Contract-based Design: Theories and Applications

Thesis by
Tung Phan-Minh

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

CALIFORNIA INSTITUTE OF TECHNOLOGY
Pasadena, California

2021
Defended December 18, 2020

Tung Phan-Minh
ORCID: 0000-0002-1403-5197

ACKNOWLEDGEMENTS

I am much obliged to Richard M. Murray, Joel W. Burdick, John C. Doyle, and K. Mani Chandy for serving on my thesis committee. I would like to thank Richard, my doctoral advisor, for teaching, inspiring, and generously supporting me in my quest to become a researcher. I am grateful to Prof. Burdick for his advice and encouragement during the early years of my PhD and Mani for his insights on the *Rules of the Road* work. It has been a great honor to learn from all of you.

This journey would have been much less enjoyable had it not been for my friends and collaborators. In particular, I would like to thank Ioannis Filippidis and Sumanth Dathathri for introducing me to "the world of LTL" and providing general advice/mentorship. I would like to thank Eric Wolff and the prediction team at Aptiv/nuTonomy (now Motional), especially Elena Corina Grigore, Freddy Boulton, for two fun and fruitful summer internships. It has also been my greatest pleasure to work with Karena Cai, Yuxiao Chen, Josefine Gräbener, Steve Guo, and Bastian Schürmann in the various projects that led to this dissertation.

I owe a lot of the decision to go seek a PhD to Art Moore and his way of convincing students of the beauty of mathematics and physics. I am beholden to the Jack Kent Cooke Foundation for enabling and generously providing financial support for my undergraduate and graduate studies. I also would like to convey my deep appreciation to the NSF VeHICaL program and DENSO North America for directly funding my research. Last but not least, I wish to thank the Mechanical and Civil Engineering Department staff, Claudia Andrade, and Monica Nolasco for their excellent organizational support throughout the years.

It goes without saying that I am forever indebted to my father Trịnh, my mother Thúy, my grandparents, uncles, aunts, siblings, cousins, my wife and her parents for their love and unconditional support.

ABSTRACT

Most things we know only exist in relation to one another. Their states are strongly coupled due to dependencies that arise from such relations. For a system designer, acknowledging the presence of these dependencies is as crucial to guaranteeing performance as studying them. As the roles played by technology in fields such as transportation, healthcare, and finance continue to be more profound and diverse, modern engineering systems have grown to be more reliant on the integration of technologies across multiple disciplines and their requirements. The need to ensure proper division of labor, integration of system modules, and attribution of legal responsibility calls for a more methodological look into co-design considerations. Originally conceived in computer programming, contract-based reasoning is a design approach whose promise of a formal compositional paradigm is receiving attention from a broader engineering community. Our work is dedicated to narrowing the gap between the theory and application of this yet nascent framework.

In the first half of this dissertation, we introduce a model interface contract theory for input/output automata with guards and a formalization of the directive-response architecture using assume-guarantee contracts and show how these may be used to guide the formal design of a traffic intersection and an automated valet parking system respectively. Next, we address a major drawback of assume-guarantee contracts, i.e., the problem of a void contract due to antecedent failure. Our proposed solution is a reactive version of assume-guarantee contracts that enables direct specification at the assumption and guarantee level along with a novel synthesis algorithm that exposes the effects of failures on the contract structure. This is then used to help optimize, adapt, and robustify our design against an uncertain environment.

In light of ongoing development of autonomous driving technologies and its potential impact on the safety of future transportation, the second half of this work is dedicated to the application of the design-by-contract framework to the distributed control of autonomous vehicles. We start by defining and proving properties of "assume-guarantee profiles," our proposed approach to transparent distributed multi-agent decision making and behavior prediction. Next, we provide a local conflict resolution algorithm in the context of a quasi-simultaneous game which guarantees safety and liveness to the composition of autonomous vehicle systems in this game. Finally, to facilitate the extension of these frameworks to real-life urban driving settings,

we also supply an effective method to predict agent behavior that utilizes recent advances in machine learning research.

PUBLISHED CONTENT AND CONTRIBUTIONS

Cai, Karena X., Tung Phan-Minh, Soon-Jo Chung, and Richard M. Murray (2021). "Rules of the Road: Towards Safety and Liveness Guarantees for Autonomous Vehicles." Submitted.
TP-M made key contributions to the conceptual framework, problem formulation, theoretical approach and proofs.

Graebener, Josefine B., Tung Phan-Minh, and Richard M. Murray (2021). "Failure-Tolerant Contract-Based Design of an Automated Valet Parking System using a Directive-Response Architecture." Submitted.
TP-M made key contributions to the conceptual framework, problem formulation and proofs.

Phan-Minh, Tung and Richard M. Murray (2021). "Contracts of Reactivity." Submitted.
TP-M developed the theoretical framework, solution approaches and case study.

Chen, Yuxiao, Sumanth Dathathri, Tung Phan-Minh, and Richard M. Murray (2020). "Counter-example Guided Learning of Bounds on Environment Behavior." In: *Conference on Robot Learning*, pp. 898–909. URL: http://proceedings.mlr.press/v100/chen20b.html.
TP-M provided a derivation for the learning algorithm and contributes to a case study.

Phan-Minh, Tung, Elena Corina Grigore, Freddy A. Boulton, Oscar Beijbom, and Eric M. Wolff (2020). "Covernet: Multimodal Behavior Prediction using Trajectory Sets." In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14074–14083. DOI: 10.1109/CVPR42600.2020.01408.
TP-M made key contributions to the problem reformulation, solution approach, implementation and experimentation.

Phan-Minh, Tung, Karena X. Cai, and Richard M. Murray (2019). "Towards Assume-Guarantee Profiles for Autonomous Vehicles." In: *2019 IEEE 58th Conference on Decision and Control (CDC)*. IEEE, pp. 2788–2795. DOI: 10.1109/CDC40024.2019.9030068.
TP-M made key contributions to the conceptual framework, problem formulation, theoretical approach and proofs.

Phan-Minh, Tung, Steve Guo, Bastian Schuermann, Matthias Althoff, and Richard M. Murray (2019). "A Modal Interface Contract Theory for Guarded Input/Output Automata with an Application in Traffic System Design." In: *2019 American Control Conference (ACC)*. IEEE, pp. 1704–1711. DOI: 10.23919/ACC.2019.8814789.
TP-M made key contributions to the theory, proofs and demonstration.

CONTENTS

Chapter 1

INTRODUCTION

> "I don't spend my time pontificating about high-concept things; I spend my time solving engineering and manufacturing problems."
>
> *Elon Musk*

Many systems we build have to interact with the world around them. Their operation usually depends on other systems that they have no control over: a solar (photovoltaic) module does not control the amount of sunlight it gets. As a solar module alone often does not produce sufficient power, a solar panel system often includes multiple solar modules, arranged in a particular way. To provide a reliable supply of useful energy, it also needs, among others, an inverter, a battery, a battery regulator, and wires to connect these components together. Every single one of these systems is, by itself, an *open* system. This means they require one another to produce the expected behavior, namely, harvesting and storing energy.

Co-design is a process that often proceeds as follows. Some engineers in different areas of expertise want to build a product. They get together, draft up a plan, build the parts, then integrate their designs, hoping that the final product will meet their expectations. How long does it take for this to happen? When it does, how do they guarantee its performance? The answers depend on a lot of factors, the most important of which are experience, ingenuity, and sometimes luck. When there is insufficient knowledge in the subsystems and the room for mistakes is small, the co-design problem becomes even more difficult. It is true that people have been building many systems by trial and error and accomplished a lot from so doing and that experiments and testing are indeed necessary to discover new knowledge about the world. However, at a certain point, when sufficient knowledge has already been attained, it proves worthwhile to be methodological when specifying systems. Especially, in the long run, extra benefits such as customizability, component re-use, and drastic cuts in costs may outweigh the initial time investment.

Writing specifications for system modules is a nontrivial task. Common approaches include "top-down," "bottom-up," and the "V-process" which can be seen as a combination of the first two. The top-down approach puts an emphasis on organization and a thorough understanding of the system while the bottom-up approach is all about implementation and early testing/validation. In the top-down approach, we begin with a high-level description of the system, and decompose it into descriptions of its modules. Each of these descriptions may then be refined by adding more details as appropriate depending on the local environment they are intended for, and optionally further decomposed, until they are ready for implementation. In the top-down approach, the testing of the modules is delayed until the design is done. In bottom-up approach, modules may be implemented without much regard as to how they interact with other parts of the system. There are many frameworks for specifying open systems. All require knowledge and skills for correct execution as specifications are delicate and often open to "loopholes" and unexpected meanings.

A common theme to open system specifications is that the system being specified is nearly always described with two separate sets of properties. One of them is called the assumption, and the other the guarantee. The former describes the environments of the system and the latter specifies its obligations. Describing a component with an assumption and a guarantee effectively constrains the design space to a favorable region while leaving enough room for other systems to operate in given constraints that are already present such as the laws of physics and mathematics. This also enables said system to be thought of as a black box, a critical requirement for modular design. The assume-guarantee paradigm plays an essential role in the design-by-contract framework, a brief history of which is now in order.

1.1 History

There are a few early variants of the assume-guarantee formalism attempting to capture the conditional dependence of components. One of the earliest attempts to formalize this method of reasoning is Hoare logic (Hoare, 1969). This makes use of triples consisting of a pre-condition, a program description, and a post-condition to prove invariants. Misra and Chandy introduced the rely-guarantee approach for safety properties of distributed networks (Misra and Chandy, 1981). In this approach, each component (process) in the network is described with a pair of assertions that are similar to pre- and post-conditions of Hoare logic. The verification of the assembled network is reduced to the application of inference rules to the components' specifications. Meyer can be credited with being the first to use

the phrase "design-by-contract" (Meyer, 1992b). The related object-oriented Eiffel programming language supports its philosophy by allowing the use of assertions and an approach to inheritance that is based on the notion of subcontracting (Meyer, 1992a). The last decade or so has seen an increase in contract theories and applications in cyber-physical systems (Sangiovanni-Vincentelli, Damm, and Passerone, 2012; Nuzzo, 2015; Kim, Arcak, and Seshia, 2015; Censi, 2016a; Filippidis, 2019; Foster et al., 2020). Prominent among these is a contract metatheory by Benveniste et al. that tries to unify many forms of compositional thinking with algebraization (Benveniste et al., 2015).

1.2 Contributions

Recent interests in contract theories from researchers in cyber-physical systems have sparked off multiple definitions for what a contract is. Depending on the applications of interest, these formulations can vary from continuous to discrete time, with components assuming various forms that range from concrete (e.g., black boxes with input/output ports connectable to one another through wires) to abstract (e.g., sets of variables with constraints) (Censi, 2016b; Filippidis, 2019). In terms of applications, while some systems like aircraft electric power system (Nuzzo et al., 2013) exist, "non-toy" examples are still quite limited. These facts motivate us to adopt the general contract perspective given by metatheory of contracts and further back it by showing how objects in the metatheory get mapped to some contract theories and applications that are of engineering interests. Specifically, we provide an example involving a traffic coordinator and relatively complex control system for a parking garage with different levels of commands. We also focus on an often neglected theme in contract theory which is reaction to failures. Our introduction of "reactive contracts" directly generalizes attempts to address this issue from earlier works (Kim, Arcak, and Seshia, 2017; Kim, Sadraddini, et al., 2017). Lastly, we formulate an assume-guarantee framework for self-driving cars in response to recent and ongoing discussions about safety and responsibility for these types of systems (Shalev-Shwartz, Shammah, and Shashua, 2017). From this, we present a discrete model of autonomous vehicles and road networks that is provably consistent and complete. We also offer an effective learning-based trajectory prediction method to help generalize this framework to more practical settings.

1.3 Outline of Thesis

In Chapter 2, some background on the contract metatheory is presented. This will serve as a guiding theoretical basis for the rest of the dissertation. In Chapter 3, we construct a metatheory-compliant contract theory for modal interface input/output automata with guards and apply it to the design of a correct-by-construction traffic intersection control system. Next, in Chapter 4, we develop an assume-guarantee contract framework to model the directive-response architecture of components of an automated valet parking system. Then in Chapter 5, we introduce the theory of reactive contracts and associated algorithms to allow for meta-specifications at the assume-guarantee level. This directly addresses the problem of a non-enforceable contract due to an expected failure invalidating one of its pre-conditions. The second half of the thesis focuses on the application of the contract framework to the development of autonomous driving technologies, specifically the distributed control of autonomous vehicles. In Chapter 6, we lay out a theoretical framework for transparent distributed multi-agent decision making and behavior prediction using "assume-guarantee profiles." In Chapter 7, a provably correct distributed conflict resolution algorithm for quasi-simultaneous games is provided to guarantee safety and liveness for a composed system of autonomous vehicle agents. In Chapter 8, we propose a competitive neural-network based multi-modal behavior prediction technique as a step toward extending our contract-based frameworks to real-world systems. Finally, some conclusions and a discussion of possible future directions are provided in Chapter 9.

References

Benveniste, Albert, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Tom Henzinger, and Kim G. Larsen (July 2015). *Contracts for Systems Design: Theory*. Research Report RR-8759. Inria Rennes Bretagne Atlantique ; INRIA, p. 86. URL: https://hal.inria.fr/hal-01178467.

Censi, Andrea (2016a). "A class of co-design problems with cyclic constraints and their solution." In: *IEEE Robotics and Automation Letters* 2.1, pp. 96–103.

– (Sept. 2016b). *A Mathematical Theory of Co-Design*. Tech. rep. Submitted and conditionally accepted to IEEE Transactions on Robotics. Laboratory for Information and Decision Systems, MIT. URL: https://arxiv.org/abs/1512.08055.

Filippidis, Ioannis (2019). "Decomposing formal specifications into assume-guarantee contracts for hierarchical system design." PhD thesis. California Institute of Technology.

Foster, Simon, Ana Cavalcanti, Samuel Canham, Jim Woodcock, and Frank Zeyda (2020). "Unifying theories of reactive design contracts." In: *Theoretical Computer Science* 802, pp. 105–140.

Hoare, Tony (1969). "An axiomatic basis for computer programming." In: *Commun. ACM* 12, pp. 576–580.

Kim, Eric S., Murat Arcak, and Sanjit A. Seshia (2015). "Compositional controller synthesis for vehicular traffic networks." In: *2015 54th IEEE Conference on Decision and Control (CDC)*. IEEE, pp. 6165–6171.

– (2017). "A small gain theorem for parametric assume-guarantee contracts." In: *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control*, pp. 207–216.

Kim, Eric S., Sadra Sadraddini, Calin Belta, Murat Arcak, and Sanjit A. Seshia (2017). "Dynamic contracts for distributed temporal logic control of traffic networks." In: *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*. IEEE, pp. 3640–3645.

Meyer, Bertrand (1992a). "Applying Design by Contract." In: *Computer* 25.10, pp. 40–51.

– (1992b). "Design by Contract." In: *IEEE Computer* 25, p. 10.

Misra, Jayadev and K. Mani Chandy (1981). "Proofs of Networks of Processes." In: *IEEE Transactions on Software Engineering* SE-7.4, pp. 417–426. DOI: 10.1109/TSE.1981.230844.

Nuzzo, Pierluigi (Aug. 2015). "Compositional Design of Cyber-Physical Systems Using Contracts." PhD thesis. EECS Department, University of California, Berkeley. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-189.html.

Nuzzo, Pierluigi, Huan Xu, Necmiye Ozay, John B Finn, Alberto L Sangiovanni-Vincentelli, Richard M Murray, Alexandre Donzé, and Sanjit A Seshia (2013). "A contract-based methodology for aircraft electric power system design." In: *IEEE Access* 2, pp. 1–25.

Sangiovanni-Vincentelli, Alberto, Werner Damm, and Roberto Passerone (2012). "Taming Dr. Frankenstein: Contract-based design for cyber-physical systems." In: *European journal of control* 18.3, pp. 217–238.

Shalev-Shwartz, Shai, Shaked Shammah, and Amnon Shashua (2017). "On a formal model of safe and scalable self-driving cars." In: *arXiv preprint arXiv:1708.06374*.

Chapter 2

THE CONTRACT METATHEORY

Various works in the contract literature are not in complete agreement when it comes to the precise meaning of the word "contract". To avoid this confusion, we adopt the following relatively general definitions from the metatheory.

2.1 What is a contract?

Two main types of objects that the metatheory concerns itself with are *components* and *contracts*. For components, *composability* is identified as the key property. Syntax-wise, for any two composable components M_1 and M_2, we will denote their composition as $M_1 \oplus M_2$. In the metatheory, we require $\oplus$ to be an associative and commutative binary operator on the set of components. These requirements enable a simple yet powerful perspective of composition that can be understood roughly as an agreement on the values of shared variables. Effectively, when we compose two components, we are restricting what they can do by constraining the behavior that they experience to be the same. A component is defined to be *closed* if it cannot be composed with any other component, otherwise it is *open*.

Each contract C is defined to be a tuple $(\mathcal{I}_C, \mathcal{E}_C)$ where $\mathcal{I}_C$ and $\mathcal{E}_C$ are two sets of components. These are referred to as the set of *implementations* and the set of *environments* of C, respectively. It is further required that any pair of elements not from the same set are composable. A contract therefore has a "built-in" assume-guarantee semantics in the sense that a set of implementations is paired to a set of "intended" environments.

The sizes of the sets of implementations and environments of a contract can be used to establish how coarse or fine it is. This allows us to talk about the descriptiveness of a contract, a useful concept for vertical design where we often want to refer to more specific implementations than the ones in question.

Definition 2.1.1 (Refinement)**.** We say that C_1 refines C_2 and write $C_1 \preceq C_2$ if and only if $\mathcal{I}_{C_1} \subseteq \mathcal{I}_{C_2}$ and $\mathcal{E}_{C_2} \subseteq \mathcal{E}_{C_1}$.

Observe that the more refined contract C_1 in Definition 2.1.1 has a smaller implementation set since it is more specific. At the same time, having fewer implemen-

tations may relax the composability constraint. Thus, it is allowed to have a larger environment set. Clearly, $\preceq$ effectively defines a partial order on the set of contracts.

The contract conjunction operator is used to enforce two aspects of a design. It describes the coarsest contract that simultaneously refines two given contracts and is useful when we would like to fuse requirements from those contracts.

Definition 2.1.2 (Conjunction)**.** The conjunction $C_1 \wedge C_2$ of C_1 and C_2 is the largest contract with respect to the refinement ordering $\preceq$ that refines both C_1 and C_2.

Conversely, the contract disjunction is used to describe alternatives or expand the set of qualified implementations in a controlled way.

Definition 2.1.3 (Disjunction)**.** The disjunction $C_1 \vee C_2$ of C_1 and C_2 is the smallest contract with respect to the refinement ordering $\preceq$ that is refined by both C_1 and C_2.

Composing contracts is the most important operation of the metatheory. It characterizes the specifications for the larger system that results from coupling the specifications of two open systems that are supposed to interact with one another. It is required that the composition of any two implementations of contracts C_1, C_2 should implement their composition $C_1 \otimes C_2$, and any environment for $C_1 \otimes C_2$ composed with an implementation for C_1, should give an environment for C_2 and vice-versa. Formally, we have the following.

Definition 2.1.4 (Composition)**.** The composition $C_1 \otimes C_2$ is the smallest contract $C = (\mathcal{E}_C, \mathcal{I}_C)$ that has the composition property, namely, for any $M_1 \in \mathcal{I}_{C_1}$, $M_2 \in \mathcal{I}_{C_2}$, and $E \in \mathcal{E}_C$, we have $M_1 \oplus M_2 \in \mathcal{I}_C$, $M_1 \oplus E \in \mathcal{E}_{C_2}$, and $M_2 \oplus E \in \mathcal{E}_{C_1}$.

Finally, the quotient operator allows us to find the missing specifications for an unspecified part of a given system.

Definition 2.1.5 (Quotient)**.** The quotient C_1/C_2 is the largest contract C such that $C \otimes C_2 \preceq C_1$.

These operators form the basis of the canonical metatheory. Recently, more operators have been found/defined, enabling more expressiveness (Passerone, Íncer, and Sangiovanni-Vincentelli, 2019; Íncer et al., 2020).

Each contract "flavor" of the metatheory is defined by a set of components and how they can be composed through $\oplus$. To concretize these definitions, we will follow up with an informal example.

Figure 2.1: An element of $\mathfrak{I}$ in the example from Section 2.2.

Figure 2.2: The result of composing an element of $\mathfrak{I}$ and an element of P in the example from Section 2.2.

2.2 An Example: Contracts on Image Annotating Functions

Let $\mathfrak{I}$ be a set of images like the one shown in Figure 2.1. Let P be a set of functions each of which takes each image in $\mathfrak{I}$ and returns an *annotated* image in $\mathfrak{I}$. By "annotating", we mean a process that adds an extra layer of information to an image. For each $p \in P$ and $i \in \mathfrak{I}$, we will denote by $p \oplus i$ or $i \oplus p$ the result of annotating i with p. An example of this is shown in Figure 2.2, in which the original image in Figure 2.1 was annotated with a red bounding box around the sole vehicle in the

Figure 2.3: A possible visualization of $p_1 \oplus p_2 \oplus i$ from 2.2.

scene.

For any two such functions p_1 and p_2 in P, we define their composition $p_1 \oplus p_2$ to be simply the function that adds both annotations from p_1 and p_2 to each image from $\mathfrak{I}$. For example, if p_1 is the function that annotates any image from $\mathfrak{I}$ by putting a rectangular box around each vehicle in it and p_2 is the function that annotates each image from $\mathfrak{I}$ with yellow arrows that describe potential future trajectories for each vehicle, then the function $p_1 \oplus p_2$ would return an image like the one in Figure 2.3 assuming the input is again the image i from Figure 2.1. Assuming the order of annotations does not matter, we can see that $\oplus$ is associative and commutative on the set of components defined as $P \cup \mathfrak{I}$ as long as the composition is well-defined (that is, if it contains at most one element from $\mathfrak{I}$).

Now, let

- $\mathfrak{I}_{\texttt{unannotated}}$ be a subset of $\mathfrak{I}$ such that each element of $\mathfrak{I}_{\texttt{unannotated}}$ is unannotated.
- $\mathfrak{I}_{\texttt{freeway}}$ be a subset of $\mathfrak{I}_{\texttt{unannotated}}$ such that each element of $\mathfrak{I}_{\texttt{freeway}}$ is a picture of freeway traffic.
- $\mathcal{I}_{\texttt{prog,car}}$ be a set of *computer programs* that annotate each freeway image with a bounding box around each car that is present in it.

- $\mathcal{I}_{\texttt{prog,bike}}$ be a set of *computer programs* that annotate each freeway image with a bounding box around each motorbike that is present in it.
- $\mathcal{I}_{\texttt{nn,box}}$ be a set of *neural networks* that annotate each image with a bounding box around each traffic agent (car, motorbike, bicycle, pedestrian...) that is present in it.

The specifications "Design a program that puts bounding boxes around each car on freeways" and "Design a program that puts bounding boxes around each motorbike on freeways" may correspond to the contracts $C_1 = (\mathfrak{I}_{\texttt{freeway}}, \mathcal{I}_{\texttt{prog,car}})$ and $C_2 = (\mathfrak{I}_{\texttt{freeway}}, \mathcal{I}_{\texttt{prog,bike}})$, respectively.

The contract conjunction $C_1 \wedge C_2$ exists and corresponds to the specification "Design a program that puts bounding boxes around every car and motorbike on freeways" while the disjunction $C_1 \vee C_2$ corresponds to "Design a program such that, for each (unannoted) freeway image, it either 1) consistently annotates every car with a bounding box or 2) consistently annotates every motorbike with a bounding box". Note that consistency means that for all images in $\mathfrak{I}_{\texttt{freeway}}$, the program must either box all the cars or all the bikes. That is, there exists no pair of images where one of them has all the cars boxed but not all the bikes while the other has all the bikes boxed but not all the cars.

On the other hand, the specification "Implement a neural network that puts a bounding box around each traffic agent" may correspond to $C_3 = (\mathfrak{I}_{\texttt{unannotated}}, \mathcal{I}_{\texttt{nn,box}})$. We can see that C_3 is a more specific contract than even the conjunction of C_1 and C_2, namely, $C_3 \preceq C_1 \wedge C_2$ because $\mathcal{I}_{\texttt{nn,box}} \subseteq \mathcal{I}_{\texttt{prog,car}} \cap \mathcal{I}_{\texttt{prog,bike}}$.

Lastly, consider the specification "Implement a neural network that, given an unannoted image, puts a bounding box around each traffic agent and provides for that same agent a predicted trajectory", which corresponds to a contract $C = (\mathfrak{I}_{\texttt{unannotated}}, \mathcal{I}_{\texttt{nn,box+pred}})$. Let $\mathfrak{I}_{\texttt{unboxed}}$ be a subset of $\mathfrak{I}_{\texttt{unannotated}}$ consisting of images with no bounding boxes. Let us assume that a contract $C_4 = (\mathfrak{I}_{\texttt{unboxed}}, \mathcal{I}_{\texttt{nn,box}})$ has been assigned to a team for implementation, namely they must, for each image in $\mathfrak{I}_{\texttt{unboxed}}$, provide the bounding box for each agent in it. What should our implementation satisfy in order for us to reuse whatever implementation the other team comes up with, in the sense that their composition would satisfy C? By the metatheory, we only need to satisfy any refinement of contract C/C_4. In particular and as may be expected, $(\mathfrak{I}_{\texttt{unpred}}, \mathcal{I}_{\texttt{nn,pred}})$ where $\mathfrak{I}_{\texttt{unpred}}$ is a

subset of $\mathfrak{I}$ with no predicted trajectories and $\mathcal{I}_{\mathtt{nn,pred}}$ is the set of neural networks that add a predicted trajectory to each agent present in any image from $\mathfrak{I}_{\mathtt{unpred}}$.

2.3 Properties

The definitions from the metatheory immediately result in the following properties (their proofs can be found in (Benveniste et al., 2015)).

The first two are clear from the definitions of refinement and conjunction.

Proposition 1. *If $C_1 \preceq C_2$, then*

1. *Any implementation of C_1 is an implementation of C_2.*
2. *Any environment of C_2 is an environment of C_1.*

Proposition 2 (Shared refinement)**.** *The following is true for the conjunction of two contracts.*

1. *Any contract that refines $C_1 \wedge C_2$ also refines C_1 and C_2.*
2. *Any implementation of $C_1 \wedge C_2$ is a shared implementation of C_1 and C_2.*
3. *Any environment of C_1 or C_2 is an environment of $C_1 \wedge C_2$.*

The next proposition says that it is safe to refine constituent contracts independently.

Proposition 3 (Independent implementability)**.** *If $C_1' \preceq C_1$ and $C_2' \preceq C_2$, then $C_1' \otimes C_2' \preceq C_1 \otimes C_2$.*

The contract composition definition also allows for flexibility in how contracts for subsystems are grouped and ordered for composition. If four subsystems are supposed to complement each other, we can frame the design problem as that which involves composing any two larger subsystems, each of which is a composition of two of the original subsystems.

Proposition 4 (Architecture flexibility)**.** $(C_1 \otimes C_2) \otimes (C_3 \otimes C_4) = (C_1 \otimes C_3) \otimes (C_2 \otimes C_4)$.

Additionally, the following relation about composing and conjoining systems implies that composing two systems with "functionally congruent" subsystems can be reduced (or refined) to fusing the requirements of each pair of congruent subsystems and composing the results afterwards.

Proposition 5 (Sub-distributivity)**.** $(C_1 \wedge C_3) \otimes (C_2 \wedge C_4) \preceq (C_1 \otimes C_2) \wedge (C_3 \otimes C_4)$.

The following follows from the definition of the quotient operator. It is consistent with the intuition that composing a refinement of a quotiented out contract with the already existing part should refine the original contract for the composite system.

Proposition 6 (Quotient)**.** $C \preceq C_1/C_2 \Leftrightarrow C \otimes C_2 \preceq C_1$.

References

Benveniste, Albert, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Tom Henzinger, and Kim G. Larsen (July 2015). *Contracts for Systems Design: Theory*. Research Report RR-8759. Inria Rennes Bretagne Atlantique ; INRIA, p. 86. URL: https://hal.inria.fr/hal-01178467.

