test execution for Experiment 4.9.

blocking the system quadruped from directly navigating to the goal state through the center row of the grid by realizing the cuts shown in Figure 4.24. Instead, the system quadruped is forced to navigate through the intermediate states by alternating between the upper and lower edges of the grid. We see that the test agent waits for the system to enter the corresponding intermediate state, and then moves up into the state corresponding to the restricted transition for the next history variable. After the test objective is satisfied, it leaves the grid by entering into a *park* state to allow the system to navigate to its goal.

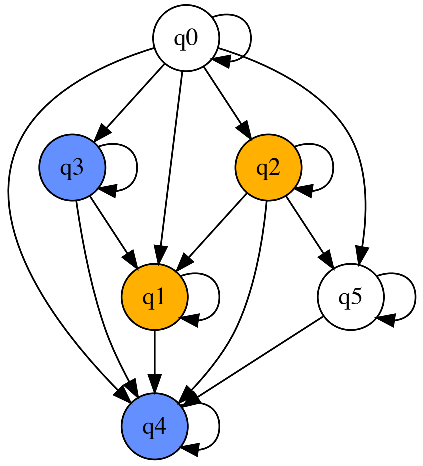
Hardware Experiments

The following two experiments were executed on hardware using two Unitree A1 quadrupeds.

Experiment 4.10 (Patrolling)**.** In this example, the grid world is shown in Figure 4.26a, the system starts in the lower right corner labeled S and wants to reach the target T in the lower left corner. The fuel capacity is 10 units and every transition



(a) Grid world layout.



(b) Specification product $\mathcal{B}_\pi$.

Figure 4.26: Grid world layout and specification product for Experiment 4.10.

on the grid decreases the fuel level by 1 unit. The refuel station is labeled R, which resets the fuel level to the maximum. The system state is $\mathbf{x} = (x, y, f)$, where x and y are the coordinates and f corresponds to the current fuel level. Originally, the grid does not contain any obstacles. The test agent can traverse up and down the grid in the third column, denoted by the yellow arrow in Figure 4.26a. It is also allowed to leave the grid from the top or bottom cell of the third column, this is denoted as the *park* states. Transitions into states that are not in the test agent's area can be restricted by placing static obstacles. The system objective is given as

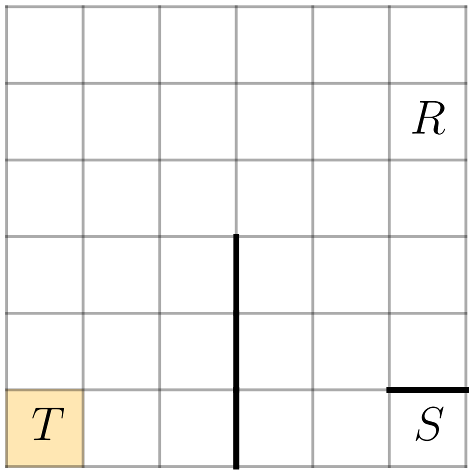
$$\varphi_{\text{sys}} = \Diamond T \wedge \Box\neg(f = 0).$$

The test objective wants to observe the system having to refuel on the way to T and is thus given as

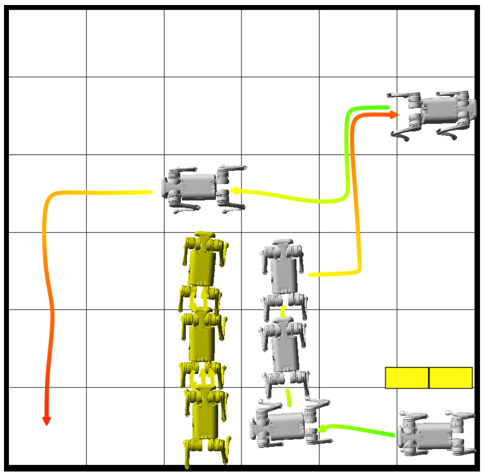
$$\varphi_{\text{test}} = \Diamond(d < f),$$

where $d = |(x, y) - \mathbf{x}_{\text{goal}}|$, is the distance to the goal state T. The specification product $\mathcal{B}_\pi$ corresponding to the system and test objective is shown in Figure 4.26b. Note that from $\mathcal{B}_\pi$ we can already observe that to route all test executions from the initial state q_0 through the test objective acceptance state (blue) before reaching the system acceptance states (yellow) it is sufficient to restrict system actions for the history variable q_0. The restrictions found by **MILP-Agent** and the counterexample-guided search are shown in Figure 4.27a. Due to its location outside of the test agent's moving area, the restriction in the lower right corner is a static cut, meaning that it will be present for all history variables in the form of a static obstacle. The cuts in the center can be realized by the reactive agent. The trace of the hardware execution is shaded in a color gradient according to the current fuel level and shown in Figure 4.27b and snapshots of the hardware during the test execution are shown in Figure 4.27c. We can see that the test agent blocks the system quadruped at the locations corresponding to the cuts (see panel 2, 3, and 4 in Figure 4.27c. Then the system quadruped decides to refuel (see panel 5) and the test agent retreats back down the grid to give way to the system quadruped (see panel 6). If the system quadruped would have chosen a different strategy, such as deciding to refuel earlier or even immediately, the test agent would not have blocked the system at all. However, the static obstacle was placed such that even if the system decided to refuel immediately, its fuel level would have been depleted enough to ensure that a refueling is necessary to successfully reach the goal state.

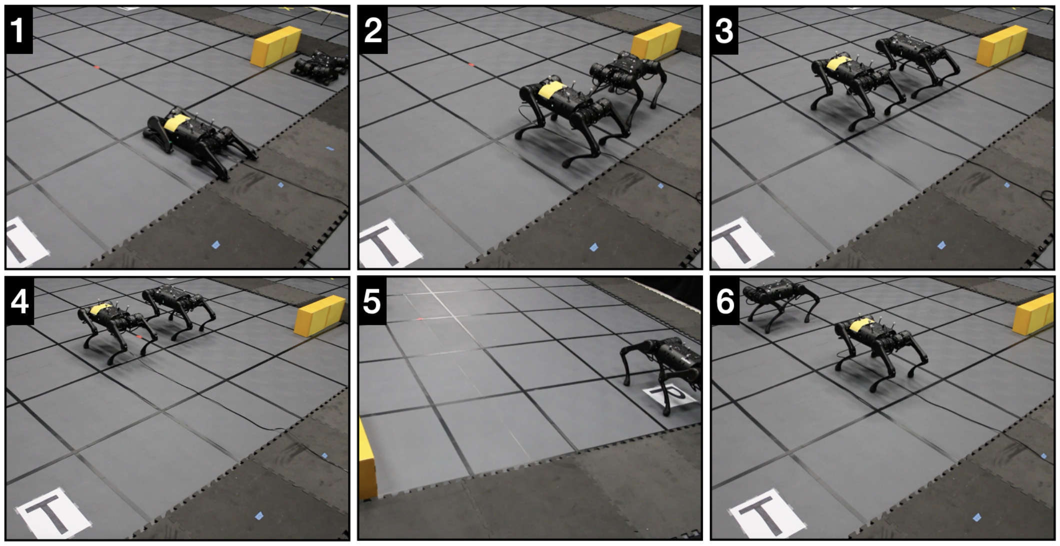
Experiment 4.11 (Grid World 2)**.** In this experiment, the quadruped starts in the lower left corner of the grid, labeled S, and wants to reach the top right corner, labeled T. The grid world layout is shown in Figure 4.28a and contains permanent obstacles, the cells shaded in dark gray that neither the system nor the test agent can occupy. The test agent quadruped can move along the center row and the center column of the grid, according to the yellow arrows. The system objective is given as $\varphi_{\text{sys}} = \Diamond T$. The test objective is the multiple visit task $\varphi_{\text{test}} = \Diamond I_2 \wedge \Diamond I_2$, where I_1 and I_2 correspond to the especially denoted cells on the grid, where the order of the visits is arbitrary. The specification product $\mathcal{B}_\pi$ is shown in Figure 4.28b. The initial history variable is q_0 and to route the test execution through the test objective accepting states (blue) before reaching the system objective acceptance states (yellow), the framework will restrict actions in q_0, q_6, and q_7. The counterexample-guided search using **MILP-AGENT** finds the reactive cuts shown in Figure 4.29. In this case, the optimization was able to find a solution that can be realized by the dynamic test agent alone, without the need for static obstacles. The trace of the hardware test execution is shown in Figure 4.30, and the corresponding snapshots are shown in Figure 4.31. The system quadruped is shown in gray and starts in the lower left corner of the grid. The test agent is the yellow quadruped which starts in the center of the grid. The initial history variable is q_0 and the test agent realizes the cuts in Figure 4.29a for this state of the test execution (see panel 1 in Figure 4.31). Initially, the system quadruped is free to choose one of the two paths toward the goal state, and it chooses the top path via I_2 (see panel 2). Once the system quadruped passes I_2, the test execution moves to the history variable q_7 and the test agent moves upwards to block the system (see panel 3). The system now realizes that the path is blocked,



(a) Reactive and static cuts for q_0.

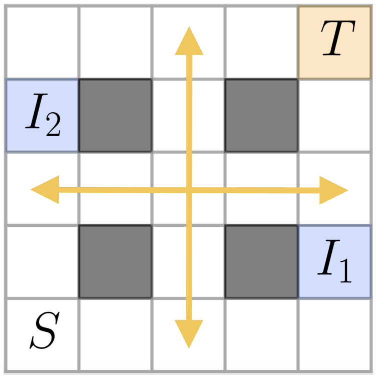


(b) Hardware trace.

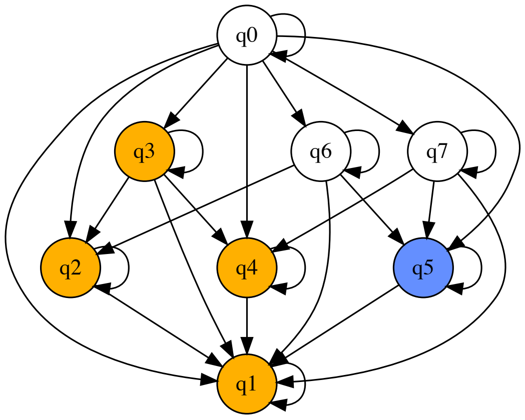


(c) Snapshots from Experiment 4.10.

Figure 4.27: Reactive cuts and hardware test execution trace and snapshots for Experiment 4.10.

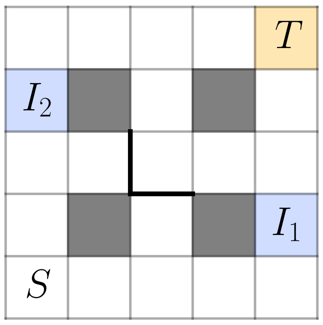


(a) Grid world layout.

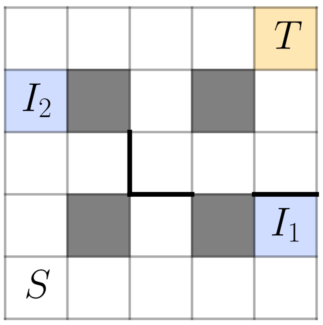


(b) Specification product $\mathcal{B}_\pi$.

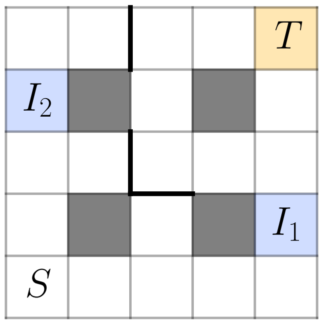
Figure 4.28: Grid world layout and specification product for Experiment 4.11.



(a) Reactive cuts in q_0.



(b) Reactive cuts in q_6.



(c) Reactive cuts in q_7.

Figure 4.29: Reactive cuts corresponding to the history variables for Experiment 4.11.

navigates back, and tries to pass through the center, but this is also blocked by the test agent (corresponding to the cuts in Figure 4.29c and panel 4 in Figure 4.31). The system quadruped now tries to navigate to T via the lower path, and as this path leads it through I_2 the test agent does not block the system anymore (see panel 5). The test execution does not go through the history variable q_6, as this history variable corresponds to seeing I_1 first. If the system quadruped had decided to try the lower path first, the system would have transitioned from q_0 to q_6 upon reaching I_1. The high-level system controller in this example does not correspond to a GR(1) controller, as the system could assume that the test agent blocks it forever in the worst case. We implement this as multiple high-level GR(1) controllers, where once the system is blocked, the controller is automatically re-synthesized with knowledge of the currently blocked states. In this example, the system controller is re-synthesized twice during the course of the test execution (shown in panel 3 and 4 in Figure 4.31).

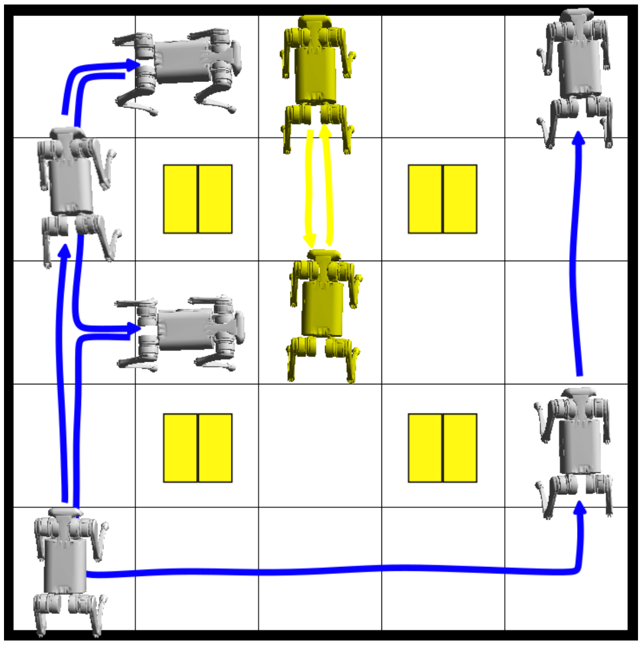


Figure 4.30: Hardware trace for Experiment 4.11.

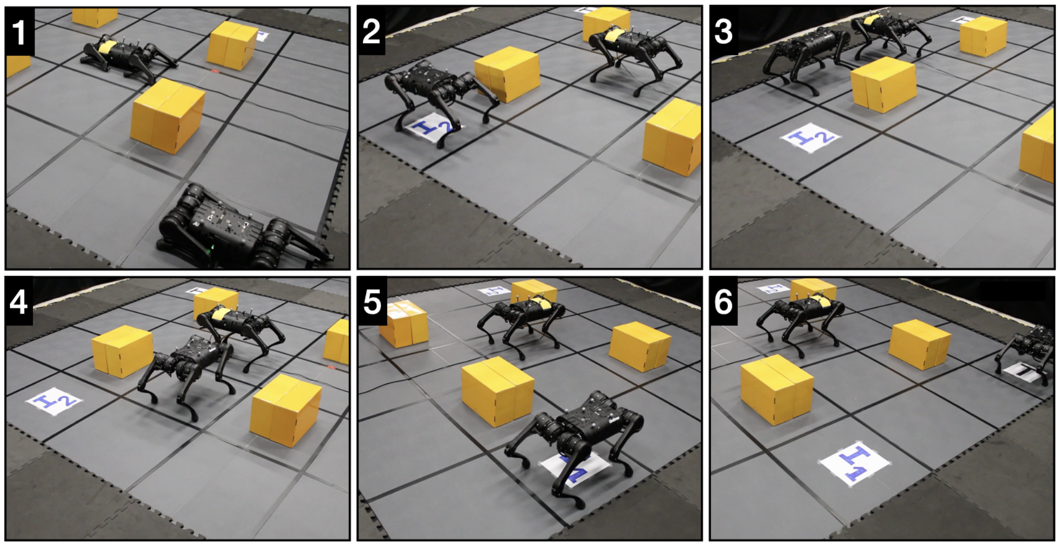


Figure 4.31: Snapshots for Experiment 4.11.