Íncer, Íñigo, Leonardo Mangeruca, Tiziano Villa, and Alberto Sangiovanni-Vincentelli (Sept. 2020). *The Quotient in Preorder Theories*. Tech. rep. UCB/EECS-2020-179. EECS Department, University of California, Berkeley. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-179.html.

Passerone, Roberto, Íñigo Íncer, and Alberto Sangiovanni-Vincentelli (2019). "Coherent Extension, Composition, and Merging Operators in Contract Models for System Design." In: *ACM Transactions on Embedded Computing Systems (TECS)* 18, pp. 1–23.

Chapter 3

A MODEL INTERFACE THEORY FOR GUARDED INPUT/OUTPUT AUTOMATA

3.1 Introduction

The growth in scale and complexity of engineering systems has created stronger demands for formal approaches to modular design (Baldwin and Clark, 2006; Huang and Kusiak, 1998). Generally speaking, modular design means breaking up a system into more or less standalone modules for a reduction in complexities. To guarantee correct product integration, it is therefore not only a matter of convenience but also of necessity for design choices intended for a module to be made available to others. One way of dealing with this dependency is to divide tasks of designing a module into two parts: specifying an *interface* and ensuring that the *implementation* satisfies it. The interface of a module contains all information about the interactions it can offer to other modules. An implementation should satisfy the specifications of the interface. The idea is that changes to an implementation of a module should not affect the overall behavior of the assembled system as long as the implementation still satisfies the requirements of the interface.

A lightweight automata-theoretic approach to represent interfaces was introduced by de Alfaro and Henzinger (2001), in which the temporal behavior of an interface is described by a game-based model in the form of an input/output (I/O) automaton, a formalism by Lynch and Tuttle (1987). This was soon followed by modal specifications by Larsen (1989), which can state whether an action is *optional* or *obligatory*. Later Raclet unified modal specifications and interface automata, paving the way for a preliminary theory of *modal interface contracts* (Raclet et al., 2009). More recently, Benveniste et al. subsumed this theory under the metatheory (Benveniste, Benoit Caillaud, et al., 2012). In the application domain, however, the semantics of the theory is limited by a lack of clear restrictions on when a transition can trigger and by a peculiar rule for composing actions, namely, requiring that the action obtained from composing an input action with an output action to be an output action, which, while preserving the "interface" semantics, obscures the distinction between open and closed interfaces/systems. These drawbacks make it difficult and sometimes impossible to specify systems whose variables assume a large or infinite

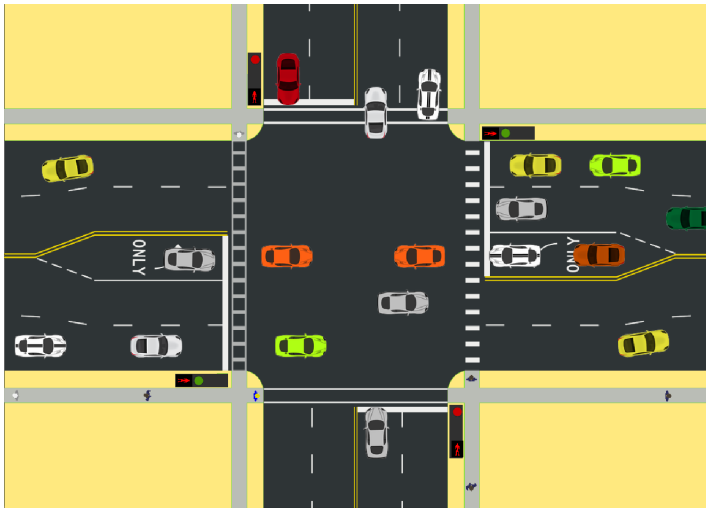


Figure 3.1: A snapshot of an implementation of a networked control system of traffic presented in (Phan-Minh, 2018).

set of values.

With an aim to bringing the benefits of the theory of modal interface contract automata to more real-time systems, inspired by symbolic transducers (Veanes, Hooimeijer, Livshits, et al., 2012), we develop a new theory that includes Boolean guards, a more intuitive definition of how the I/O actions interact, an introduction of a special state that enriches the semantics of the contract object, and a simplification in the definitions of interfaces. In addition, we prove that the algebraic operations defined for our contract theory also have metatheoretic properties, implying compatibility with many existing contract frameworks. We then implement a set of tools that carry out the contract algebra in a similar manner to *Mica*, which implements the modal interface contract in (Raclet et al., 2011). As an illustrative example application of our theory, we introduce a method for setting up an autonomous traffic system where various interfaces communicate with each other while abiding by the contract protocol. Our concrete case study involves a real-time simulation of a traffic intersection (see Fig. 3.1) whose components interact with each other in accordance with the contract objects we devise.

3.2 Interface Contract Theory

Many real-world applications ranging from online payment services to autonomous robots require networking protocols to control sequential exchanges of information between many subsystems or participating agents. By “sequential” we mean that the interactions must occur in a well-specified, agreed upon temporal order. A good

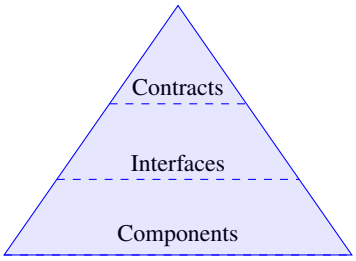


Figure 3.2: Contract design hierarchy pyramid.

implementation of these protocols presupposes the notion of a set of rules for each subsystem that not only restricts what action the subsystem can perform from a certain state at a certain time, but can also be compared to or combined with other sets of rules corresponding to other subsystems. To illustrate these ideas, we will provide definitions for three formal objects, arranged in the hierarchy pyramid in Fig. 3.2 by their level of abstraction. The higher the object sits, the more abstract it is and the fewer ways there are to implement it. First, we will introduce the *interface*.

Interface

Our formalism rests upon the assumption that each state of the universe fixes the values for a master set of variables $\mathcal{U}$ whose temporal and algebraic behaviors are governed by mathematical and physical laws. The *interface of a component* (or *interface* for short) may be described by a subset of $\mathcal{U}$ (these are called its *reference variables*), and can interact with the rest of the universe (i.e., one of its *environments*) through a set of *actions*. Any action from this interface may belong to exactly one of the following three classes. An *input* action of an interface corresponds to the receiving of mass or energy from its environment. An *output* action, on the other hand, corresponds to sending mass or energy to its environment. An *internal* or *rendezvous* action represents an interconnection. Intuitively, an interconnection implies the existence of at least one input and one output action and we can think of it as being internally exchanged *within* the component. As a syntactic reminder, we will prefix input actions with an exclamation mark (!), output actions with a question mark (?), and rendezvous actions with a hash symbol (#).

An interface is *closed* if and only if its corresponding set of actions is empty or only consists of rendezvous actions. An interface is called *open* otherwise. For example, the universe, in its entirety has a closed interface. A wireless router has an open interface because it takes inputs from a modem and emits radiowaves.

For any set of variables $\mathcal{V} \subseteq \mathcal{U}$, a valuation e of $\mathcal{V}$ denoted by $e[\mathcal{V}]$ is a legal

assignment of values to each of the variables in $\mathcal{V}$. By legal, we mean that each variable is assigned a value that is in the value set specified by its type (e.g., the reals). In mathematical logic terms, e is a ground substitution of variables in $\mathcal{V}$. The set $E_{\mathcal{V}} \coloneqq \{e \mid e \text{ is a valuation of } \mathcal{V}\}$ contains all possible valuations of $\mathcal{V}$. Since one of our interests is in "connecting" different interfaces, it is important to specify the conditions under which this can happen. For this purpose and also to obtain a compact automaton representation of interfaces, we will invoke the notion of guards.

Definition 3.2.1 (Guard)**.** Let $\top \coloneqq \texttt{True}$ and $\bot \coloneqq \texttt{False}$. A *guard* g defined on a set of variables $\mathcal{V}$ is a predicate on the variables in this set, namely, $g : E_{\mathcal{V}} \to \{\top, \bot\}$. The set of all predicates on $\mathcal{V}$ is denoted by $G_{\mathcal{V}}$.

For example, when $\mathcal{V}$ is a set of Boolean variables, then a guard on $\mathcal{V}$ is a map from $2^{\mathcal{V}}$ to $\{\top, \bot\}$. Below, we will use the tilde symbol $\sim$ as a wild card character that acts as a placeholder for one of !, ?, and #.

Definition 3.2.2 (Interface)**.** Each *interface* M is defined by a set of reference variables $\mathcal{V}$ and a tuple $\mathcal{A} = (S, s_0, \mathfrak{F}, A, \to)$ where

(i) S is a finite set of operational states

(ii) $s_0 \in S$ is the start state

(iii) $\mathfrak{F} \notin S$ is *the failure state*

(iv) A is a set of actions. We will often write A as the partition

$$A = ?A \cup !A \cup \#A,$$

where $?A, !A, \#A$ are the smallest sets containing all the input, output, and internal actions of A, respectively.

(v) $\to \subset S \times [G_{\mathcal{V}} \mid A] \times \bar{S}$ is a guarded transition relation with $\bar{S} = S \cup \{\mathfrak{F}\}$ (note the asymmetry between the start and end sets). For all s_1, s_2, g, a such that $s_1 \times [g \mid \sim a] \times s_2 \in \to$, we say that the action $\sim a$ is only available to M in those states of the universe where g evaluates to $\top$. We also require the transitions to be *deterministic* by requiring that, from each state, there is only one transition per unmasked (namely, disregarding ?, !, or #) action.

We will be writing $q \xrightarrow{[g|\alpha]} p$ in place of $q \times [g_E \mid \alpha] \times p \in \longrightarrow$ as a predicate. The fact that our interfaces are deterministic allows us also to use the shorthand $q \xrightarrow{[g|\alpha]}$ to say with little ambiguity that there exists a state $p \in \bar{S}$ such that $q \xrightarrow{[g_E|\alpha]} p$. Now we are ready to define the interface composition operator, which we will denote by the symbol $\times$. Intuitively, given two interfaces M_1 and M_2, let us assume that M_1 is currently in state s_1 and M_2 is in s_2. If, for example, from s_1, M_1 has a transition $[g_1 \mid ?a]$ to some state s_1' and from s_2, M_2 has a transition $[g_2 \mid !a]$ to state s_2', then if M_1 and M_2 were to be composable, it would make sense to require M_1 and M_2 to exchange the action a with one another. This then reduces to checking if $g_1 \wedge g_2$ is satisfiable (there exists a valuation of the variables in $g_1 \wedge g_2$ that makes it evaluate to `True`). If there is at least one satisfying assignment, then the two interfaces must handshake or "rendezvous" on it, otherwise, they are not composable.

Definition 3.2.3 (Composition of Transitions)**.** The *composition of two transitions* $t_1 = q_1 \xrightarrow{[g_1|\sim a_1]} q_1'$ and $t_2 = q_2 \xrightarrow{[g_2|\sim a_2]} q_2'$, from automata M_1 and M_2, respectively, is possible if $g_1 \wedge g_2$ is satisfiable and $a_1 = a_2$. If these conditions are satisfied, the composed transition is given by $t = (q_1, q_2) \xrightarrow{[g|a]} (q_1', q_2')$, where $g := g_1 \wedge g_2$ and $a := \sim a_1 + \sim a_2$ where + is a binary operator acting on $\sim a_1$ and $\sim a_2$ such that,

$$\sim a_1 + \sim a_2 := \begin{cases} \#a_1 & (\sim a_1 \in\, !A_1 \wedge \sim a_2 \in\, ?A_2) \\ & \vee (\sim a_2 \in\, !A_2 \wedge \sim a_1 \in\, ?A_1) \\ & \vee \sim a_1 = \#a_1 \\ & \vee \sim a_2 = \#a_2 \\ ?a_1 & \sim a_1 \in\, ?A_1 \wedge \sim a_2 \in\, ?A_2 \\ !a_1 & \sim a_1 \in\, !A_1 \wedge \sim a_2 \in\, !A_2. \end{cases}$$

Any input supplied to an automaton that is an output of the other becomes a rendezvous action of the composed automaton since the composed automata should represent the interconnection of the two automata. Observe also that input (or output) actions that compose with themselves are not converted to rendezvous actions, but rather remain unchanged. For convenience, we define u to be a universal "unmasking" function that maps all prefixed actions to pure actions, namely, $u : \sim a \mapsto a$. As an abuse of notation, for a set A of prefixed actions, we write $u(A)$ to mean the set $\{u(a) : a \in A\}$. The composition M of two interfaces M_1 and M_2 is defined as follows:

Definition 3.2.4 (Interface Composition)**.** For two interfaces M_1 defined by $\mathcal{V}_1$ and $(S_1, s_{0,1}, \mathfrak{F}_2, A_1, \rightarrow_1)$ and M_2 defined by $\mathcal{V}_2$ and $(S_2, s_{0,2}, \mathfrak{F}_2, A_2, \rightarrow_2)$, their *composition* $M = M_1 \times M_2$ is defined by $\mathcal{V} \coloneqq \mathcal{V}_1 \cup \mathcal{V}_2$ and $(S, s_0, \mathfrak{F}, A, \rightarrow)$ such that

(i) $S \coloneqq S_1 \times S_2$

(ii) $s_0 \coloneqq s_{0,1} \times s_{0,2}$

(iii) A is partitioned into $\#A \cup !A \cup ?A$, with

$$\begin{aligned} \#A &\coloneqq \#A_1 \cup \#A_2 \bigcup_{i=1}^{2} \#(u(?A_i) \cap u(!A_{3-i})) \\ ?A &\coloneqq ?A_1 \cup ?A_2 \\ !A &\coloneqq !A_1 \cup !A_2 \end{aligned}$$

(iv) $\rightarrow$ is the Cartesian product of transitions in $\rightarrow_1$ and $\rightarrow_2$ with respect to composition as defined in definition 3.2.3. More specifically, $\forall t_1 \in S_1 \times [G_{\mathcal{V}_1} \mid A_1] \times \bar{S_1}, t_2 \in S_2 \times [G_{\mathcal{V}_2} \mid A_2] \times \bar{S_2}$ such that t_1 and t_2 are composable, $t_1 \times t_2 \in \rightarrow$.

(v) associate each (reachable) composite state in $\{\mathfrak{F}_1\} \times S_2 \cup S_1 \times \{\mathfrak{F}_2\}$ with the failure state $\mathfrak{F}$.

Observe that due to Definition 3.2.4(iv) and the fact that each constituent interface is deterministic, from each composite state there can be at most one transition per unmasked action, so the composition is also a deterministic interface.

Proposition 1 (Associativity and Commutativity)**.** *The interface composition operator* $(\times)$ *is associative and commutative. Namely, for all interfaces* M_1, M_2, M_3*, we have*

$$M_1 \times M_2 = M_2 \times M_1 \tag{3.1}$$

and

$$(M_1 \times M_2) \times M_3 = M_1 \times (M_2 \times M_3). \tag{3.2}$$

Proof. (3.1) easily follows from Definitions 3.2.3 and 3.2.4. To see why (3.2) holds, first we note that it is trivial to show that the resulting sets of operational states,

the initial states, and the sets of states that eventually become the failure state from both sides of Equation (3.2) are equal. In what follows, we will be referring to the constituents (e.g., action set) of each interface M_i by their standard notations. Now observe from the last two items in Definition 3.2.4(iii), that the input and output action sets of the resulting interfaces on both sides of Equation (3.2) are equal. For the internal action set, a direct calculation with an application of the set distributive law shows that either side of (3.2) can be reduced to $\bigcup_{i=1}^{3} \#A \bigcup_{j=1, j\neq i}^{3} \#(u(?A_i) \cap u(!A_j))$. By Definition 3.2.4(iv), it remains to be shown that the transition sets are identical, which is to prove

$$\forall t_1, t_2, t_3. \bigwedge_{i=1}^{3} (t_i \in \rightarrow_i) \implies (t_1 \times t_2) \times t_3 = t_1 \times (t_2 \times t_3). \tag{3.3}$$

By Definition 3.2.3, we can assume that $a_1 = a_2 = a_3 = a$. Let us suppose that the resulting action of $(t_1 \times t_2) \times t_3$ is an internal action (the cases for input and output actions are straightforward). Then either

- at least one of $\sim a_1$ or $\sim a_2$ or $\sim a_3$ is an internal action, in which case the resulting action of $t_1 \times (t_2 \times t_3)$ must also be an internal action since # "absorbs" any other type of action.
- or among $\sim a_1, \sim a_2, \sim a_3$, there are exactly two types of actions, namely, input and output, in which case, the resulting action of $t_1 \times (t_2 \times t_3)$ is also an internal action.

Both cases show that (3.3) holds. □

Below, whenever we make a reference to an interface M_i and later a contract, we will be using the same notations used in their respective definitions with the appropriate subscripts to talk about their constituents (set of actions, start states etc.). Given two interfaces, knowledge of whether they are comparable to one another can be very useful. One way to enable this comparison is to check whether one interface can "imitate" or *simulate* the other.

Definition 3.2.5 (Simulation)**.** Let M_1 and M_2 be two interface automata. For $i = 1, 2$, let q_i be a state of M_i. q_2 *simulates* q_1, written $q_1 \lesssim q_2$, if for all $\alpha \in A$

$$q_1 \xrightarrow{[g_1|\alpha]} q_1' \implies \exists q_2' : q_2 \xrightarrow{[g_2|\alpha]} q_2' \wedge (g_1 \implies g_2) \wedge (q_1' \lesssim q_2'). \tag{3.4}$$

M_2 simulates M_1, written $M_1 \lesssim M_2$, when $s_{0,1} \lesssim s_{0,2}$.

The following fact will be useful later when we define contract composition.

Proposition 2. *If $M_1 \preccurlyeq M_2$ and $M_3 \preccurlyeq M_4$, then $M_1 \times M_3 \preccurlyeq M_2 \times M_4$.*

Proof. The start state of $M_1 \times M_3$ can be written as (q_1, q_3) where q_1 and q_3 are start states of M_1 and M_3. By Definition 3.2.5, the start states q_2, q_4 of M_2 and M_4 satisfy $q_1 \preccurlyeq q_2$ and $q_3 \preccurlyeq q_4$. We claim $(q_1, q_3) \preccurlyeq (q_2, q_4)$. Suppose for some q_1', q_3' such that $(q_1, q_3) \xrightarrow{[g_1 \wedge g_3 | \alpha]} (q_1', q_3')$ where $q_1 \xrightarrow{[g_1|\alpha_1]} q_1'$, $q_3 \xrightarrow{[g_3|\alpha_2]} q_3'$ and $\alpha = \alpha_1 + \alpha_2$. By Definition 3.2.5, we have $q_2 \xrightarrow{[g_2|\alpha_1]} q_2' \wedge (g_1 \Rightarrow g_2) \wedge q_1' \preccurlyeq q_2'$ and $q_4 \xrightarrow{[g_4|\alpha_2]} q_4' \wedge (g_3 \Rightarrow g_4) \wedge q_3' \preccurlyeq q_4'$. These two yield the transition $(q_2, q_4) \xrightarrow{[g_2 \wedge g_4 | \alpha_1 + \alpha_2]} (q_2', q_4')$ in $M_2 \times M_4$ and $g_1 \wedge g_3 \Rightarrow g_2 \wedge g_4$. By performing this argument inductively, we have $(q_1', q_3') \preccurlyeq (q_2', q_4')$ and therefore $q_1 \times q_3 \preccurlyeq q_2 \times q_4$, from which the claim follows. □

Component

While the interface is a mathematical object that specifies actions a module can exchange with its environment, a component is any structure (e.g., hardware or software, or human) that satisfies the promises of the interface for that module. To keep the interface representation compact, it is helpful to maintain a small action alphabet. In applications, this may be done by appropriately mapping the numerous actions (due to parametrization or the fact that they stem from different structures) to a small number of classes that represent the actions in the alphabet of the corresponding interface. For a set of actions A_c, an action alphabet A, a component K, and $Q : A_c \to A$ be an action equivalence map, we have the following definition.

Definition 3.2.6 (Component)**.** We say a *component* K models an interface M under the action equivalence map Q and write $K \models^{Q}_{comp} M$ if there exists a state machine representation $\bar{K}$ of K modulo Q such that $\bar{K} \preccurlyeq M$.

Immediately from the definition, we have for all interfaces M_1 and M_2, $M_1 \preccurlyeq M_2 \implies M_1 \models^{\mathcal{I}}_{comp} M_2$, where $\mathcal{I}$ is the identity map.

Contract

At the top of the contract design hierarchy is the contract object, which is defined as follows:

Definition 3.2.7 (Guarded Modal Interface Contracts)**.** A *guarded modal interface contract* C consists of a set of reference variables $\mathcal{V}$ and a tuple of the form $\mathfrak{A} = (S, s_0, \mathfrak{F}, A, \rightarrow, \dashrightarrow)$, where $S, s_0, \mathfrak{F}, A$ are defined as in the interface automaton object. $\rightarrow$ and $\dashrightarrow$ are two transition relations called *must* and *may*, respectively. Intuitively, a may transition with guard g and action α in the interface contract specifies that any interface implementing the contract is *allowed but not required* to perform α as long as the guard is satisfied. On the other hand, a must transition in the interface contract specifies a transition that any interface implementing it is required to include. Clearly, this implies that any must transition is also required to be a may transition, namely for $q \in S, g_1, g_2 \in G_{\mathcal{V}}$, and $\alpha \in A$, we have

$$(q \xrightarrow{[g_1|\alpha]} \Longrightarrow\ q \overset{[g_2|\alpha]}{\dashrightarrow}) \wedge (g_1 \implies g_2). \tag{3.5}$$

(3.5) says that the existence of a must transition implies the existence of a may transition with a weaker guard. A modal interface contract C naturally induces two interface automata M_{must} and M_{may} with only $\rightarrow$ and $\dashrightarrow$ as transition relations, respectively, and fixes a set of *environments* of the contract, denoted by E_C. An environment $E \in E_C$ is an interface automaton such that $E \times M_{\text{may}}$ is closed (by (3.5), $E \times M_{\text{must}}$ is also closed) and for each reachable state $(q_E, q_M) \in E \times M_{\text{may}}$ and any (unprefixed) action a

$$q_E \xrightarrow{[g_E|!a]} \Longrightarrow\ q_M \xrightarrow{[g_M|?,\#a]} \wedge(g_E \implies g_M), \tag{3.6}$$

$$q_E \xrightarrow{[g_E|?a]} \Longrightarrow\ q_M \xrightarrow{[g_M|!,\#a]} \wedge(g_E \implies g_M). \tag{3.7}$$

Here ?, # indicate that α can be either input or internal. Together, these mean that any time the environment is only willing to emit an output or request an input if M_{may} can accept it. C also fixes a set of interfaces M_C that *implement* C, such that $M \in M_C$ if

$$M_{\text{must}} \lesssim M \lesssim M_{\text{may}}. \tag{3.8}$$

which is essentially stating that all reachable must transitions must be included in M, and M can only use may transitions of the contract. Below, we will use as a shorthand $\lesssim_{\text{may(must)}}$ as the simulation relation with respect to the may (must) transitions only in the contract object. Contract refinement, conjunction, and composition are defined as follows

Definition 3.2.8 (Modal Refinement)**.** Let C_1 and C_2 be two guarded modal interface contracts. Then C_2 *refines* C_1, written $C_2 \preceq C_1$ if and only if

$$M_{2,\text{may}} \lesssim M_{1,\text{may}}, \tag{3.9}$$

$$M_{1,\text{must}} \lesssim M_{2,\text{must}}. \tag{3.10}$$

Proposition 3. *A more refined contract allows for more environments, namely*

$$C_2 \preceq C_1 \implies E_{C_2} \supseteq E_{C_1}. \tag{3.11}$$

Proof. Let $E \in E_{C_1}$. By Definitions 3.2.5 and 3.2.8, we have $M_{2,\text{may}} \lesssim M_{1,\text{may}}$ and for any reachable state (q_E, q_2) of $E \times M_{2,\text{may}}$, there exists a reachable state (q_E, q_1) in $E \times M_{1,\text{may}}$ such that $q_2 \lesssim q_1$. So for any outgoing transition of (q_E, q_2) in $E \times M_{2,\text{may}}$ doing an action α, there is a corresponding transition from (q_E, q_1) in $E \times M_{1,\text{may}}$ that also does α. Since $E \times M_{1,\text{may}}$ is closed, α must be an internal action. Therefore $E \times M_{2,\text{may}}$ is also closed. Furthermore,

$$q_E \xrightarrow{[g_E|!a]} \overset{E\in E_{C_1}}{\Longrightarrow} q_1 \xrightarrow{[g_1|?,\#a]} \overset{(3.9)}{\Longrightarrow} q_2 \xrightarrow{[g_2|?,\#a]}$$

and

$$g_E \overset{E\in E_{C_1}}{\Longrightarrow} g_1 \overset{(3.9)}{\Longrightarrow} g_2$$

satisfying (3.6). Similarly, (3.7) also holds, implying $E \in E_{C_2}$. □

Proposition 4. *A contract is more refined than another if and only if its implementations are also the other's implementations:*

$$C_2 \preceq C_1 \iff M_{C_2} \subseteq M_{C_1}. \tag{3.12}$$

Proof. ($\Rightarrow$) : Let $M \in M_{C_2}$, by (3.9) we have $M \lesssim M_{2,\text{may}} \lesssim M_{1,\text{may}}$. By (3.10), we have $M_{1,\text{must}} \lesssim M_{2,\text{must}}$ and therefore $M_{1,\text{must}} \lesssim M$. This proves that M has property (3.8).
($\Leftarrow$) : First, we have $M_{2,\text{may}} \in M_{C_2} \subseteq M_{C_1} \implies M_{2,\text{may}} \lesssim M_{1,\text{may}}$. On the other hand, $M_{2,\text{must}} \in M_{C_2} \subseteq M_{C_1}$ and hence $M_{1,\text{must}} \lesssim M_{2,\text{must}}$. This shows that $C_2 \preceq C_1$. □

Propositions (3) and (4) immediately yield

Corollary 1. *Modal refinement and metatheoretic refinement are equivalent.*

Contract refinement allows us to compare levels of abstractions of contracts; for instance, a contract that involves details on how to perform local control actions may refine a contract for a car driving safely into a traffic intersection. The conjunction of two contracts C_1 and C_2 is defined as the greatest common lower bound (GCLB) of C_1 and C_2, or in other words, the most abstract contract C that refines both C_1 and C_2.

Definition 3.2.9 (Contract Conjunction)**.** Conjunction is defined for two modal interface contracts C_1 and C_2 if A_1 and A_2 are equal and have the same decomposition. Then the pre-conjunction $C_1 \underline{\wedge} C_2$ has states $S = S_1 \times S_2$, start state $s_{0,12} = s_{0,1} \times s_{0,2}$, and the same alphabet as C_1 and C_2, with transitions defined by the following relations, assuming for any subscript i, if a transition from q_i to q_i' does not exist, then we add it in with a `False` guard

$$(q_1, q_2) \xrightarrow{[g_1 \wedge g_2 | \alpha]} (q_1', q_2') \Leftrightarrow q_1 \xrightarrow{[g_1|\alpha]} q_1' \wedge q_2 \xrightarrow{[g_2|\alpha]} q_2', \tag{3.13}$$

$$q_1 \xrightarrow{[g_1|\alpha]} q_1' \vee q_2 \xrightarrow{[g_2|\alpha]} q_2' \Leftrightarrow (q_1, q_2) \xrightarrow{[g_1 \vee g_2|\alpha]} (q_1', q_2'). \tag{3.14}$$

A state (q_1, q_2) of $C_1 \underline{\wedge} C_2$ is illegal if it is inconsistent, that is, the "must implies may" condition in (3.5) does not hold. Specifically, if there exists $\alpha \in A$ such that $(q_1, q_2) \xrightarrow{[g_1|\alpha]} \wedge(q_1, q_2) \xrightarrow{[g_2|\alpha]}$ but $g_1 \not\Rightarrow g_2$, then we prune it by deleting all may transitions leading to (q_1, q_2); if (q_1, q_2) is a start state, it will simply get removed. Repeating this procedure and deleting all non-may reachable states yields the conjunction $C_1 \wedge C_2$ (note that the deletion must terminate because the number of states is finite).

Proposition 5. *$C_1 \wedge C_2$ has a start state if and only if C_1 and C_2 have a common lower bound.*

Proof. ($\Rightarrow$) : We prove $C_1 \wedge C_2 \preceq C_1, C_2$, by showing (3.9) and (3.10). Letting (q_1, q_2) be the start state of $C_1 \wedge C_2$, we have for $i \in \{1, 2\}$ and any (q_1', q_2') in $C_1 \wedge C_2$ such that $(q_1, q_2) \xrightarrow{[g_1' \wedge g_2'|\alpha]} (q_1', q_2')$, we have by (3.13) that $q_i \xrightarrow{[g_i'|\alpha]} q_i'$ and continuing inductively, we conclude $(q_1, q_2) \lesssim_{\text{may}} q_i$. Fixing i, for any q_i'', β such that $q_i \xrightarrow{[g_i''|\beta]} q_i''$, by (3.14), $(q_1, q_2) \xrightarrow{[g_1'' \vee g_2''|\beta]} (q_1'', q_2'')$ in $C_1 \underline{\wedge} C_2$. Since (q_1, q_2) is not illegal, so is (q_1'', q_2''), because otherwise, the may transition that performs β from (q_1, q_2) to (q_1'', q_2'') would have been deleted during pruning, violating (3.5) for (q_1, q_2). This shows that $q_i \lesssim_{\text{must}} (q_1, q_2)$.
($\Leftarrow$) : Suppose $C \preceq C_1, C_2$. Instead of showing that the start state of $C_1 \wedge C_2$ is not pruned, we will prove a stronger result, namely that $C \preceq C_1 \wedge C_2$. Indeed, if q is the start state of C, then by definition, the start state q_i of C_i for $i = 1, 2$ satisfies for $\alpha \in A$, $(q \xrightarrow{[g'|\alpha]} q' \Rightarrow q_i \xrightarrow{[g_i'|\alpha]} q_i') \wedge (g' \Rightarrow g_i') \wedge (q' \lesssim_{\text{may}} q_i')$. Thus $q \xrightarrow{[g'|\alpha]} q' \Rightarrow (q_1 \xrightarrow{[g_1'|\alpha]} q_1' \wedge q_2 \xrightarrow{[g_2'|\alpha]} q_2')$ with $g' \Rightarrow g_1' \wedge g_2'$ and $q' \lesssim_{\text{may}} q_i'$. By (3.9), we have $q \xrightarrow{[g'|\alpha]} q' \Rightarrow (q_1, q_2) \xrightarrow{[g_1' \wedge g_2'|\alpha]} (q_1', q_2')$ where

(q_1, q_2) and (q_1', q_2') are states of $C_1 \underline{\wedge} C_2$. Fixing i, for any β

$$q_i \xrightarrow{[g_i''|\beta]} q_i'' \Rightarrow q \xrightarrow{[g''|\beta]} q'' \wedge (g_i'' \Rightarrow g'') \wedge (q_i'' \lesssim_{\text{must}} q'').$$

Also by (3.14)

$$(q_1, q_2) \xrightarrow{[g_1'' \vee g_2''|\beta]} (q_1'', q_2'').$$

Since q is not illegal, there is an $\alpha \in A$ such that $\beta = \alpha$ and also $g'' \Rightarrow g'$; then by determinism $q' = q''$ and $q_i' = q_i''$. Clearly, $g_1'' \vee g_2'' \Rightarrow g'' \Rightarrow g' \Rightarrow g_1' \wedge g_2'$ so that the must transition from (q_1, q_2) to (q_1', q_2') doing β is also legal. Finally, continuing this chain of inductive reasoning, we obtain (3.9) and (3.10) for C and $C_1 \wedge C_2$, proving the claim. □

Proposition 5 and the stronger result shown in the reverse direction of its proof imply the following.

Proposition 6. *Modal conjunction and metatheoretic conjunction are equivalent, that is, the modal conjunction of two contracts is their GCLB.*

Definition 3.2.10 (Contract Composition)**.** *Contract composition* is denoted by the operator $\otimes$. The pre-composition $C_1 \underline{\otimes} C_2$ of two contracts C_1 and C_2 is given by

$$M_{1\underline{\otimes}2,must} = M_{1,must} \times M_{2,must},$$

$$M_{1\underline{\otimes}2,may} = M_{1,may} \times M_{2,may}.$$

A state (q_1, q_2) of $C_1 \underline{\otimes} C_2$ is illegal if one automaton attempts to supply an input, but the other refuses it. Furthermore, state (q_1, q_2) is illegal if it is impossible for the guards to match in input/output matching, resulting in the input being rejected. Define $SAT_{\mathcal{V}}$ to be the set of satisfiable predicates over $\mathcal{V}$. For $i \in \{1, 2\}$, assuming g_i is satisfiable, then (q_i, q_{3-i}) is illegal if there exists $\alpha_i \in A_i$ such that

$$(q_i \overset{[g_i|\alpha_i]}{\dashrightarrow} \wedge \alpha_i \in ?A_{3-i}) \nRightarrow q_{3-i} \xrightarrow{[g_{3-i}|\alpha_i]} \wedge (g_i \wedge g_{3-i} \in SAT_{\mathcal{V}}).$$

Pruning of states is done as in contract conjunction. This new contract is $C_1 \otimes C_2$. And we have the following result.

Proposition 7. *Modal composition is equivalent to metatheoretic contract composition.*

Proof. It suffices to show that, for $M_1 \in M_{C_1}$, $M_2 \in M_{C_2}$,

1. $M_1 \times M_2 \in M_{C_1 \otimes C_2}$.

2. For all $E \in E_{C_1 \otimes C_2}$, $E \times M_2 \in E_{C_1}$ and $E \times M_1 \in E_{C_2}$.

3. $C_1 \otimes C_2$ is the least contract with respect to refinement that satisfies these.

First we show that $C_1 \otimes C_2$ satisfies conditions 1 and 2. Let $C = C_1 \otimes C_2$. Then condition 1 is equivalent to $M_{\text{must}} \lesssim M_1 \times M_2 \lesssim M_{\text{may}}$. Since, for $i \in \{1, 2\}$, $M_i \in M_{C_i}$, $M_{i,\text{must}} \lesssim M_i \lesssim M_{i,\text{may}}$, the desired result immediately follows from Proposition 2. Next consider some $E \in E_{C_1 \otimes C_2}$, so $E \times (M_{1,\text{may}} \times M_{2,\text{may}})$ is closed. It follows that $(E \times M_{1,\text{may}}) \times M_{2,\text{may}}$ and $(E \times M_{2,\text{may}}) \times M_{1,\text{may}}$ are also closed. By definition, $M_1 \lesssim M_{1,\text{may}}$ and $M_2 \lesssim M_{2,\text{may}}$, so $(E \times M_1) \times M_{2,\text{may}}$ and $(E \times M_2) \times M_{1,\text{may}}$ are closed. It then remains to show that $E \times M_1$ and $E \times M_2$ satisfy (3.6) and (3.7) of Definition 3.2.7. First, since E is an environment of $C_1 \otimes C_2$, we have for any reachable state $(q_E, q_{1\otimes 2})$ in $E \times M_{1\otimes 2,\text{may}}$

$$q_E \xrightarrow{[g_E|!\alpha]} \Rightarrow q_{1\otimes 2} \xrightarrow{[g_{1\otimes 2}|?,\#\alpha]} \wedge(g_E \Rightarrow g_{1\otimes 2}),$$

$$q_E \xrightarrow{[g_E|?\alpha]} \Rightarrow q_{1\otimes 2} \xrightarrow{[g_{1\otimes 2}|!,\#\alpha]} \wedge(g_E \Rightarrow g_{1\otimes 2}).$$

Note that since $M_{1\otimes 2,\text{must}} = M_{1,\text{must}} \times M_{2,\text{must}}$, these are equivalent to

$$q_E \xrightarrow{[g_E|!\alpha]} \Rightarrow q_1 \xrightarrow{[g_1|\sim_1\alpha]} \wedge q_2 \xrightarrow{[g_2|\sim_2\alpha]} \wedge(g_E \Rightarrow g_1 \wedge g_2),$$

$$q_E \xrightarrow{[g_E|?\alpha]} \Rightarrow q_1 \xrightarrow{[g_1|\sim_1\alpha]} \wedge q_2 \xrightarrow{[g_2|\sim_2\alpha]} \wedge(g_E \Rightarrow g_1 \wedge g_2).$$

where $\sim_1$ and $\sim_2$ are action types such that their composition matches that of $C_1 \otimes C_2$. The following chart demonstrates possible action types of this transition.

E	C_1	C_2	$E \times M_{1,\text{must}}$	$E \times M_{2,\text{must}}$
!	?	?	#	#
!	#	#	#	#
!	#	?	#	#
!	#	!	#	!
?	!	!	#	#
?	#	#	#	#
?	#	!	#	#
?	#	?	#	?

The proof proceeds as follows. For M_1 implementing C_1 and M_2 implementing C_2, note that for $i \in \{1, 2\}$

$$q_i \xrightarrow{[g_i|?\alpha]} \Rightarrow q_{M_i} \xrightarrow{[g_{M_i}|?\alpha]} \wedge(g_i \Rightarrow g_{M_i}).$$

So in $E \times M_1$, state (q_E, q_{M_1}) is reachable if state (q_E, q_1) is reachable in $E \times C_{1,must}$. Consider the first row of the chart, where the environment is outputting α. Then from our result above, we have

$$q_E \xrightarrow{[g_E|!\alpha]} \Rightarrow q_{M_1} \xrightarrow{[g_{M_1}|?\alpha]} \wedge q_2 \xrightarrow{[g_2|?\alpha]} \wedge(g_E \Rightarrow g_{M_1} \wedge g_2),$$

so the composition of the transitions from q_E and q_{M_1} in $E \times M_1$ yields

$$q_{(E,M_1)} \xrightarrow{[g_E \wedge g_{M_1}|\#\alpha]} \Rightarrow q_2 \xrightarrow{[g_2|?\alpha]} \wedge(g_E \wedge g_{M_1} \Rightarrow g_2)$$

and the equivalent result for $E \times M_2$ yields

$$q_{(E,M_2)} \xrightarrow{[g_E \wedge g_{M_2}|!\alpha]} \Rightarrow q_1 \xrightarrow{[g_1|?\alpha]} \wedge(g_E \wedge g_{M_2} \Rightarrow g_1).$$

The latter result is precisely (3.7) with respect to $E \times M_2$ and C_1. It can be easily verified in a similar manner that the rest of the combinations yield similar results. For condition 3, it suffices to show that $C_1 \otimes C_2$ is the greatest lower bound of all contracts that satisfy conditions 1 and 2. Thus, for any C_* satisfying 1 and 2, then $C_1 \otimes C_2 \lesssim C_*$. This follows immediately from condition 1 and Proposition 2, since $M_1 \times M_2 \in M_{C_*}$ yields, as desired

$$\begin{aligned}
&M_{*,\text{must}} \lesssim M_1 \times M_2 \lesssim M_{*,\text{may}} \\
&\Rightarrow M_{*,\text{must}} \lesssim M_{1,\text{must}} \times M_{2,\text{must}} \lesssim M_1 \times M_2 \\
&\lesssim M_{1,\text{may}} \times M_{2,\text{may}} \lesssim M_{*,\text{may}} \\
&\Rightarrow M_{*,\text{must}} \lesssim M_{1\otimes 2,\text{must}} \lesssim M_{1\otimes 2,\text{may}} \lesssim M_{*,\text{may}}.
\end{aligned}$$

□

So far, we have only defined contract operations for contracts with matching alphabet conditions. Alphabet equalization is achieved via the same procedure described in (Benveniste, Benoît Caillaud, et al., 2015). May self-loops are temporarily added during the computation of the conjunction and must self-loops added for composition both having $\top$ as their guards.

3.3 An Application in Traffic Control

We will apply the developed theory to the contract-based design of the real-time networked control traffic system illustrated in Fig. 3.1. A full simulation of this system is available in (Phan-Minh, 2018). This system consists of 4 interacting components whose temporal behaviors are described by the contracts C_{lights}, $C_{pedestrian}$, $C_{vehicle}$, and $C_{scheduler}$ shown in Fig. 3.3. These specify the desired models for pedestrians, traffic lights, cars, and a scheduler in the intersection. We note that the many continuous variables involved in the timers and execution conditions of these components would have made producing and deciphering their contracts in the vanilla modal interface framework significantly more challenging due to the need for numerous potentially confounding auxiliary states and actions.

The traffic lights, in addition to some timing constraints on the duration of the "red," "green," "yellow" signals, are also required to have an "all red" phase that lasts for `t_c` seconds, a period long enough for cars to clear the intersection before the walk signal with duration `t_w` is turned on. Pedestrians should only attempt to cross when they are capable of successfully landing on the other island for the duration of the walk signal. All (or at least some) vehicles involved are robots that can be informed by a centralized planner on how to proceed past the intersection without causing accidents. These directions must be requested by the robots upon entrance. The traffic lights and the pedestrians form a subsystem $C_{lights} \otimes C_{pedestrian}$ that operates orthogonally to the subsystem $C_{vehicle} \otimes C_{scheduler}$ defined by the cars and the scheduler. By orthogonality, the overall system is simply $(C_{lights} \otimes C_{pedestrian}) \wedge (C_{vehicle} \otimes C_{scheduler})$.

To simplify the process of writing the interface contract for the traffic lights, we specify and compose two separate subcontracts for traffic lights in each direction, $C_{horizontal\_lights}$ and $C_{vertical\_lights}$, for the east-west and north-south directions respectively, that is $C_{lights} = C_{horizontal\_lights} \otimes C_{vertical\_lights}$. Since $C_{horizontal\_lights}$ and $C_{vertical\_lights}$ are symmetric with the exception of the start state (the former starts at node 0 while the latter starts at node 3), we only show the former in Fig. 3.3. The variables for $C_{horizontal\_lights}$ are `h`, `h'`, `h_timer`, which represent the traffic lights' current state, the next state after performing a related action and a special timer to specify the minimum durations to allow for vehicles to finish clearing the intersection `t_c` and for the walk signal `t_w`. The traffic light states are `r` for red, `y` for yellow, and `g` for green. The output actions are `!r_h` and `!h_walk` which serve to announce that the current state is red or that the walk sign for lanes in the

north-south directions is on. The input signal is `?r_v`, denoting a safety check with the state of the lights in the north-south direction. As can be seen in the automaton, via the may transition, we also allow the traffic lights to potentially bypass the yellow phase in transitioning from green to red. The contract $C_{pedestrian}$ is more simple. Its variable is `t_cross` which denotes the minimum time it takes the pedestrian to cross the street and the input actions are `?h_walk` and `?v_walk` which, in that order, denote a crossing action in the north-south and east-west directions of the pedestrian (both of these actions need to synchronize with a walk signal from the corresponding traffic lights). Note that both transitions in this contract are optional. The composition $C_{horizontal\_lights} \otimes C_{vertical\_lights}$ was computed automatically with the code in (Phan-Minh, 2018) and shown in Fig. 3.4. Though not included here to economize space, composing this with $C_{pedestrian}$ closes all the remaining output actions in Fig. 3.4.

The scheduler contract automaton has access to a variable `len(request_queue)`, which is the length of the request queue. In addition, $C_{scheduler}$ has one input action, `?request`, which denotes a check for whether there is a new request from a vehicle trying to travel through the intersection. Its output actions are `!reject` and `!accept` denoting whether the scheduler decides to accept or reject the request, and `!primitives` denoting the sending of controlling signals to the requesting vehicle. The internal action is `#processing`, corresponding to the internal computation of the controller. Observe that the scheduler must be able to accept requests under any condition (by the `True` guard), but can only process the request if the queue length is greater than 0. $C_{vehicle}$ has a variable `not_done` which keeps track of whether the original request has been carried out to completion. As can be expected, the car can make a request with `!request` and receive signals from the scheduler with the action `?reject`, `?accept`, and `?primitives`. Composing $C_{scheduler}$ with $C_{vehicle}$ yields the third system shown in Fig. 3.3. Observe that this system is also closed.

By Definition 3.2.6, checking that an implementation is a component whose interface satisfies the corresponding contract involves finding action equivalence maps between the implementation and the interface. To illustrate this process, consider the action of sending and receiving primitive commands of the scheduler and the vehicle, `!primitives` and `?primitives`. For a reasonable autonomous traffic intersection, the class of actions that qualify as the action `primitives` are those control signals that result in a safe and deadlock-free operation of all vehicles. We propose an implementation based on computing robust controllers or

"primitives" that can restrict the vehicles to a waypoint graph structure even in the presence of stochastic disturbance. In particular, the vehicle dynamics are given by $\dot{v} = a + w_1$, $\dot{\theta} = \frac{v}{L}\tan(\delta + w_2)$, $\dot{x} = v\cos(\theta)$, $\dot{y} = v\sin(\theta)$, with velocity v, orientation θ, positions x and y as state variables; acceleration $a \in [-9.8, 9.8]\frac{\mathrm{m}}{\mathrm{s}^2}$ and steering angle $\delta \in [-0.9, 0.9]\frac{\mathrm{rad}}{\mathrm{s}}$ as controllable inputs; $w_1 \in [-1.1, 1.1]\frac{\mathrm{m}}{\mathrm{s}^2}$ and $w_2 \in [-0.065, 0.065]\frac{\mathrm{rad}}{\mathrm{s}^2}$ as uncontrollable disturbances; and vehicle length $L = 2.8\mathrm{m}$.

We use a formal, set-based algorithm (Schürmann and Althoff, 2017a; Schürmann and Althoff, 2017b; Schürmann, Heß, et al., 2017) to obtain controllers that steer cars from one node to another on the waypoint graph with each node being a set of states of the car's dynamics around a nominal state. The reason a set of states is used is because of the disturbance present. This low-level controller ensures the satisfaction of input constraints and provides the occupancy sets of the vehicles, each of which represents a directed edge in the graph. The set-based controller computes a reference trajectory, a feedback controller to track this reference trajectory, and the corresponding reachable set of states. For any states p, q of the vehicle, let $\mathcal{X}_0(p)$ and $\mathcal{X}_f(q)$ denote the initial and final sets around p and q, respectively. By construction, the primitive controller steers in a fixed time $t_{1,2}$ from the initial set $\mathcal{X}_0(p_1)$ around a nominal waypoint p_1 to a final set $\mathcal{X}_f(p_2)$ around p_2. Constraining $\mathcal{X}_f(p_2) \subseteq \mathcal{X}_0(p_2)$ allows any trajectory in the edge that starts from $\mathcal{X}_0(p_1)$ and ends in $\mathcal{X}_f(p_2)$ to be concatenated with any trajectory starting in $\mathcal{X}_0(p_2)$. In this way, long chains of primitive commands that span multiple (directed) edges can be formed from unit commands spanning a single edge, which justifies treating the scheduling problem as a graph routing problem to which we propose Algorithm 1 as a solution. In Algorithm 1, the SCHEDULE($request\_queue, timetable$) function, taking two variables representing a queue of requests and a scheduling timetable is called repeatedly to rendezvous with new requests. Each time, it extracts the request from a certain car in the form of a starting configuration and an ending configuration. From this information, the scheduling algorithm finds a path in the primitive graph that connects these configurations and consults with the scheduling timetable to see if the path is safe and legal. If it is, the scheduler will send the primitives (each one of a fixed, known time length) to the requesting car; otherwise, to improve efficiency, it will attempt to find a safe and legal transit node along the path to temporarily send the car to. If such a node is found, it will send the corresponding primitives. If not, the request will be rejected. Proof details regarding the correctness of this algorithm mainly rely on the use of the timetable to avoid conflicts and illegal actions. Under

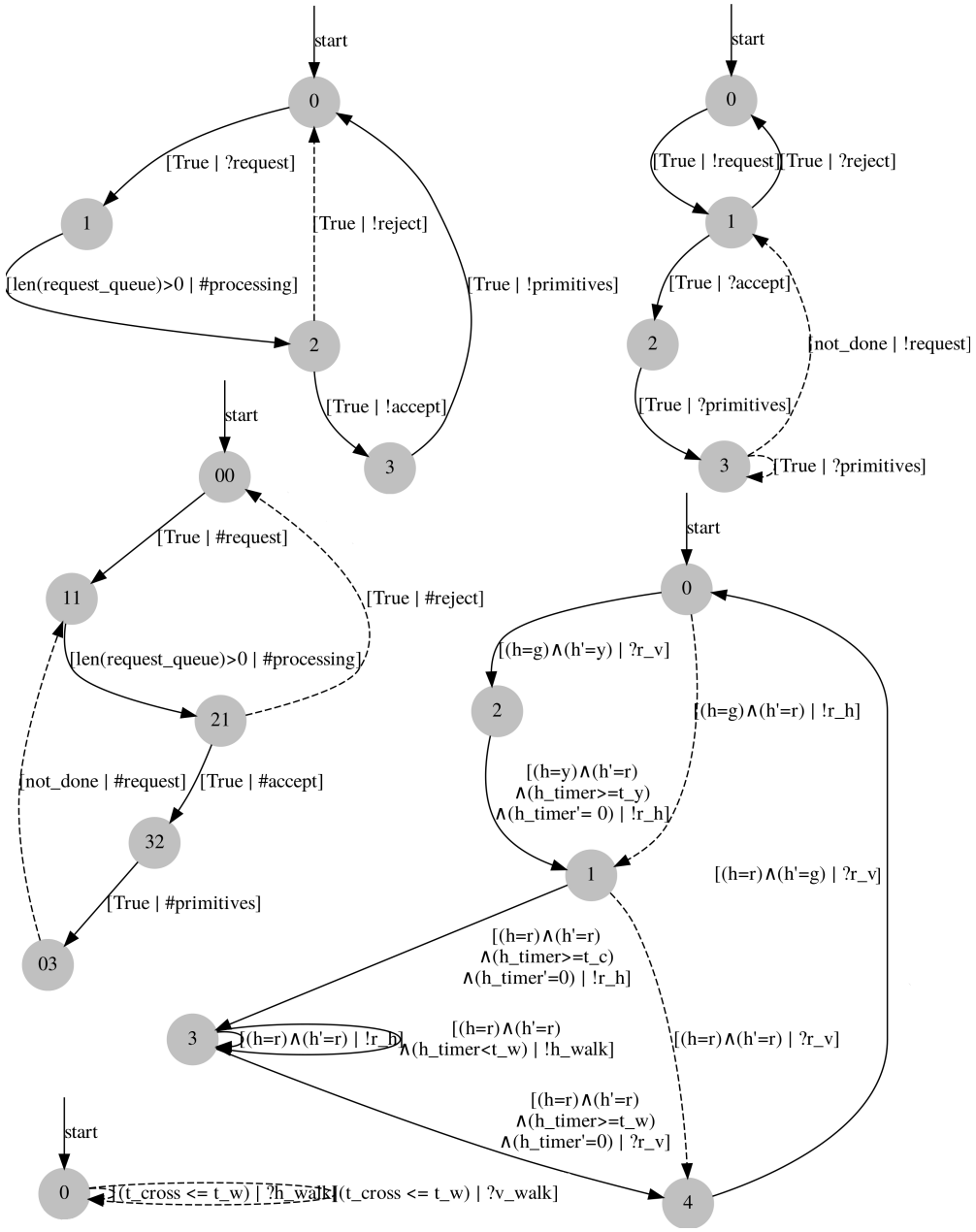


Figure 3.3: Reading from left to right, then top to bottom: $C_{scheduler}$, C_{car}, $C_{scheduler} \otimes C_{car}$, $C_{horizontal\_lights}$, $C_{pedestrian}$.

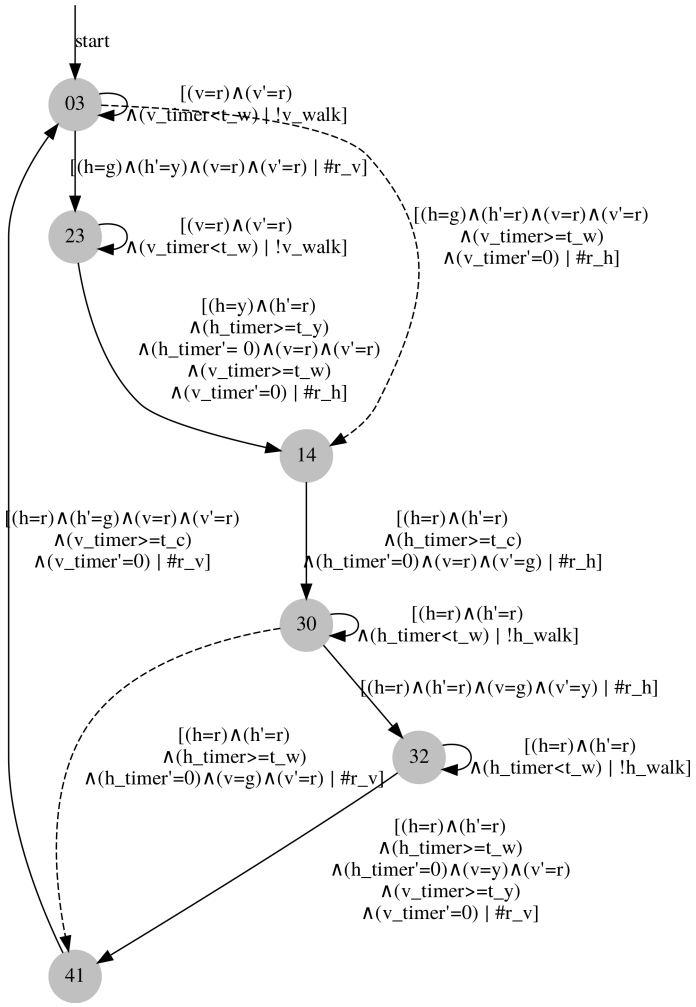


Figure 3.4: $C_{lights} = C_{horizontal\_lights} \otimes C_{vertical\_lights}$

this algorithm, `!primitives` and `?primitives` actions therefore correspond to the 2 SEND_PRIMITIVES$(\cdot)$ calls in the pseudocode.

Algorithm 1 The scheduling algorithm

function SCHEDULE($request\_queue, timetable$)
  $path, car \leftarrow$ EXTRACT_REQUEST($request\_queue$)
  if IS_SAFE($path, timetable$) **then**
    SEND_PRIMITIVES($path, car, timetable$)
  else if EXISTS_TRANSIT_NODE($path, timetable$) **then**
    $transit \leftarrow$ FIND_TRANSIT_PATH($path, timetable$)
    SEND_PRIMITIVES($transit, car, timetable$)
    $request\_queue$.ADD_LEG($path, car, transit$)
  else
    $request\_queue$.READD($path, car$)

3.4 Conclusion

In this chapter, we introduce a theory of guarded modal interface contracts that is compliant with the metatheory. We further demonstrated how this theory can be used to specify in a compact manner various components involved in a traffic intersection with variables taking on a continuous range of values. In Chapter 4, we will explore another contract-based design example with assume-guarantee contracts.

References

Alfaro, Luca de and Thomas A Henzinger (2001). "Interface Automata." In: *ESEC/FSE*, pp. 109–120.

Baldwin, Carliss Y. and Kim B. Clark (2006). "Modularity in the design of complex engineering systems." In: *Complex engineered systems*. Springer, pp. 175–205.

Benveniste, Albert, Benoit Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, et al. (Nov. 2012). *Contracts for System Design*. Research Report RR-8147. INRIA, p. 65. URL: https://hal.inria.fr/hal-00757488.

Benveniste, Albert, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, et al. (July 2015). *Contracts for Systems Design: Methodology and Application cases*. Research Report RR-8760. Inria Rennes Bretagne Atlantique ; INRIA, p. 63. URL: https://hal.inria.fr/hal-01178469.

Huang, Chun-Che and Andrew Kusiak (1998). "Modularity in design of products and systems." In: *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans* 28.1, pp. 66–77.

Larsen, Kim Guldstrand (1989). "Modal specifications." In: *International Conference on Computer Aided Verification*. Springer, pp. 232–246.

Lynch, Nancy A. and Mark R. Tuttle (1987). "Hierarchical correctness proofs for distributed algorithms." In: *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. ACM, pp. 137–151.

Phan-Minh, Tung (2018). *Traffic Intersection*. https://github.com/tungminhphan/traffic-intersection.

Raclet, Jean-Baptiste, Eric Badouel, Albert Benveniste, Benoit Caillaud, et al. (2009). "Modal interfaces: unifying interface automata and modal specifications." In: *Proceedings of the seventh ACM international conference on Embedded software*. ACM, pp. 87–96.

– (2011). "A modal interface theory for component-based design." In: *Fundamenta Informaticae* 108.1-2, pp. 119–149.