Table 4.2: Graph construction run times (with mean and standard deviation) for random 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
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
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

4.10 Run Times

In this section, we analyze the scalability of this framework for different randomized grids for increasing sizes and different specification patterns. Detailed run times for the simulated and hardware experiments can be found in Appendix A.1. For the randomized runtime experiments, we characterize the system and test objective by the number of propositions used to construct the formulae. For the reachability objectives, the system objective is $\varphi_{\text{sys}} = \diamond p_0$ and test objectives are $\varphi_{\text{test}} = \bigwedge_{i=1}^{n} \diamond p_i$ for i visit tasks. The number of propositions is then found as $|AP| = |p_0, \dots, p_n|$. In the case of reachability and reaction objectives, the system objective is given as $\varphi_{\text{sys}} = \diamond p_1 \bigwedge_{i=2}^{n} \square(p_i \rightarrow \diamond q_i)$, containing i reaction tasks and the task to visit the goal. The test objective corresponds to the triggers for each reaction task, $\varphi_{\text{test}} = \diamond p_0 \bigwedge_{i=2} \diamond p_i$. In this case, we find the number of propositions as $|AP| = |p_0, \dots, p_n, q_2, \dots, q_n|$. For example, testing a single reaction task corresponds to $|AP| = 3$, while testing two reaction tasks corresponds to $|AP| = 5$. Lastly, we also consider reachability and safety tasks. In this case the test objective is simply given as $\varphi_{\text{test}} = \diamond p_0$, and the system objective is $\varphi_{\text{sys}} = \diamond p_1 \bigwedge_{i=2}^{n} \square \neg p_i$, containing the goal visit task and i avoidance tasks. In this case, we have $|AP| = |p_0, \dots, p_n|$. Table 4.2 shows the run times for constructing the virtual product graph for these objective for different grid sizes. Increasing grid sizes correspond to an increase in the size of the transition system T. The grid worlds were generated with random grid cells being assigned to each atomic proposition in the formula. If the generated grid layout is infeasible, we remove it from the analysis and replace it by another randomly generated grid world. In total we run 25 instances for each grid size.

Table 4.3: Optimization run times (with mean and standard deviation) for random experiments using MILP-REACTIVE

Experiment		5×5	10×10	15×15	20×20
$\|AP\|$	$\|\mathcal{B}_\pi\|$	Optimization [s]			
Reachability:					
2	(4, 9)	5.63 ± 13.43	64.62 ± 38.75	67.38 ± 25.47	68.63 ± 31.12
3	(8, 27)	23.36 ± 38.15	61.68 ± 35.12	91.54 ± 31.41	117.82 ± 34.89
Reachability & Reaction:					
3	(6, 21)	5.97 ± 13.21	61.06 ± 34.67	71.64 ± 41.03	85.20 ± 19.49
5	(20, 155)	17.19 ± 25.51	78.44 ± 34.71	159.91 ± 76.63	280.16 ± 148.88
Reachability & Safety:					
3	(6, 18)	0.76 ± 1.52	70.82 ± 89.70	63.68 ± 27.54	80.58 ± 20.79
4	(6, 18)	0.150 ± 0.29	71.47 ± 80.61	59.59 ± 38.92	76.02 ± 27.11

Table 4.4: Optimization run time for random experiments using MILP-STATIC

Experiment		5×5	10×10	15×15	20×20
$\|AP\|$	$\|\mathcal{B}_\pi\|$	Optimization [s]			
Reachability:					
2	(4, 9)	8.17 ± 13.14	54.07 ± 17.98	60.168 ± 0.12	60.17 ± 0.10
3	(8, 27)	27.78 ± 21.71	60.17 ± 0.10	60.484 ± 0.86	74.02 ± 38.71
Reachability & Reaction:					
3	(6, 21)	10.62 ± 14.86	60.09 ± 0.06	60.23 ± 0.24	60.34 ± 0.46
5	(20, 155)	20.41 ± 19.21	67.77 ± 31.90	95.35 ± 116.82	268.68 ± 222.41
Reachability & Safety:					
3	(6, 18)	1.27 ± 1.47	60.08 ± 0.06	57.27 ± 12.61	60.32 ± 0.24
4	(6, 18)	0.17 ± 0.23	60.06 ± 0.05	60.14 ± 0.10	60.30 ± 0.19

The run times for solving the MILP in the reactive and static setting using Gurobi [66] are shown in Tables 4.3 and 4.4, respectively. We added a callback function to the optimization, such that if the optimal solution is not found, the optimization times out after 10 minutes. Once a feasible solution is found, the optimization terminates after 60 s and returns the current optimal solution. For the grid sizes in Tables 4.3 and 4.4 the optimization found a feasible solution for every instance within the allowed time frame.

4.11 Conclusion

In this chapter, we presented the flow-based synthesis framework to synthesize least-restrictive test strategies corresponding to a desired test behavior encoded in the system and test objectives. To analyze the different temporal events during a test execution we construct the virtual product graph from the system model and the system and test objectives. In addition, we construct the system product graph to keep track of the system's perspective. We formulate the routing optimization using flow networks as a MILP corresponding to the available test environment (**MILP-static, MILP-reactive**, or **MILP-agent**). In the static and reactive setting the reactive test strategy can be realized directly in the form of static obstacles or reactive obstacles such as doors or gates. For test environments consisting of a dynamic test agent we employ a counterexample-guided search and GR(1) synthesis to find a solution to **MILP-agent** that is realizable by the available test agent. We show that the routing problem is NP-hard, but in practice, this framework using the MILP can handle problems of medium size. The resulting test strategies ensure that a correctly implemented system can satisfy its objective, corresponding to a test that is not impossible from the system's perspective. Additionally, if the system fails to meet its system objective, it is due to a faulty system and not the fault of the test environment. The framework was illustrated on several simulated and hardware experiments for test environments with static and reactive obstacles, as well as test environments with dynamic test agents and static obstacles.

Chapter 5

SYSTEM DIAGNOSTICS USING ASSUME-GUARANTEE CONTRACTS

5.1 Introduction

Testing is an important step in the development of any system because it helps to identify errors, defects, or issues in a system or product before it is released to users or taken into operation. Through testing, system designers can establish a high level of confidence in the system's operation, which is crucial for ensuring its safety and overall performance. Diagnostics refers to the process of identifying, analyzing, and resolving problems that become apparent during the operation of a system, product, or process. It involves using various techniques and tools, depending on the nature of the problem, to pinpoint the root cause and find a solution to resolve the issue. Diagnostics consists of analyzing data, and symptoms and evaluating possible causes of the violation to determine the underlying cause.

Our diagnostics approach differs from existing work, as we are focusing on a system-level failure and seek to identify the root cause of this failure on the component level. Consider an autonomous vehicle consisting of multiple components, where the system designers are confident in the operation of the system. During a test, the system unexpectedly exhibits unsafe behavior, a violation of the system-level safety guarantees. As the system can consist of many components that are responsible for different tasks, but still interact with each other, finding the cause for a system-level failure is a very involved process. Our approach utilizes assume-guarantee reasoning and leverages the syntax of specifications to facilitate tracing the causes of violated system-level guarantees to potential subsystems. One major benefit of using assume-guarantee contracts (defined in Chapter 2) is the notion of blame. When the assumptions and guarantees are specified in this manner, it is possible to determine whether a component malfunctioned by monitoring whether the assumptions and guarantees were satisfied. For example, suppose we have a trace for a component. If this trace violates the assumption of the component, the component cannot be blamed for any undesired behavior. On the other hand, under satisfied assumptions, the behavior has to satisfy the promised guarantees. If the component does not deliver its guarantees in this case, then it did not satisfy its specification. One can

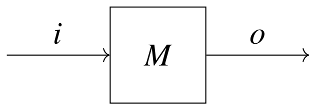


Figure 5.1: Component M block diagram with i and o, input and output variables.

now analyze the component further and determine whether the implementation was faulty or if anything was missed when defining the specification.

To be able to analyze the assumptions and guarantees of every component in a system, we would need access to all parameters in a system, including all internal variables, capturing relevant component inputs, outputs, and internal parameters. In some cases, it might be possible to trace a system-level output to a set of components by only focusing the analysis on components that directly interact with the component whose output was violated, and expanding the search from there, but this can still result in a large dataset that needs to be analyzed. Depending on the technology that is tested, ease of access to internal data may vary. For autonomous vehicles, test logs can be readily available, albeit enormous in size, while for other industries, such as synthetic biology, only certain limited data can be gathered. The framework we present in this chapter aims to pinpoint possible failure causes by identifying which component's guarantees to assess. This systematic approach allows for a targeted search of relevant data in the test logs or can provide insights on how to instrument the system for further insights.

The ultimate objective of this approach is to assist system designers in the failure diagnosis process by automatically analyzing violations and identifying potential culprits. This information can be valuable in guiding further testing, including sensor placement. To achieve this we use Pacti [79], a tool for system analysis and design using assume guarantee contracts, and build our framework by augmenting Pacti's contract composition function.

Pacti - A Tool for Compositional System Analysis and Design

Pacti [79] is an open-source Python package for compositional system analysis and design. Components can be defined using assume-guarantee contracts and contract operations can be performed, such as composition, merging, and quotient. Contracts in Pacti are defined over a universe of behaviors $\mathcal{B} = \prod_{V \in \textbf{VarSet}} \mathcal{B}_V$, where **VarSet** is the set of variables in the system. In the standard definitions, contracts are defined over sets of behaviors, which are not conducive to implementation. To alleviate this, a Boolean algebra T, called a *term algebra*, was introduced in [79]. Given this term algebra T, contracts can be written as $C = (a, g)$, where $a \in T$ and $g \in T$ are terms. An IO contract for each component can be defined as follows.

Definition 5.1 (IO Contract [79])**.** Let V be a set of variables. An *IOContract* is the tuple $(I, O, \mathfrak{a}, \mathfrak{g})$, where $I, O \in V$ are disjoint sets of input and output variables respectively, $\mathfrak{a} \in T$ a set of assumptions, and $\mathfrak{g} \in T$ a set of guarantees.

Defining contracts over the term algebra allows us to compute contract operations, for example the composition of contract $C = (a, g)$ and contract $C' = (a', g')$ can be directly computed as $C_c = C \parallel C' = ((a \wedge a') \vee (a \wedge \neg g') \vee (a' \wedge \neg g), (g \vee \neg a) \wedge (g' \vee \neg a'))$. This composition is the most refined contract that a system composed of two components M and M' will satisfy, provided that M and M' were implemented such that they satisfy their corresponding contracts C, and C', respectively. In practice, this composition might not prove to be very useful for multiple reasons. It may contain many terms in the assumptions and the guarantees, thus making it difficult for humans to understand. In addition, it might contain variables that we do not want to consider at this level of the composition (i.e. internal variables that we do not have access to).

To make the composed contract more user-friendly, it can be relaxed by either refining the assumptions, relaxing the guarantees, or both. Intuitively, refining the assumptions corresponds to 'shrinking' the size of the valid assumption set. Relaxing the guarantees corresponds to increasing the size of the guarantee set. For two assumptions a and a', we denote that a' refines a, $a \geq a'$, if the set corresponding to a' is a subset of the set corresponding to a. Similarly, for guarantees g and g', we say that g' is a relaxation of g, $g \leq g'$, if the set corresponding to g is a subset of the set corresponding to g'. When thinking about assumptions and guarantees each as conjunctions of terms (or constraints), refining a contract informally corresponds to either assuming *more*, guaranteeing *less*, or both. Pacti makes use of this contract relaxation to eliminate internal variables and make the contract more readable. Any relaxed contract that is computed in this way will be satisfied by a correct implementation of the system. Nevertheless, for a contract to be useful, we need to compute the relaxation systematically, as in the extreme case a contract that guarantees *True*, i.e. every possible behavior, is a valid refinement, yet pointless in the context of capturing the system's behavior.

When computing a composition of contracts C and C', the assumptions of the

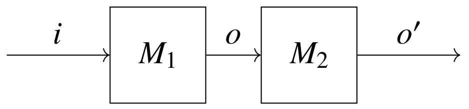


Figure 5.2: Block diagram for composition in Example 5.2.

composed contract are given as follows:

$$a_c = \overbrace{(a \wedge a')}^{\text{stem}} \vee \overbrace{(a \wedge \neg g') \vee (a' \wedge \neg g)}^{\text{error terms}} \geq (a \wedge a'). \tag{5.1}$$

We refer to the first term as the *stem*, as this is where the composed system should operate—where the assumptions of both components are satisfied. The second and third terms are referred to as *error terms*, where each term refers to one of the components not delivering their guarantees. As we want the composition to live in the stem, we can use the error terms to modify the stem into the desired form (eliminate unwanted variables). Once the error terms are no longer useful to transform the stem, we can define the assumption of the relaxed contract as the transformed stem by contract abstraction. The transformation of the guarantees follows a similar argument, but ensures that the variables are eliminated by relaxing the guarantees. The guarantees of a composition are given as follows:

$$\begin{aligned} g_c &= (g \vee \neg a) \wedge (g' \vee \neg a') \\ &= \underbrace{(g \wedge g')}_{\text{stem}} \vee (\neg a \wedge g') \vee (\neg a' \wedge g) \vee (\neg a \wedge \neg a') \\ &\leq (g \wedge g') \vee \neg(a \wedge a'), \end{aligned}$$

where the *stem* again refers to the desired area of operation, when both components satisfy their guarantees. The stem may contain variables that should be eliminated. We can use the remaining terms to transform the stem to eliminate these variables, we will refer to these terms as the *context* terms.

Example 5.1. Given two components M_1 and M_2 and their inputs and outputs as illustrated in Figure 5.2 and their IO contracts as $C_1 = (\{i\}, \{o\}, \mathfrak{a}_1, \mathfrak{g}_1)$, where $\mathfrak{a}_1 = \{i \leq 2\}$, and $\mathfrak{g}_1 = \{o = i\}$ and $C_2 = (\{o\}, \{o'\}, \mathfrak{a}_2, \mathfrak{g}_2)$, where $\mathfrak{a}_2 = \{o \leq 1\}$, and $\mathfrak{g}_2 = \{o = o'\}$. The stem term of the composed assumptions contains the internal variable o, to eliminate it, we make use of the error terms as follows.

$$\begin{aligned}
a_c &= (\overbrace{(i \leq 2 \land o \leq 1)}^{\text{stem}} \lor \overbrace{(i \leq 2 \land \neg(o = i)) \lor (o \leq 1 \land \neg(o = o'))}^{\text{error terms}}\\
&= (i \leq 2) \land (o \leq 1 \lor \neg(o = i)) \lor (o \leq 1 \land \neg(o = o'))\\
&= (i \leq 2) \land (o \leq 1 \land o = i \lor \neg(o = i)) \lor (o \leq 1 \land \neg(o = o'))\\
&= (i \leq 2) \land (i \leq 1 \land o = i \lor \neg(o = i)) \lor (o \leq 1 \land \neg(o = o'))\\
&= (i \leq 2 \land i \leq 1) \lor (i \leq 2 \land \neg(o = i)) \lor (o \leq 1 \land \neg(o = o'))\\
&= \underbrace{(i \leq 1)}_{\text{transformed stem}} \lor (i \leq 2 \land \neg(o = i)) \lor (o \leq 1 \land \neg(o = o'))
\end{aligned}$$

This allows us to define the assumptions of the relaxed composed contract as $\mathfrak{a} = \{i \leq 1\}$, which now only contains the top-level input variable i. When computing the guarantees, we can make use of the transformed assumptions when relaxing the guarantees to eliminate the internal variable o from the stem.

$$\begin{aligned}
g_c &= \overbrace{(o = i \land o = o')}^{\text{stem}} \lor (\neg(i \leq 2) \land (o = o')) \lor (\neg(o \leq 1) \land (o = i))\\
&= (i = o') \lor (\neg(i \leq 2) \land (o = o')) \lor (\neg(o \leq 1) \land (o = i))\\
&\leq (i = o') \lor \neg(i \leq 1)
\end{aligned}$$

In this example, the stem contained all necessary terms to eliminate the internal variable without using any of the additional terms. Lastly, we could relax the guarantees by substituting the error terms for the negation of the transformed stem of the assumptions. This is now the saturated form of the guarantees. The resulting contract is $C = (\{i\}, \{o'\}, \{i = o'\}, \{i \leq 1\})$. In the above example, each contract only had a single term in their respective assumption and guarantee set. In the case of multiple terms, the assumptions and guarantees are the conjunction of these terms. During the relaxation of the contract, Pacti includes additional steps during the transformation of each term that we will call *eliminating redundancies* and *filtering*. To eliminate redundancies, Pacti uses standard methods to eliminate redundant terms [105, 147] to check whether a term is redundant and remove it from the context. Filtering identifies which other terms are useful during the transformation; we will refer to these useful terms as the *relevant context*. More details on the theory and implementation can be found in [79] and more details on the filtering procedure can be found in [80].

Pacti currently supports linear inequalities with real coefficients as the theory to define the terms. When performing a composition, Pacti computes the top-level assumptions and guarantees by relaxing the guarantees and refining the assumptions to eliminate internal variables. During this process for each term in the assumptions and guarantees of the composed component, only specific terms are used in their generation, which are identified during the filtering procedure. The framework presented in this thesis makes use of instrumenting this filtering step to store the history for each computed assumption and guarantee which then allows us to provide a systematic diagnostics procedure for the identification of responsible components.

All operations in Pacti are computed under the assumption that the components operate correctly as specified by their contracts. The entire premise of testing lies in the difference between a 'perfectly' specified system and its real-world implementation. In a real-world system, specifications might be incomplete, or implementations might be faulty. A single component failure might present itself in multiple ways, it might result in a system-level guarantee violation, or it might not. If a component failure is latent, meaning it does not show itself in the system-level guarantee violation, this framework cannot detect it. If the fault does show itself in a system-level guarantee violation, we can trace it to the responsible component(s) by tracking which relevant component's guarantees were violated under satisfied assumptions.

5.2 Tracing System Guarantees

In this section, we will focus on how we can trace top-level system guarantees down to the responsible components. This is beneficial in the diagnostics process, because if a system-level guarantee is violated, we will be able to identify what components could have contributed to this failure. We will present a methodological approach to how the system (and its subsystems) shall be modeled such that we can make use of Pacti, a tool for compositional system-level analysis and design, and augment this tool to provide a systematic diagnosis procedure.

Defining the components

Each component of the system can be modeled as an assume-guarantee contract, where the environment in which the component is expected to operate is constrained by the assumptions, and the guarantees describe the component's expected behavior. We express the assumptions and guarantees using sets of linear inequalities. Additionally, we are required to define the input and output variables to create an IO contract according to Definition 5.1 to enable the composition of the components

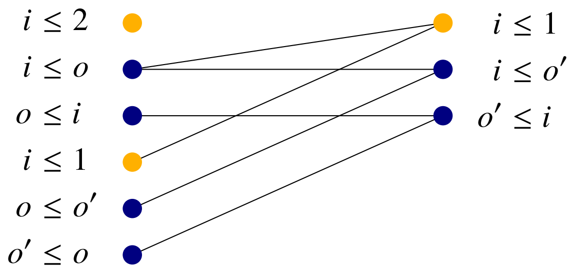


Figure 5.3: Composition graph for Example 5.1, where each vertex corresponds to a term in the assumptions (yellow) or guarantees (blue) of the contracts.

in Pacti. For example, for component M illustrated in Figure 5.1 we can define the corresponding IO contract $C = (I, O, \mathfrak{a}, \mathfrak{g})$ where the set $\mathfrak{a}$ contains the term $i <= 2$ and the guarantee set $\mathfrak{g}$ contains the term $o <= 2i + 1$, where $I = \{i\}$, and $O = \{o\}$ are the singleton sets of the input and output variables.

Definition 5.2 (Faulty component)**.** Given a component M and the corresponding contract $C = (I, O, \mathfrak{a}, \mathfrak{g})$, M is *faulty* if it does not satisfy the guarantees $\mathfrak{g}$, but the assumptions $\mathfrak{a}$ are satisfied. That is, for a faulty component the implementation does not satisfy the contract, $M \not\models C$.

Composing two components

As explained in the background section of this chapter, Pacti uses a filtering procedure to determine the relevant context terms when computing the assumptions and guarantees of the composed component. During this procedure, for every term the subroutine `transform` is called, which identifies the relevant context terms. We implemented an ID system, that allows us to store which term was used to generate another term. For each composition operation, we can define a composition graph, that allows us to map the composed assumption and guarantee terms to the terms that were used in their transformation. Simply stated, a composition graph (see for example in Figure 5.3) consists of a set of vertices, where each vertex corresponds to a term in the assumptions or guarantees of the individual contracts (shown on the left), and the composed contract (shown on the right). The edges in the composition graph connect the vertices if the corresponding individual contract term was used to generate the composed contract term, shown as the edges from left to right in Figure 5.3. Therefore, by analyzing the vertices and edges in a composition graph, we can identify which terms from the individual contracts were used in the generation

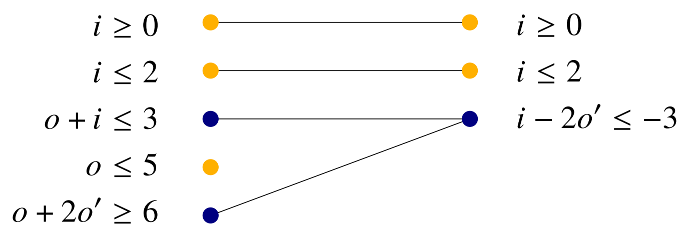


Figure 5.4: Composition graph for Example 5.2, where each vertex corresponds to a term in the assumptions or guarantees of the contracts.

of each term in the composed contract.

Definition 5.3 (Composition Graph)**.** Let there be two components M_1 and M_2 and their corresponding IO contracts $C_1 = (I_1, O_1, \mathfrak{a}_1, \mathfrak{g}_1)$, and $C_2 = (I_2, O_2, \mathfrak{a}_2, \mathfrak{g}_2)$ and their composed IO contract $C = (I, O, \mathfrak{a}, \mathfrak{g})$. A *composition graph* is defined as a directed graph $G = (V, E)$, with a set of vertices V and a set of edges E. Each vertex corresponds to a term of the individual component contracts and their composition, meaning the sets of vertices $V_{i,a}$ and $V_{i,g} \subseteq V$ correspond to the assumptions $\mathfrak{a}_i$ and guarantees $\mathfrak{g}_i$ of component i, and $V_a, V_g \subseteq V$ correspond to the composed assumptions and guarantees $\mathfrak{a}$, and $\mathfrak{g}$, respectively. For simplicity, for each term $s \in \mathfrak{a} \cup \mathfrak{g} \bigcup_{i=1}^{2} (\mathfrak{a}_i \cup \mathfrak{g}_i)$ we will use the same notation to denote the term and the corresponding vertex $s \in V$. Vertices $u, v \in V$ are connected by an edge $(u, v) \in E$ iff the term corresponding to u was used to generate the term v. We define a path on G between two vertices $s, t \in V$ if there exists a sequence of edges $e \subseteq E$ that connect vertex s with vertex t, and denote it as $\mathtt{path}(G, s, t)$.

Example 5.1 (continued)**.** Figure 5.3 shows the composition graph for the composition shown in Figure 5.2. Equalities are represented as two inequalities in Pacti, thus $i \leq o'$ and $o' \leq i$ capture the composed guarantee that $i = o'$ (and similarly for the component guarantees).

Example 5.2. Given two components M_1 and M_2 and their inputs and outputs as illustrated in Figure 5.2 and their IO contracts as $C_1 = (\{i\}, \{o\}, \mathfrak{a}_1, \mathfrak{g}_1)$, where $\mathfrak{a}_1 = \{i \geq 0, i \leq 2\}$, and $\mathfrak{g}_1 = \{o + i \leq 3\}$ and $C_2 = (\{o\}, \{o'\}, \mathfrak{a}_2, \mathfrak{g}_2)$, where $\mathfrak{a}_1 = \{o \leq 5\}$, and $\mathfrak{g}_1 = \{o + 2o' \geq 6\}$. The composition results in the contract $C = (\{i\}, \{o'\}, \mathfrak{a}, \mathfrak{g})$, where $\mathfrak{a} = \{i \geq 0, i \leq 2\}$, and $\mathfrak{g} = \{i - 2o' \leq -3\}$. The composition graph corresponding to the composition $C_1 \parallel C_2$ is shown in Figure 5.4.

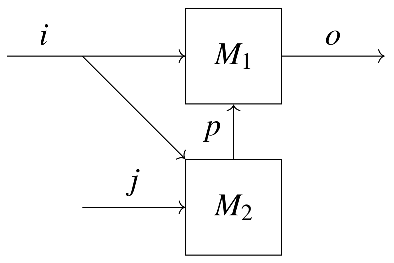


Figure 5.5: Block diagram for composition in Example 5.3.

Components do not necessarily have to be connected in series to form a system. Their inputs and outputs can be interconnected in different ways, except for feedback if both components make assumptions on their respective input feedback variable. From the examples presented in this section, we can observe that for a composition, the resulting diagnostics graph is very sparse. This allows us to trace a composition-level guarantee back to a small set of terms on the component level. This observation allows us to extend the analysis to compositions of multiple components and to limit the blame from a violated system-level guarantee to specific components.

Example 5.3. Given two components M_1 and M_2 and their inputs and outputs as illustrated in Figure 5.5 and their IO contracts as $C_1 = (\{i, p\}, \{o\}, \mathfrak{a}_1, \mathfrak{g}_1)$, where $\mathfrak{a}_1 = \{i \leq 2, i \geq 0, p \geq 0\}$, and $\mathfrak{g}_1 = \{o + p \leq 3, o - i \geq 0\}$ and $C_2 = (\{i, j\}, \{p\}, \mathfrak{a}_2, \mathfrak{g}_2)$, where $\mathfrak{a}_2 = \{i \leq 5, j \geq 0, i + j \leq 10\}$, and $\mathfrak{g}_2 = \{p + j \geq 6, p - i \geq 0\}$. The resulting composition contract is $C = (\{i, j\}, \{o\}, \mathfrak{a}, \mathfrak{g})$, where $\mathfrak{a} = \{i \geq 0, i \leq 2, j \geq 0, i + j \leq 10\}$, and $\mathfrak{g} = \{o - j \leq -3, i - o \leq 0, i + o \leq 3\}$. The composition graph corresponding to the composition $C_1 \parallel C_2$ is shown in Figure 5.6.

Composing the system

We can now create a diagnostics graph for the composition of two components and their corresponding IO contracts. To build the overall system, we need to compose multiple components. Contract composition is a binary operation. Therefore, to compose the entire system, we need to compose two components at a time, and then compose their composition with the next component. There are many different ways of composing the same system, and the resulting contract for the composition is dependent on the order of composition. As Pacti tries to hide internal variables, the composition has to be chosen carefully such that the variables necessary for future compositions are kept. This can be done either by deciding on a specific composition order or by setting a flag in Pacti that ensures that the desired variable is kept as the output variable.

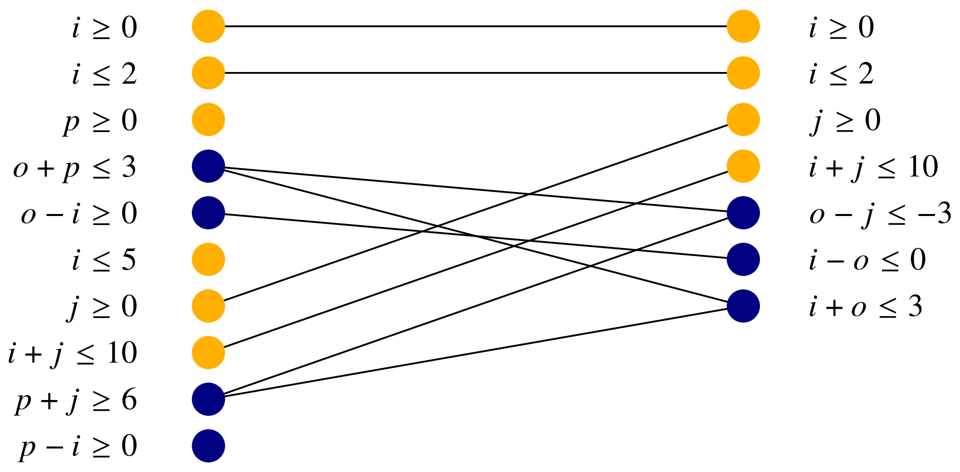


Figure 5.6: Compostition graph for Example 5.3, where each vertex corresponds to a term in the assumptions or guarantees of the contracts.

Another important aspect that can guide the composition order is the availability of component data. When there is a lack of available information from inside a block of components it might be beneficial to compose these components first and treat them as a meta-component—a grouping of multiple components. If the analysis ends up pointing to this meta-component as the possible cause, a more detailed analysis can still be set up focusing on these components.

While every computed contract is a correct relaxation of the composed system, in practice, different composition orders can provide more or less useful results. As guarantees can be relaxed, for some composition order, the guarantee that we might be looking for could have been removed from the set of guarantees during the relaxation. We are therefore required to choose a composition order that ensures that the violated guarantee is part of the resulting guarantee set. For this framework, we assume that when a composition order is chosen, this order will be maintained for the remainder of the diagnostics process.

Definition 5.4 (Composition Order)**.** The component contracts are provided in the form of a *composition order* `CompOrd` $= [C_1, \ldots, C_N]$, where N is the number of components in the system. `CompOrd` contains the component contracts in the order of composition when computing the system specification. The composition up to component contract k is defined as $C_{\text{comp},k} := C_1 \parallel \ldots \parallel C_k$.

This composition order requires composing the system starting from a single component and building the system up from there. If it is desired to start by composing certain regions of the system first, this framework can easily be extended to include this approach. Otherwise, the component contracts in the composition order need to be provided at the level of granularity such that they can be composed according to a composition order defined in Definition 5.4. We will now define the diagnostics graph that corresponds to the composition of multiple components and outline how it is constructed. The union of two graphs G_1 and G_2 is defined as $G = (V_1 \cup V_2, E_1 \cup E_2)$, and we will denote it by $G = G_1 \cup G_2$. Simply stated, the diagnostics graph consists of multiple composition graphs. The system is composed step-by-step according to the composition order, and for every composition the diagnostics graph is extended by the composition graph for this composition. An example is shown in Figure 5.8, where we can see the union of three composition graphs. Each 'column' of vertices corresponds to the individual contract terms in a composition, where the top nodes correspond to the already composed system (or the first component for the first composition), and the bottom vertices correspond to the next contract in the composition order.

Definition 5.5 (Diagnostics Graph)**.** Given component contracts in a composition order $\texttt{CompOrd} = [C_1, \ldots, C_N]$, the *diagnostics graph* $G = (V, E)$ is constructed as follows. For each i, $2 \leq i \leq N$, we compute the composition $C_{\text{comp,i-1}} \parallel C_i$ with the corresponding graph G_i. Then the diagnostics graph G is defined as $G = \bigcup_{i=2}^{N} G_i$.

Definition 5.6 (Diagnostics Map)**.** Let $\texttt{CompOrd}$ be the composition order for N components and their contracts. Let $C = \parallel_{i=1}^{N} C_i = (\mathfrak{a}, \mathfrak{g})$ be the system-level composition according to the composition order, and let the corresponding diagnostics graph be G. Then, we can define the *diagnostics map* $\texttt{CM} : \mathfrak{g} \longrightarrow 2^{(\bigcup_{i=1}^{N}(\mathfrak{a}_i \cup \mathfrak{g}_i)) \times \{C\}_{i=1}^{N}}$, that maps each composed assumption and guarantee term to a set of component level assumption or guarantee terms through the diagnostics graph. That is, for system-level term s and the corresponding vertex $s \in V$, we have

$$\begin{aligned} \texttt{CM}(s) = &\{(t, C_i) \mid \forall C_i \in \texttt{CompOrd},\ t \in \mathfrak{g}_i \cup \mathfrak{a}_i,\ \text{if} \\ &\exists \texttt{path}(G, t, s) \text{ and } \forall u \in V, u \neq t \implies \nexists (u, t) \in E\}, \end{aligned} \tag{5.2}$$

where $t \in V$ corresponds a component-level term used to generate s, and i is the index of the component the term belongs to.

The diagnostics map finds the leaf nodes in the diagnostics graph, that have a path to the vertex corresponding to the violated guarantee. For each leaf node, it returns the term corresponding to the leaf node and the contract that this term belongs to

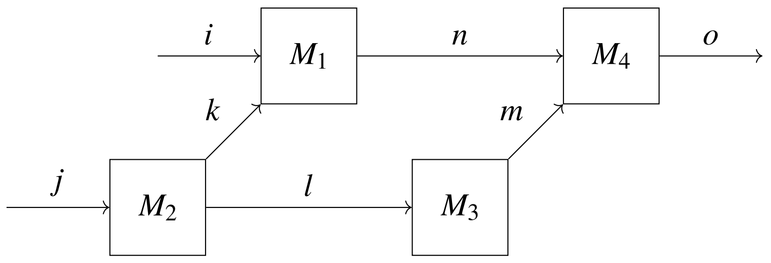


Figure 5.7: Block diagram for composition in Example 5.4.

in the form of a tuple. With this information we can now focus our attention on these terms first and start the diagnostics process by evaluating these terms using the available test data.