Schürmann, Bastian and Matthias Althoff (2017a). "Guaranteeing Constraints of Disturbed Nonlinear Systems Using Set-Based Optimal Control in Generator Space." In: *Proc. of the 20th IFAC World Congress*, pp. 12020–12027.

Schürmann, Bastian and Matthias Althoff (2017b). "Optimal Control of Sets of Solutions to Formally Guarantee Constraints of Disturbed Linear Systems." In: *Proc. of the American Control Conference*, pp. 2522–2529.

Schürmann, Bastian, Daniel Heß, Jan Eilbrecht, Stursberg, et al. (2017). "Ensuring Drivability of Planned Motions Using Formal Methods." In: *Proc. of the Intelligent Transportation Systems Conference*, pp. 1661–1668.

Veanes, Margus, Pieter Hooimeijer, Benjamin Livshits, et al. (Jan. 2012). "Symbolic Finite State Transducers: Algorithms and Applications." In: *SIGPLAN Not.* 47.1, pp. 137–150. ISSN: 0362-1340. DOI: 10.1145/2103621.2103674. URL: http://doi.acm.org/10.1145/2103621.2103674.

Chapter 4

DIRECTIVE-RESPONSE ASSUME-GUARANTEE CONTRACTS FOR AN AUTOMATED VALET PARKING SYSTEM[1]

4.1 Introduction

Formally guaranteeing safe and reliable behavior for modern cyber-physical systems has a scalability problem (Benveniste et al., 2018). Managing these highly complex architectures requires a design process that explicitly defines the dependencies and interconnections of system components to enable guaranteed safe behavior of the implemented system (Censi, 2015). Contract-based design reduces the complexity of the design and verification process by decomposing the system tasks into smaller tasks for the components to satisfy. From the composition of these components, overall system properties can be inferred or proved. Contract-based architectures have been demonstrated for several cyberphysical systems applications (Damm, Hungar, et al., 2011; Damm, Votintseva, et al., 2005; Nuzzo et al., 2013; Maasoumy, Nuzzo, and A. Sangiovanni-Vincentelli, 2015). Our goal here is to adapt and extend this framework to model a directive-response architecture on an automated valet parking system with the following features:

1. Discrete and continuous decision making components, which have to interact with one another.
2. Different components have different temporal requirements.
3. A natural hierarchy between the different components in our system that may be thought of as different layers of abstraction.
4. The system involves both human and non-human agents, the number of which is allowed to change over time.

One example of industry efforts to commercialize such a system is the automated valet parking system developed by Bosch in collaboration with Mercedes-Benz, which has been demonstrated in the Mercedes-Benz Museum parking garage in Stuttgart, Germany. Bosch and Daimler also later announced in 2020 that they

[1] The material in this chapter comes from joint work with Josefine B. Graebener and Richard M. Murray.

would set up a commercially operating AVP at the Stuttgart airport (Bosch, 2020a). Another commercial AVP system is planned to be set up by Bosch in downtown Detroit as a collaboration with Bedrock and Ford (Bosch, 2020b). Other examples include efforts by Siemens (Siemens, 2020) and DENSO (Yamazaki et al., n.d.). Our contributions include the formulation of a formal contract structure for an automated valet parking system with multiple layers of abstraction with a directive-response architecture for failure-handling. By implementing this system in Python, we aim to bridge the gap between the abstract contract metatheory and such non-trivial engineering applications. In addition, we incorporated error handling into the contracts and demonstrated the use of this architecture and approach towards writing specifications in the context of the automated valet parking example. Finally, we proved that the composed implementation satisfies the composite contract, adding this example of a large scale control system, involving a dynamic set of agents that are allowed to fail, to the small and slowly growing list of examples of formal assume-guarantee contract-based design.

4.2 Theoretical Background

Directive-Response Architecture

In a centralized approach for contingency management, recovery from failures is achieved by communicating with nearly every module in the system from a central module, hence increasing the system's complexity and potentially making it more error-prone (Wongpiromsarn and Murray, 2008). The Mission Data system (MDS), developed by JPL as a multi-mission information and control architecture for robotic exploration spacecraft, was an approach to unify the space system software design architecture. MDS includes failure handling as an integral part of the design (Dvorak et al., 2000; Ingham et al., 2005). It is based on the state analysis framework, a system engineering methodology that relies on a state-based control architecture and explicit models of the system behavior. Fault detection in MDS is executed at the level of the modules, which report if they cannot reach the active goal and possible recovery strategies. Resolving "expected" failures is one of the tasks the system was designed in advance to be capable of (Dvorak et al., 2000; Rasmussen, 2001). Another architecture based on the state analysis framework is the Canonical Software Architecture (CSA) used on the autonomous vehicle Alice by the Caltech team in the DARPA Urban Challenge in 2007. The CSA enables decomposition of the planning system into a hierarchical framework, respecting the different levels of abstraction at which the modules are reasoning, and the communication between

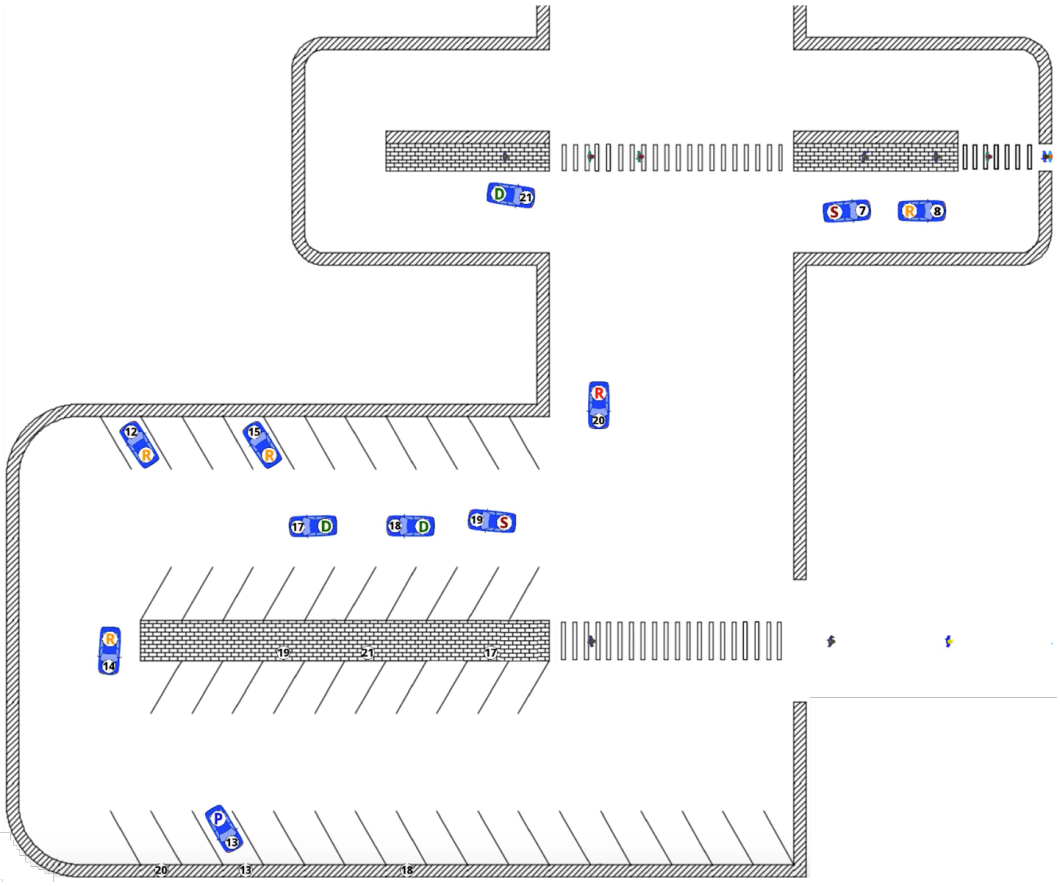


Figure 4.1: Snapshot from our AVP implementation showing human agents and vehicles as well as the parking lot topology.

the modules is via a directive-response framework (Burdick et al., 2007). This framework enables the system to detect and react to unexpected failure scenarios, which might arise from changes in the environment or hardware and software failures in the system (Wongpiromsarn and Murray, 2008). We are trying to capture the MDS and CSA approaches by incorporating directive-response techniques into a contract framework.

Directive-Response Contract Framework

We propose a contract-based design framework incorporating a directive-response architecture to enable reactivity to failures in the system. System components can be abstracted as *black boxes* constrained by assume-guarantee contracts that specify the behavior of the integrated system. Components communicate with one another by exchanging directives and responses, potentially acting according to a contingency plan that specifies how to react to possible failures. The higher module sends a

directive, and the lower module chooses its responses according to its status in achieving the directive's intended goal. The system components are composed to satisfy the overall system requirements while interacting with the environment, such as safety and liveness specifications.

4.3 Mathematical Formulation

Geometry

Definition 4.3.1 (Path)**.** A *path* is a continuous map $p : [0, 1] \to \mathbb{R}^2$. $p(0)$ is called the *start point* of p and $p(1)$ is called the *end point* of p. For each path p, let $p_h : [0, 1] \to (-180, 180]$ be such that $p_h(s)$ is the *heading angle* measured in degrees from the abscissa to $p'(s)$, the derivative vector of p with respect to s. For $t \in [0, 1]$, let $\tilde{p}(t)$ denote the element $p(t) \times p_h(t)$ of $\mathbb{R}^3$.

We will denote the set of all paths by $\mathbf{P}$ and, by abuse of notation, we will also use p to denote $p([0, 1])$, the image of $[0, 1]$ under p.

Definition 4.3.2 (Curvature feasibility)**.** Given $\kappa > 0$ and a path p, κ-*feasible*(p) is set to `True` if and only if p is twice differentiable on $[0, 1]$, and its curvature $\frac{|\det(p'(s), p''(s))|}{\|p'(s)\|^3} < \kappa$ for $s \in [0, 1]$.

Definition 4.3.3 (δ-corridor)**.** Let $\mathbb{B} \coloneqq \{\texttt{True}, \texttt{False}\}$. If $p \in \mathbf{P}$, and $\delta : \mathbf{P} \times [0, 1] \times \mathbb{R}^3 \to \mathbb{B}$ is such that the corresponding subset:

$$\Gamma_\delta(p) \coloneqq \bigcup_{s \in [0,1]} \Gamma_\delta(p, s),$$

where $\Gamma_\delta(p, s) \coloneqq \{(x, y, \theta) \in \mathbb{R}^3 \mid \delta(p, s, (x, y, \theta)) = \texttt{True}\}$ such that $\Gamma_\delta(p, s)$ is open and contains $\tilde{p}(s)$, then we say that $\Gamma_\delta(p)$ is a δ-*corridor* for p.

AVP World

Building Blocks

In this section, we will introduce naming symbols for objects that exist in the AVP world.

Definition 4.3.4 (AVP World)**.** The *AVP world* consists of the following:

1. A distinguished set of indexing symbols $\mathbf{T} := \{t, t', t'', ...\}$ denoting time.
2. A set of typed variables $\mathcal{U}$ to denote actions, states, channels, etc.

3. The following set of constants: $\mathbf{C}$, $\mathbf{G}$ where

    a) $\mathbf{C}$, a set of symbols, is called the customer set.

    b) $\mathbf{G}$, a set of symbols, is called the garage set containing the following constant values:

        i. $\mathbf{G}.drivable\_area \subseteq \mathbb{R}^3$, the set of configurations that vehicles are allowed to be in;

        ii. $\mathbf{G}.walkable\_area \subseteq \mathbb{R}^2$, the area that pedestrians are allowed to walk on;

        iii. $\mathbf{G}.entry\_configurations \subseteq \mathbb{R}^3$, a set of configurations that the customers can deposit their car in;

        iv. $\mathbf{G}.return\_configurations \subseteq \mathbb{R}^3$, a set of configurations that the car should be returned in;

        v. $\mathbf{G}.parking\_spots \in \mathbb{N}$, the number of parking spots available in the parking lot;

        vi. $\mathbf{G}.interior \subseteq \mathbb{R}^2$, the area inside the parking garage.

Directive-Response Message Types

Each channel in the system is associated with a unique message type. The following are all the message types in our AVP system.

1. $\mathbf{A}(\cdot)$, directive types:

    a) $\mathbf{A}(\textit{CustomerInterface}) \coloneqq \{\texttt{Park}, \texttt{Retrieve}\}$,

    b) $\mathbf{A}(\textit{Supervisor}) \coloneqq \mathbb{R}^6$,

    c) $\mathbf{A}(\textit{Planner}) \coloneqq \mathbf{P}$,

    d) $\mathbf{A}(\textit{Tracker}) \coloneqq \mathbf{I} \subseteq \mathbb{R}^2$, the set of all control inputs.

2. $\mathbf{B}(\cdot)$, response types:

    a) $\mathbf{B}(\textit{CustomerInterface}) \coloneqq \{\texttt{Failed}\}$,

    b) $\mathbf{B}(\textit{Supervisor}) \coloneqq \{\texttt{Rejected}, \texttt{Accepted}, \texttt{Returned}\}$,

    c) $\mathbf{B}(\textit{Planner}) = \mathbf{B}(\textit{Tracker}) \coloneqq \{\texttt{Blocked}, \texttt{Failed}, \texttt{Completed}\}$.

For each type $\mathbf{T}$, we will denote by $\tilde{\mathbf{T}}$ the product type $\mathbf{T} \times \mathbf{C}$ which will be used to associate a message of type $\mathbf{T}$ with a specific customer in $\mathbf{C}$. In addition, we will use $\mathbf{Id}$ to denote the set of message IDs.

Behavior

For each variable $u \in \mathcal{U}$, we denote by $\text{type}(u)$ the *type* of u, namely, the set of values that it can take. The types of elements of $\mathbf{T}$ are taken to be $\mathbb{R}_{\geq 0}$.

Definition 4.3.5 (Behavior)**.** Let Z be an ordered subset of variables in $\mathcal{U}$. A *Z-behavior* is an element of $\mathcal{B}(Z) := (\prod_{z \in Z} \text{type}(z))^{\mathbb{R}_{\geq 0}}$. Given $\sigma_Z \in \mathcal{B}(Z)$ and $\tau \in \mathbf{T}$, we will call $\sigma_Z(\tau)$ the *valuation* of Z at time τ. If $z \in Z$, we will also denote by $z(\tau)$ the value of z at time τ.

Note that each behavior in $Z \subseteq \mathcal{U}$ can be "lifted" to a set of behaviors in $\mathcal{U}$ by letting variables that are not contained in Z assume all possible values in their domains. Additionally, the set of behaviors $\mathcal{B}(Z)$ can be lifted to a set of behaviors in $\mathcal{B}(\mathcal{U})$ in a similar way. To ease notational burden for the reader, we will take the liberty of not explicitly making any reference to the "lifting" operation when they are in use unless there is any ambiguity that may result from doing so.

Definition 4.3.6 (Constraint)**.** A *constraint* k on a set of variables Z is a function that maps each behavior of Z to an element of $\mathbb{B}$, the Boolean domain. In other words, $k \in \mathbb{B}^{\mathcal{B}(Z)}$.

Note that by "lifting", a constraint on a set of variables Z is also a constraint on $\mathcal{U}$.

Definition 4.3.7 (Channel variables)**.** For each component X and another component Y, we can define two types of *channel variables*:

- $X_{\leftarrow Y}$, denoting an incoming information flow from Y to X,
- $X_{\rightarrow Y}$, denoting an outcoming information flow from X to Y.

In this work, we assume that $X_{\rightarrow Y}$ is always identical to $Y_{\leftarrow X}$. Each channel variable must have a well-defined message type and each message m has an ID denoted by $\text{id}(m) \in \mathbf{Id}$. If the message has value v, then we will denote it by $[v, \text{id}(m)]$, but we will often refer to it as $[v]$ whereby we omit the ID part to simplify the presentation. Intuitively, given a behavior, a channel variable x is a function that maps each time step to the message the associated channel is broadcasting at that time step.

Definition 4.3.8 (System)**.** A *system* M consists of a set of each of the following

1. internal variables/constants var_X^M,

2. output channel variables var_Y^M,
3. input channel variables var_U^M,
4. constraints con_M on $\text{var}_X^M \cup \text{var}_Y^M \cup \text{var}_U^M$.

A behavior of a system M is an element of the set of behaviors that correspond to $\text{var}_X^M \cup \text{var}_Y^M \cup \text{var}_U^M$ subject to con_M. This is denoted by $\mathcal{B}(M)$.

Directive-response

Before introducing directive-response systems, for any predicates A and B, we define the following syntax:

$$A \rightsquigarrow B \coloneqq \forall t :: A(t) \Rightarrow \exists t' \geq t :: B(t'). \qquad \text{(“leads to”)}$$

$$A \preceq B \coloneqq \forall t :: B(t) \Rightarrow \exists t' \leq t :: A(t'). \qquad \text{(“precedes”)}$$

$$\Box_{\geq t} A \coloneqq \forall t' \geq t :: A(t'). \qquad \text{(“always from } t\text{”)}$$

$$\text{starts\_at}(A, t) \coloneqq A(t) \wedge \forall t' < t :: \neg A(t'). \tag{4.1}$$

If M is a set-valued variable, then we define

$$\text{persistent}(M) \coloneqq \forall t :: \forall m :: m \in M(t) \Rightarrow \Box_{\geq t}(m \in M). \tag{4.2}$$

Definition 4.3.9 (Directive-response system)**.** A *directive-response system* M is a system such that for each output (resp., input) channel variable $chan$ there is an internal variable $send_{chan}$ (resp., $receive_{chan}$) whose domain is a collection of sets of messages that are of the type associated with $chan$. If $chan$ is an output channel variable, there is a causality constraint $k_{chan} \in \text{con}_M$ defined by

$$k_{chan} \coloneqq m \in send_c \preceq m = chan. \tag{4.3}$$

That is, a message must be sent before it shows in the channel. Otherwise if $chan$ is an input channel variable, then

$$k_{chan} \coloneqq m = chan \preceq m \in receive_{chan}. \tag{4.4}$$

Namely, a message cannot be received before it is broadcasted.

Definition 4.3.10 (Lossless directive-response system)**.** A lossless directive-response system is a directive-response system such that if $chan$ is an output channel, then

$$\text{persistent}(\text{send}_{chan}) \wedge (m \in \text{send}_{chan} \rightsquigarrow m = chan), \tag{4.5}$$

and if $chan$ is an input channel

$$\text{persistent}(\text{receive}_{chan}) \wedge (m = chan \rightsquigarrow m \in \text{receive}_{chan}). \tag{4.6}$$

Definition 4.3.11 (Assume-guarantee contracts for directive-response systems)**.** An assume-guarantee contract C for a directive-response system M consists of a pair of behaviors A, G of M and denoted by $C = (A, G)$. An environment for C is any set of all behaviors that are contained in A while an implementation of C is any set of behaviors that is contained in $A \Rightarrow G$. C is said to be saturated if the guarantee part satisfies $G = (\neg A \vee G) = (A \Rightarrow G)$.

Note that any contract can be converted to the saturated form without changing its sets of environments and implementations. The saturated form is useful in making contract algebra less cumbersome in general. If M is a system, then we say M satisfies C if $\mathcal{B}(M) \subseteq (A \Rightarrow G)$. Furthermore, the system composition $M_1 \times M_2$ of M_1 and M_2 is a system whose behavior is equal to $\mathcal{B}(M_1) \cap \mathcal{B}(M_2)$.

Definition 4.3.12 (Customer)**.** A *customer* is an element of $\mathbf{C}$. Corresponding to each $c \in \mathbf{C}$ is a set of $\mathcal{U}$ variables $\text{var}(c)$ that include $c.x$, $c.y$ (the coordinates of the customer him/herself), $c.car.x$, $c.car.y$, $c.car.\theta$ (the coordinates and heading of the customer's car), $c.car.healthy$, whether the car is healthy, $c.controls.v$, $c.controls.\varphi$ (the velocity and steering inputs to the vehicle), $c.car.\ell$ (the length of the car), $c.car.towed$ (whether the car is being towed). We will use the shorthand $c.car.state$ to mean the 3-tuple $(c.car.x, c.car.y, c.car.\theta)$.

For each behavior in $\mathcal{B}(\mathcal{U})$, we require each $c \in \mathbf{C}$ for which $c.car.towed$ is `False` to satisfy the following constraints that describe the Dubins car model:

$$\begin{aligned}
\frac{d(c.car.x)}{dt}(t) &= c.controls.v(t)\cos(c.car.\theta(t)) \\
\frac{d(c.car.y)}{dt}(t) &= c.controls.v(t)\sin(c.car.\theta(t)) \\
\frac{d(c.car.\theta)}{dt}(t) &= \frac{c.controls.v(t)}{c.car.\ell}\tan(c.controls.\varphi(t)).
\end{aligned} \tag{4.7}$$

Table 4.1: *CustomerInterface* directive-response system.

Internal variables/constants var_X	
C	The set of all customers in the AVP world.
Outputs var_Y	
$CustomerInterface_{\rightarrow Supervisor}$	An output channel of type $\tilde{\mathbf{A}}(CustomerInterface)$.
Inputs var_U	
$CustomerInterface_{\leftarrow Supervisor}$	An input channel of type $\tilde{\mathbf{B}}(Supervisor)$.
$CustomerInterface_{\leftarrow Tracker}$	An input channel of type $\tilde{\mathbf{A}}(Tracker)$.
Constraints con_M	
Vehicle dynamics	See (4.7)
Car and pedestrian limits	(4.8) and (4.9).

AVP System

By treating the *CustomerInterface* as an external component, the AVP system consists of three internal components: *Supervisor*, *Planner* , and *Tracker*. These systems are described below.

CustomerInterface

The environment in which the system shall operate consists of the customers and the pedestrians which we will call a *CustomerInterface*. A customer drops off the car at the drop-off location and is assumed to make a request for the parked car back from the garage eventually. The pedestrians are also controlled by the environment. When a pedestrian was generated by the environment, they start walking on the crosswalks. Pedestrians are confined to the pedestrian path, meaning they will not leave the crosswalk and walkway areas and their dynamics are continuous, meaning no sudden jumps. The cars move according to their specified dynamics. This includes a breaking distance depending on their velocity and maximum allowed curvature. For a formal description, refer to Table 4.1. Below are some constraints we impose on this module.

$$\begin{aligned}\forall c \in \mathbf{C} :: \square(v_{\min} \leq c.controls.v \wedge c.controls.v \leq v_{\max} \\ \wedge \varphi_{\min} \leq c.controls.\varphi \wedge c.controls.\varphi \leq \varphi_{\max})\end{aligned} \tag{4.8}$$

$$\forall c \in \mathbf{C} :: \forall s. \left\| \left(\frac{d(c.x)}{dt}(s), \frac{d(c.y)}{dt}(s) \right) \right\| \leq v_{ped,\max}. \tag{4.9}$$

Supervisor

A *Supervisor* component is responsible for the high level decision making in the process. It receives the *CustomerInterface*: requests and processes them by sending

Table 4.2: *Supervisor* directive-response system.

Internal variables/constants var_X	
$\mathbf{G}.*$	All $\mathbf{G}$ objects.
$num\_active\_customers$	The number of cars currently being served in the parking lot.
Outputs var_Y	
$Supervisor_{\rightarrow CustomerInterface}$	An output channel of type $\tilde{\mathbf{B}}(Supervisor)$.
$Supervisor_{\rightarrow Planner}$	An output channel of type $\tilde{\mathbf{A}}(Supervisor)$.
Inputs var_U	
$Supervisor_{\leftarrow CustomerInterface}$	An input channel of type $\tilde{\mathbf{A}}(CustomerInterface)$.
$Supervisor_{\leftarrow Planner}$	An input channel of type $\tilde{\mathbf{B}}(Planner)$.
Constraints con_M	
Parking lot topology	Any specific geometric constraints on $\mathbf{G}.*$.
Number of active customers	$num\_active\_customers$ must be equal to the number of cars that have been accepted but not yet left the parking lot.

Table 4.3: *Planner* directive-response system.

Interval variables/constants var_X	
$\mathbf{G}.*$	All $\mathbf{G}$ objects.
$\{c.car.x, c.car.y, c.car.\theta \mid c \in \mathbf{C}\}$	The configurations of all cars in AVP world.
κ	Maximum allowable curvature.
Outputs var_Y	
$Planner_{\rightarrow Supervisor}$	An output channel of type $\tilde{\mathbf{B}}(Planner)$.
$Planner_{\rightarrow Tracker}$	An output channel of type $\tilde{\mathbf{A}}(Planner)$.
Inputs var_U	
$Planner_{\leftarrow Supervisor}$	An input channel of type $\tilde{\mathbf{A}}(Supervisor)$.
$Planner_{\leftarrow Tracker}$	An input channel of type $\tilde{\mathbf{B}}(Tracker)$.
Constraints con_M	
Parking lot topology	Any specific geometric constraints on $\mathbf{G}.*$.
κ	Maximum allowable curvature given car dynamics and input constraints.

the appropriate directives to the *Planner* to fulfill a task. A *Supervisor* determines whether a car can be accepted into the garage or rejected. It also receives responses from the *Planner*. A *Supervisor* is to be aware of the reachability, the vacancy, and occupied spaces in the lot, as well as the parking lot layout. Formally, a *Supervisor* is a lossless directive-response system described by Table 4.2.

Planner

A *Planner* system receives directives from the *Supervisor* to make a car reach a specific location in the parking lot. A *Planner* system may have access to a planning graph determined from the parking lot layout, and thus can generate executable trajectories for the cars to follow. The *Planner* is aware of the locations of the agents and the obstacles in the parking lot from the camera system. A *Planner* is a lossless directive-response system described by Table 4.3.

Table 4.4: *Tracker* directive-response system.

Interval variables/constants var_X	
δ	Corridor map.
$\varepsilon_{\text{min},car}$	Minimum safety distance to other cars.
$\varepsilon_{\text{min},people}$	Minimum safety distance to pedestrians.
Outputs var_Y	
$Tracker_{\rightarrow Planner}$	An output channel of type $\tilde{\mathbf{B}}(Tracker)$.
$Tracker_{\rightarrow CustomerInterface}$	An output channel of type $\tilde{\mathbf{A}}(Tracker)$.
Inputs var_U	
$Tracker_{\leftarrow Planner}$	An input channel of type $\tilde{\mathbf{A}}(Planner)$.
Constraints con_M	
Corridor constraints	In our implementation, we define the δ-corridor for any path p to be the open set containing points whose distance to the closest point in p does not exceed 3 meters.
$\varepsilon_{\text{min},car}, \varepsilon_{\text{min},people}$	These values are determined based on the dynamics and the uncertainty Δ_{Car}.

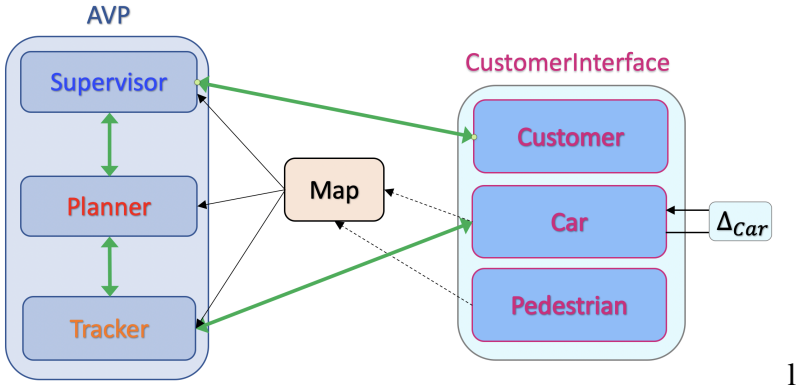


Figure 4.2: Contracts between the components of the AVP system.

Tracker

A *Tracker* system is responsible for the safe control of cars that are accepted into the garage by a *Supervisor*. It receives directives from a *Planner* consisting of executable paths to track and send responses based on the task status to a *Planner*. See Table 4.4.

4.4 AVP Contracts

In this section we will define the contracts for each of the modules in our system. These contracts are the guidelines for the implementation, and will be used to verify each of the components, as well as the composed system. In Figure 4.2, the green arrows represent directive-response assume-guarantee contracts, solid black arrows represent communication, and dashed black arrows represent passive information flow (observing movement of the agents). The Δ in the car component represents the possibility of failure and uncertainty.

Contract 1 ($C_{CustomerInterface}$)**.** The following is the contract for the *CustomerInterface*.

- Assumes
  - If the *CustomerInterface* sends a request to the *Supervisor*, then they will receive a response from the *Supervisor*:

$$\begin{aligned}\forall c \in \mathbf{C} :: ([m, c] \in \text{send}_{CustomerInterface_{\rightarrow Supervisor}} \rightsquigarrow \\ \exists r \in \mathbf{B}(Supervisor) :: [r, c] \in \text{receive}_{CustomerInterface_{\leftarrow Supervisor}}).\end{aligned} \tag{4.10}$$

  - If the car is healthy and accepted by the garage, it will be returned after being summoned:

$$\begin{aligned}\forall c \in \mathbf{C} :: (\square_{\geq 0}(c.car.healthy) \\ \wedge([\texttt{Accepted}, c] \in \text{receive}_{CustomerInterface_{\leftarrow Supervisor}} \wedge \\ [\texttt{Retrieve}, c] \in \text{send}_{CustomerInterface_{\rightarrow Supervisor}}) \rightsquigarrow \\ [\texttt{Returned}, c] \in \text{receive}_{CustomerInterface_{\leftarrow Supervisor}}).\end{aligned} \tag{4.11}$$

- Guarantees
  - When the request is accepted, the *CustomerInterface* should not tamper with the car controls until the car is returned (i.e., control signals should match the directive) :

$$\begin{aligned}\forall c \in \mathbf{C} :: \forall t :: \forall (v, \varphi) \in \mathbf{I} :: \\ ([\texttt{Accepted}, c] \in \text{receive}_{CustomerInterface_{\leftarrow Supervisor}}(t) \\ \wedge \neg([\texttt{Returned}, c] \in \text{receive}_{CustomerInterface_{\leftarrow Supervisor}}(t)) \Rightarrow \\ ([(v, \varphi), c] \in \text{receive}_{CustomerInterface_{\leftarrow Tracker}}(t) \\ \wedge \forall t' < t :: [(v, \varphi), c] \notin \text{receive}_{CustomerInterface_{\leftarrow Tracker}}(t') \Rightarrow \\ c.controls.v(t) = v \wedge c.controls.\varphi(t) = \varphi)).\end{aligned} \tag{4.12}$$

  - When the *CustomerInterface* is not receiving any new input signal, then it keeps the control inputs at zero:

$$\begin{aligned}\forall c \in \mathbf{C} :: \forall t :: \\ ([\texttt{Accepted}, c] \in \text{receive}_{CustomerInterface_{\leftarrow Supervisor}}(t) \\ \wedge \neg([\texttt{Returned}, c] \in \text{receive}_{CustomerInterface_{\leftarrow Supervisor}}(t)) \Rightarrow \\ (\forall (v, \varphi) \in \mathbf{I} :: [(v, \varphi), c] \in \text{receive}_{CustomerInterface_{\leftarrow Tracker}}(t) \\ \Rightarrow \exists t' < t :: [(v, \varphi), c] \in \text{receive}_{CustomerInterface_{\leftarrow Tracker}}(t')) \Rightarrow \\ c.controls.v(t) = 0 \wedge c.controls.\varphi(t) = 0))).\end{aligned} \tag{4.13}$$

– From sending a request until receiving a response, the car must stay in the deposit area:

$$\begin{aligned}\forall c \in \mathbf{C} :: \Box_{\geq 0}([\texttt{Park}, c] \in \text{send}_{CustomerInterface_{\rightarrow Supervisor}} \\ \wedge [\texttt{Accepted}, c] \notin \text{receive}_{CustomerInterface_{\leftarrow Supervisor}} \\ \wedge [\texttt{Rejected}, c] \notin \text{receive}_{CustomerInterface_{\leftarrow Supervisor}} \\ \Rightarrow c.car.state \in \mathbf{G}.entry\_configurations).\end{aligned} \tag{4.14}$$

– After the car is deposited, the customer will eventually summon it:

$$\begin{aligned}\forall c \in \mathbf{C} :: [\texttt{Accepted}, c] \in \text{receive}_{CustomerInterface_{\leftarrow Supervisor}} \rightsquigarrow \\ [\texttt{Retrieve}, c] \in \text{send}_{CustomerInterface_{\leftarrow Supervisor}}.\end{aligned} \tag{4.15}$$

– Pedestrians will only walk on "walkable" area:

$$\forall c \in \mathbf{C} :: \Box_{\geq 0}((c.x, c.y) \in \mathbf{G}.walkable\_area). \tag{4.16}$$

– Pedestrians will not stay on crosswalks forever:

$$\begin{aligned}\forall c \in \mathbf{C} :: ((c.x, c.y) \in \mathbf{G}.walkable\_area \cap \mathbf{G}.drivable\_area \\ \rightsquigarrow (c.x, c.y) \notin \mathbf{G}.walkable\_area \cap \mathbf{G}.drivable\_area).\end{aligned} \tag{4.17}$$

– If the car is not healthy and not towed, it cannot move:

$$\begin{aligned}\forall c \in \mathbf{C} :: \Box_{\geq 0}(\neg c.car.healthy \wedge \neg c.car.towed \Rightarrow \\ c.controls.v = 0 \wedge c.controls.\varphi = 0).\end{aligned} \tag{4.18}$$

– Sending a `Retrieve` message must always be preceded by receiving an `Accepted` message from the *Supervisor*:

$$\begin{aligned}\forall c \in \mathbf{C} :: [\texttt{Accepted}, c] \in \text{receive}_{CustomerInterface_{\leftarrow Supervisor}} \\ \preceq [\text{Retrieve}, c] \in \text{send}_{CustomerInterface_{\rightarrow Supervisor}}.\end{aligned} \tag{4.19}$$

– If a customer receives `Rejected` or `Returned` from the *Supervisor*, then they must leave the lot forever:

$$\begin{aligned}\forall c \in \mathbf{C} :: \forall t :: [\texttt{Rejected}, c] \in \text{receive}_{CustomerInterface_{\leftarrow Supervisor}}(t) \\ \vee [\texttt{Returned}, c] \in \text{receive}_{CustomerInterface_{\leftarrow Supervisor}}(t) \Rightarrow \\ \exists t' > t :: \Box_{\geq t'}((c.car.x, c.car.y) \notin \mathbf{G}.interior).\end{aligned} \tag{4.20}$$

Contract 2 ($C_{Supervisor}$)**.** The contract for the *Supervisor* is as follows.

- Assumes

– Towing eventually happens after the *Supervisor* is alerted of car failure:

$$\begin{aligned}\forall c \in \mathbf{C} :: \forall t :: [\texttt{Failed}, c] \in \text{receive}_{Supervisor_{\leftarrow Planner}}(t) \Rightarrow \\ \exists t' :: \Box_{\geq t'}(c.car.towed \wedge (c.car.x, c.car.y) \notin \mathbf{G}.interior).\end{aligned} \tag{4.21}$$

– If a car fails, then the *Planner* reports `Failed`:

$$\forall c \in \mathbf{C} :: \neg c.car.healthy \rightsquigarrow [\texttt{Failed}, c] \in \text{receive}_{Supervisor_{\leftarrow Planner}}. \tag{4.22}$$

– Cars making requests are deposited correctly by the customer:

$$\begin{aligned}\forall c \in \mathbf{C} :: \Box_{\geq 0}([\texttt{Park}, c] \in \text{receive}_{Supervisor_{\leftarrow CustomerInterface}} \\ \wedge([\texttt{Accepted}, c] \notin \text{send}_{Supervisor_{\rightarrow CustomerInterface}} \\ \vee [\texttt{Rejected}, c] \notin \text{send}_{Supervisor_{\rightarrow CustomerInterface}}) \\ \Rightarrow c.car.state \in \mathbf{G}.entry\_configurations).\end{aligned} \tag{4.23}$$

– If a car is healthy and summoned, then it will eventually appear at the return area and the *Planner* will send a `Completed` signal to the *Supervisor*:

$$\begin{aligned}\forall c \in \mathbf{C} :: (\Box_{\geq 0} c.car.healthy \wedge [\texttt{Retrieve}, c] \in \text{receive}_{Supervisor_{\leftarrow Planner}} \\ \rightsquigarrow ([\texttt{Completed}, c] \in \text{receive}_{Supervisor_{\leftarrow Planner}} \wedge \\ c.car.state \in \mathbf{G}.return\_configurations)).\end{aligned} \tag{4.24}$$

- Guarantees

– All requests from customers will be replied:

$$\begin{aligned}\forall c \in \mathbf{C} :: ([m, c] \in \text{receive}_{Supervisor_{\leftarrow CustomerInterface}} \rightsquigarrow \\ \exists r \in \mathbf{B}(Supervisor) :: [r, c] \in \text{send}_{Supervisor_{\rightarrow CustomerInterface}}).\end{aligned} \tag{4.25}$$

– The *Supervisor* cannot send a `Returned` message to the *CustomerInterface* unless it has received a `Completed` message from the *Planner* and the car is in the return area:

$$\begin{aligned}\forall c \in \mathbf{C} :: [\texttt{Completed}, c] \in \text{receive}_{Supervisor_{\leftarrow Planner}} \\ \wedge c.car.state \in \mathbf{G}.return\_configurations \preceq \\ [\texttt{Returned}, c] \in \text{send}_{Supervisor_{\rightarrow CustomerInterface}}.\end{aligned} \tag{4.26}$$

– If a car is healthy and a `Retrieve` message is received, then the last thing sent to the *Planner* should be a directive to the return area (the second configuration

should be one of the return configurations).

$$\begin{aligned}\forall c \in \mathbf{C} :: (\square_{\geq 0} c.car.healthy \Rightarrow [\texttt{Retrieve}, c] \in \text{receive}_{Supervisor_{\leftarrow Planner}} \wedge \\ \exists p_0, p_1 \in \mathbb{R}^3 :: [(p_0, p_1), c] \in \text{send}_{Supervisor_{\rightarrow Planner}} \wedge \forall p_0', p_1' \in \mathbb{R}^3 :: \\ [(p_0', p_1'), c] \in \text{send}_{Supervisor_{\rightarrow Planner}} \preceq [(p_0, p_1), c] \in \text{send}_{Supervisor_{\rightarrow Planner}} \\ \Rightarrow p_1 \in \mathbf{G}.return\_configurations.\end{aligned} \tag{4.27}$$

– If the car is healthy and if it is ever summoned, then the *Supervisor* will send a `Returned` message to its owner:

$$\begin{aligned}\forall c \in \mathbf{C} :: (\square_{\geq 0} c.car.healthy \Rightarrow \\ [\texttt{Retrieve}, c] \in \text{receive}_{Supervisor_{\leftarrow CustomerInterface}} \rightsquigarrow \\ [\texttt{Returned}, c] \in \text{send}_{Supervisor_{\rightarrow CustomerInterface}}).\end{aligned} \tag{4.28}$$

– If there is a not-yet-responded-to `Park` request and the parking lot capacity is not yet reached, then the *Supervisor* should accept the request:

$$\begin{aligned}\forall c \in \mathbf{C} :: \forall t :: \exists [\texttt{Park}, c] \in \text{receive}_{Supervisor_{\leftarrow CustomerInterface}}(t) \\ \wedge \forall t' \leq t :: [\texttt{Rejected}, c] \notin \text{send}_{Supervisor_{\rightarrow CustomerInterface}}(t') \\ \wedge [\texttt{Accepted}, c] \notin \text{send}_{Supervisor_{\rightarrow CustomerInterface}}(t') \\ \wedge num\_active\_customers(t) < \mathbf{G}.parking\_spots \\ \Rightarrow \exists t'' > t :: [\texttt{Accepted}, c] \in \text{send}_{Supervisor_{\rightarrow CustomerInterface}}(t'').\end{aligned} \tag{4.29}$$

– For every `Accepted` to or `Retrieve` from the *CustomerInterface* or `Blocked` from the *Planner*, the *Supervisor* sends a pair of configurations to the *Planner*, the first of which is the current configuration of the car and such that there exists a path of allowable curvature :

$$\begin{aligned}\forall c \in \mathbf{C} :: [\texttt{Accepted}, c] \in \text{send}_{Supervisor_{\rightarrow CustomerInterface}} \\ \vee [\texttt{Retrieve}, c] \in \text{receive}_{Supervisor_{\leftarrow CustomerInterface}} \\ \vee [\texttt{Blocked}, c] \in \text{receive}_{Supervisor_{\leftarrow Planner}} \rightsquigarrow \\ \exists k_0, k_1 \in \mathbb{R}^3 :: [(k_0, k_1), c] = Supervisor_{\rightarrow Planner} \\ \wedge k_0 = c.car.state \wedge \exists p \in \mathbf{P} :: \kappa\text{-feasible}(p) \\ \wedge \tilde{p}(0) = k_0 \wedge \tilde{p}(1) = k_1.\end{aligned} \tag{4.30}$$

Contract 3 ($C_{Planner}$)**.** The contract for the *Planner* is as follows:

- Assumes

– When the *Tracker* completes its task according to the corridor map δ, it should send a report to the *Planner*:

$$\begin{aligned} &\forall c \in \mathbf{C} :: \exists p \in \mathbf{P} :: \forall p' \in \mathbf{P} :: \\ &[p', c] \in \text{send}_{Planner_{\rightarrow Tracker}} \preceq [p, c] \in \text{send}_{Planner_{\rightarrow Tracker}} \wedge \\ &c.car.state \in \Gamma_\delta(p, 1) \rightsquigarrow [\texttt{Completed}, c] \in \text{receive}_{Planner_{\leftarrow Tracker}}. \end{aligned} \tag{4.31}$$

– If the *Tracker* sees a failure, it should report to the *Planner*:

$$\forall c \in \mathbf{C} :: \neg c.car.healthy \rightsquigarrow [\texttt{Failed}, c] \in \text{receive}_{Planner_{\leftarrow Tracker}}. \tag{4.32}$$

• Guarantees

– When receiving a pair of configurations from the *Supervisor*, the *Planner* should send a path to the *Tracker* such that the starting and ending configurations of the path match the received configurations, or if this is not possible, send `Blocked` to the *Supervisor*:

$$\begin{aligned} &\forall c \in \mathbf{C} :: \exists (p_0, p_1) \in \mathbb{R}^6 :: [(p_0, p_1), c] \in \text{receive}_{Planner_{\leftarrow Supervisor}} \\ &\rightsquigarrow (\exists p \in \mathbf{P} :: \tilde{p}(0) = p_0 \wedge \tilde{p}(1) = p_1 \wedge [p, c] \in \text{send}_{Planner_{\rightarrow Tracker}} \\ &\qquad \vee [\texttt{Blocked}, c] \in \texttt{send}_{Planner_{\rightarrow Supervisor}}). \end{aligned} \tag{4.33}$$

– Only send safe paths with κ-feasible curvature:

$$\begin{aligned} \forall c \in \mathbf{C} :: \exists p \in \mathbf{P} :: [p, c] \in \text{send}_{Planner_{\rightarrow Tracker}} \Rightarrow \\ \kappa\text{-feasible}(p) \wedge \Gamma_\delta(p) \subseteq \mathbf{G}.drivable\_area. \end{aligned} \tag{4.34}$$

– If receiving a task status update from the *Tracker*, eventually forward it to the *Supervisor*:

$$\begin{aligned} &\forall c \in \mathbf{C} :: [m, c] \in \text{receive}_{Planner_{\leftarrow Tracker}} \\ &\wedge m \in \{\texttt{Failed}, \texttt{Completed}\} \rightsquigarrow \\ &[m, c] \in \text{send}_{Planner_{\rightarrow Supervisor}}. \end{aligned} \tag{4.35}$$

– If the *Planner* receives a `Blocked` signal from the *Tracker*, it attempts to fix it, otherwise forwards it to the *Supervisor*:

$$\begin{aligned} &\forall c \in \mathbf{C} :: \forall t :: [\texttt{Blocked}, c] \in \text{receive}_{Planner_{\leftarrow Tracker}}(t) \Rightarrow \\ &\exists \varepsilon \in \mathbf{R}_{\geq 0} :: (\exists p \in \mathbf{P} :: [p, c] \in \text{send}_{Planner_{\rightarrow Tracker}}(t + \varepsilon) \wedge \\ &\qquad \forall t' < t + \varepsilon :: [p, c] \notin \text{send}_{Planner_{\rightarrow Tracker}}(t') \\ &\qquad \vee [\texttt{Blocked}, c] \in \text{send}_{Planner_{\rightarrow Supervisor}}(t + \varepsilon)). \end{aligned} \tag{4.36}$$

Contract 4 ($C_{Tracker}$)**.** The contract for the tracking component is as follows:

- Assumes
  - Any path command from the *Planner* is always κ-feasible, the corresponding corridor is drivable, and the car configuration upon receiving the command is in the initial portion of the corridor:

$$\forall c \in \mathbf{C} :: \forall p \in \mathbf{P} :: \forall t :: \text{starts\_at}([p, c] \in \text{receive}_{Tracker_{\leftarrow Planner}}, t) \wedge \kappa\text{-}feasible(p) \wedge c.car.state(t) \in \Gamma_\delta(p, 0) \wedge \Gamma_\delta(p) \subseteq \mathbf{G}.drivable\_area. \tag{4.37}$$

  - Commands are not modified by the *CustomerInterface*:

$$\text{See (4.12) and (4.13).} \tag{4.38}$$

- Guarantees
  - Make sure car stays in the latest sent p's corridor $\Gamma_\delta(p)$:

$$\begin{aligned} &\forall c \in \mathbf{C} :: \forall t :: \exists p \in \mathbf{P} :: \forall p' \in \mathbf{P} :: \\ &(([p, c] \in \text{receive}_{Tracker_{\leftarrow Planner}}(t) \wedge [p', c] \in \text{receive}_{Tracker_{\leftarrow Planner}}(t)) \Rightarrow \\ &([p', c] \in \text{receive}_{Tracker_{\leftarrow Planner}} \preceq [p, c] \in \text{receive}_{Tracker_{\leftarrow Planner}})) \Rightarrow \\ &c.car.state(t) \in \Gamma_\delta(p). \end{aligned} \tag{4.39}$$

  - Tracking command inputs are compatible with cars:

$$\begin{aligned} &\forall c \in \mathbf{C} :: \square_{\geq 0}([(v, \varphi), c] \in \text{send}_{Tracker_{\rightarrow CustomerInterface}} \Rightarrow \\ &v_{\min} \leq v \wedge v \leq v_{\max} \wedge \varphi_{\min} \leq \varphi \wedge \varphi \leq \varphi_{\max}). \end{aligned} \tag{4.40}$$

  - Never drive into a dynamic obstacle (customer or car):

$$\begin{aligned} &\forall c_1, c_2 \in \mathbf{C} :: \square_{\geq 0}((c_1 \neq c_2 \Rightarrow \\ &\|(c_1.car.x, c_1.car.y) - (c_2.car.x, c_2.car.y)\| \geq \varepsilon_{\min,car}) \wedge \\ &\|(c_1.car.x, c_1.car.y) - (c_2.x, c_2.y)\| \geq \varepsilon_{\min,people})). \end{aligned} \tag{4.41}$$

  - If a car fails, it must report to the *Planner*:

$$\forall c \in \mathbf{C} :: \neg c.car.healthy \rightsquigarrow [\texttt{Failed}, c] \in \text{send}_{Tracker_{\leftarrow Planner}}. \tag{4.42}$$

  - If a car is healthy, then it must "track" the last sent path from the *Planner*:

$$\begin{aligned} &\forall c \in \mathbf{C} :: \square_{\geq 0} c.car.healthy \wedge \exists p \in \mathbf{P} :: \forall p' \in \mathbf{P} :: \\ &[p', c] \in \text{receive}_{Tracker_{\leftarrow Planner}} \preceq [p, c] \in \text{receive}_{Tracker_{\leftarrow Planner}} \\ &\Rightarrow \exists t :: c.car.state(t) \in \Gamma_\delta(p, 1). \end{aligned} \tag{4.43}$$

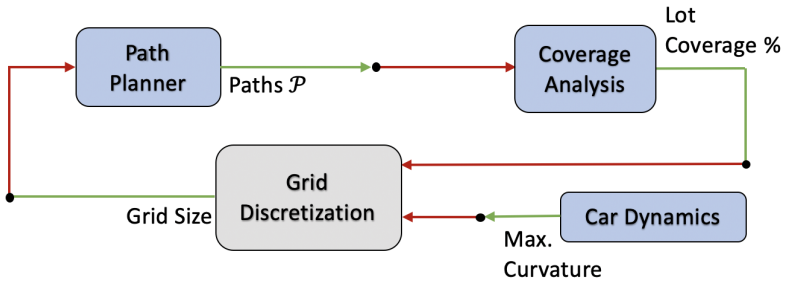


Figure 4.3: Implementation of the Planner component.

– When the *Tracker* completes its task according to a corridor map δ, it should send a report to the *Planner* module:

$$\text{See (4.31).} \tag{4.44}$$

– If a car is blocked (i.e., there is a failed car in its current corridor), then the *Tracker* must report `Blocked` to the *Planner*:

$$\begin{aligned} &\forall c \in \mathbf{C} :: \forall t :: \exists p \in \mathbf{P} :: ([p,c]\text{receive}_{Tracker_{\leftarrow Planner}}(t) :: \\ &\qquad \forall p' \in \mathbf{P} :: [p',c] \in \text{receive}_{Tracker_{\leftarrow Planner}}(t') \\ &\Rightarrow [p',c] \in \text{receive}_{Tracker_{\leftarrow Planner}} \preceq [p,c] \in \text{receive}_{Tracker_{\leftarrow Planner}}) \Rightarrow \\ &\quad (\exists c' \in \mathbf{C} :: c' \neq c \wedge \neg c'.car.healthy \wedge c'.car.state(t) \in \Gamma_\delta(p) \\ &\qquad \rightsquigarrow [\texttt{Blocked}, c] \in \text{send}_{Tracker_{\rightarrow Planner}}). \end{aligned} \tag{4.45}$$

4.5 System Design

Simulation Environment and Implementation

The proposed design framework was demonstrated via simulation of an automated valet parking (AVP) system (Graebener, 2020). It consists of the layout of a parking lot (Fig. 4.1), as well as multiple cars that arrive at the drop off location of the parking lot and are parked in one of the vacant spots by the AVP system. Once the customer requests their car, it is returned to the pick-up location. The asynchronicity is captured by modeling each component as a concurrent process using Python `async` library `Trio` (Smith, 2017). The communication between the layers is implemented using `Trio`'s `memory_channel`. In particular, each channel is a first-in-first-out queue which ensures losslessness. The architecture is described in Figure 4.2. In this setup, the cars may experience failures and report them to the *Tracker* module. The failures considered in this demonstration are a blocked path, a blocked parking spot, and a total engine failure resulting in immobilization. The benefit of the directive-response architecture becomes apparent when failures are introduced into

the system. Upon experiencing a failure, a component that is higher in the hierarchy will be alerted through the response it receives. If possible, the failure will be resolved, e.g., through the re-planning of the path or assigning a different spot. Every layer has access to its contingency plan, consisting of several predetermined actions according to the possible failure scenarios and corresponding responses it receives. In some cases (e.g., complete blockage of a car), when no action can resolve the issue, the cars have to wait until the obstruction is removed. We assume that only broken cars can be towed, and when a car breaks down, it will take a specified amount of time until it is towed.

***CustomerInterface* Modeling**

In our simulation, customers are responsible for driving their cars into the parking garage and depositing them at the drop-off area with an admissible configuration before sending a `Park` directive to the *Supervisor* and stay there until they get a response. This is satisfied as long as the customer drops off their vehicle behind the green line such that the heading of the vehicle is within the angle bounds $\underline{\alpha}$ and $\overline{\alpha}$ as shown in Figure 4.4 with the projection w of the vehicle onto the green edge of the blown-up entrance box shown in Figure 4.5. Therefore, *CustomerInterface* satisfies $G_{(4.14)}$. If the `Park` directive is `Rejected` by the *Supervisor*, the customer is assumed to be able to leave the garage safely (satisfying $G_{(4.20)}$). If the car is `Accepted`, then the customer will leave the control of the car to the *Tracker* (satisfying $G_{(4.12)}$ and $G_{(4.13)}$). The customer is assumed to always eventually send a `Retrieve` directive to the *Supervisor*, after their car is `Accepted` (satisfying $G_{(4.15)}$ and $G_{(4.19)}$). Once the vehicle is `Returned`, the customer is assumed to be able to pick it up and drive safely away. All pedestrians in the parking lot are customers, and they are constrained to only walk on the walkable area and never stay on a crosswalk forever (thus satisfying $G_{(4.16)}$ and $G_{(4.17)}$). When a car fails, it becomes immobilized until it is towed ($G_{(4.18)}$). From this, it follows that *CustomerInterface* satisfies $C_{CustomerInterface}$.

***Supervisor* Implementation**

At any time, the *Supervisor* knows the total number of cars that have been accepted into the garage, which is represented by the variable $num\_active\_customers$, and is designed to accept new cars when this number is strictly less than the total number of parking spots $\mathbf{G}.parking\_spots$. This implies that $G_{(4.29)}$ is satisfied. Overall, this ensures that all directives will get a response, yielding $G_{(4.25)}$. Whenever the

Supervisor receives a `Completed` signal, it will check if the car is in the return area. If it is, then the *Supervisor* will send a `Returned` signal to the *CustomerInterface* in compliance with $G_{(4.26)}$. If the *Supervisor* ever accepts a new car, or receives a `Blocked` signal from the *Planner*, or a `Retrieve` request, it will send a start configuration compatible with the car's current state as well as an end configuration to one of the parking spaces in the former case and to a place in the return area in the latter. This guarantees $G_{(4.30)}$.

Proposition 7. *$M_{Supervisor}$ satisfies $C_{Supervisor}$.*

Proof. Let M denote our implementation of the *Supervisor* and $\sigma \in M$. We want to show that

$$\sigma \in \bigwedge_{i=4.21}^{4.24} A_{(i)} \Rightarrow \sigma \in \bigwedge_{i=4.25}^{4.30} G_{(i)}.$$

From the description of the *Supervisor* implementation, we conclude that $\sigma \in G_{(4.25)} \wedge G_{(4.26)} \wedge G_{(4.27)} \wedge G_{(4.29)} \wedge G_{(4.30)}$. Since $\sigma \in A_{(4.24)}$ and because in our implementation whenever the *Supervisor* receives a `Completed` signal it will alert the customer of the corresponding status, our implementation satisfies $G_{(4.28)}$. □

Planner Implementation

The *Planner* computes paths that cover the parking spots, as well as the entry and exit areas of the parking garage, which are κ-feasible for a car that satisfies (4.7) such that the corresponding δ-corridor is on $\mathbf{G}.drivable\_area$. Given a maximum allowable curvature, a grid discretization scheme is based on a planning grid whose size is computed to provide full lot coverage and satisfy the curvature bounds, as depicted in Figure 4.3. For every specified grid size, the algorithm will check if the planning graph is appropriate by determining how well the parking lot is covered. Only a grid size that provides full coverage of the lot is chosen for path planning. The dynamical system specified in (4.7) is differentially flat (Fliess et al., 1995). In particular, it is possible to compute all states and inputs to the system, given the outputs x, y, and their (in this case, up to second order) derivatives. Specifically, the steering input is given by

$$\varphi(t) = \arctan(\ell\kappa(t)), \tag{4.46}$$

where $\kappa(t)$ is the curvature of the path traced by the midpoint of the rear axle at time t given by

$$\kappa(t) = \frac{\ddot{y}(t)\dot{x}(t) - \ddot{x}(t)\dot{y}(t)}{\left(\dot{x}^2(t) + \dot{y}^2(t)\right)^{\frac{3}{2}}}. \tag{4.47}$$

The task of tracking a given path can be shown to depend only on how $\varphi(t)$ is constrained. For practical purposes, let us assume $|\varphi(t)| \leq B$ for some $B > 0$. Then by Equation (4.46), tracking feasibility depends on whether the maximum curvature of that path exceeds $\frac{\tan(B)}{\ell}$. For our implementation, this is assumed to be $0.2\ m^{-1}$. This problem has been studied in (Cowlagi and Tsiotras, 2011) in the context of rectangular cell planning. We apply the algorithm described therein for a Type 1 path (CBTA-S1) to a rectangular cell while constraining the exit configuration to a heading difference of $\pm 5^\circ$ and a deviation of $\pm 0.5\ m$ from the nominal path. The setup and the resulting initial configuration, for which traversal is guaranteed, are shown in Figure 4.4 and Figure 4.5. The initial car configuration can be anywhere on the grid segment entry edge, as long as it is between the lower bound $\underline{\alpha}$ and the upper bound $\overline{\alpha}$. By passing through this initial funnel segment, the car will transition itself onto the planning grid. Therefore, it remains to be verified that each path generated from the grid is guaranteed to have a maximum curvature that is smaller than κ. An example path and its curvature are provided in Figure 4.5. Combining the parking lot coverage, initial grid segment traversability, and the curvature analysis, a grid size is determined to be $3.0\ m$ for the path planner, according to Figure 4.3. The synthesized grid size and path smoothing technique used in our *Planner* guarantee that all trajectories generated meet this maximum curvature requirement. In addition to satisfying $G_{(4.35)}$, any execution of the *Planner* also satisfies $G_{(4.33)}$ and $G_{(4.34)}$ because either the *Planner* can generate a feasible path or it will send a `Blocked` signal to the *Supervisor*. When the *Planner* receives a `Blocked` signal from the *Tracker*, it will either attempt to find a different path on the planning graph or report this to the *Supervisor*. This satisfies $G_{(4.36)}$.