Example 5.4. Given a list of component contracts in the composition order $\texttt{CompOrd} = [C_1, C_2, C_3, C_4]$ corresponding to the block diagram shown in Figure 5.7. The contracts are given as: $C_1 = (\{i, k\}, \{n\}, \mathfrak{a}_1, \mathfrak{g}_1)$, where $\mathfrak{a}_1 = \{i \leq 2, i \geq 0, k \leq 5\}$ and $\mathfrak{g}_1 = \{n + i \leq 3, n \geq 0\}$; $C_2 = (\{j\}, \{k, l\}, \mathfrak{a}_2, \mathfrak{g}_2)$, where $\mathfrak{a}_2 = \{j \leq 5, j \geq 0\}$, and $\mathfrak{g}_2 = \{j + k + l \geq 6, k \leq 5, l \leq 5\}$; $C_3 = (\{l\}, \{m\}, \mathfrak{a}_3, \mathfrak{g}_3)$, where $\mathfrak{a}_3 = \{l \leq 5, l \geq 0\}$ and $\mathfrak{g}_3 = \{l+m \geq 6, m \leq 10\}$; $C_4 = (\{n, m\}, \{o\}, \mathfrak{a}_4, \mathfrak{g}_4)$, where $\mathfrak{a}_4 = \{m \leq 10, m \geq 0, n \leq 5, n \leq 0\}$ and $\mathfrak{g}_4 = \{m + o \geq 6, n + o \geq 0, o \leq 5\}$. The system-level contract is $C = (\{i, j\}, \{o\}, \mathfrak{a}, \mathfrak{g})$, where $\mathfrak{a} = \{j \leq 1, i \leq 2, i \geq 0, j \geq 0\}$, and $\mathfrak{g} = \{i - o \leq 3, o \leq 5\}$. The diagnostics graph for this system composition is shown in Figure 5.8.

The violating trace is given as the following variable assignment: $i = 2$, $j = 1$, $k = 4$, $l = 2$, $m = 2$, $n = 1$, and $o = -2$. We observe that the system-level assumptions $\mathfrak{a}$ are satisfied but the system-level guarantee $g_v := i - o \leq 3$ is not satisfied, as $i - o = 2 + 2 = 4 \nleq 3$.

The diagnostics map for this guarantee is $\texttt{CM}(g_v) = \{(n+i \leq 3, C_1), (n+o \geq 0, C_4)\}$, derived from the diagnostics graph, illustrated by the red edges in Figure 5.8. Therefore we will start the diagnostics process by evaluating these guarantees of components M_1 and M_4 first. Checking that the guarantee $i + n = 2 + 1 \leq 3$ from contract C_1 is satisfied, we move on to component M_4. Subsequently, we observe that $n + o \geq 0$ is not satisfied, as $n + o = 1 - 2 = -1 \ngeq 0$, narrowing our focus on component M_4. The assumptions of C_4 are satisfied, i.e. $0 \leq m \leq 10$ and $0 \leq n \leq 5$, thus, allowing us to identify component M_4 the culprit of the violation. To come to this conclusion, we only needed to evaluate these two component guarantees, and the assumptions of C_4, allowing us to check only 6 statements instead of having to

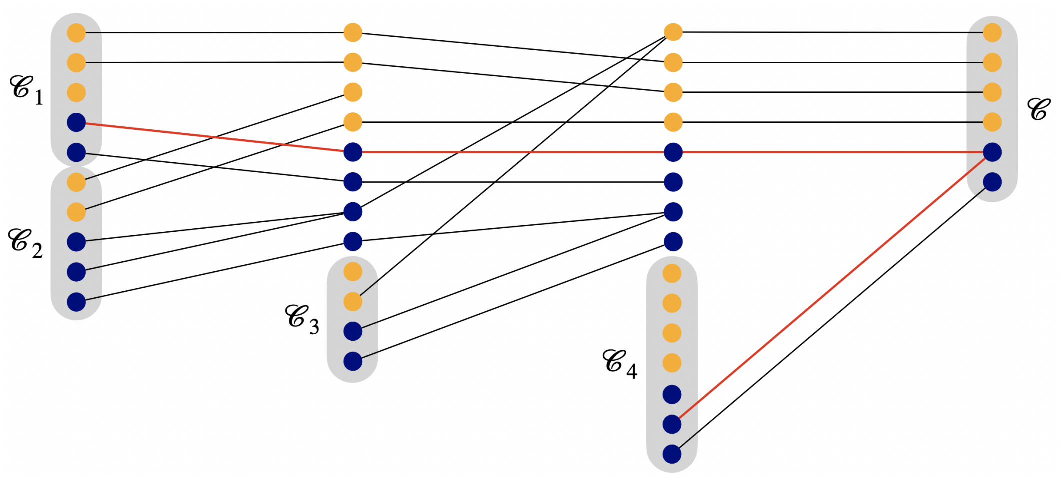


Figure 5.8: Diagnostics graph for Example 5.4. Vertices corresponding to assumptions are shown in yellow, and vertices corresponding to guarantees are shown in blue.

evaluate all 21 component assumptions and guarantees.

5.3 Identifying Relevant Information

Given a system, its block diagram, and the diagnostics graph corresponding to the composition, we can identify which information is relevant to determine which components are the cause of the observed top-level guarantee failure. For autonomous systems, it is reasonable to assume that a log exists that contains all the information gathered from the system and its components during the test. With this framework, we can identify which relevant information to look for in the log to reduce the time required to analyze the entire log. When we do not have access to the full system information but instrumentation can be added to rerun the test, we can identify required sensor locations to access the relevant information. In the case of sparse sensing, i.e., for a biological experiment, this framework allows for tracing back the failure to an area limited by the available system information.

Using the diagnostics graph and the system block diagram we can identify the component assumptions and guarantees that were relevant to the generation of the top-level system guarantee that was violated. We can then focus on the relevant components of the system block diagram and analyze whether the guarantee was satisfied or violated. Once we find a component where the guarantee was violated we can shift our focus to the assumptions of this term. From then on two different scenarios can occur: i) the assumptions are satisfied, or ii) the assumptions are violated.

Case i). If the assumptions are satisfied, this component is the cause of the failure. In this case, we either have a component failure or the system designer forgot to specify an assumption. At this point, the analysis at this level terminates and the component designer needs to analyze the behavior of this component.
Case ii). In the case of violated assumptions, the component is likely not the cause of the failure, as another component's behavior resulted in the violation of the assumptions. To identify this component, we need to trace the violated assumption back to a component guarantee (or multiple). To achieve this we find in the composition order when this component was composed with another component and make use of the Pacti function ElimVarsbyRefinement to find out which terms were used when transforming this assumption. Then we refer to the diagnostics graph again which allows us to trace these newly identified guarantees back.

It is important to note that this process of tracking assumptions only applies to components whose assumptions are not solely dependent on the overall system input variables. A system-level failure is defined as having satisfied the system-level assumptions, but failing to satisfy the guarantees of the system. Thus, the system-level assumptions are satisfied—this will ensure that component-level assumptions that are only dependent on the system-level input variables are also satisfied.

Identifying Causes for Violated Assumptions. Assume we are given the component M, its corresponding contract as $C = (I, O, \mathfrak{a}, \mathfrak{g})$, component M_{other}, the component in the composition order that is composed with component M, and its corresponding contract $C_{\text{other}} = (I_{\text{other}}, O_{\text{other}}, \mathfrak{a}_{\text{other}}, \mathfrak{g}_{\text{other}})$. The component assumption that was violated is denoted $a_v \subseteq \mathfrak{a}$. We will use Pacti to find the relevant context used to refine this assumption by calling the augmented version of ElimVarsbyRefinement, denoted by

$$\textsc{FindCauseForAssumption}(a_v, \mathfrak{a}_{\text{other}} \cup \mathfrak{g}_{\text{other}}).$$

This function call in Pacti transforms the assumption a_v with the use of $\mathfrak{a}_{\text{other}} \cup \mathfrak{g}_{\text{other}}$ as the context to eliminate any unwanted variables. We can make use of the same function augmentation that we created to compute the diagnostics graph to analyze the transformation at this level. The instrumentation of the filtering step will return to us the relevant context terms $\mathfrak{c}_r \subseteq \mathfrak{a}_{\text{other}} \cup \mathfrak{g}_{\text{other}}$ in the assumptions and guarantees of C_{other}. Once we have determined $\mathfrak{c}_r$ can refer to the diagnostics map and trace back the terms in $\texttt{CM}(c)$ for each $c \in \mathfrak{c}_r$ to the responsible component level terms. The entire diagnostics procedure is outlined in Algorithm 5.2.

Algorithm 5.1 Trace Term g_v

```
1: procedure TRACE($g_v$, CompOrd, CM, Log)
   Input: guarantee to trace $g_v$, composition order CompOrd = $[C_1, \ldots, C_N]$, causality map CM, log data Log, components $M_1, \ldots, M_N$
   Output: set of failed components $C_f$
2:     $C_f \leftarrow \emptyset$                                ▷ Initialize empty set of failed components
3:     for $(t, C_i) \in$ CM$(g_v)$ do          ▷ Component-level term $t$, component index $i$
4:         if $t \in \mathfrak{g}_i$ then                          ▷ $\mathfrak{g}_i$ are the guarantees of $C_i$
5:             if NOTSATISFIED($t$, Log) then     ▷ Check if $t$ is satisfied in log data
6:                 AssumptionsSatisfied ← True          ▷ Initialize flag as True
7:                 for $a_i \in \mathfrak{a}_i$ do
8:                     if NOTSATISFIED($a_i$, Log) then           ▷ Check $a_i$ in log data
9:                         AssumptionsSatisfied ← False
10:                        $C_{\text{other}} \leftarrow C_1 \parallel \ldots \parallel C_{i-1}$
11:                        $\mathfrak{c}$ ←FINDCAUSEFORASSUMPTION($a_i$, $\mathfrak{a}_{\text{other}} \cup \mathfrak{g}_{\text{other}}$)
12:                        for $c_k \in \mathfrak{c}$ do
13:                            $C_f \leftarrow C_f \cup$ TRACE($c_k$, CompOrd, CM)
14:                if AssumptionsSatisfied then
15:                    $C_f \leftarrow C_f \cup M_i$                    ▷ Add component $i$ to the list
16:    return $C_f$
```

Algorithm 5.2 Diagnosing Violated Guarantee g_v

```
1: procedure DIAGNOSE($g_v$, CompOrd, Log)
   Input: failed guarantee $g_v$, composition order CompOrd = $[C_1, \ldots, C_N]$, log data Log
   Output: set of failed components $C_f$
2:     G ← $(\emptyset, \emptyset)$                             ▷ Initialize empty diagnostics graph
3:     for $C_i \in$ CompOrd do
4:         $C_{\text{comp},i-1} \leftarrow C_1 \parallel \ldots \parallel C_{i-1}$
5:         $G_i \leftarrow$ COMPOSITIONGRAPH($C_{\text{comp},i-1}$, $C_i$))
6:         $G \leftarrow G \cup G_i$               ▷ Add composition graph to diagnostics graph
7:     CM ← define causality map according to equation (5.2)
8:     $C_f \leftarrow$ TRACE($g_v$, CompOrd, CM)              ▷ Find set of failed components
9:     return $C_f$
```

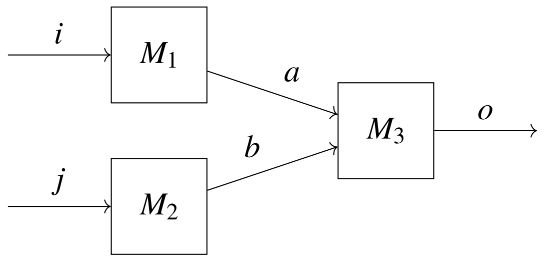


Figure 5.9: Block diagram for composition in Example 5.5.

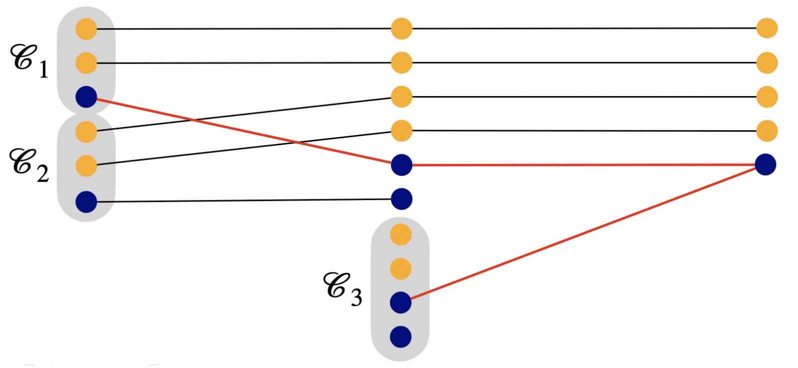


Figure 5.10: Diagnostics graph for Example 5.5.

Note that this approach can identify multiple component faults under certain conditions. As stated previously, we can only find the faulty component, if a system-level guarantee is violated. If two component faults end up cancelling each other out, (i.e. are not observable at the system level), then this approach cannot identify them as no system-level guarantee was violated. If a faulty component results in violated assumptions for another component, we cannot determine whether the component with the violated assumption also failed. This is due to the fact that per definition, for a contract with violated assumptions, any behavior is allowed. Under the condition that all faulty components are independent (i.e. a faulty component does not lead to violated assumptions of another faulty component), this procedure is able to identify all faulty components.

Example 5.5. Let there be a system consisting of three component contracts in the composition order `CompOrd` $= [C_1, C_2, C_3]$ with their inputs and outputs as illustrated in Figure 5.9. The IO contracts are given as $C_1 = (\{i\}, \{a\}, \mathfrak{a}_1, \mathfrak{g}_1)$, where $\mathfrak{a}_1 = \{i \leq 2, i \geq 0\}$, and $\mathfrak{g}_1 = \{a \leq 2\}$ and $C_2 = (\{j\}, \{b\}, \mathfrak{a}_2, \mathfrak{g}_2)$, where $\mathfrak{a}_2 = \{j \leq 2, j \geq 0\}$, and $\mathfrak{g}_2 = \{b \leq 3\}$ and $C_3 = (\{a, b\}, \{o\}, \mathfrak{a}_3, \mathfrak{g}_3)$, where $\mathfrak{a}_3 = \{a \leq 5, b \leq 5\}$, and $\mathfrak{g}_3 = \{o \leq a, o \leq b\}$.

The system-level contract is computed as $C = (\{i, j\}, \{o\}, \mathfrak{a}, \mathfrak{g}$, where $\mathfrak{a} = \{i \leq$

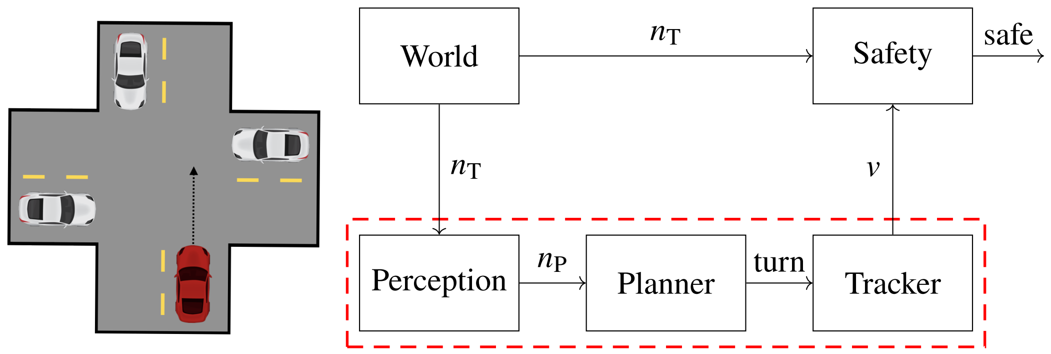
$2, i \geq 0, j \leq 2, j \geq 0\}$, and $\mathfrak{g} = \{o \leq 2\}$. Given the violating trace in the form of the variable assignment $i = 1$, $j = 1$, $a = 2$, $b = 7$, $o = 3$, the system-level guarantee $g_v := o \leq 2$ is violated, and the diagnostics map is $\mathtt{CM}(g_v) = \{(a \leq 2, C_1), (o \leq a, C_3)\}$. Thus we check the guarantee of component M_1 first and see that $a \leq 2$ is satisfied. Next, we check $o \leq a$, which is not satisfied, as $3 \nleq 2$. This narrows our analysis on component M_3. Evaluating the assumptions of contract C_3, we find that $a \leq 5$ is satisfied, but $b \leq 5$ is not. Therefore, M_3 is not responsible for the violation. We can now trace which terms were used to transform $b \leq 5$ to find which terms to evaluate next in our search for the failed component. For this, we compute $C_{\text{other}} = C_1 \parallel C_2$ from the composition order and evaluate $\textsc{FindCauseForAssumption}(b \leq 5, \mathfrak{a}_{\text{other}} \cup \mathfrak{g}_{\text{other}})$, which returns the relevant context term as the following guarantee from component contract C_2, $b \leq 3$. This guarantee is not satisfied as $7 \nleq 3$. Subsequently, we check the assumptions for C_2, $0 \leq j \leq 2$, which are satisfied, leading to the identification of M_2 as the component responsible for the violation. This example required checking 6 terms, instead of all 10 terms, which shows that even though we needed to trace the cause for a violated assumption this process resulted in checking fewer terms than evaluating all component-level terms using the log data.