Tracker Implementation

The *Tracker* receives directives from the *Planner* consisting of trackable paths and sends responses according to the task status to the *Planner*. The *Tracker* sees all agents in $\mathbf{G}.interior$ and guarantees no collisions by sending a brake signal when necessary to ensure a minimum safe distance is maintained at all times. The tracking algorithm that we use is an off-the-shelf MPC algorithm from (Sakai et al., 2018).

To ensure that the vehicles stay in the δ-corridors, given knowledge of the vehicle's dynamics, we can synthesize motion primitives that are robust to a certain disturbance set Δ_{Car} (see Figure 4.2). Algorithms for achieving this have been proposed and implemented, for example, in (Schürmann and Althoff, 2017) for nonlinear, continuous-time systems and for affine, discrete-time systems in (Filippidis et al.,

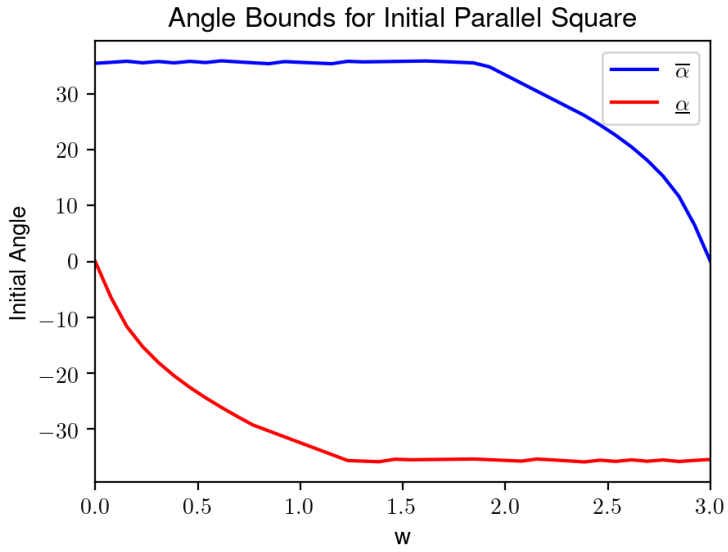


Figure 4.4: Possible initial car configuration along the entrance region (green) corresponding to a grid square as defined in Fig. 4.5.

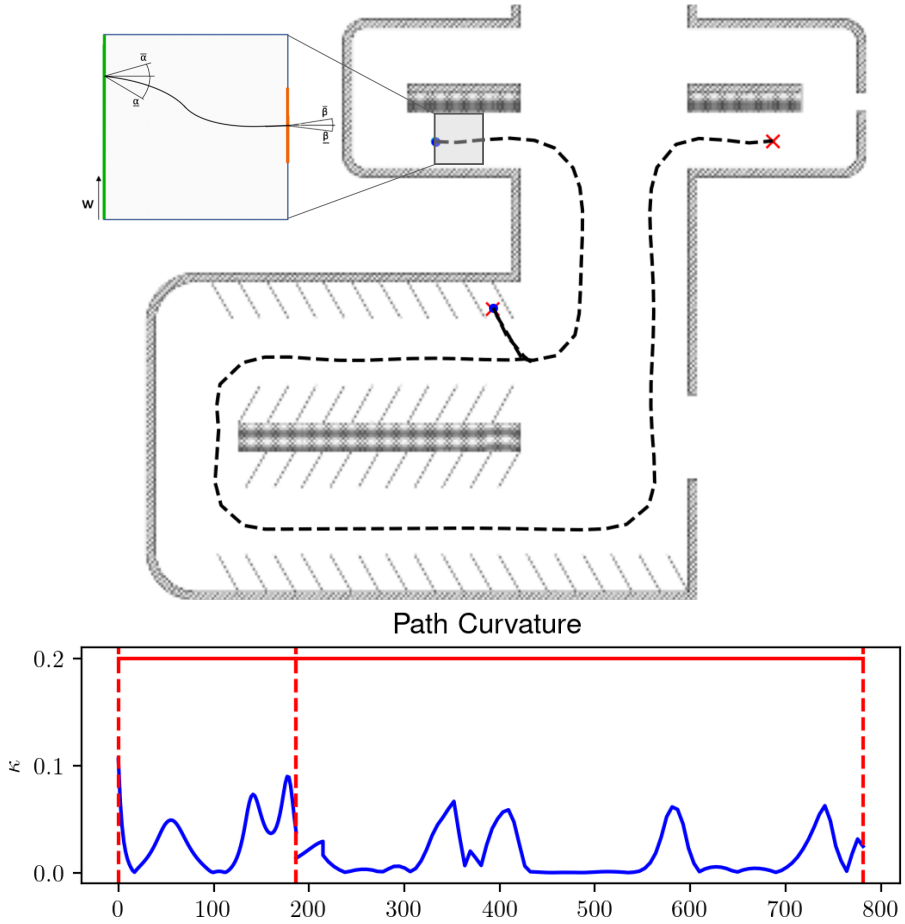


Figure 4.5: Example path through the parking lot and corresponding curvature and initial grid segment layout.

2016).

By $A_{(4.37)}$, any new path command $[p, c]$ sent down from the *Planner* module is assumed to be κ-feasible and have a drivable δ-corridor, the initial portion of which contains $c.car$ at that time. In our implementation, we ensure that every time this happens, $c.car$ is stationary. And under this condition, we were able to confirm by testing that a car controlled by the MPC algorithm can track the corresponding δ-corridors of a diverse enough set of paths, thus satisfying $G_{(4.39)}$ and $G_{(4.43)}$. The MPC algorithm is configured to output properly bounded control inputs, thus satisfying $G_{(4.40)}$. In addition, our implementation satisfies $G_{(4.42)}$, $G_{(4.44)}$, and $G_{(4.45)}$ by construction. And finally, we can guarantee $G_{(4.41)}$ by Property 4.48.

Correctness of the Composed System

As a final verification step, we will be taking the composition of the system contracts and showing that our overall system implementation satisfies this composition. This will imply that the composition is consistent.

Given two saturated contracts C_1 and C_2, their composition $C_1 \otimes C_2 = (A, G)$ given by (Benveniste et al., 2018):

$$G = G_1 \wedge G_2 \text{ and } A = \bigvee_{A' \in \mathcal{A}} A',$$

where

$$\mathcal{A} = \left\{ A' \;\middle|\; \begin{array}{c} A' \wedge G_2 \Rightarrow A_1 \\ \text{and} \\ A' \wedge G_1 \Rightarrow A_2. \end{array} \right\}$$

A nice property of the composed contract is that if M_1 satisfies C_1 and M_2 satisfies C_2, then $M_1 \times M_2$ satisfies $C_1 \otimes C_2$. Using the fact that the composition operator $\otimes$ is associative and commutative, a straightforward calculation yields the following more explicit form for the composition of N saturated contracts $(A_i, G_i)_{i=1}^N$.

$$G = \bigwedge_{i=1}^{N} G_i \text{ and } A = \bigwedge_{i=1}^{N} A_i \vee \neg\Big(\bigwedge_{i=1}^{N} G_i \Big)$$

If $A \neq \varnothing$, then the composed contract is compatible. The contract is consistent if there exists an implementation for it, namely $G \neq \varnothing$ if it is saturated. For our AVP system, we will show that our composed implementation also satisfies the composed contract in a non-vacuous way, meaning it satisfies all guarantees of the component contracts simultaneously. In the composition, an acceptable behavior satisfies the following properties:

1. If rejected, then by $G_{(4.20)}$, the car will have to leave the parking garage.

2. For any customer, if their car never fails, they can by $G_{(4.14)}$ send a `Park` request to the *Supervisor* while the car is at the deposit area, thus satisfying $A_{(4.23)}$ while waiting for a response from the *Supervisor*, which is guaranteed by $G_{(4.25)}$ and satisfies $A_{(4.10)}$. One of the following can happen:

    a) Accepted, no contingency: which is guaranteed to happen by $G_{(4.28)}$ if the parking lot capacity has not been reached. Then the *CustomerInterface* has to leave the control of the car to the *Tracker* by $G_{(4.12)}$ and $G_{(4.13)}$, thus satisfying $A_{(4.38)}$. After this, the *Supervisor* must send a directive in the form of a pair of configurations to the *Planner* $G_{(4.30)}$, which in turn must send to the *Tracker* a safe and feasible path (satisfying $A_{(4.37)}$) such that the starting and ending configurations of the path match the received configurations ($G_{(4.33)}$, $G_{(4.34)}$). Upon receiving the path from the *Planner*, the *Tracker* ensures that the car stays in the corridor of the path $G_{(4.39)}$ and ensures that it will make progress on that path (this satisfies $G_{(4.43)}$). It will accomplish this while sending compatible inputs to the customer's car $G_{(4.40)}$ and not driving it into people and other cars $G_{(4.41)}$. By $A_{(4.19)}$, the *CustomerInterface* may send a `Retrieve` command after being accepted by the *Supervisor*. The above process repeats with the *Supervisor*, which ensures that the last configuration is in the return area, thus satisfying $G_{(4.27)}$. If this is the last sent path, then upon reaching the end of the path, it should notify the *Planner* module that it has completed the task by $G_{(4.44)}$ which satisfies $A_{(4.24)}$, $A_{(4.29)}$, and $A_{(4.31)}$. The *Supervisor* alerts the *CustomerInterface* of the completed return by $G_{(4.26)}$, satisfying $A_{(4.11)}$. Then by $G_{(4.15)}$, the car will be picked up by the *CustomerInterface*.

    b) Accepted, with problems: If the car is accepted and at any time during the above process:

        i. If the car fails (hence, cannot move by $G_{(4.18)}$), the *Tracker* will send a `Failed` message to the *Planner* by $G_{(4.42)}$ satisfying $A_{(4.32)}$, and by $G_{(4.35)}$, this will be forwarded to the *Supervisor*. This satisfies $A_{(4.22)}$, which together with $A_{(4.21)}$, will imply that the failed car will eventually be towed.

        ii. If the car is `Blocked`, the *Tracker* will report to the *Planner* by $G_{(4.45)}$, which will try to resolve or alert the *Supervisor* satisfying

$G_{(4.36)}$.

We will show specifically that the composed system satisfies the following two properties ($G_{(4.41)}$ and $G_{(4.28)}$):

Property 1 (Safety)**.**

$$\begin{aligned} \forall c_1, c_2 \in \mathbf{C} :: \Box((c_1 \neq c_2 \Rightarrow \\ \|(c_1.car.x, c_1.car.y) - (c_2.car.x, c_2.car.y)\| \geq \varepsilon_{\min,car}) \wedge \\ \|(c_1.car.x, c_1.car.y) - (c_2.x, c_2.y)\| \geq \varepsilon_{\min,people})). \end{aligned} \tag{4.48}$$

Proof (Sketch). For each vehicle in the parking lot, the following invariance is maintained. There will be no collisions, as the *Tracker* checks the spatial region in front of the car and brings it to a full stop in case the path is blocked by another agent (car or pedestrian). The minimum distance to an obstacle is determined by a minimum braking distance. Furthermore, the environment does not take actions, which will lead to an inevitable collision due to the constraints on the pedestrian dynamics 4.8. □

Property 2 (Liveness)**.**

$$\begin{aligned} \forall c \in \mathbf{C} :: (\Box c.car.healthy \Rightarrow \\ [\texttt{Retrieve}, c] \in \text{receive}_{Supervisor_{\leftarrow CustomerInterface}} \rightsquigarrow \\ [\texttt{Returned}, c] \in \text{receive}_{CustomerInterface_{\leftarrow Supervisor}}). \end{aligned} \tag{4.49}$$

Proof (Sketch). Consider the parking lot topology shown in Figure 4.1. Let $c \in \mathbf{C}$ and $c.car.healthy$. Assume that c sends a `Retrieve` message to the *Supervisor* . For each t, let us define $f(t)$ to be the number cars between $c.car$ and its destination. Clearly, $f(t) \geq 0$ for any t and $f(t)$ is well-defined because for the topology being considered, we can trace out a line that starts from the entrance area, going to any one of the parking spots and ending at the return area without having to retrace our steps at any time. We will show that there exists a $t' \geq t$ such that $f(t') = 0$, implying that there is no longer any obstacle between c and its destination. Next, we claim that $\forall t, t' :: t' > t :: f(t) \geq f(t')$. This is true because:

- The parking lot topology and the safety measures do not allow for overtaking.
- The area reservation strategy implemented in the *Supervisor* prevents an increase in f upon re-routing to avoid a failed car. A notable detail is that

if $c.car$ is trying to back out of a parking spot, a stream of cars passing by can potentially block it forever. This is resolved by having $c.car$ reserve the required area so that once any other car has cleared this area, $c.car$ is the only one that has the right to enter it.

Finally, we will show that $\forall t :: \exists t' :: t' > t :: f(t) > f(t')$. Let c' be such that $c'.car$ is between $c.car$ and its destination. By the dynamical constraint on pedestrians and by assumptions $A_{(4.16)}$ and $A_{(4.17)}$, they will not block cars forever. Our algorithm guarantees that one of the following will happen at some time $t' > t$:

1. $c'.car$ is picked up by c'.
2. $c'.car$ is parked and $c.car$ drives past it
3. $c'.car$ drives past $c.car$'s destination.
4. $c'.car$ breaks down and by $A_{(4.21)}$ is eventually towed.

It is easy to see that each of these events implies that $f(t) > f(t')$. Since f is an integer and cannot drop below 0, the result follows. □

4.6 Conclusion

We have formalized an assume-guarantee contract variant with communication via a directive-response framework. We then used it to write specifications and verified the correctness of an AVP system implementation (Graebener, 2020). This was done separately for each module and everything together as a complete system. The application of this framework in the AVP can be extended to more agent types, for example, human-driven cars and pedestrians that do not necessarily follow traffic rules at all times. A contract between the valet driven cars and the human-driven cars will be needed to ensure the safe operation of the parking lot, and in the event that a human-driven car violates the contract, cars controlled by the system need to be able to react to this situation safely. More failure scenarios such as communication errors (message loss, cyberphysical attacks etc.) may also be included.

References

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

Bosch (2020a). *Automated valet parking service*. URL: https://www.bosch-mobility-solutions.com/en/products-and-services/passenger-cars-and-light-commercial-vehicles/automated-parking/automated-valet-parking/.

– (2020b). *Ford, Bedrock and Bosch are exploring highly automated vehicle technology in Detroit to help make parking easier*. URL: https://www.bosch-presse.de/pressportal/de/en/ford-bedrock-and-bosch-are-exploring-highly-automated-vehicle-technology-in-detroit-to-help-make-parking-easier-217984.html.

Burdick, Joel W., Noel du Toit, Andrew Howard, Christian Looman, Jeremy Ma, Richard M. Murray, and Tichakorn Wongpiromsarn (2007). *Sensing, navigation and reasoning technologies for the DARPA Urban Challenge*. Tech. rep. California Institute of Technology and Jet Propulsion Lab.

Censi, Andrea (2015). "A mathematical theory of co-design." In: *arXiv preprint arXiv:1512.08055*.

Cowlagi, Raghvendra V. and Panagiotis Tsiotras (2011). "Hierarchical motion planning with dynamical feasibility guarantees for mobile robotic vehicles." In: *IEEE Transactions on Robotics* 28.2, pp. 379–395.

Damm, Werner, Hardi Hungar, Bernhard Josko, Thomas Peikenkamp, and Ingo Stierand (2011). "Using contract-based component specifications for virtual integration testing and architecture design." In: *2011 Design, Automation & Test in Europe*. IEEE, pp. 1–6.

Damm, Werner, Angelika Votintseva, Alexander Metzner, Bernhard Josko, Thomas Peikenkamp, and Eckard Böde (2005). "Boosting re-use of embedded automotive applications through rich components." In: *Proceedings of Foundations of Interface Technologies*.

Dvorak, Daniel, Robert Rasmussen, Glenn Reeves, and Allan Sacks (2000). "Software architecture themes in JPL's Mission Data System." In: *2000 IEEE Aerospace Conference. Proceedings*. Vol. 7. IEEE, pp. 259–268.

Filippidis, Ioannis, Sumanth Dathathri, Scott C. Livingston, Necmiye Ozay, and Richard M Murray (2016). "Control design for hybrid systems with TuLiP: The temporal logic planning toolbox." In: *2016 IEEE Conference on Control Applications (CCA)*. IEEE, pp. 1030–1041.

Fliess, Michel, Jean Lévine, Philippe Martin, and Pierre Rouchon (1995). "Flatness and defect of non-linear systems: introductory theory and examples." In: *International Journal of Control* 61.6, pp. 1327–1361.

Graebener, Josefine (2020). *Automated Valet Parking Simulation*. URL: https://youtu.be/dtDz9zlj46w.

Ingham, Michel D., Robert D. Rasmussen, Matthew B. Bennett, and Alex C. Moncada (2005). "Engineering complex embedded systems with state analysis and the mission data system." In: *Journal of Aerospace Computing, Information, and Communication* 2.12, pp. 507–536.

Maasoumy, Mehdi, Pierluigi Nuzzo, and Alberto Sangiovanni-Vincentelli (2015). "Smart buildings in the smart grid: Contract-based design of an integrated energy management system." In: *Cyber Physical Systems Approach to Smart Electric Power Grid*. Springer, pp. 103–132.

Nuzzo, Pierluigi, Huan Xu, Necmiye Ozay, John B Finn, Alberto L Sangiovanni-Vincentelli, Richard M Murray, Alexandre Donzé, and Sanjit A Seshia (2013). "A contract-based methodology for aircraft electric power system design." In: *IEEE Access* 2, pp. 1–25.

Rasmussen, Robert D. (2001). "Goal-based fault tolerance for space systems using the Mission Data System." In: *2001 IEEE Aerospace Conference Proceedings (Cat. No. 01TH8542)*. Vol. 5. IEEE, pp. 2401–2410.

Sakai, Atsushi, Daniel Ingram, Joseph Dinius, Karan Chawla, Antonin Raffin, and Alexis Paques (2018). "Pythonrobotics: a python code collection of robotics algorithms." In: *arXiv preprint arXiv:1808.10703*.

Schürmann, Bastian and Matthias Althoff (2017). "Guaranteeing constraints of disturbed nonlinear systems using set-based optimal control in generator space." In: *IFAC-PapersOnLine* 50.1, pp. 11515–11522.

Siemens (2020). *Improving Autonomous Valet Parking with simulation and testing*. URL: https://blogs.sw.siemens.com/simcenter/autonomous-valet-parking-using-simulation-and-testing-for-safe-and-robust-system-and-algorithms-developments/.

Smith, Nathaniel J. (2017). "Trio: a friendly Python library for async concurrency and I/O." In: *https://trio.readthedocs.io/en/latest/, accessed 03/24/2020*.

Wongpiromsarn, Tichakorn and Richard M Murray (2008). "Distributed mission and contingency management for the DARPA urban challenge." In: *International Workshop on Intelligent Vehicle Control Systems (IVCS)*. Vol. 5.

Yamazaki, Akio, Yuki Izumi, Katsuya Yamane, Tetsuya Nomura, and Yasushi Seike (n.d.). *Development of Control Technology for Controlling Automated Valet Parking*. URL: https://www.denso-ten.com/business/technicaljournal/pdf/Vol03-03.pdf.

Chapter 5

REACTIVE CONTRACTS

5.1 Introduction

A premise central to formal methods is the idea that the model being verified is a correct description of the system in question. Oftentimes, especially for physical systems, this is not the case since adding too many details would make the model too large to be verified efficiently, if at all (Henzinger et al., 1998). Model uncertainties, environmental disturbances, simplifying assumptions etc. must be accounted for separately, often using heuristics. In the reactive synthesis setting (Bloem, 2015), attempts have been made to automatically generate systems that satisfy specifications with some measure of robustness to certain classes of uncertainties. For instance, (Livingston et al., 2013) exploits "locality" to compute modal μ-calculus fixpoints (Arnold and Niwinski, 2001) that enable patching system strategies in the presence of updated information about the game graph. Similarly, (Dathathri, Livingston, and Murray, 2017) discusses a way to recover from a finite number of unexpected actuation or sensing errors with pre-computed safe strategies that attempt to bring the system back to the nominal control trajectory. In addition to making specific assumptions on the environment, these methods also rely on the fact that such recovery strategies are always feasible for the original set of objectives. On the specification side of things, (Azzopardi, Pace, and Schapachnik, 2014) represents an elementary and direct means to add "reparation" handling to contract automata. Unfortunately, the obligation to define contract states and transitions explicitly does not make the automaton approach amenable to expressing and extracting complex properties.

Consider an *assume-guarantee* specification φ_S for a parking garage system S operated by robot valets of the form $\varphi_S := A \Rightarrow G$ (see (Phan-Minh, 2019b) for a concrete specification in TLA$^+$), where A encodes a set of input constraints or *assumptions* such as garage characteristics, quantity of valets, how fast they can park and retrieve cars, and G consists of *guarantees* the system must provide in the event that the constraints in A are satisfied, such as an upper bound on the maximum wait times for customers. Under this assume-guarantee framework, any system that satisfies the requirement

unless a constraint in A is violated, all guarantees in G are provided

is said to implement the contract. In practice, we tend to come up with an assumption-guarantee pair that is

1. *fragile*: suppose that a car has a flat tire and thereby invalidates the assumption on the performance of the valet assigned to it, are other valets suddenly allowed to indiscriminately abandon their responsibilities? According to φ_S, the answer is yes even though the "intuitively correct" answer is that they should not.

2. *underpromising*: the guarantee must often be very conservative when it must hold against worst case assumptions such as a single abnormally slow valet in the fleet that biases the worst wait time.

3. *maladaptive*: with only one assume-guarantee pair, the implementation does not have to adapt when the current environment "slightly" deviates from the assumption. Even if more pairs are specified and combined using, for example, the "contract conjunction" operator, there is no clear procedure on how to specify controlling or switching between specific assumptions and guarantees when such an opportunity arises.

To address these issues, we propose a contract formalism that explicitly takes into account the notion of uncertainty in the system being modelled and emphasizes its obligation to adapt to possible changes in the behavior of the environment. The structure of the chapter is as follows: in Section 5.2, basic notations are defined and a formal definition of assume-guarantee contracts is given as a point of reference. In Section 5.3, our theory of reactive contracts is presented and contrasted with the non-reactive formalism. In Section 5.4, we specialize this theory to the context of GR(1) games and derive a procedure for formulating and synthesizing their reactive contracts. In Section 5.5, we apply results from Section 5.4 to a concrete example/simulation and explore notions of optimal and robust reactivity.

5.2 Systems and Contracts

To keep things concise without losing generality, we will only cover assume-guarantee contracts defined over a common set of Boolean variables $\mathcal{V}$ called the *alphabet*. For any set X, let X^ω be the set of infinite sequences generated from X, namely $\{\langle x_i \rangle_{i=0}^{\infty} = \langle x_0, x_1, \ldots \rangle \mid x_i \in X\}$, X^* be the set of finite sequences, and

2^X be the powerset of X. Define X^∞ as $X^* \cup X^\omega$. In *implementations* a contract. accordance with the metatheory, we term any pair of collections of *environments* and *implementations* a contract. The theory of assume-guarantee contracts is a model of the metatheory and can be described as follows.

Definition 5.2.1 (Behaviors and Assertions)**.** A behavior σ is an element of $\mathcal{B} \coloneqq (2^{\mathcal{V}})^\omega$. An assertion A is a subset of $\mathcal{B}$, namely, $A \in 2^{\mathcal{B}}$.

We lift the set of all assertions $2^{\mathcal{B}}$ to a Boolean algebra by defining a unary operator $\neg$ and two binary operators $\wedge, \vee$ on it in the standard way: if A, A_1, A_2 are assertions, then $\neg A \coloneqq \mathcal{B} \setminus A$, $A_1 \vee A_2 \coloneqq A_1 \cup A_2$, $A_1 \wedge A_2 \coloneqq A_1 \cap A_2$. The induced partial ordering relation $\leq$ on $2^{\mathcal{B}}$ is simply the subset relation $\subseteq$. Additionally, we define a secondary binary operator $\Rightarrow$ in $A_1 \Rightarrow A_2$ as a shorthand for $\neg A_1 \vee A_2$.
A *component* M is an assertion designated as such. Via locality assumptions, the assertion characterizing a component is often restricted to a subset of $\mathcal{V}$ (with no constraints on variables outside the set). If M_1 and M_2 are components, the *interconnection* binary operator $\oplus$ is defined by $M_1 \oplus M_2 \coloneqq M_1 \wedge M_2$. That is, the set of behaviors of the interconnection consists only of those common to (or witnessed by) both implementations. We note that depending on how the variables and assertions making up the components being interconnected are defined, $\oplus$ can assume the meaning of either a parallel, series, coproduct, or feedback connection (Censi, 2015). In fact, contracts and the corresponding algebra can be used to constrain components satisfying them so that the meaning of their interconnection will be clear.

Definition 5.2.2 (Contracts)**.** An *assume-guarantee contract* C is a pair of assertions (A, G), called the *assumption* and the *guarantee* respectively. The set of environments of C, denoted by $\mathcal{E}_C$, captures all components E such that

$$E \leq A. \tag{5.1}$$

In other words, $\mathcal{E}_C = 2^A$. The set of implementations of C, denoted by $\mathcal{M}_C$, consists of all components M such that

$$\forall E \in \mathcal{E}_C . M \oplus E \leq G. \tag{5.2}$$

Example 5.2.1. Let $\mathcal{V} = \{x, y\}, C = (A, G)$ where $A \coloneqq \{\sigma \mid x \in \sigma_i \Leftrightarrow i \mod 2 = 0\}$, $G \coloneqq \{\sigma \mid y \in \sigma_i \Leftrightarrow i \mod 2 \neq 0\}$, $\sigma_1 \coloneqq \langle \{x\}, \{y\}, \{x\}, \emptyset, \{x\}, \emptyset, \ldots \rangle$ and $\sigma_2 \coloneqq \langle \{x\}, \{y\}, \{x\}, \{y\}, \ldots \rangle$. If $E \coloneqq \{\sigma_1, \sigma_2\}$, then $E \in \mathcal{E}_C$ because $\sigma_1, \sigma_2 \in A$.

Let $M_1 \coloneqq \{\sigma_1\}$ and $M_2 \coloneqq \{\sigma_2\}$. Then $M_1 \notin \mathcal{M}_C$ because $M_1 \oplus E = \{\sigma_1\} \notin G$. However, one can check that $M_2 \in \mathcal{M}_C$. Note that if $M_3 \coloneqq \varnothing$, then $M_3 \in \mathcal{M}_C$ as well. The interpretation here is that M_3 satisfies the assume-guarantee semantics of C vacuously.

Since $2^{\mathcal{B}}$ is a Boolean algebra, we infer from inequality (5.2) and the definition of $\oplus$ that M is an implementation if $\forall E \in \mathcal{E}_C . M \leq \neg E \vee G$. Choosing $E = A$ in inequality (5.1) gives $M \leq \neg A \vee G$. Conversely, satisfying $M \leq \neg A \vee G$ implies that M is an implementation because $\neg E \vee G$ is antitone in E. Thus, we have the following proposition.

Proposition 8. *Given $C = (A, G)$, a component M satisfies $M \in \mathcal{M}_C$ if and only if*

$$M \leq A \Rightarrow G. \tag{5.3}$$

Proposition 8 characterizes implementations M of C as those components whose behaviors either do not conform to the behaviors specified in A or are compatible with G. Specifically, it says that $\mathcal{M}_C = 2^{A \Rightarrow G}$. Furthermore, since

$$A \Rightarrow (A \Rightarrow G) = \neg A \vee (\neg A \vee G) = \neg A \vee G = A \Rightarrow G, \tag{5.4}$$

inequalities (5.1) and (5.3) yield the following proposition.

Proposition 9. *If $C = (A, G)$ and $C^* = (A, A \Rightarrow G)$, then $\mathcal{M}_C = \mathcal{M}_{C^*}$ and $\mathcal{E}_C = \mathcal{E}_{C^*}$.*

It may be seen from equation (5.4) why any assume-guarantee contract of the form $C = (A, A \Rightarrow G)$ is called *saturated*. By the metatheory, we will consider contracts that have the same sets of environments and implementations to be equal, and so by Proposition 9, every contract C has a unique saturated *canonical* form C^*. This saturated form makes contract algebra more convenient and sheds light on the meaning of the conjunction operation, which motivates our development of "reactive contracts." To describe the conjunction, we will need the idea of contract refinement whose definition is repeated here for ease of reference.

Definition 5.2.3. We say contract $C_1 = (A_1, G_1)$ refines a contract $C_2 = (A_2, G_2)$ and write $C_1 \preceq C_2$ if $\mathcal{M}_{C_1} \subseteq \mathcal{M}_{C_2}$ and $\mathcal{E}_{C_2} \subseteq \mathcal{E}_{C_1}$.

The *conjunction* of two contracts C_1 and C_2, denoted by $C_1 \wedge C_2$, is a contract that is their largest lower bound (or meet) with respect to $\preceq$. For any contract C, we

have $\mathcal{E}_C = 2^A$ and $\mathcal{M}_C = 2^{A \Rightarrow G}$. Therefore, if $C \leq C_1, C_2$, then by Definition 5.2.3, $\mathcal{M}_C \subseteq \mathcal{M}_{C_1} \cap \mathcal{M}_{C_2} = 2^{A_1 \Rightarrow G_1} \cap 2^{A_2 \Rightarrow G_2} = 2^{(A_1 \Rightarrow G_1) \wedge (A_2 \Rightarrow G_2)}$ and $2^{A_1 \vee A_2} = 2^{A_1} \cup 2^{A_2} = \mathcal{E}_{C_1} \cup \mathcal{E}_{C_2} \subseteq \mathcal{E}_C$ (since the intersection/union of powersets of two sets is the powerset of their intersection/union). By the fact that $(A_1 \Rightarrow G_1) \wedge (A_2 \Rightarrow G_2)$ is saturated, we have $C_1 \wedge C_2 = (A_1 \vee A_2, (A_1 \Rightarrow G_1) \wedge (A_2 \Rightarrow G_2))$. By induction, we can conclude the following:

Proposition 10. *If for $i = 1, 2, \ldots, n$, C_i are assume-guarantee contracts, then*

$$\wedge_{i=1}^{n} C_i = (\vee_{i=1}^{n} A_i, \wedge_{i=1}^{n} A_i \Rightarrow G_i). \tag{5.5}$$

Note that we can apply an analogous argument to the disjunction of contracts (defined as their join) to conclude that the set of all saturated contracts forms a complete lattice. Equation (5.5) shows that the "parametric contract" formalism given in (Kim, Arcak, and Seshia, 2017) is exactly the result of applying the conjunction operation to the constituent contracts. That is, if M is an implementation of the conjunction $\wedge_i C_i$ containing σ such that $\sigma \in E$ where E is an environment of $\wedge_i C_i$, then there exists at least a $k \in \{1, 2, \ldots, n\}$ such that $\sigma \in A_k$ and for all such k, $\sigma \in G_k$. In other words, for any behavior σ in which the environment satisfies any assumption A_k in $\{A_1, A_2, \ldots, A_n\}$, the system must *react* by providing the corresponding guarantee G_k. Thus in contract conjunctions, 1) the reactions are defined by pairing each A_k with the corresponding G_k, and 2) $A_k \Rightarrow G_k$ must hold over the sequence σ in its entirety. The second restriction is partially relaxed in the "dynamic contract" formalism, used for instance in (Kim, Sadraddini, et al., 2017), where assumptions are allowed to change over fixed time intervals. Our reactive contract framework will 1) remove the one-to-one restriction to allow for a more flexible assumption-guarantee pairing process, 2) enforces *immediate* guarantee reactions to assumption changes *directly* on each element $\sigma \in \mathcal{B}$, and 3) enables automated synthesis.

5.3 Reactive Contracts

Reactivity

For each $\sigma \in \mathcal{B}$, and $i, j \in \mathbb{N}_0 : i < j$, we denote by $\sigma_{i \to j}$ the subsequence of σ spanning from σ_i up to *but not including* σ_j, namely $\langle \sigma_i \rangle_{k=i}^{j-1}$, and by $\sigma_{i \to \infty}$ the infinite sequence $\langle \sigma_k \rangle_{k=i}^{\infty}$. For $A \subseteq \mathcal{B}$ and $k \geq 0$, denote by $\text{Pref}_k(A)$ the set of all prefixes of behaviors in A of length k, $\text{Pref}_k(A) \coloneqq \{\sigma_{0 \to k} \mid \sigma \in A\}$ and $\text{Pref}(A) \coloneqq \cup_{k=1}^{\infty} \text{Pref}_k(A)$, the set of all prefixes of A. Let $\cdot$ be the standard string

sequence concatenation operator mapping from $(\text{Pref}(\mathcal{B})\times\text{Pref}(\mathcal{B}))\cup(\text{Pref}(\mathcal{B})\times\mathcal{B})$ to $\text{Pref}(\mathcal{B})\cup\mathcal{B}$.

Definition 5.3.1 (Witness)**.** Let $\sigma\in\mathcal{B}, A\subseteq\mathcal{B}$ and $i,j\in\mathbb{N}_0\cup\{\infty\}: i<j$. We say that σ is a *witness* for A from i up until j and write $\sigma\models_{i\to j}A$ if $\sigma_{i\to j}\in\text{Pref}(A)\cup A$. If $j\neq\infty$, we consider the witness relation as being *strict* and write $\sigma\models^s_{i\to j}A$ if $\sigma\models_{i\to j}A$, but $\sigma\not\models_{i\to j+1}A$. If $j=\infty$, the witness relation is always strict.

To describe and keep track of assumption changes, we appeal to the notion of assigning *signatures* (or labels) to each behavior that undergoes those changes.

Definition 5.3.2 (Signature)**.** Given a set of assertions $\mathcal{A}\subseteq 2^{\mathcal{B}}$, an $\mathcal{A}$-signature is any nonempty assertion sequence $\alpha=\langle\alpha_k\rangle_{k=0}^m\in\mathcal{A}^\infty$ where $m\in\mathbb{N}\cup\{\infty\}$. If $m<\infty$, we say $\sigma\in\mathcal{B}$ is a witness for α and put $\sigma\models\alpha$ if there exists a partitioning sequence $\langle i_k\rangle_{k=0}^m$ in $\mathbb{N}_0$ satisfying $0=i_0<i_1<\ldots<i_m$ such that with $i_{m+1}:=\infty$, we have

$$\forall k\in\{0,1,\ldots,m\}.\sigma\models_{i_k\to i_{k+1}}\alpha_k. \tag{5.6}$$

We say that σ is a strict witness for the signature α and write $\sigma\models^s\alpha$ if the witness relation in equation (5.6) is strict. Analogously, if $m=\infty$, then $\sigma\models\alpha$ if there exists a (strictly monotone) partitioning sequence $\langle i_k\rangle_{k=0}^\infty$ in $\mathbb{N}_0$ satisfying $i_0=0$ and equation (5.6) with $\{0,1,\ldots,m\}$ replaced by $\mathbb{N}_0$.

In general, a given $\sigma\in\mathcal{B}$ may be a strict witness for more than one signature in $\mathcal{A}^\infty$. For example, if $\mathcal{A}=\{A_1,A_2\}$ where $A_1\cap A_2\neq\varnothing$, then any behavior $\sigma\in A_1\cap A_2$ satisfies $\sigma\models^s\langle A_j\rangle$ for $j\in\{1,2\}$. This may still be the case even when $A_1\cap A_2=\varnothing$. For $\mathcal{V}=\{x,y\}$, $A_1=\{\langle\{x\}\rangle_{i=0}^\infty\}$ and $A_2=\{\langle\{y\}\rangle_{i=0}^\infty\}\cup\{\sigma\}$ where σ satisfies $\sigma_k=\{x\}$ for $k=0$ and $\sigma_k=\{y\}$, otherwise. Then $\sigma\models^s\langle A_1,A_2\rangle$ and $\sigma\models^s\langle A_2\rangle$. This non-uniqueness makes it unclear as to which assumption change sequence should be considered and how/when to properly react to it. Being able to restrict the set of assumptions so that this does not happen is necessary because in order to react at all, the system must be able to consistently detect which assumption to operate under next. The following proposition gives a necessary and sufficient condition.

Proposition 11. *Let $\mathcal{A}$ be a collection of assertions, then*

$$\forall\alpha,\beta\in\mathcal{A}^\infty.\forall\sigma\in\mathcal{B}.(\sigma\models^s\alpha\wedge\sigma\models^s\beta)\Rightarrow\alpha=\beta, \tag{5.7}$$

if and only if

$$\forall A_1, A_2 \in \mathcal{A}. A_1 \neq A_2 \Rightarrow \mathit{Pref}_1(A_1) \cap \mathit{Pref}_1(A_2) = \emptyset. \tag{5.8}$$

Proof. First, assume that $\mathcal{A}$ does not satisfy Formula (5.8). Let A_1, A_2 be such that $A_1 \neq A_2$ and $\mathrm{Pref}_1(A_1) \cap \mathrm{Pref}_1(A_2) \neq \emptyset$. Observe that for any $k \in \mathbb{N}_0$ and $\sigma \in \mathcal{B}$, if $\sigma_{0\to k+1} \in \mathrm{Pref}_{k+1}(A_1) \cap \mathrm{Pref}_{k+1}(A_2)$ then $\sigma_{0\to k} \in \mathrm{Pref}_k(A_1) \cap \mathrm{Pref}_k(A_2)$. Hence, either $A_1 \cap A_2 \neq \emptyset$, in which case (5.7) clearly does not hold, or there exists a $k' \geq 1$ such that $\mathrm{Pref}_{k'}(A_1) \cap \mathrm{Pref}_{k'}(A_2) \neq \emptyset$ and $\mathrm{Pref}_{k'+1}(A_1) \cap \mathrm{Pref}_{k'+1}(A_2) = \emptyset$. Let $\sigma' \in \mathrm{Pref}_{k'}(A_1) \cap \mathrm{Pref}_{k'}(A_2)$ and $\sigma_1 \in A_1$, $\sigma_2 \in A_2$ be such that $\sigma_{10\to k'} = \sigma_{20\to k'} = \sigma'$. If for all $0 < i < k'$, we have $\sigma'_i = \sigma'_0$, then $\langle \sigma'_0 \rangle_{i=0}^{\infty} \models^s \langle A_1 \rangle$ or $\langle \sigma'_0 \rangle_{i=0}^{\infty} \models^s \langle A_1 \rangle_{i=0}^{\infty}$ while $\langle \sigma'_0 \rangle_{i=0}^{\infty} \models^s \langle A_2 \rangle$ or $\langle \sigma'_0 \rangle_{i=0}^{\infty} \models^s \langle A_2 \rangle_{i=0}^{\infty}$. On the other hand, if there is an $0 < k < k'$ such that $\sigma'_k \neq \sigma'_0$, then for $\sigma'' \coloneqq \sigma'_{0\to k+1} \cdot \sigma'_{0\to k+1} \cdot \ldots$, we have $\sigma'' \models^s \langle A_1 \rangle_{i=0}^{\infty}$ and $\sigma'' \models^s \langle A_1 \rangle_{i=0}^{\infty}$. Both of these cases contradict Formula (5.7).
For the other direction, assume that $\mathcal{A}$ satisfies (5.8). Let $\alpha, \beta \in \mathcal{A}^{\infty}$ and $\sigma \in \mathcal{B}$ be such that $\sigma \models^s \alpha$ and $\sigma \models^s \beta$. Let $m \in \mathbb{N} \cup \{\infty\}$ be the length of α. By the fact that $\sigma_0 \in \mathrm{Pref}_1(\alpha_0) \cap \mathrm{Pref}_1(\beta_0)$ and $\forall A \in \mathcal{A}. A \neq \alpha_0 \Rightarrow \mathrm{Pref}_1(A) \cap \mathrm{Pref}_1(\alpha_0) = \emptyset$, we conclude that $\alpha_0 = \beta_0$. Suppose that up to $n < m$, $\alpha_k = \beta_k$ for all k satisfying $0 \leq k \leq n$. We will show that α_{n+1} and β_{n+1} are defined and equal to one another. Indeed, since $n < m$, the $(n+1)^{\text{th}}$ term of α, α_{n+1}, exists. Let $\langle i_k \rangle_{k=0}^{m}$ be a partitioning sequence for $\sigma \models^s \alpha$ given by Definition 5.3.2. By strictness and the induction hypothesis, we have $\sigma_{i_n \to (i_{n+1}+1)} \not\models \alpha_n = \beta_n$. Since $\sigma \models^s \beta$, it follows that β_{n+1} exists as well. From $\sigma \models^s \alpha$, if $n+2 \leq m$, we have $\sigma \models^s_{i_{n+1} \to i_{n+2}} \alpha_{n+1}$ and, in particular, $\sigma_{i_{n+1}} \in \mathrm{Pref}_1(\alpha_{n+1}) \cap \mathrm{Pref}_1(\beta_{n+1})$, which by Formula (5.8) yields $\alpha_{n+1} = \beta_{n+1}$. If $n+2 > m$, then $\sigma \models^s_{i_{n+1} \to \infty} \alpha_{n+1}$, arguing similarly, we arrive at the additional conclusion that β also has length m. This implies Formula (5.7). □

Any $\mathcal{A}$ that satisfies (5.8) is called *initially disjoint*. Hence, Proposition 11 says that $\mathcal{A}$ is a set of assertions that are initially disjoint if and only if any behavior is a strict witness for at most one $\mathcal{A}$-signature. Let $\mathcal{B}_{\mathcal{A}} \coloneqq \{\sigma \in \mathcal{B} \mid \exists \alpha \in \mathcal{A}^{\infty}. \sigma \models^s \alpha\}$, the set of behaviors that have $\mathcal{A}$-signatures. If $\mathcal{A}$ is initially disjoint, then the function $\mathcal{U}_{\mathcal{A}} : \mathcal{B}_{\mathcal{A}} \to \mathcal{A}^{\infty}$ mapping each $\sigma \in \mathcal{B}_{\mathcal{A}}$ to the unique signature $\mathcal{U}_{\mathcal{A}}(\sigma) \in \mathcal{A}^{\infty}$ for which it is a witness is well-defined. Lastly, for any $M \subseteq \mathcal{B}_{\mathcal{A}}$, we denote by $\mathcal{U}_{\mathcal{A}}(M)$ the set of signatures generated by M, namely, $\{\mathcal{U}_{\mathcal{A}}(\sigma) \mid \sigma \in M\}$.

Contracts

Definition 5.3.3 (Reactive contracts)**.** A reactive assume-guarantee contract C is a 4-tuple $(\mathcal{A}, \mathcal{G}, \Delta, R)$ where

1. $\mathcal{A}, \mathcal{G} \subseteq 2^{\mathcal{B}}$ are called the *assumption* and *guarantee sets*, respectively. $\mathcal{A}$ is required to be initially disjoint.
2. $\Delta \subseteq \mathcal{A}^{\infty}$ is called the *contingency set*, consisting of assumption change scenarios that may happen.
3. $R \subseteq (\mathcal{A} \times \mathcal{G})^{\infty}$ is called the *reaction set*.

Observe that $\mathcal{A}$ and $\mathcal{G}$ are not necessarily of the same cardinality and that from each $r \in R$, we can obtain a unique $\mathcal{A}$-signature by "projecting away the $\mathcal{G}$ dimension." We denote the projection function by $\Pi_{\mathcal{A}} : R \to \mathcal{A}^{\infty}$ so that $\Pi_{\mathcal{A}}(\langle A_k, G_k \rangle_{k=0}^{m}) := \langle A_k \rangle_{k=0}^{m}$ for any $\langle A_k, G_k \rangle_{k=0}^{m} \in R$.

Definition 5.3.4 (Environment)**.** An *environment* for $C = (\mathcal{A}, \mathcal{G}, \Delta, R)$ is any $E \subseteq \mathcal{B}_{\mathcal{A}}$ such that $\mathcal{U}_{\mathcal{A}}(E) \subseteq \Delta$, namely each $\sigma \in E$ is a strict witness for some $\mathcal{A}$-signature in Δ.

Thus, for a reactive contract, assumptions about its environment's behaviors are allowed to change according to the contingency specified in Δ. As these assumptions change, the system should provide the corresponding guarantees as specified by the reaction set R. We characterize R via the following definitions.

Definition 5.3.5 (Reactive satisfaction)**.** Let $\sigma \in \mathcal{B}$, $r = \langle (A_k, G_k) \rangle_{k=0}^{m} \in R$. We say that σ *reactively satisfies* r and write $\sigma \models^{\rho} r$ if the following hold

1. $\sigma \models^{s} \Pi_{\mathcal{A}}(r)$ with the partitioning sequence $\langle i_k \rangle_{k=0}^{m}$.
2. a) If $m < \infty$, then $\forall k \in \{0, 1, \ldots, m\}. \sigma_{i_k \to i_{k+1}} \models G_k$ with $i_{m+1} := \infty$;
   b) otherwise, $\forall k \in \mathbb{N}_0. \sigma_{i_k \to i_{k+1}} \models G_k$.

Definition 5.3.6 (Implementation)**.** An implementation of a reactive contract $C = (\mathcal{A}, \mathcal{G}, \Delta, R)$ is any $M \subseteq \mathcal{B}$ such that for any environment E of C, we have

$$\forall \sigma \in (M \cap E). \exists r \in R. \sigma \models^{\rho} r \wedge \Pi_{\mathcal{A}}(r) = \mathcal{U}_{\mathcal{A}}(\sigma).$$

Intuitively, an implementation consists of all behaviors σ in which either the assumptions do not change according to Δ, i.e., $\mathcal{U}_{\mathcal{A}}(\sigma) \notin \Delta$, or the system reacts according to instructions specified by the set R, namely there exists a reaction $r \in R$, such that σ reactively satisfies r, in which the system must satisfy the guarantee corresponding to the current assumption for as long as the latter holds and is required to immediately adapt to any new assumption by committing itself to the corresponding new obligation. Let us compare this formalism to "standard" assume-guarantee contracts. First, we mention that the following holds.

Proposition 12. *Corresponding to each standard assume-guarantee contract $C = (A, G)$ is a reactive assume-guarantee contract $C_r = (\mathcal{A}, \mathcal{G}, \Delta, R)$ with $\mathcal{A} = \{A\}, \mathcal{G} = \{G\}, \Delta = \{\langle A\rangle\}$, and $R = \{\langle (A, G)\rangle\}$ such that $C = C_r$ in the sense that they have same sets of environments and implementations.*

Recall that any parametric assume-guarantee contract is a standard assume-guarantee contract obtained by taking the conjunction of a set of standard assume-guarantee contracts. Therefore, by Proposition 12, each parametric assume-guarantee contract has a reactive version. In particular, when all assumptions are initially disjoint, we have the following generalization of Proposition 12.

Proposition 13. *If $n \geq 1$, $\{A_1, A_2, \ldots, A_n\}$ is a set of initially disjoint assertions and for $i \in \{1, 2, \ldots, n\}$, $C_i = (A_i, G_i)$ are assume-guarantee contracts, then there exists a reactive assume-guarantee contract C_r such that $\wedge_{i=1}^{n} C = C_r$.*

Proof. Let $C_r = (\mathcal{A}, \mathcal{G}, \Delta, R)$ be defined with

- $\mathcal{A} \coloneqq \{A_1, A_2, \ldots, A_n\}$,
- $\mathcal{G} \coloneqq \{G_1, G_2, \ldots, G_n\}$,
- $\Delta \coloneqq \{\langle A_1\rangle, \langle A_2\rangle, \ldots, \langle A_n\rangle\}$,
- $R \coloneqq \{\langle (A_1, G_1)\rangle, \langle (A_2, G_2)\rangle, \ldots, \langle (A_n, G_n)\rangle\}$.

Then, $E \in \mathcal{E}_C$ if and only if

$$\begin{aligned} & \forall \sigma \in E.\sigma \in \vee_{i=1}^{n} A_i \\ \Leftrightarrow \quad & \forall \sigma \in E.\exists i \in \{1, 2, \ldots, n\}.\sigma \in A_i \\ \Leftrightarrow \quad & \forall \sigma \in E.\exists i \in \{1, 2, \ldots, n\}.\mathcal{U}_{\mathcal{A}}(\sigma) = \langle A_i\rangle \subseteq \Delta \end{aligned}$$

which holds if and only if $E \in \mathcal{E}_{C_r}$. Also, $M \in \mathcal{M}_C \Leftrightarrow \forall \sigma \in M.\sigma \in \wedge_{i=1}^{n}(A_i \Rightarrow G_i)$. Since the A_i's are initially disjoint, and therefore disjoint, there are two cases: either $\sigma \in \wedge_{i=1}^{n} \neg A_i$, in which case $\mathcal{U}_{\mathcal{A}}(\sigma) \notin \Delta$, or there is an A_i such that $\sigma \in A_i \wedge G_i$, in which case, $\sigma \models^{\rho} \langle (A_i, G_i) \rangle$ and $\mathcal{U}_{\mathcal{A}}(\sigma) = \Pi_{\mathcal{A}}(\langle (A_i, G_i) \rangle) = \langle A_i \rangle$. This implies $M \in \mathcal{M}_{C_r}$. On the other hand, $M \in \mathcal{M}_{C_r}$ implies $\forall \sigma \in M$, either $\sigma \models^{\rho} \langle (A_i, G_i) \rangle$ for some $i \in \{1, 2, \ldots, n\}$, which by Definition 5.3.5, shows that $\sigma \in A_i \wedge G_i$, or $\mathcal{U}_{\mathcal{A}}(\sigma) \notin \Delta$, which implies that $\sigma \in \wedge_{i=1}^{n} \neg A_i$. □

The following example shows the greater flexibility offered by reactive contracts over parametric ones.

Example 5.3.1. Let A_1, A_2 be initially disjoint and $C = (A_1 \vee A_2, (A_1 \Rightarrow G_1) \wedge (A_2 \Rightarrow G_2))$ and $\tilde{C}_r = (\tilde{\mathcal{A}}, \tilde{\mathcal{G}}, \tilde{\Delta}, \tilde{R})$ where $\tilde{\mathcal{A}} = \{A_1, A_2\}$, $\tilde{\mathcal{G}} = \{G_1, G_2\}$, $\tilde{\Delta} = \{\langle A_1 \rangle, \langle A_2 \rangle, \langle A_1, A_2 \rangle\}$, $\tilde{R} = \{\langle (A_1, G_1) \rangle, \langle (A_2, G_2) \rangle, \langle (A_1, G_1), (A_2, G_2) \rangle\}$. We can verify that $\tilde{C}_r \preceq C$ using the fact that by Proposition 13, $C = C_r = (\mathcal{A}, \mathcal{G}, \Delta, R)$ where $A = \tilde{\mathcal{A}}$, $G = \tilde{\mathcal{G}}$, $\Delta = \{\langle A_1 \rangle, \langle A_2 \rangle\}$, and $R = \{\langle (A_1, G_1) \rangle, \langle (A_2, G_2) \rangle\}$. With the inclusion of $\langle (A_1, G_1), (A_2, G_2) \rangle$ in $\tilde{R}$, $\tilde{C}_r$ is receptive to environments whose behaviors exhibit a change in assumptions from A_1 to A_2 and requires implementations to adapt accordingly by changing their guarantee from G_1 to G_2. On the other hand, C_r only specifies the set of implementations to be those behaviors in which either neither A_1 or A_2 is satisfied or at least a pair (A_i, G_i) is always satisfied.

Algebra

Let $\mathcal{A}$ be a set of initially disjoint assumptions and $\mathcal{G}$ be a set of guarantees. We can construct an algebra on the set $\mathfrak{C}_{(\mathcal{A},\mathcal{G})}$ of all reactive contracts obtained from $\mathcal{A}$ and $\mathcal{G}$ as follows. Let $\mathfrak{R} := (\mathcal{A} \times \mathcal{G})^{\infty}$, the set of all $(\mathcal{A} \times \mathcal{G})$-signatures and $\Delta \Rightarrow R := \{r \mid r \in \mathfrak{R} \wedge \Pi_{\mathcal{A}}(r) \notin \Delta\} \cup R$, the set of all $(\mathcal{A} \times \mathcal{G})$-signatures that are either a reaction in R or have an assumption change sequence not specified in the contingency Δ. Also, let $R_{\downarrow\Delta} = \{r \in R \mid \Pi_{\mathcal{A}}(r) \in \Delta\}$ and $\Delta_{\setminus\mathcal{R}} = \{\delta \in \Delta \mid \forall r \in R.\Pi_{\mathcal{A}}(r) \notin \Delta\}$. From these definitions, we have:

Proposition 14. *If $C = (\mathcal{A}, \mathcal{G}, \Delta, R)$ is a reactive contract, then $C^{\star} := (\mathcal{A}, \mathcal{G}, \Delta, \Delta \Rightarrow R)$ satisfies $C^{\star} = C$.*

Proof. First, by Definition 5.3.4, it is clear that $\mathcal{E}_C = \mathcal{E}_{C^{\star}}$. Also, $M \in \mathcal{M}_C$ is

equivalent to

$$\begin{aligned} & \forall E \in \mathcal{E}_C.\forall \sigma \in (M \cap E).\exists r \in R.\sigma \models^\rho r \wedge \Pi_{\mathcal{A}}(r) = \mathcal{U}_{\mathcal{A}}(\sigma) \\ \Leftrightarrow \quad & \forall E \in \mathcal{E}_{C^\star}.\forall \sigma \in (M \cap E).\exists r \in R.\sigma \models^\rho r \wedge \Pi_{\mathcal{A}}(r) = \mathcal{U}_{\mathcal{A}}(\sigma) \\ \Leftrightarrow \quad & \forall E \in \mathcal{E}_{C^\star}.\forall \sigma \in (M \cap E).\exists r \in \Delta \Rightarrow R.\sigma \models^\rho r \wedge \Pi_{\mathcal{A}}(r) = \mathcal{U}_{\mathcal{A}}(\sigma) \end{aligned}$$

which is equivalent to $M \in \mathcal{M}_{C^\star}$. Note that the forward direction of the last "$\Leftrightarrow$" follows from the fact that $R \subseteq \Delta \Rightarrow R$. The reverse direction holds because for any $\sigma \in M \cap E$ where $E \in \mathcal{E}_{C^\star} = \mathcal{E}_C$, we have $\mathcal{U}_{\mathcal{A}}(\sigma) \in \Delta$, which implies that for any $r' \in \{r \mid r \in \mathfrak{R} \wedge \Pi_{\mathcal{A}}(r) \notin \Delta\}$, we obtain $\sigma \not\models^s \Pi_{\mathcal{A}}(r')$ by the initial disjointness of $\mathcal{A}$. By the first condition of Definition 5.3.5, we have $\sigma \not\models^\rho r'$. Therefore, the r that satisfies $\Delta \Rightarrow R$ must satisfy $r \in R$. □

In light of this, we will say that a reactive contract $C = (\mathcal{A}, \mathcal{G}, \Delta, R)$ is in *canonical* form if $R = \Delta \Rightarrow R$. We will also denote by $E_{C,\max}$ the set $\{e \in \mathcal{B} \mid \exists \delta \in \Delta.\, e \models^s \delta\}$. Observe that $\mathcal{M}_C \coloneqq \{m \in \mathcal{B} \mid m \notin E_{C,\max} \vee (m \in E_{C,\max} \wedge \exists r \in R.m \models^\rho r\}) = \mathcal{B} - \{m \in \mathcal{B} \mid m \in E_{C,\max} \wedge \neg(\exists r \in R.m \models^\rho r)\} = \mathcal{B} - \{m \in E_{C,\max} \mid \neg(\exists r \in R.m \models^\rho r)\}$. In the following, for $i \in \{1, 2\}$, let $C_i = (\mathcal{A}, \mathcal{G}, \Delta_i, R_i)$ be canonical reactive contracts. The next lemma follows from the fact that $\mathcal{A}$ is initially disjoint:

Lemma 1. $\Delta_2 \subseteq \Delta_1 \Leftrightarrow E_{C_2,\max} \subseteq E_{C_1,\max}$.