Proposition 5.1. Given a complete log of test data for a violating trace $\mathtt{Log}$ and K component-level terms, where each term corresponds to a component assumption or guarantee, diagnosing a failed system-level guarantee g_v according to Algorithm 5.2 requires evaluating $L \leq K$ terms from the test log data.

Proof. Given a diagnostics map $\mathtt{CM}$, according to Algorithm 5.2 only component guarantees that are in $\mathtt{CM}(g_v)$ and their assumptions need to be evaluated. If an assumption is violated additional terms need to be checked. The equality holds when in the worst case the violated guarantee requires checking every single component guarantee and the corresponding assumptions, resulting in evaluating K terms using the log data. □

Theorem 5.2. *Suppose we have a list of components $M_1, \ldots, M_N$, their contracts in a composition order* $\mathtt{CompOrd} = [C_1, \ldots C_N]$*, a violated system-level guarantee g_v, and the complete log data of a failing trace* $\mathtt{Log}$*. If g_v is a guarantee of the composed system, we can identify the faulty component(s) using Algorithm 5.2.*

Proof. Let us denote the composed contract as $C = (I, O, \mathfrak{a}, \mathfrak{g})$. For a given composition and the corresponding contract C, under satisfied system-level assumptions



(a) Intersection layout.

(b) Diagram of Alice's components.

Figure 5.11: Layout of the intersection and Alice's component block diagram.

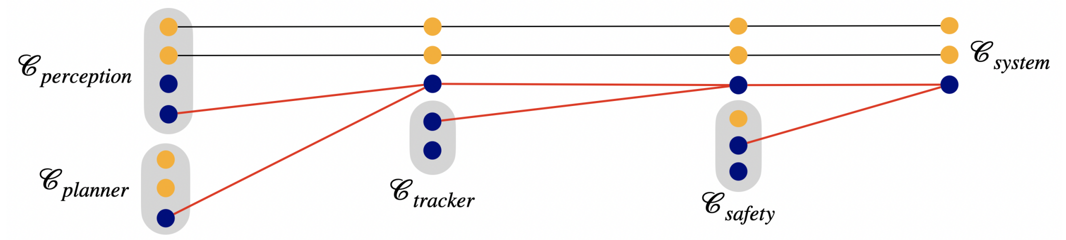


Figure 5.12: Diagnostics graph for Alice at 4-Way Intersection Example.

$\mathfrak{a}$ and a violated system-level guarantee $g_v \in \mathfrak{g}$, by construction of the composition, there exists at least one faulty component. From `CompOrd`, we can construct a diagnostics map `CM`. If $g_v \subseteq \mathfrak{g}$, $\mathtt{CM}(g_v)$ is guaranteed to contain at least one component-level guarantee $g_k \in \mathfrak{g}_k$, a guarantee of contract C_k, where $1 \leq k \leq N$. For each $g \in \mathtt{CM}(g_v)$, we evaluate from the trace `Log` whether it is satisfied or violated. If g_k is violated, we have two different cases: i) if the assumptions $\mathfrak{a}_k$ of contract C_k, are satisfied, then M_k is added to the list of responsible components; in case ii), if the assumptions of C_k are violated, from the composition operation in Pacti, we can identify which component-level terms were used to refine this assumption and identify which terms to evaluate next. In the worst case, this procedure requires checking all component assumptions and guarantees to find the component that did not deliver its guarantees under satisfied assumptions. By definition of assume-guarantee contracts, an implementation of a contract where the assumptions are satisfied and whose guarantees are violated is faulty. Any component in the analysis that violated its guarantees under satisfied assumptions is faulty. □

5.4 Example: Autonomous Vehicle at Intersection

The following examples were inspired by Alice, Caltech's entry in the 2007 DARPA Urban Challenge [33]. While conducting the pre-challenge testing campaign, Alice faced scenarios where it failed to accomplish its objective because of an unforeseen behavior arising from the interaction of various subsystems in that particular situation. In this section, we will illustrate scenarios that are loosely based on the real-world scenarios that Alice faced in the DARPA Urban Challenge, which are described in detail in [33]. We will characterize simplified specifications for several of Alice's subsystems and demonstrate how to diagnose the cause of the failure from the violation when a system trace is given.

Case 1: Alice at a 4-Way Intersection

In this example, the test was set up such that Alice was approaching an intersection with multiple cars already waiting at the intersection as shown in Figure 5.11a. While Alice was approaching, its sensors detected the other cars in the intersection and commanded Alice to stop and give way to the other cars. The unforeseen circumstance was that the deceleration tilted the LADARs forward and towards the ground such that Alice lost sight of the other cars momentarily. Once Alice came to a full stop, the line of sight of the LADARs tilted back up and detected the cars again, but now Alice was under the impression that the cars just arrived, leading to the control system commanding Alice to drive into the intersection and leading to unsafe behavior. We model the components in Alice's control architecture as shown in Figure 5.11b. As a real-world observation, we define the variable n_T, which captures whether there are other cars in the intersection, that Alice would have to give way to. For $n_T = 1$ other cars are present, for $n_T = 0$, the intersection is clear. We model the contract for the perception subsystem as follows: $C_{\text{perception}} = (\{n_T\}, \{n_P\}, \{n_T \leq 1, n_T \geq 0\}, \{n_P \geq n_T, n_P \leq n_T\})$, where n_P is the perceived state of the intersection. In particular, this contract ensures that the perceived state of the intersection is the same as the actual state of the intersection, $n_T = n_P$. The planner subsystem contract is given as $C_{\text{planner}} = (\{n_P\}, \{\text{turn}\}, \{n_P \leq 1, n_P \geq 0\}, \{\text{turn} + n_P <= 1\})$. The variable turn encodes whether it is Alice's turn in the intersection. This contract ensures that Alice will only take its turn, when the perceived state of the intersection is clear. The tracker component controls Alice's speed v, it is given as $C_{\text{tracker}} = (\{\text{turn}\}, \{v\}, \{\}, \{v \leq \text{turn}, v \geq \text{turn}\})$, and ensures that the speed is 1 when it's Alice's turn, and it is 0 otherwise. Next we describe the safety component, this component maps Alice's behavior to a variable denoted by 'safe', which captures whether the system is safe. The safety component is captured in the following contract, $C_{\text{safety}} = (\{n_T, v\}, \{\text{safe}\}, \{n_T \geq 0\}, \{1 - n_T - v \leq \text{safe}, 1 - n_T - v \geq \text{safe}\})$. Depending on the assignment of the variables, safe can take on three values, safe $= -1$ corresponds to the case of Alice having a nonzero speed and the intersection being occupied—an unsafe situation; safe $= 0$ corresponds to Alice stopping at the intersection which is occupied—correct safe behavior; and safe $= 1$, which corresponds to Alice waiting at an empty intersection—safe but incorrect behavior. In this example, we are only interested in preventing unsafe behavior, thus we only want to exclude behavior where safe $= -1$.

The composition order is given as $\texttt{CompOrd} = [C_{\text{perception}}, C_{\text{planner}}, C_{\text{tracker}}, C_{\text{safety}}]$. The composed system-level contract is $C = (\{n_T\}, \{\text{safe}\}, \{n_T \geq 0, n_T \leq 1\}, \{\text{safe} \geq 0\})$, and the diagnostics graph is shown in Figure 5.12. As we can see, if Alice behaves correctly, we see that 'safe' is guaranteed to be zero or positive, ensuring that Alice will not drive into an occupied intersection and possibly create an unsafe scenario.

We are now given a violating trace, where $n_T = 1$ and safe $= -1$. The system level guarantee safe ≥ 0 is violated. The diagnostics map returns the following tuples:

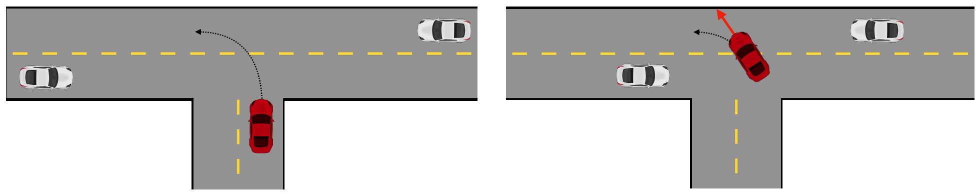
$$\begin{aligned}\texttt{CM}(\text{safe} \geq 0) = \{&(1 - n_T - v \leq \text{safe}, C_{\text{safety}}),\\ &(v \leq \text{turn}, C_{\text{tracker}})\\ &(n_P + \text{turn} \leq 1, C_{\text{planner}})\\ &(n_T \leq n_P, C_{\text{perception}})\},\end{aligned}$$

which is illustrated by the red edges in Figure 5.12.

Given the remaining log data containing the internal variables, we can now follow the diagnostics procedure outlined in Algorithm 5.2. First, we check the guarantee of the safety component. From the log data we get that $v = 1$, therefore we see that $1 - n_T - v \leq$ safe is satisfied. Next, we evaluate the tracker component. From the log data we see that turn $= 1$, and the guarantee $v \leq$ turn is satisfied, as $1 \leq 1$. We move on to the planner component, whose guarantee $n_P +$ turn ≤ 1 is satisfied, as $n_P = 0$. Finally, we evaluate the guarantee $n_T \leq n_P$ from the perception component, which turns out to be violated, as $1 \nleq 0$. Checking the assumptions of the perception component shows that they are satisfied, $0 \leq n_T \leq 1$, for $n_T = 1$. This results in the perception component being responsible for the failure, as the perceived state of the intersection was clear, but the real state of the intersection was occupied.

Case 2: Alice at T-Intersection

In this example, Alice approaches a T-intersection and is supposed to make a left turn into oncoming traffic. This situation requires Alice to decide on whether the



(a) Intersection layout and planned path. (b) Alice's final position.

Figure 5.13: Layout and failure of Alice's T-intersection crossing test.

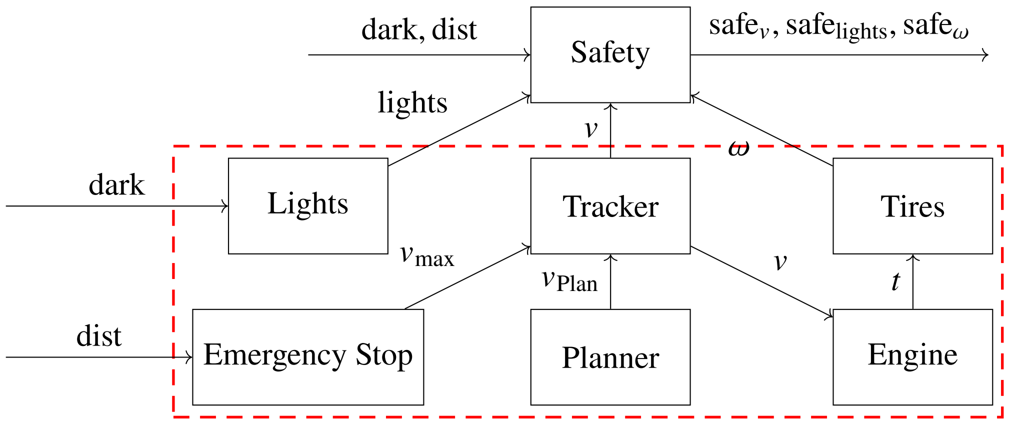


Figure 5.14: Block diagram for Case 2, Alice's components are enclosed by the red dashed outline.

gap in the oncoming traffic is sufficient to finish the left turn while staying clear of the approaching vehicles. Alice successfully identified a gap, but then during the execution of the turn, Alice's emergency braking system detected the concrete barrier ahead and initiated an emergency stop. This left Alice stranded in the middle of the intersection leading to an unsafe traffic scenario. The layout of the intersection and Alice's final position are illustrated in Figure 5.13. We will now abstract the components that made up Alice's path planning and tracking modules and use the known vehicle trace to identify the cause of the sudden stopping maneuver. In this analysis we will add an additional component that controls the lights, which was not part of Alice but is added to show how to different functionalities can be included. A block diagram of Alice's components and the safety component is shown in Figure 5.14. Alice's components are shown in the red dashed box, and comprise the planner, tracker, emergency stop, engine, tires, and lights. The composition order is $\texttt{CompOrd} = [C_{\text{planner}}, C_{\text{emergency}}, C_{\text{tracker}}, C_{\text{engine}}, C_{\text{tires}}, C_{\text{lights}}, C_{\text{safety}}]$. The planner component adheres to the following contract,

$$C_{\text{planner}} = (\{\}, \{v_{\text{max}}\}, \{\}, \{v_{\text{max}} \leq 1, v_{\text{max}} \geq 1\}),$$

where v_{max} is the maximum speed allowed, which is set by the planning component to be 1. The emergency stop component is given as

$$C_{\text{emergency}} = (\{\text{dist}\}, \{v_{\text{safe}}\}, \{\text{dist} \geq 0, \text{dist} \leq 1\}, \{\text{dist} \leq v_{\text{safe}}, \text{dist} \geq v_{\text{safe}}\}),$$

where dist is the distance to an obstacle, ranging from 0 (too close) to 1 (far enough away) and v_{safe} is the allowed safe speed. This contract ensures that if an obstacle is too close, dist = 0, the allowed speed is $v_{\text{safe}} = 0$, otherwise v_{safe} is set to 1. The tracker component contract is

$$\begin{aligned} C_{\text{tracker}} = (&\{v_{\text{safe}}, v_{\text{max}}\}, \{v\},\\ &\{v_{\text{max}} \geq 0, v_{\text{max}} \leq 1, v_{\text{safe}} \geq 0, v_{\text{safe}} \leq 1\},\\ &\{v \leq v_{\text{max}}, v \leq v_{\text{safe}}, v \geq v_{\text{safe}}\}), \end{aligned}$$

where v is Alice's speed that is commanded by the tracker. The tracker component guarantees that the speed v is less than the maximum speed, and is also set to the safe speed. The contract for the engine,

$$C_{\text{engine}} = (\{v\}, \{t\}, \{v \leq 1\}, \{t \leq 3v, t \geq 3v\}),$$

states that the torque t is proportional to the commanded speed. The contract describing the tire component behavior is

$$C_{\text{tires}} = (\{t\}, \{\omega\}, \{\}, \{\omega \leq 100, \omega \geq 2t\}),$$

where ω is the rate of tire revolutions, which is proportional to the engine torque t and upper bounded by 100. The following component contract,

$$\begin{aligned} C_{\text{lights}} = (&\{\text{dark}\}, \{\text{lights}\},\\ &\{\text{dark} \geq 0, \text{dark} \leq 1\}, \{\text{dark} \leq \text{lights}, \text{dark} \geq \text{lights}\}), \end{aligned}$$

describes the requirements on Alice's light system. the illumination conditions are captured in the variable dark, which ranges from daytime (dark = 0) to nighttime (dark = 1). Alice is expected to operate in any illumination conditions. Furthermore, the contract guarantees that the Alice's lights have to be on when it is dark (i.e. dark = 1 requires lights = 1), and need to be turned off during daytime (i.e. dark = 0 requires lights = 0). The safety layer is captured in the following contract:

$$\begin{aligned} C_{\text{safety}} = (&\{\text{dark}, \text{dist}, \text{lights}, v, \omega\}, \{\text{safe}_v, \text{safe}_{\text{lights}}, \text{safe}_\omega\}, \{v \leq 10\},\\ &\{\text{safe}_v \geq v - \text{dist} + 1, \text{safe}_{\text{lights}} \geq \text{dark} - \text{lights} + 1,\\ &\text{safe}_\omega \geq 100 - \omega + 1\}). \end{aligned}$$

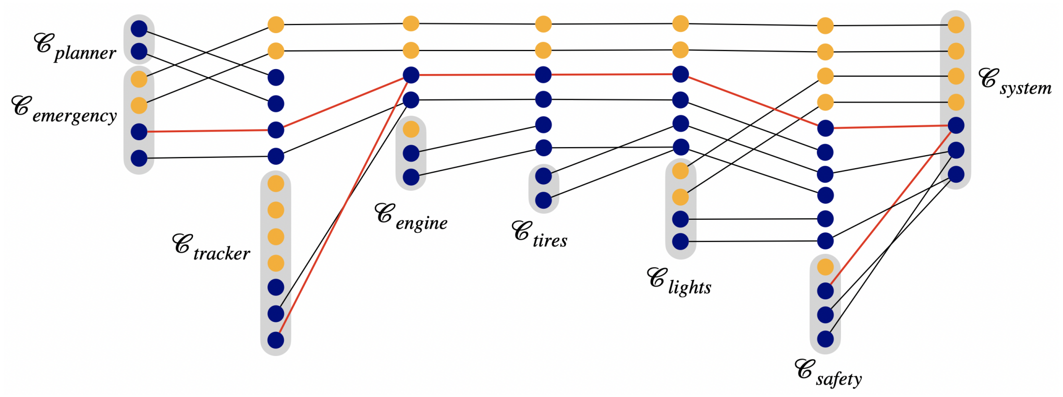


Figure 5.15: Diagnostics graph for Alice at T-Intersection Example.

The safety contract inputs are the variables dark, dist, lights, v, and ω, and the outputs are the safety flags: $\text{safe}_{\text{lights}}$, safe_v, and safe_ω. Each safety flag captures the desired safe behavior, such that the composition using Pacti will result in a bound on the safety variables that can be used for analysis.

Composing Alice's subsystems together with the safety layer results in the following system-level contract $C = (\{\text{dist}, \text{dark}\}, \{\text{safe}_v, \text{safe}_{\text{lights}}, \text{safe}_\omega\}, \{\text{dist} \geq 0, \text{dist} \leq 1, \text{dark} \geq 0, \text{dark} \leq 1\}, \{\text{safe}_v \geq 1, \text{safe}_{\text{lights}} \geq 1, \text{safe}_\omega \geq 1\})$. As expected, Pacti was able to reduce Alice's behavior to the desired safety flags: safe_v, $\text{safe}_{\text{lights}}$, and safe_ω. For this example, a safety variable of 1 or greater implies that the system is safe with respect to that variable, while a safety variable of 0 corresponds a violation. The guarantees on the safety variables were derived from Alice's correct behavior, i.e. when driving in the 'dark', Alice is expected to turn on its 'lights', this behavior is captured by $\text{safe}_{\text{lights}} \geq 1$. In this example, we specifically observed that Alice came to a full stop in the intersection. This can be observed as $\text{safe}_v = 0$ — a violated system-level guarantee. The system-level assumptions were satisfied, as the test was executed during daytime, dark = 0, and the path was clear, dist = 1. From the diagnostics map, we trace back the violated system-level guarantee and find

$$\begin{aligned}\texttt{CM}(\text{safe}_v = 0) = \{&(\text{safe}_v \geq v - \text{dist} + 1, C_{\text{safety}}),\\ &(\text{dist} \leq v_{\text{safe}}, C_{\text{emergency}}),\\ &(v_{\text{safe}} \leq v, C_{\text{tracker}})\}.\end{aligned}$$

We can now focus our attention on the safety and emergency stop, and tracker components and evaluate their behavior. First we evaluate the safety component. The relevant variable assignments from the violating trace are $\text{safe}_v = 0$, $v = 0$, and dist = 1. We check that the safety component correctly evaluated the safety flag, $\text{safe}_v \geq v - \text{dist} + 1 = 0 - 1 + 1 = 0$, which is satisfied. Next we evaluate the emergency stop component. From the violating trace we learn that $v_{\text{safe}} = 0$. Therefore the guarantee $\text{dist} \leq v_{\text{safe}}$ is violated, as $1 \nleq 0$. On the other hand, the assumptions of the emergency stop component are satisfied, as $0 \leq \text{dist} \leq 1$, making the emergency stop component a cause of the failure. Lastly, for completeness, we evaluate the tracker component, the guarantee to check is $v_{\text{safe}} \leq v$, which is satisfied as $0 \leq 0$. Therefore we could isolate the failure to the emergency stop component. In Alice's case the distance sensor measured straight ahead and not along the driving path, leading to the stop in the intersection, even though the path is clear. It is important to note that analysis cannot explain why the failure happened, but it can help isolate the faulty component such that it can be further analyzed by experts.

5.5 Conclusion

In this chapter, we proposed a methodological approach to diagnose which component is responsible for a system-level failure using Pacti, a tool for compositional system analysis and design. We have characterized when a system-level guarantee failure can be traced to a component, defined how the components need to be composed, and which information needs to be stored to facilitate the diagnostics process. The framework presented in this chapter includes composing the system, creating a diagnostics map, and systematically checking component-level guarantees and assumptions using the log data to identify failed components. Applying this framework can reduce the number of statements that need to be evaluated, as the most likely culprits are identified and checked first. In the worst case, all component-level assumptions and guarantees need to be evaluated, ensuring that if a component did not satisfy its guarantees, it will be found. We illustrated the approach on abstract examples and two examples inspired by real-world test outcomes representing a simplified abstraction of a dynamical system.

Chapter 6

CONCLUSION AND FUTURE WORK

Test and evaluation of autonomous systems are critical to ensure their safe operation. To allow widespread deployment of these systems in the future, testing, as well as design of these systems need to be standardized. In this thesis we present novel approaches to reason over tests in the form of formal specifications, automatically synthesize test strategies, and diagnose system-level failures.

6.1 Reasoning over Tests

In Chapter 3, a framework building on assume-guarantee contracts was developed, that allows to characterize tests in the form of a test structure to allow reasoning over these test structures. We showed how to compare, combine, and split tests and illustrated this approach with examples. We identified under which conditions combined tests require temporal constraints to ensure that the combined test captures the information required to infer satisfaction of the individual unit tests. Additionally, we proposed how to find a strategy for test agents that ensures that the desired test behavior is observed and a pre-defined difficulty metric is maximized using a winning-set-based approach together with Monte Carlo Tree Search.

6.2 Flow-Based Test Synthesis

In Chapter 4, we propose a framework for automated test environment construction, including placement of static and reactive obstacles, as well as finding the strategies for dynamics test agents. We characterize the desired test behavior in the form of a system objective and a test objective, where the test objective is unknown to the system. From the system model and the system and test objectives, we construct the virtual product graph and the system product graph. These graphs allow reasoning over the test executions from the perspective of the entire test and the system perspective. Next, we formulated a network flow optimization as a MILP that captures the requirements that every test execution where the system behaves according to its system objective, will also satisfy the test objective. It is important to note that this framework does not help the system achieve its goals but ensures that at any point during the test execution, a correct system will be able to make progress towards its goal. We then propose a counterexample-guided approach that makes use of the MILP and GR(1) synthesis to find the strategy for a dynamic test agent. Finally, the proposed approach was demonstrated in simulations and hardware experiments.

6.3 System Diagnostics using Assume-guarantee Contracts

In Chapter 5, we propose a methodology to identify component-level failures from a violated system-level behavior and the failing trace of the system. We augment the functionality of Pacti, a tool for compositional system design, to find the diagnostics graph for a composed system. We develop an algorithm that makes use of this graph to trace the failure back to candidate component assumptions and guarantees. This allows us to pinpoint the important information required to identify the component culpable for the failed system-level trace. We show that if the violating trace contains the valuations of the entire system state, this framework will identify the failed component. In the worst case, this corresponds to checking the assumptions and guarantees of all components. However, by identifying the relevant information this framework focuses on the most likely components first and can result in evaluating fewer component-level statements. We illustrate the framework on abstract examples and two examples inspired by a real-world autonomous system test.

6.4 Future Work

The approaches proposed in this thesis all aim to improve the efficiency of the testing process. They are designed to be applied in conjunction with other testing techniques and each other. Given a given list of interesting tests created by test engineers or other scenario generation frameworks, the approaches in this thesis could provide an efficient way to find and execute a testing campaign. We imagine a testing framework that combines the approaches in this thesis to find a test campaign with the corresponding test environment and required instrumentation to diagnose any violations. In the following paragraphs, we will outline the proposed improvements and envisioned synergies for each framework.

Reasoning over Tests

The current approach to reason over tests allows for a systematic combining and splitting of test structures but still requires domain knowledge to refine the test structures according to the desired test behavior. One interesting application for these operations would be test compression and testing for coverage. To enable this, we would require an automated approach including accessing domain knowledge, possibly in the form of a formal set of rules, to determine which tests to combine. Test compression could then return a test campaign that is optimal according to user-defined metrics, such as test effort, test time, or cost. Testing for coverage requires defining the type of coverage that we want to observe, be it in the form of different test structures, or different ways to execute tests for the same test structure. With this approach, one could reactively choose the next tests, depending on what was observed during the test campaign so far. Additionally, applying techniques such as symbolic reasoning could allow scaling up this framework to handle test structures with larger contracts and test campaigns consisting of a larger set of test structures. Finally, combining this framework with test synthesis, or scenario generation could close the loop from unit tests in the form of test specifications to finding a test campaign with corresponding test strategies for the system under test.

Flow-Based Test Synthesis

In this thesis, we presented how to find the strategy for a given dynamic test agent. One approach to generalize this framework would be defining a tester library, from which the synthesis procedure can then select which test agents to deploy. The selection could account for test effort, and return a test agent configuration that is optimal according to a test effort metric, which could be the cost of each agent, test time, or number of agents. As the test agents are controlled by the test engineers, it is reasonable to assume that they can communicate with each other. Thus, their centralized controller can be found using coordinated synthesis, reducing the complexity of coordination between multiple agents. In the current framework, the information about the test agent in the MILP is limited to the areas that the agent can occupy. We could investigate, whether we can capture the dynamics of the agent in the optimization constraints to reduce the number of counterexamples required to find a realizable solution. Additionally, we would like to apply this framework to applications other than ground-operating robots, from robots flying in 3D space to systems that do not physically move such as power grids. To apply this framework, we need to have a system model in the form of a transition system and the ability to block system transitions statically or reactively. Depending on the application this might require different auxiliary constraints that represent how restricting a system transition might affect other transitions as well. The MILP performs well for medium-sized problems, but for larger problems, the runtime increases significantly. For future work, we would like to investigate whether there is a tight convex relaxation for this MILP that results improves the run times and results in the desired solutions. The time required for the construction of the virtual product graph also depends on the size of the problem and can become significant. Techniques such as symbolic reasoning could improve the run times of the proposed framework.

System Diagnostics

The proposed framework allows the tracing of a violated system-level guarantee to the responsible component. Inherently, there is a connection to explainability of robot behaviors, which is an active area of research. This framework uses the contract composition operator for diagnostics. A similar approach could be employed using the contract quotient, which finds missing components in a system. Adapting the implemented functions to trace the terms resulting from a quotient could allow finding missing components and provide insight into the origin of certain terms, aiding in explainability.

The procedure used to trace the violated system-level guarantee could prove beneficial to testing and not just diagnostics. One could imagine an approach for *testing for diagnostics*, where the test is set up in such a way as to facilitate any required system diagnostics. Specifically, from a system-level composition, we can find the relevant terms for each system-level guarantee, and the corresponding sensor location for testing a specific guarantee. In the case of a violation, the resulting test could then efficiently lead to a diagnosis.

In particular, we would like to investigate how to automate the framework proposed in Chapter 3. Instead of tracing a system-level guarantee back to the components, we could use this approach to check whether a test combination is allowed. For a combined test, we can find the combined test guarantees and determine whether we need to enforce temporal constraints by checking what terms each guarantee depends on. In its current form, Pacti only supports linear inequalities. Therefore, tests would need to be specified in linear inequalities, or a temporal logic would need to be implemented in Pacti. Another benefit of implementing propositional logic or temporal logic in Pacti is that it would facilitate specifying the component contracts for digital systems in the proposed diagnostics framework.

BIBLIOGRAPHY

[1] Houssam Abbas and Georgios Fainekos. "Linear hybrid system falsification through local search". In: *International Symposium on Automated Technology for Verification and Analysis*. Springer. 2011, pp. 503–510.

[2] Wasif Afzal, Richard Torkar, and Robert Feldt. "A systematic review of search-based testing for non-functional system properties". In: *Information and Software Technology* 51.6 (2009), pp. 957–976.

[3] Venkatesh Agaram, Frank Barickman, Felix Fahrenkrog, Edward Griffor, Ibro Muharemovic, Huei Peng, Jeremy Salinger, Steven Shladover, and William Shogren. "Validation and verification of automated road vehicles". In: *Road Vehicle Automation 3* (2016), pp. 201–210.

[4] Bernhard K Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. "Killing strategies for model-based mutation testing". In: *Software Testing, Verification and Reliability* 25.8 (2015), pp. 716–748.

[5] Ali Ajdari and Hashem Mahlooji. "An adaptive exploration-exploitation algorithm for constructing metamodels in random simulation using a novel sequential experimental design". In: *Communications in Statistics-Simulation and Computation* 43.5 (2014), pp. 947–968.

[6] Rajeev Alur, Costas Courcoubetis, and Mihalis Yannakakis. "Distinguishing tests for nondeterministic and probabilistic machines". In: *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*. 1995, pp. 363–372.

[7] Paul E Ammann, Paul E Black, and William Majurski. "Using model checking to generate tests from specifications". In: *Proceedings second international conference on formal engineering methods (Cat. No. 98EX241)*. IEEE. 1998, pp. 46–54.

[8] James M Anderson, Kalra Nidhi, Karlyn D Stanley, Paul Sorensen, Constantine Samaras, and Oluwatobi A Oluwatola. *Autonomous vehicle technology: A guide for policymakers*. Rand Corporation, 2014.

[9] Anneliese Andrews, Mahmoud Abdelgawad, and Ahmed Gario. "World model for testing urban search and rescue (USAR) robots using petri nets". In: *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. IEEE. 2016, pp. 663–670.

[10] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram 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*. Springer. 2011, pp. 254–257.

[11] Dejanira Araiza-Illan, Tony Pipe, and Kerstin Eder. “Model-based testing, using belief-desire-intentions agents, of control code for robots in collaborative human-robot interactions”. In: *arXiv preprint arXiv:1603.00656* (2016).

[12] *Automotive dynamics simulation | Applied Intuition*. URL: https://www.appliedintuition.com/products/carsim.

[13] Apurva Badithela, Josefine B Graebener, Inigo Incer, and Richard M Murray. “Reasoning over Test Specifications using Assume Guarantee Contracts”. In: *NASA Formal Methods: 15th International Symposium, NFM 2023, Houston, TX, USA, May 16–18, 2023, Proceedings*. 2023, tbd. URL: https://link.springer.com/chapter/10.1007/978-3-031-33170-1_17.

[14] Apurva Badithela, Josefine B Graebener, Wyatt Ubellacker, Eric V Mazumdar, Aaron D Ames, and Richard M Murray. “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)*. 2023, pp. 12430–12436. DOI: 10.1109/ICRA48891.2023.10160841. URL: https://ieeexplore.ieee.org/abstract/document/10160841.

[15] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[16] Ezio Bartocci, Roderick Bloem, Benedikt Maderbacher, Niveditha Manjunath, and Dejan Ničković. “Adaptive testing for specification coverage in CPS models”. In: *IFAC-PapersOnLine* 54.5 (2021), pp. 229–234.

[17] Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Ničković, and Sriram Sankaranarayanan. “Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications”. In: *Lectures on Runtime Verification: Introductory and Advanced Topics*. 2018.

[18] Andreas Bauer, Martin Leucker, and Christian Schallhart. “Runtime verification for LTL and TLTL”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20.4 (2011), pp. 1–64.

[19] Richard Vernon Beard. “Failure accomodation in linear systems through self-reorganization.” PhD thesis. Massachusetts Institute of Technology, 1971.

[20] Armin Beer and Rudolf Ramler. “The role of experience in software testing practice”. In: *2008 34th Euromicro Conference Software Engineering and Advanced Applications*. IEEE. 2008, pp. 258–265.

[21] Michael Behrisch, Laura Bieker, Jakob Erdmann, and Daniel Krajzewicz. “SUMO–simulation of urban mobility: an overview”. In: *Proceedings of SIMUL 2011, The Third International Conference on Advances in System Simulation*. ThinkMind. 2011.

[22] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. "Testing advanced driver assistance systems using multi-objective search and neural networks". In: *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 2016, pp. 63–74.

[23] Albert Benveniste, Benoît Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos 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*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 200–225. ISBN: 978-3-540-92188-2. DOI: 10.1007/978-3-540-92188-2_9. URL: https://doi.org/10.1007/978-3-540-92188-2%5C_9.

[24] Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas A Henzinger, Kim G Larsen, et al. "Contracts for system design". In: *Foundations and Trends® in Electronic Design Automation* 12.2-3 (2018), pp. 124–400.

[25] Kevin M Betts, Mikel D Petty, et al. "Automated search-based robustness testing for autonomous vehicle software". In: *Modelling and Simulation in Engineering* 2016 (2016).

[26] Andreas Blass, Yuri Gurevich, Lev Nachmanson, and Margus Veanes. "Play to test". In: *Formal Approaches to Software Testing: 5th International Workshop, FATES 2005, Edinburgh, UK, July 11, 2005, Revised Selected Papers 5*. Springer. 2006, pp. 32–46.

[27] Roderick Bloem, Goerschwin Fey, Fabian Greif, Robert Könighofer, Ingo Pill, Heinz Riener, and Franz Röck. "Synthesizing adaptive test strategies from temporal logic specifications". In: *Formal methods in system design* 55.2 (2019), pp. 103–135.

[28] Markus Borg, Cristofer Englund, Krzysztof Wnuk, Boris Duran, Christoffer Levandowski, Shenjian Gao, Yanwen Tan, Henrik Kaijser, Henrik Lönn, and Jonas Törnqvist. "Safely entering the deep: A review of verification and validation for machine learning and a challenge elicitation in the automotive industry". In: *arXiv preprint arXiv:1812.05389* (2018).

[29] Adrian Boteanu, Jacob Arkin, Siddharth Patki, Thomas Howard, and Hadas Kress-Gazit. "Robot-initiated specification repair through grounded language interaction". In: *arXiv preprint arXiv:1710.01417* (2017).

[30] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. "A survey of monte carlo tree search

methods". In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012), pp. 1–43.

[31] J. Richard Büchi. "On a Decision Method in Restricted Second Order Arithmetic". In: *The Collected Works of J. Richard Büchi*. New York, NY: Springer New York, 1990, pp. 425–435. ISBN: 978-1-4613-8928-6. DOI: 10.1007/978-1-4613-8928-6_23. URL: https://doi.org/10.1007/978-1-4613-8928-6_23.

[32] Martin Buehler, Karl Iagnemma, and Sanjiv Singh. *The DARPA urban challenge: autonomous vehicles in city traffic*. Vol. 56. Springer Science & Business Media, 2009.

[33] Joel W Burdick, Noel DuToit, Andrew Howard, Christian Looman, Jeremy Ma, Richard M Murray, and Tichakorn Wongpiromsarn. "Sensing, navigation and reasoning technologies for the DARPA Urban Challenge". In: *DARPA Urban Challenge Final Report, Tech. Rep* (2007).

[34] John Callahan, Francis Schneider, Steve Easterbrook, et al. "Automated software testing using model-checking". In: *Proceedings 1996 SPIN workshop*. Vol. 353. 1996.

[35] IPG CarMaker. "Users guide version 4.5. 2". In: *IPG Automotive, Karlsruhe, Germany* 1 (2014).

[36] Jie Chen and Ron J Patton. *Robust model-based fault diagnosis for dynamic systems*. Vol. 3. Springer Science & Business Media, 2012.

[37] Yuanfang Chi, Yanjie Dong, Z Jane Wang, F Richard Yu, and Victor CM Leung. "Knowledge-based fault diagnosis in industrial internet of things: a survey". In: *IEEE Internet of Things Journal* 9.15 (2022), pp. 12886–12900.

[38] Stephen A Cook. "The complexity of theorem-proving procedures". In: *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*. 2023, pp. 143–152.

[39] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[40] Anthony Corso, Peter Du, Katherine Driggs-Campbell, and Mykel J Kochenderfer. "Adaptive stress testing with reward augmentation for autonomous vehicle validatio". In: *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*. IEEE. 2019, pp. 163–168.

[41] Alexandre David, Kim G Larsen, Shuhao Li, and Brian Nielsen. "Cooperative testing of timed systems". In: *Electronic Notes in Theoretical Computer Science* 220.1 (2008), pp. 79–92.

[42] Giuseppe De Giacomo, Riccardo De Masellis, and Marco Montali. "Reasoning on LTL on finite traces: Insensitivity to infiniteness". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 28. 1. 2014.

[43] Giuseppe De Giacomo, Moshe Y Vardi, et al. “Linear Temporal Logic and Linear Dynamic Logic on Finite Traces.” In: *Ijcai*. Vol. 13. 2013, pp. 854–860.

[44] Johan De Kleer and Brian C Williams. “Diagnosing multiple faults”. In: *Artificial intelligence* 32.1 (1987), pp. 97–130.

[45] Edsger W Dijkstra. “Guarded commands, nondeterminacy and formal derivation of programs”. In: *Communications of the ACM* 18.8 (1975), pp. 453–457.

[46] Simulink Documentation. *Simulation and Model-Based Design*. 2020. URL: `https://www.mathworks.com/products/simulink.html`.

[47] Alexandre Donzé. “Breach, a toolbox for verification and parameter synthesis of hybrid systems”. In: *International Conference on Computer Aided Verification*. Springer. 2010, pp. 167–170.

[48] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. “CARLA: An open urban driving simulator”. In: *Conference on robot learning*. PMLR. 2017, pp. 1–16.

[49] John Eason and Selen Cremaschi. “Adaptive sequential sampling for surrogate model generation with artificial neural networks”. In: *Computers & Chemical Engineering* 68 (2014), pp. 220–232.

[50] André Engels, Loe Feijs, and Sjouke Mauw. “Test generation for intelligent networks using model checking”. In: *Tools and Algorithms for the Construction and Analysis of Systems: Third International Workshop, TACAS’97 Enschede, The Netherlands, April 2–4, 1997 Proceedings 3*. Springer. 1997, pp. 384–398.

[51] Jeff A Estefan et al. “Survey of model-based systems engineering (MBSE) methodologies”. In: *Incose MBSE Focus Group* 25.8 (2007), pp. 1–12.

[52] Daniel J Fagnant and Kara Kockelman. “Preparing a nation for autonomous vehicles: opportunities, barriers and policy recommendations”. In: *Transportation Research Part A: Policy and Practice* 77 (2015), pp. 167–181.

[53] Georgios E Fainekos and George J Pappas. “Robustness of temporal logic specifications for continuous-time signals”. In: *Theoretical Computer Science* 410.42 (2009), pp. 4262–4291.

[54] Francesca Favarò, Sky Eurich, and Nazanin Nader. “Autonomous vehicles’ disengagements: Trends, triggers, and regulatory limitations”. In: *Accident Analysis & Prevention* 110 (2018), pp. 136–148.

[55] Shuo Feng, Haowei Sun, Xintao Yan, Haojie Zhu, Zhengxia Zou, Shengyin Shen, and Henry X Liu. “Dense reinforcement learning for safety validation of autonomous vehicles”. In: *Nature* 615.7953 (2023), pp. 620–627.

[56] Lester R. Ford and Delbert R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962. ISBN: 9780691625393.

[57] Gordon Fraser and Paul Ammann. "Reachability and propagation for LTL requirements testing". In: *2008 The Eighth International Conference on Quality Software*. IEEE. 2008, pp. 189–198.

[58] Gordon Fraser and Franz 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*. 2007, pp. 64–74.

[59] Gordon Fraser, Franz Wotawa, and Paul E Ammann. "Testing with model checkers: a survey". In: *Software Testing, Verification and Reliability* 19.3 (2009), pp. 215–261.

[60] Daniel J Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L Sangiovanni-Vincentelli, and Sanjit 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*. 2019, pp. 63–78.

[61] Daniel J Fremont, Edward Kim, Yash Vardhan Pant, Sanjit A Seshia, Atul Acharya, Xantha Bruso, Paul Wells, Steve Lemke, Qiang Lu, and Shalin Mehta. "Formal scenario-based testing of autonomous vehicles: From simulation to the real world". In: *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*. IEEE. 2020, pp. 1–8.

[62] Alessio Gambi, Marc Müller, and Gordon Fraser. "Asfault: Testing self-driving car software using search-based procedural content generation". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE. 2019, pp. 27–30.

[63] Zhiwei Gao, Carlo Cecati, and Steven X Ding. "A survey of fault diagnosis and fault-tolerant techniques—Part I: Fault diagnosis with model-based and signal-based approaches". In: *IEEE transactions on industrial electronics* 62.6 (2015), pp. 3757–3767.

[64] Josefine B Graebener, Apurva Badithela, Denizalp Goktas, Wyatt Ubellacker, Eric V Mazumdar, Aaron D Ames, and Richard M Murray. "Flow-Based Synthesis of Reactive Tests for Discrete Decision-Making Systems with Temporal Logic Specifications". In: *arXiv preprint* (2024). arXiv: `2404.09888 [cs.FL]`. URL: `https://arxiv.org/abs/2404.09888`.

[65] Josefine B Graebener, Apurva Badithela, and Richard M Murray. "Towards Better Test Coverage: Merging Unit Tests for Autonomous Systems". In: *NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, Proceedings*. 2022, pp. 133–155. URL: `https://dl.acm.org/doi/abs/10.1007/978-3-031-06773-0_7`.

[66] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2023. URL: `https://www.gurobi.com`.

[67] Seana Hagerman, Anneliese Andrews, and Stephen Oakes. “Security testing of an unmanned aerial vehicle (UAV)”. In: *2016 Cybersecurity Symposium (CYBERSEC)*. IEEE. 2016, pp. 26–31.

[68] Robert Hammett. “Design by extrapolation: an evaluation of fault tolerant avionics”. In: *IEEE aerospace and electronic systems magazine* 17.4 (2002), pp. 17–25.

[69] Klaus Havelund and Grigore Rosu. “Monitoring programs using rewriting”. In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE. 2001, pp. 135–143.

[70] Philipp Helle, Wladimir Schamai, and Carsten Strobel. “Testing of autonomous systems–Challenges and current state-of-the-art”. In: *INCOSE international symposium*. Vol. 26. 1. Wiley Online Library. 2016, pp. 571–584.

[71] R.M. Hierons. “Applying adaptive test cases to nondeterministic implementations”. In: *Information Processing Letters* 98.2 (2006), pp. 56–60. ISSN: 0020-0190. DOI: `https://doi.org/10.1016/j.ipl.2005.12.001`. URL: `https://www.sciencedirect.com/science/article/pii/S0020019005003418`.

[72] Robert M Hierons, Kirill Bogdanov, Jonathan P Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, et al. “Using formal specifications to support testing”. In: *ACM Computing Surveys (CSUR)* 41.2 (2009), pp. 1–76.

[73] Gergő Horányi, Zoltán Micskei, and István Majzik. “Scenario-based automated evaluation of test traces of autonomous systems”. In: *SAFECOMP 2013-Workshop DECS (ERCIM/EWICS Workshop on Dependable Embedded and Cyber-physical Systems) of the 32nd international conference on computer safety, reliability and security*. 2013, NA.

[74] T Chiang Hu. “Multi-commodity network flows”. In: *Operations research* 11.3 (1963), pp. 344–360.

[75] David Husch and John Albeck. “Trafficware SYNCHRO 6 user guide”. In: *TrafficWare, Albany, California* 11 (2004).

[76] Rasheed Hussain and Sherali Zeadally. “Autonomous cars: Research results, issues, and future challenges”. In: *IEEE Communications Surveys & Tutorials* 21.2 (2018), pp. 1275–1313.

[77] The MathWorks Inc. *Statistics and machine learning toolbox*. Natick, Massachusetts, United States, 2022. URL: `https://www.mathworks.com/help/stats/index.html`.

[78] Inigo Incer. “The Algebra of Contracts”. PhD thesis. EECS Department, University of California, Berkeley, 2022.

[79] Inigo Incer, Apurva Badithela, Josefine Graebener, Piergiuseppe Mallozzi, Ayush Pandey, Sheng-Jung Yu, Albert Benveniste, Benoit Caillaud, Richard M Murray, Alberto Sangiovanni-Vincentelli, et al. “Pacti: Scaling assume-guarantee reasoning for system analysis and design”. In: *arXiv preprint arXiv:2303.17751* (2023).

[80] Inigo Incer, Albert Benveniste, Richard M Murray, Alberto Sangiovanni-Vincentelli, and Sanjit A Seshia. “Context-Aided Variable Elimination for Requirement Engineering”. In: *arXiv preprint arXiv:2305.17596* (2023).

[81] Inigo Incer, Leonardo Mangeruca, Tiziano Villa, and Alberto Sangiovanni-Vincentelli. “The quotient in preorder theories”. In: *arXiv:2009.10886* (2020).

[82] Inigo Incer, Alberto L. Sangiovanni-Vincentelli, Chung-Wei Lin, and Eunsuk Kang. “Quotient for Assume-Guarantee Contracts”. In: *16th ACM-IEEE International Conference on Formal Methods and Models for System Design*. MEMOCODE’18. Beijing, China, 2018, pp. 67–77. ISBN: 9781538661956. DOI: 10.1109/MEMCOD.2018.8556872.

[83] Craig Innes and Subramanian Ramamoorthy. “Automated Testing With Temporal Logic Specifications for Robotic Controllers Using Adaptive Experiment Design”. In: *2022 International Conference on Robotics and Automation (ICRA)*. 2022, pp. 6814–6821. DOI: 10.1109/ICRA46639.2022.9811579.

[84] International Standards Organization (ISO). *Road vehicles – Functional safety*. Norm. 2011.

[85] International Standards Organization (ISO). *Road vehicles – Safety of the intended functionality*. Norm. 2022.

[86] Antona-Makoshi Jacobo, Uchida Nobuyuki, Yamazaki Kunio, Ozawa Koichiro, Kitahara Eiichi, and Taniguchi Satoshi. “Development of a safety assurance process for autonomous vehicles in Japan”. In: *Proceedings of ESV Conference*. 2019.

[87] Andreas Junghanns, Jakob Mauss, and Mugur Tatar. “Testautomation based on Computer Chess Principles”. In: *7th international CTI symposium innovative automotive transmissions, Berlin*. 2008, pp. 2–3.

[88] Nidhi Kalra and Susan M. Paddock. *Driving to Safety: How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability?* Santa Monica, CA: RAND Corporation, 2016. DOI: 10.7249/RR1478.

[89] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. “Autoware on board: Enabling autonomous vehi-

cles with embedded systems”. In: *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE. 2018, pp. 287–296.

[90] OM Kirovskii and VA Gorelov. “Driver assistance systems: analysis, tests and the safety case. ISO 26262 and ISO PAS 21448”. In: *IOP Conference Series: Materials Science and Engineering*. Vol. 534. 1. IOP Publishing. 2019, p. 012019.

[91] Michael Kläs, Thomas Bauer, Andreas Dereani, Thomas Söderqvist, and Philipp Helle. “A large-scale technology evaluation study: effects of model-based analysis and testing”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. IEEE. 2015, pp. 119–128.

[92] Jack PC Kleijnen, Wim Van Beers, and Inneke Van Nieuwenhuyse. “Constrained optimization in expensive simulation: Novel approach”. In: *European journal of operational research* 202.1 (2010), pp. 164–174.

[93] Johannes Kloos, Tanvir Hussain, and Robert Eschbach. “Risk-Based Testing of Safety-Critical Embedded Systems Driven by Fault Tree Analysis”. In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. 2011, pp. 26–33. DOI: 10.1109/ICSTW.2011.90.

[94] Alessia Knauss, Jan Schröder, Christian Berger, and Henrik Eriksson. “Paving the roadway for safety of automated vehicles: An empirical study on testing challenges”. In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2017, pp. 1873–1880.

[95] Levente Kocsis and Csaba Szepesvári. “Bandit based monte-carlo planning”. In: *European conference on machine learning*. Springer. 2006, pp. 282–293.

[96] Nathan Koenig and Andrew Howard. “Design and use paradigms for gazebo, an open-source multi-robot simulator”. In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Vol. 3. IEEE. 2004, pp. 2149–2154.

[97] Philip Koopman and Michael Wagner. “Challenges in autonomous vehicle testing and validation”. In: *SAE International Journal of Transportation Safety* 4.1 (2016), pp. 15–24.

[98] Rick Kuhn, Raghu Kacker, Yu Lei, and Justin Hunter. “Combinatorial Software Testing”. In: *Computer* 42.8 (2009), pp. 94–96. DOI: 10.1109/MC.2009.253.

[99] Leslie Lamport. “win and sin: Predicate transformers for concurrency”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990), pp. 396–428.

[100] Li Li, Wu-Ling Huang, Yuehu Liu, Nan-Ning Zheng, and Fei-Yue Wang. “Intelligence testing for autonomous vehicles: A new approach”. In: *IEEE Transactions on Intelligent Vehicles* 1.2 (2016), pp. 158–166.

[101] Yihao Li, Jianbo Tao, and Franz Wotawa. "Ontology-based test generation for automated and autonomous driving functions". In: *Information and Software Technology* 117 (2020), p. 106200. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2019.106200. URL: https://www.sciencedirect.com/science/article/pii/S0950584918302271.

[102] Liangkai Liu, Sidi Lu, Ren Zhong, Baofu Wu, Yongtao Yao, Qingyang Zhang, and Weisong Shi. "Computing Systems for Autonomous Driving: State of the Art and Challenges". In: *IEEE Internet of Things Journal* 8.8 (2021), pp. 6469–6486. DOI: 10.1109/JIOT.2020.3043716.

[103] Piergiuseppe Mallozzi, Inigo Incer, Pierluigi Nuzzo, and Alberto Sangiovanni-Vincentelli. "Contract-Based Specification Refinement and Repair for Mission Planning". In: *2023 IEEE/ACM 11th International Conference on Formal Methods in Software Engineering (FormaliSE)*. IEEE. 2023, pp. 29–38.

[104] Shahar Maoz and Jan Oliver Ringert. "GR (1) synthesis for LTL specification patterns". In: *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 2015, pp. 96–106.

[105] Alexandre Maréchal and Michaël Périn. "Efficient elimination of redundancies in polyhedra by raytracing". In: *Verification, Model Checking, and Abstract Interpretation: 18th International Conference, VMCAI 2017, Paris, France, January 15–17, 2017, Proceedings 18*. Springer. 2017, pp. 367–385.

[106] Malte Mauritz, Falk Howar, and Andreas Rausch. "Assuring the Safety of Advanced Driver Assistance Systems Through a Combination of Simulation and Runtime Monitoring". In: vol. 9953. Oct. 2016, pp. 672–687. ISBN: 978-3-319-47168-6. DOI: 10.1007/978-3-319-47169-3_52.

[107] Bertrand Meyer. "Applying 'design by contract'". In: *Computer* 25.10 (1992), pp. 40–51.

[108] Christoph C Michael, Gary E McGraw, Michael A Schatz, and Curtis C Walton. "Genetic algorithms for dynamic test data generation". In: *Proceedings 12th IEEE International Conference Automated Software Engineering*. IEEE. 1997, pp. 307–308.

[109] Ali Mili, Bojan Cukic, Yan Liu, and Rahma Ben Ayed. "Towards the verification and validation of online learning adaptive systems". In: *Software Engineering with Computational Intelligence* (2003), pp. 173–203.

[110] Andreas Morgenstern, Manuel Gesell, and Klaus Schneider. "An asymptotically correct finite path semantics for LTL". In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer. 2012, pp. 304–319.

[111] Galen E Mullins, Paul G Stankiewicz, R Chad Hawthorne, and Satyandra K Gupta. “Adaptive generation of challenging scenarios for testing and evaluation of autonomous vehicles”. In: *Journal of Systems and Software* 137 (2018), pp. 197–215.

[112] Radu Negulescu. “Process Spaces”. In: *CONCUR 2000 — Concurrency Theory*. Ed. by Catuscia Palamidessi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 199–213. ISBN: 978-3-540-44618-7.

[113] Cu D Nguyen, Simon Miles, Anna Perini, Paolo Tonella, Mark Harman, and Michael Luck. “Evolutionary testing of autonomous software agents”. In: *Autonomous Agents and Multi-Agent Systems* 25 (2012), pp. 260–283.

[114] Henrik Niemann. “A setup for active fault diagnosis”. In: *IEEE Transactions on Automatic Control* 51.9 (2006), pp. 1572–1578.

[115] Pierluigi Nuzzo, Alberto L Sangiovanni-Vincentelli, Davide Bresolin, Luca Geretti, and Tiziano Villa. “A platform-based design methodology with contracts and related tools for the design of cyber-physical systems”. In: *Proceedings of the IEEE* 103.11 (2015), pp. 2104–2132.

[116] ASAM OpenSCENARIO. “ASAM OpenSCENARIO: User Guide [EB/OL]”. In: (2022).

[117] Christos H Papadimitriou. “Computational complexity”. In: *Encyclopedia of computer science*. 2003, pp. 260–265.

[118] Roberto Passerone, Inigo Incer, and Alberto L Sangiovanni-Vincentelli. “Coherent extension, composition, and merging operators in contract models for system design”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 18.5s (2019), pp. 1–23.

[119] Michael Paulweber. “Validation of highly automated safe and secure systems”. In: *Automated driving: Safer and more efficient future driving* (2017), pp. 437–450.

[120] Doron Peled, Moshe Y Vardi, and Mihalis Yannakakis. “Black box checking”. In: *International Conference on Protocol Specification, Testing and Verification*. Springer. 1999, pp. 225–240.

[121] Alexandre Petrenko and Nina Yevtushenko. “Adaptive testing of nondeterministic systems with FSM”. In: *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*. IEEE. 2014, pp. 224–228.

[122] Nir Piterman, Amir Pnueli, and Yaniv Sa’ar. “Synthesis of reactive (1) designs”. In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer. 2006, pp. 364–380.

[123] Erion Plaku, Lydia E Kavraki, and Moshe Y Vardi. “Falsification of LTL safety properties in hybrid systems”. In: *International Journal on Software Tools for Technology Transfer* 15.4 (2013), pp. 305–320.

[124] Amir Pnueli. “The temporal logic of programs”. In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. IEEE. 1977, pp. 46–57.

[125] Amir Pnueli and Roni Rosner. “On the synthesis of a reactive module”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, pp. 179–190.

[126] Jeremie Pouly and Sylvain Jouanneau. “Model-based specification of the flight software of an autonomous satellite”. In: *Embedded Real Time Software and Systems (ERTS2012)*. 2012.

[127] V Dimitra Pyrialakou, Christos Gkartzonikas, J Drew Gatlin, and Konstantina Gkritza. “Perceptions of safety on a shared road: Driving, cycling, or walking near an autonomous vehicle”. In: *Journal of safety research* 72 (2020), pp. 249–258.

[128] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. “ROS: an open-source Robot Operating System”. In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[129] Vasumathi Raman and Hadas Kress-Gazit. “Explaining impossible high-level robot behaviors”. In: *IEEE Transactions on Robotics* 29.1 (2012), pp. 94–104.

[130] Qing Rao and Jelena Frtunikj. “Deep learning for self-driving cars: Chances and challenges”. In: *Proceedings of the 1st international workshop on software engineering for AI in autonomous systems*. 2018, pp. 35–38.

[131] Raymond Reiter. “A theory of diagnosis from first principles”. In: *Artificial intelligence* 32.1 (1987), pp. 57–95.

[132] Stefan Riedmaier, Thomas Ponn, Dieter Ludwig, Bernhard Schick, and Frank Diermeyer. “Survey on scenario-based safety assessment of automated vehicles”. In: *IEEE access* 8 (2020), pp. 87456–87477.

[133] SAE-J3016. *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. Tech. rep. SAE International, 2021.

[134] Francesca Saglietti and Matthias Meitner. “Model-driven structural and statistical testing of robot cooperation and reconfiguration”. In: *Proceedings of the 3rd Workshop on Model-Driven Robot Software Engineering*. 2016, pp. 17–23.

[135] Alberto L. Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. “Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems”. In: *Eur. J. Control* 18.3 (2012), pp. 217–238. DOI: `10.3166/ejc.18.217-238`. URL: `https://doi.org/10.3166/ejc.18.217-238`.

[136] Dirk van Schrick. "Remarks on terminology in the field of supervision, fault detection and diagnosis". In: *IFAC Proceedings Volumes* 30.18 (1997), pp. 959–964.

[137] Alan C Schultz, John J Grefenstette, and Kenneth A De Jong. "Test and evaluation by genetic algorithms". In: *IEEE expert* 8.5 (1993), pp. 9–14.

[138] Sanjit A Seshia, Dorsa Sadigh, and S Shankar Sastry. "Towards verified artificial intelligence". In: *arXiv preprint arXiv:1606.08514* (2016).

[139] Azim Shariff, Jean-François Bonnefon, and Iyad Rahwan. "Psychological roadblocks to the adoption of self-driving vehicles". In: *Nature Human Behaviour* 1.10 (2017), pp. 694–696.

[140] Santokh Singh. *Critical reasons for crashes investigated in the national motor vehicle crash causation survey*. Tech. rep. 2015.

[141] Tong Duy Son, Ajinkya Bhave, and Herman Van der Auweraer. "Simulation-based testing framework for autonomous driving development". In: *2019 IEEE International Conference on Mechatronics (ICM)*. Vol. 1. IEEE. 2019, pp. 576–583.

[142] Qunying Song, Emelie Engström, and Per Runeson. "Concepts in testing of autonomous systems: Academic literature and industry practice". In: *2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN)*. IEEE. 2021, pp. 74–81.

[143] Cameron Stark, Christopher Medrano-Berumen, and Mustafa İlhan Akbaş. "Generation of Autonomous Vehicle Validation Scenarios Using Crash Data". In: *2020 SoutheastCon*. 2020, pp. 1–6.

[144] Kaushik Subramanian, Jonathan Scholz, Charles L Isbell, and Andrea L Thomaz. "Efficient exploration in Monte Carlo tree search using human action abstractions". In: *Proceedings of the 30th international conference on neural information processing systems, NIPS*. Vol. 16. 2016.

[145] Gregory S Tallant, James M Buffington, Walter A Storm, Peter O Stanfill, and Bruce H Krogh. "Validation & verification for emerging avionic systems". In: *National Workshop on aviation software systems: Design for certifiably dependable systems. NITRD. Retrieved February*. Vol. 25. 2006, p. 2022.

[146] Li Tan, Oleg Sokolsky, and Insup Lee. "Specification-based testing with linear temporal logic". In: *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, 2004. IRI 2004.* IEEE. 2004, pp. 493–498.

[147] Jan Telgen. "Identifying redundant constraints and implicit equalities in systems of linear constraints". In: *Management Science* 29.10 (1983), pp. 1209–1222.

[148] Jan Tretmans. “Conformance testing with labelled transition systems: Implementation relations and test generation”. In: *Computer Networks and ISDN Systems* 29.1 (1996), pp. 49–79.

[149] Cumhur Erkan Tuncali, Georgios Fainekos, Hisahiro Ito, and James Kapinski. “Simulation-Based Adversarial Test Generation for Autonomous Vehicles with Machine Learning Components”. In: *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2018, pp. 1555–1562.

[150] Wyatt Ubellacker and Aaron D. Ames. “Robust Locomotion on Legged Robots through Planning on Motion Primitive Graphs”. In: *2023 IEEE International Conference on Robotics and Automation (ICRA), preprint*. 2023.

[151] Simon Ulbrich, Till Menzel, Andreas Reschka, Fabian Schuldt, and Markus Maurer. “Defining and substantiating the terms scene, situation, and scenario for automated driving”. In: *2015 IEEE 18th international conference on intelligent transportation systems*. IEEE. 2015, pp. 982–988.

[152] Vijay V Vazirani. *Approximation algorithms*. Vol. 1. Springer, 2001.

[153] Trent Victor, Kristofer Kusano, Tilia Gode, Ruoshu Chen, and Matthew Schwall. “Safety performance of the waymo rider-only automated driving system at one million miles”. In: *Waymo LLC* (2023).

[154] Hong Wang, Amir Khajepour, Dongpu Cao, and Teng Liu. “Ethical Decision Making in Autonomous Vehicles: Challenges and Research Progress”. In: *IEEE Intelligent Transportation Systems Magazine* 14.1 (2022), pp. 6–17. DOI: `10.1109/MITS.2019.2953556`.

[155] Waymo. *Waymo Safety Report*. `https://waymo.community/resources/waymo-safety-report-2021.html/`, Last accessed on 2024-04-13. 2021.

[156] Nick Webb, Dan Smith, Christopher Ludwick, Trent Victor, Qi Hommes, Francesca Favaro, George Ivanov, and Tom Daniel. *Waymo’s Safety Methodologies and Safety Readiness Determinations*. 2020. arXiv: `2011.00054 [cs.RO]`.

[157] Hermann Winner, Karsten Lemmer, Thomas Form, and Jens Mazzega. “PEGASUS—First steps for the safe introduction of automated driving”. In: *Road Vehicle Automation 5*. Springer. 2019, pp. 185–195.

[158] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M Murray. “Receding horizon temporal logic planning”. In: *IEEE Transactions on Automatic Control* 57.11 (2012), pp. 2817–2830.

[159] Tichakorn Wongpiromsarn, Ufuk Topcu, Necmiye Ozay, Huan Xu, and Richard 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*. 2011, pp. 313–314.

[160] Mihalis Yannakakis. “Testing, optimization, and games”. In: *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.* IEEE. 2004, pp. 78–88.

[161] Jin Zhang and Jingyue Li. “Testing and verification of neural-network-based safety-critical control software: A systematic literature review”. In: *Information and Software Technology* 123 (2020), p. 106296.

[162] Zoox. *Putting Zoox to the test: preparing for the challenges of the road*. `https://zoox.com/journal/structured-testing/`, Last accessed on 2024-04-13. 2023.

Appendix A

TEST ENVIRONMENT SYNTHESIS

A.1 Experiment Run Times and Implementation Details

In this section, we show the implementation details and run times of for the experiments in Section 4.9. The MILP callback function used in the experiments was such that the MILP terminates if the best objective has not improved for 5 minutes.

Reactive and Static Test Environments

Table A.1 shows the sizes of the specification product, the transition system, and the virtual product graph for the experiments illustrated in Section 4.9. The run times for the construction of the virtual product graph, and the time to solve the optimization can be found in Table A.2. The experiments were conducted on a M2 Mac Pro with 16GB of RAM. Table A.2 also shows the resulting flows and number of the cuts. Details on the number of binary and continuous variables are found in Table A.3.

Table A.1: Implementation details for simulated and hardware experiments

Experiment	$\lvert\mathcal{B}_\pi\rvert$	$\lvert T\rvert$	$\lvert G\rvert$
Exp. 4.2	(4, 9)	(15, 53)	(27, 96)
Exp. 4.5	(6, 18)	(265, 1047)	(332, 1346)
Exp. 4.6	(36, 354)	(376, 1146)	(4073, 17251)
Example 4.2	(8, 27)	(6, 17)	(20, 56)
Exp. 4.7	(12, 54)	(7, 19)	(15, 39)
Exp. 4.8	(16, 81)	(15, 42)	(72, 207)

Table A.2: Run times and results simulated and hardware experiments

Experiment	G[s]	Opt[s]	Flow	|cuts|
Exp. 4.2	0.0273	0.0152	3.0	14
Exp. 4.5	0.6205	0.0018	2.0	199
Exp. 4.6	77.6323	0.1716	2.0	1641
Example 4.2	0.0430	0.0001	2.0	4
Exp. 4.7	0.0532	0.0012	2.0	2
Exp. 4.8	0.4597	0.0005	3.0	15

Table A.3: Number of optimization variables for simulated and hardware experiments

Experiment	|BinVars|	|ContVars|	|Constraints|
Exp. 4.2	73	100	566
Exp. 4.5	1014	1346	19989
Exp. 4.6	13178	17251	1647774
Example 4.2	25	120	419
Exp. 4.7	8	159	451
Exp. 4.8	106	774	2632

Test Environments with Dynamic Test Agent

Table A.4 shows the sizes of the specification product, the transition system, and the virtual product graph for the experiments illustrated in Section 4.9. The run times for the construction of the virtual product graph, the time to solve the optimization, and the time required for the synthesis of the test agent controller the can be found in Table A.5. The experiments were conducted on a M2 Mac Pro with 16GB of RAM. Table A.5 also shows the resulting flows, the number of the cuts, and the number of the excluded solutions until a realizable solution was found. Details on the number of binary and continuous variables are found in Table A.6.

Table A.4: Implementation details for simulated and hardware experiments with dynamic agents

Experiment	$\lvert\mathcal{B}_{\text{sys}}\rvert$	$\lvert\mathcal{B}_{\text{test}}\rvert$	$\lvert\mathcal{B}_{\pi}\rvert$	$\lvert T\rvert$	$\lvert G\rvert$	$\lvert G_{\text{sys}}\rvert$
Exp. 4.9	(2,3)	(8,27)	(16, 81)	(26, 80)	(196, 604)	(26,80)
Exp. 4.10	(3,6)	(2,3)	(6, 18)	(386, 1153)	(210, 831)	(142,552)
Exp. 4.11	(2,3)	(4,9)	(8, 27)	(21, 66)	(80, 252)	(21,66)

Table A.5: Run times for simulated and hardware experiments with dynamic Agents

Experiment	G[s]	Opt[s]	Controller[s]	$\lvert\mathsf{C}_{\text{ex}}\rvert$	Flow	|cuts|
Exp. 4.9	1.6226	0.0010	100.0	4	1.0	3
Exp. 4.10	0.4573	6.0535	16.1191	0	1.0	13
Exp. 4.11	0.2195	0.0292	7.151	8	2.0	8

Table A.6: Number of optimization variables for simulated and hardware experiments with dynamic agents

Experiment	BinVars	ContVars	Constraints
Exp. 4.9	355	1327	5130
Exp. 4.10	621	6658	47885
Exp. 4.11	176	546	2438