Proof. Assume that $E_{C_2,\max} \subseteq E_{C_1,\max}$. For any $\delta \in \Delta_2$, choose a $\sigma \in \mathcal{B}$ such that $\sigma \models^s \delta$. By Definition 5.3.4, $\sigma \in E_{C_2,\max}$ and therefore $\sigma \in E_{C_1,\max}$. Since $\mathcal{A}$ is initially disjoint, by Proposition 5.8, any $\delta' \in \Delta_1$ such that $\sigma \models^s \delta'$ must satisfy $\delta = \delta'$. Thus, $\delta \in \Delta_1$. The other direction follows directly from the definition of the $E_{C_i,\max}$'s. □

Proposition 15. $(\Delta_2 \subseteq \Delta_1 \wedge R_1 \subseteq R_2) \Rightarrow C_1 \preceq C_2$.

Proof. We have $E \in \mathcal{E}_{C_2} \Leftrightarrow E \subseteq E_{C_2,\max} \overset{Lemma\ 1}{\Rightarrow} E \subseteq E_{C_1,\max} \Leftrightarrow E \in \mathcal{E}_{C_1}$. On the other hand, $M \in \mathcal{M}_{C_1} \Leftrightarrow M \in \mathcal{B} - \{m \in E_{C_1,\max} \mid \neg(\exists r \in R_1.m \models^\rho r)\} \overset{Lemma\ 1}{\Rightarrow} M \in \mathcal{B} - \{m \in E_{C_2,\max} \mid \neg(\exists r \in R_1.m \models^\rho r)\} \overset{R_1 \subseteq R_2}{\Rightarrow} M \in \mathcal{B} - \{m \in E_{C_2,\max} \mid \neg(\exists r \in R_2.m \models^\rho r)\} \Leftrightarrow M \in \mathcal{M}_{C_2}$. □

Lemma 2. $C_1 \preceq C_2 \Rightarrow \exists C_2' \in \mathfrak{C}_{(A,G)}.\mathcal{E}_{C_2'} = \mathcal{E}_{C_2} \wedge \mathcal{M}_{C_2'} = \mathcal{M}_{C_2} \wedge \Delta_2' \subseteq \Delta_1 \wedge R_1 \subseteq R_2'$.

Proof. We claim that C_2' can be obtained by letting $\Delta_2' \coloneqq \Delta_2$ and $R_2' \coloneqq R_1 \cup R_2$. Then, clearly, $\mathcal{E}_{C_2'} = \mathcal{E}_{C_2}$, $R_1 \subseteq R_2'$. Since $C_1 \preceq C_2$, we have $\mathcal{E}_{C_2} \subseteq \mathcal{E}_{C_1}$, and in

particular, $E_{C_2,\max} \in \mathcal{E}_{C_1}$. This implies that $E_{C_2,\max} \subseteq E_{C_1,\max}$ as $\forall E \in \mathcal{E}_{C_1}.E \subseteq E_{C_1,\max}$. Hence $\Delta_2' = \Delta_2 \subseteq \Delta_1$. Now since $\mathcal{M}_{C_1} \subseteq \mathcal{M}_{C_2}$, we have $\{m \in E_{C_2,\max} \mid \neg(\exists r \in R_2.m \models^\rho r)\} \subseteq \{m \in E_{C_1,\max} \mid \neg(\exists r \in R_1.m \models^\rho r)\}$. In particular, because $E_{C_2,\max} \subseteq E_{C_1,\max}$, $\forall m \in E_{C_2,\max}.\neg(\exists r \in R_2.m \models^\rho r) \Rightarrow \neg(\exists r \in R_1.m \models^\rho r)$. Therefore $\forall m \in E_{C_2,\max}.\neg(\exists r \in R_2.m \models^\rho r) \Leftrightarrow \neg(\exists r \in R_1 \cup R_2.m \models^\rho r)$. Therefore, $\mathcal{M}_{C_2'} = \mathcal{M}_{C_2}$. □

A reaction set $\mathcal{R}$ is said to be *unambiguous* for a contingency set Δ if for each $\delta \in \Delta$, there is at most one $r \in R$ with $\Pi_{\mathcal{A}}(r) = \delta$.

Proposition 16. *If $R_{1\downarrow\Delta_1} \cup R_{2\downarrow\Delta_2}$ is unambiguous for $\Delta_1 \cup \Delta_2$, then $C = C_1 \wedge C_2 \coloneqq (\mathcal{A}, \mathcal{G}, \Delta_1 \cup \Delta_2, R_1 \cap R_2)$ is the infimum for C_1 and C_2 in $\mathfrak{C}_{(\mathcal{A},\mathcal{G})}$.*

Proof. It is not hard to see that C is a canonical reactive contract in $\mathfrak{C}_{(A,G)}$. The main thing to check here is that if $r \in \mathfrak{R}$ and $\Pi_{\mathcal{A}}(r) \notin \Delta_1 \cup \Delta_2$, then $r \in R_1 \cap R_2$. Indeed, since $\Pi_{\mathcal{A}}(r) \notin \Delta_1 \wedge \Pi_{\mathcal{A}}(r) \notin \Delta_2$, we have $r \in R_1 \wedge r \in R_2$ since C_1 and C_2 are canonical. By Proposition 15, we have $C \preceq C_1$ and $C \preceq C_2$. Next, we will show that, if σ is a behavior such that $\{\sigma\} \in \mathcal{M}_{C_1} \cap \mathcal{M}_{C_2}$, then $\{\sigma\} \in \mathcal{M}_C$. Because any subset of an implementation is also an implementation, this will imply that $\mathcal{M}_{C_1} \cap \mathcal{M}_{C_2} \subseteq \mathcal{M}_C$ and hence $\mathcal{M}_{C_1} \cap \mathcal{M}_{C_2} = \mathcal{M}_C$. If $\sigma \notin \mathcal{B}_{\mathcal{A}}$ or $\mathcal{U}_{\mathcal{A}}(\sigma) \notin \Delta_1 \cup \Delta_2$, then this is obvious. Suppose therefore that $\mathcal{U}_{\mathcal{A}}(\sigma) \in \Delta_1 \cup \Delta_2$. Consider the following cases:

1. Without loss of generality, suppose that $\sigma \in \Delta_1$ and $\sigma \notin \Delta_2$. Then since $\{\sigma\} \in \mathcal{M}_{C_1}$, there is an $r \in R_1$ such that $\sigma \models^\rho r$. Since $\Pi_{\mathcal{A}}(r) \notin \Delta_2$ by the initial disjointness of $\mathcal{A}$ and R_2 is in canonical form, we have $r \in R_2$ and therefore $r \in R_1 \cap R_2$.

2. $\sigma \in \Delta_1 \cap \Delta_2$, then there are $r_1 \in R_1$ and $r_2 \in R_2$ such that $\sigma \models^\rho r_1 \wedge \sigma \models^\rho r_2$. By the unambiguity assumption, we have $r_1 = r_2$. Therefore, $\{\sigma\} \in \mathcal{M}_C$.

Let $C' = (\mathcal{A}, \mathcal{G}, \Delta', R')$ be such that $C' \preceq C_1$ and $C' \preceq C_2$. This implies that $\mathcal{M}_{C'} \subseteq \mathcal{M}_{C_1} \cap \mathcal{M}_{C_2}$. By the initial disjointness of $\mathcal{A}$, we have $\Delta_1 \cap \Delta_2 \subseteq \Delta'$. Since C satisfies $\mathcal{M}_C = \mathcal{M}_{C_1} \cap \mathcal{M}_{C_2}$ and $\Delta_1 \cap \Delta_2 = \Delta$ and $\mathcal{E}_{C'}$ is monotone in Δ', we have $C' \preceq C$. □

Proposition 17. *$C = C_1 \vee C_2 \coloneqq (\mathcal{A}, \mathcal{G}, \Delta_1 \cap \Delta_2, R_1 \cup R_2)$ is the supremum for C_1 and C_2 in $\mathfrak{C}_{(\mathcal{A},\mathcal{G})}$.*

Proof. Let $C' = (\mathcal{A}, \mathcal{G}, \Delta', R')$ be such that $C_1 \preceq C'$ and $C_2 \preceq C'$. Applying (the proof of) Lemma 2 twice, we obtain the contract $(\mathcal{A}, \mathcal{G}, \Delta', R' \cup R_1 \cup R_1)$ that is equal to C. In addition, since $\mathcal{E}_{C'} \subseteq \mathcal{E}_{C_1}$ and $\mathcal{E}_{C'} \subseteq \mathcal{E}_{C_2}$, $\Delta' \subseteq \Delta_1 \cap \Delta_2$. By Proposition 15, we have $C_1 \preceq C$, $C_2 \preceq C$, and $C \preceq C'$. Since C' is an arbitrary upper bound, we are done. □

Finally, for composing reactive contracts, we have the following proposition.

Proposition 18. *If $R_{1\downarrow\Delta_1} \cup R_{2\downarrow\Delta_2}$ is unambiguous for $\Delta_1 \cup \Delta_2$, then $C = C_1 \otimes C_2 \coloneqq (\mathcal{A}, \mathcal{G}, (\Delta_1 \cap \Delta_2) \cup \Delta_{1\setminus R_1} \cup \Delta_{2\setminus R_2}), R_1 \cap R_2)$ is the least contract that has the composition property, namely, for any $M_1 \in \mathcal{M}_{C_1}$, $M_2 \in \mathcal{M}_{C_2}$, and $E \in \mathcal{E}_C$, we have $M_1 \oplus M_2 \in \mathcal{M}_C$, $M_1 \oplus E \in \mathcal{E}_{C_2}$, and $M_2 \oplus E \in \mathcal{E}_{C_1}$.*

Proof. Let $\sigma \in M_1 \oplus M_2$ and $\delta \in (\Delta_1 \cap \Delta_2) \cup \Delta_{1\setminus R_1} \cup \Delta_{2\setminus R_2}$ such that $\sigma \models^s \delta$. Then as $\sigma \in M_1$ and $\sigma \in M_2$, $\delta \in \Delta_1 \cap \Delta_2$, and there exist $r_1 \in R_1$ and $r_2 \in R_2$ such that $\sigma \models^\rho r_1$ with $\Pi_{\mathcal{A}}(r_1) = \delta$ and $\sigma \models^\rho r_2$ with $\Pi_{\mathcal{A}}(r_2) = \delta$. By the initial disjointness of $\mathcal{A}$ and the unambiguity assumption on C_1 and C_2, we have $r_1 = r_2 \in R_1 \cap R_2$. Therefore $\{\sigma\} \in \mathcal{M}_C$. This implies $M_1 \oplus M_2 \in \mathcal{M}_C$. Now, let $\sigma \in M_1 \oplus E$. Then there exists $\delta \in (\Delta_1 \cap \Delta_2) \cup \Delta_{2\setminus R_2}$ such that $\sigma \models^s \delta$. As $(\Delta_1 \cap \Delta_2) \cup \Delta_{2\setminus R_2} \subseteq \Delta_2$, clearly $\{\sigma\} \in \mathcal{E}_{C_2}$, which implies $M_1 \oplus E \in \mathcal{E}_{C_2}$. Arguing similarly, we also have $M_2 \oplus E \in \mathcal{E}_{C_1}$. This shows that C has the composition property. Suppose that C' also has the composition property, then $\Delta' \subseteq \Delta$. Indeed, suppose that there is a $\delta \in \Delta'$ such that $\delta \notin (\Delta_1 \cap \Delta_2) \cup \Delta_{1\setminus R_1} \cup \Delta_{2\setminus R_2}$. Let $\sigma \models^s \delta$. If $\delta \notin \Delta_1 \cup \Delta_2$. Then clearly, $\{\sigma\} \in \mathcal{M}_{C_1} \cap \mathcal{E}_{C'}$. However, $\sigma \notin \mathcal{E}_{C_2}$. Suppose on the other hand that $\delta \in \Delta_1 \cup \Delta_2$. Without loss of generality, assume that $\delta \in \Delta_1$. Then the fact that $\delta \notin \Delta_1 \cap \Delta_2$ implies $\delta \notin \Delta_2$ and $\{\sigma\} \notin \mathcal{E}_{C_2}$. Furthermore, $\sigma \notin \Delta_{1\setminus R_1}$ means that $\{\sigma\} \in \mathcal{M}_{C_1} \cap \mathcal{E}_{C'}$, a contradiction. Next, we will show that $\mathcal{M}_C \subseteq \mathcal{M}_{C'}$. Indeed, the requirement that for any $M_1 \in \mathcal{M}_{C_1}$ and $M_2 \in \mathcal{M}_{C_2}$, $M_1 \oplus M_2 \in \mathcal{M}_{C'}$ (implying $\mathcal{M}_{C_1} \cap \mathcal{M}_{C_2} \subseteq \mathcal{M}_{C'}$) will enforce this. It suffices to show that for any $\sigma \in \mathcal{B}$ such that $\{\sigma\} \in \mathcal{M}_C$, we have $\{\sigma\} \in \mathcal{M}_{C_1} \cap \mathcal{M}_{C_2}$. We will prove the contrapositive. Let $\sigma \in \mathcal{B}$ be such that $\{\sigma\} \notin \mathcal{M}_{C_1} \cap \mathcal{M}_{C_2}$. Let us show $\{\sigma\} \notin \mathcal{M}_C$. Without loss of generality, assume that $\{\sigma\} \notin \mathcal{M}_{C_1}$. Then there is a $\delta \in \Delta_{1\setminus R_1}$ such that $\sigma \models^s \delta$. But this implies that there exists no $r \in R_1 \cap R_2$ such that $\sigma \models^\rho r$. Therefore, $\{\sigma\} \notin \mathcal{M}_C$. Thus, C' satisfies $\mathcal{M}_C \subseteq \mathcal{M}_{C'}$ and $\Delta' \subseteq \Delta$. Since $\mathcal{E}_{C'}$ is monotone in Δ', C is indeed the least contract with the composition property. □

In the next section, we will show how this formalism can be applied to reactive synthesis.

5.4 Reactive Contracts for GR(1) Games

Reactive synthesis refers to the automatic correct-by-construction synthesis of a reactive system from formal specifications. Linear temporal logic (LTL) is a language whose formulae are built from a finite set of logical (e.g, $\neg, \wedge, \vee$), temporal (e.g., $\Box, \Diamond$ for "always" and "eventually") operators and atomic propositions (Clarke Jr. et al., 2018). An LTL formula can be used to check a system trace for satisfaction of properties such as "always eventually `atom_prop` will be true" ($\Box\Diamond$`atom_prop`) or "`atom_prop` must always hold" ($\Box$`atom_prop`). In general, synthesis from LTL specifications is 2EXPTIME-complete in the length of the formula (Piterman, Pnueli, and Sa'ar, 2006). Fortunately, an expressive subset of LTL has been identified and shown to allow for relatively efficient synthesis algorithms with cubic time complexity (Maoz and Ringert, 2015). This subset is called General Reactivity of Rank 1, or GR(1), and offers the following specification format:

$$\underbrace{(\theta^e \wedge \Box\rho^e \wedge \wedge_{0\leq i\leq m}\Box\Diamond J_i^e)}_{A} \Rightarrow \underbrace{(\theta^s \wedge \Box\rho^s \wedge \wedge_{0\leq i\leq n}\Box\Diamond J_i^s)}_{G}. \tag{5.9}$$

Formula (5.9) is to be interpreted on a 2-player turn-based game between an environment e and a system (implementation) s over a set of variables $\mathcal{V}$ that can be decomposed as $\mathcal{V}^e \cup \mathcal{V}^s$. A round of the game consists of two turns with the first being taken by the environment to set values for variables in $\mathcal{V}^e$ and the second taken by the system to set values for variables in $\mathcal{V}^s$. By the end of each round, all variables in $\mathcal{V}$ will have been assigned a value. Such a valuation of $\mathcal{V}$ defines a game *state*. A *play* is an infinite sequence of states. In Formula (5.9), for $p \in \{e, s\}$, θ^p is a constraint over the set of initial states, ρ^p is a safety constraint over the current and the next states, and J_i^p is a liveness constraint specifying a set of states that are required to always eventually be visited. Without loss of generality, we can assume the variables in $\mathcal{V}$ are Boolean, in which case the contract formalism discussed earlier captures Formula (5.9) exactly. That is, the set $\mathcal{B} = (2^{\mathcal{V}})^\omega$ consists of all possible plays of the game and Formula (5.9) is a *GR(1) contract* of the form $(A, A \Rightarrow G)$ for which the powerset of the set of all plays in which the antecedent A holds is the set of all environments and the powerset of the set of all plays in which the antecedent A fails or the consequent G holds is the set of all implementations. A GR(1) contract is said to be *realizable* if the system has a strategy such that no

matter what the environment does, the resulting plays are implementations of the contract.

Being of the form $(A, A \Rightarrow G)$, GR(1) contracts are certainly not reactive: as mentioned, a play in which the system successfully incapacitates the environment from fulfilling its obligations in the antecedent is considered to be an implementation. This vacuous satisfaction may result in unintended behaviors: in fact, there has recently been work aimed at finding "environmentally-friendly" implementations that allow the environment to satisfy its promises (Majumdar, Piterman, and Schmuck, 2019). Of course, a designer may attempt to mitigate this problem by making the contract reactive, to which end, they will need to somehow come up with a proper characterization of the sets $\mathcal{A}$, $\mathcal{G}$, Δ, and R. Relatively speaking, the one that is often the "easiest" to characterize among these is $\mathcal{G}$ because system designers usually have some idea of a set of basic services the system should provide. In addition, they can also think of a set of less desirable but still acceptable services that the system may resort to when a failure occurs. The set $\mathcal{A}$ can be less straight forward to construct because it is difficult to anticipate potential failures. In addition, the multiple possible configurations (some may be unreachable) of the system/environment at which a failure happens often determine the possibility of recovery and therefore should be considered. For many such reasons, system designers may only come up with a set of fragmentary, perhaps very specific and incomplete assumptions and guarantees that they think may be relevant. In so far as $\mathcal{A}$ and $\mathcal{G}$ are fragmentary and the relationships between their elements are unknown, one is not quite ready to specify Δ and R. The contract synthesis question becomes: how do we synthesize from these sets of *coarse* assumptions and guarantees a compact and fundamental road map for specifying reactivity?

A key relation between elements of $\mathcal{A}$ and $\mathcal{G}$ is that of realizability since, after all, including an unrealizable pair in a reaction from R does not guarantee an implementation. More importantly, this relation can be efficiently computed for GR(1) games as part of the same synthesis algorithm mentioned earlier. From this point on, we will denote by $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{G}$ the *realizability relation* that satisfies $(A, G) \in \mathcal{R}$ if and only if (A, G) is realizable. From $\mathcal{R}$, an answer to the above question may be found in the form of a Galois connection between $2^{\mathcal{A}}$ and $2^{\mathcal{G}}$.

Definition 5.4.1 (Galois connection)**.** Given two partially ordered sets $(A, \leq_A)$ and $(G, \leq_G)$, a (antitone) Galois connection between these two sets consists of a pair of

maps $^* : A \to G$ and $_* : G \to A$ such that for any $a \in A$ and $g \in G$

$$a \leq_A g_* \Leftrightarrow g \leq_G a^*$$

where a^* and g_* denote the applications of * and $_*$ to $a \in A$ and $g \in G$.

Define the "forward" map * by $^* : S_{\mathcal{A}} \mapsto \{G \mid \forall A \in S_{\mathcal{A}}.(A, G) \in \mathcal{R}\}$ and the "pullback" map $_*$ by $_* : S_{\mathcal{G}} \mapsto \{A \mid \forall G \in S_{\mathcal{G}}.(A, G) \in \mathcal{R}\}$. One can verify that these maps satisfy the requirement of Definition 5.4.1 and therefore form a Galois connection. It is well know that the compositions of Galois connection operators, namely, $_*\circ^*$ and $^*\circ_*$ are dual closure operators and give rise to fixpoint pairs that form isomorphic complete lattices when ordered by set inclusion (Denecke, Erné, and Wismath, 2013). In our case, the one that comes from the coarse assumptions will be called the *assumption lattice* and the other the *guarantee lattice*.

Each of these fixpoint pairs is of the form $(S_{\mathcal{A}}, S_{\mathcal{G}})$ where $S_{\mathcal{A}}$ is the largest subset of $\mathcal{A}$, each element of which is realizable with all guarantees in the subset $S_{\mathcal{G}}$ of $\mathcal{G}$ and vice versa. Observe that since each assumption in $S_{\mathcal{A}}$ is realizable with all guarantees in $S_{\mathcal{G}}$, we can take the disjunction of all fragmentary assumptions in $S_{\mathcal{A}}$ to form a single new assumption that is also realizable with any guarantee from $S_{\mathcal{G}}$. Dually, when actions taken in the GR(1) games are reversible (hence the ability to satisfy two liveness goals separately implies the ability to satisfy them jointly), we can get a single guarantee from $S_{\mathcal{G}}$ by taking the conjunction of all its guarantees. In fact, we can get a smaller representation of the fixpoints with some more refactoring. We can simplify $S_{\mathcal{G}}$ ($S_{\mathcal{A}}$) to $S'_{\mathcal{G}}$ ($S'_{\mathcal{A}}$) by subtracting from it all guarantees (assumptions) contained in any descendant (ancestor) of the node corresponding to $S_{\mathcal{G}}$ ($S_{\mathcal{A}}$) on the guarantee (assumption) lattice. Note that operation may result in an empty set (and if that is the case, we will denote it by the symbol "$\wedge$" ("$\vee$") as in Fig. 5.3. Thus, these new and compact fixpoints $(S'_{\mathcal{A}}, S'_{\mathcal{G}})$ allow us to define new sets of assumptions and guarantees $\mathcal{A}'$ and $\mathcal{G}'$ with complete lattice structures that can be used to specify Δ and R. On the assumption lattice, the greater elements represent assumptions that are more favorable to the system and conversely on the guarantee lattice, the greater elements are more "difficult" for the system to satisfy (see Fig. 5.3). In other words, descending any chain from the assumption lattice is equivalent to experiencing more and more restrictive assumptions that may be associated with more severe failures. A chain on the guarantee lattice however represents all possible guarantees that can be realizable if the largest element of the chain is realizable.

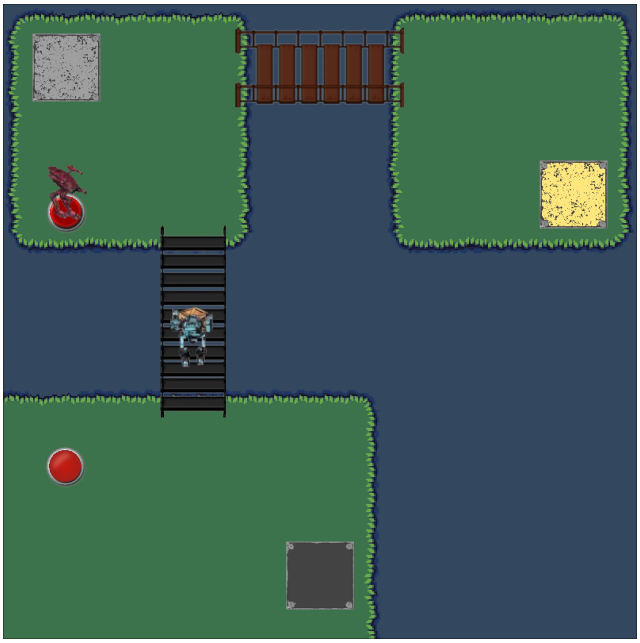


Figure 5.1: A screenshot of a simulation trace. At this point, no failure has occured. The red robot is holding down a red button for the crate-carrying blue robot to cross. The simulation and synthesis code is available at (Phan-Minh, 2019a).

Note that the most costly step in the described procedure is the computation of the realizability relation $\mathcal{R}$ which has time complexity $O(|\mathcal{R}|n^3)$ where n is the length of the longest GR(1) formula. However, this should be done offline during the design/planning process. It is, nevertheless, easy to parallelize and also optimize this computation using a priori knowledge about $\mathcal{A}$ and $\mathcal{G}$. In addition, full knowledge of $\mathcal{R}$ may not be necessarily for most applications and for that reason it can be incrementally refined. The following case study will demonstrate this method.

5.5 Case Study: a Reactive GR(1) Contract on 3 Islands

Fig. 5.1 and 5.2 are two screenshots from a simulation in (Phan-Minh, 2019a) that captures a reactive GR(1) game involving two land robots that navigate and manipulate an uncertain gridworld environment consisting of 3 islands connected by 2 bridges in order to satisfy some liveness objectives. In Fig. 5.1, the blue "transporter" robot (`r1`) is shown carrying a crate across the black bridge, which only deploys when at least one of the red buttons (which may unexpectedly fail and turn grey in the animation) is held down by either robot, which in this scene, is the red "supervisor" robot (`r2`). The brown bridge is always available but, like the buttons, can suddenly go out of service (see Fig. 5.2). The crate (movable only by the transporter robot) can only be picked up from the black square patch (a factory

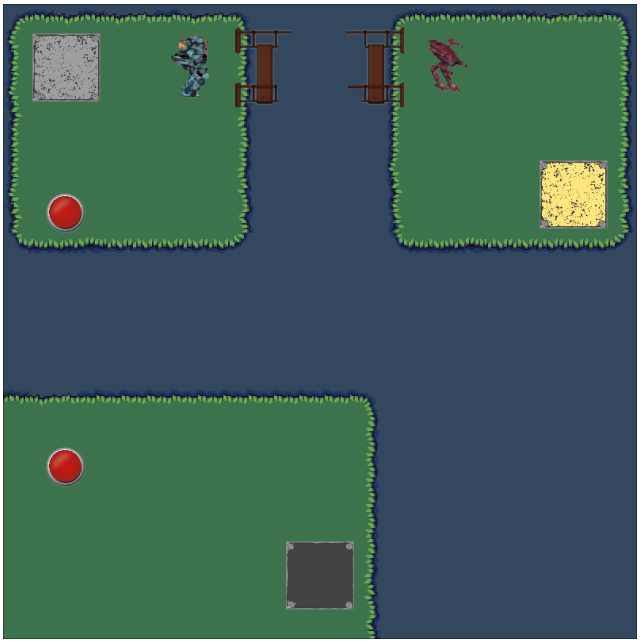


Figure 5.2: The brown bridge broke, the two robots are stranded.

that is *home* to these robots) and deposited at either the silver or yellow square patches (shipping ports), after which it will respawn at the black patch. The set of coarse guarantees $\mathcal{G}$ consists of 7 elements:

- $\square\lozenge$`box_r2_far` (resp., $\square\lozenge$`box_r2_near`): the supervisor robot `r2` stands on the yellow (resp., silver) square patch to supervise the dropping off of the crate there by the other robot infinitely often.
- $\square\lozenge$`box_far` (resp., $\square\lozenge$`box_near`): the crate gets dropped off at the yellow (resp., silver) square patch infinitely often (possibly without the supervisor robot being present).
- $\square\lozenge$`r2_far` (resp., $\square\lozenge$`r2_near`, $\square\lozenge$`r2_home`): the supervisor robot patrols the yellow (resp., silver, black) square patch infinitely often.

The set $\mathcal{A}$ that coarsely characterizes possible operating scenarios is constructed by taking the conjunction of all variables appearing in elements of the product set obtained from the sets {`r1_home`, `r1_near`, `r1_far`}, {`r2_home`, `r2_near`, `r2_far`}, and $2^{\{\square\texttt{bridge},\square\texttt{button1},\square\texttt{button2}\}}$. An element of $\mathcal{A}$ can be `r1_home` $\wedge$ `r2_far` $\wedge$ $\square$`bridge` $\wedge$ $\square$`button1`, denoting the condition that the transporter robot `r1` starts on the "home" island (the one with the black patch), the supervisor robot `r2` starts on the "far" island (yellow patch), and the button on the island with the silver patch,

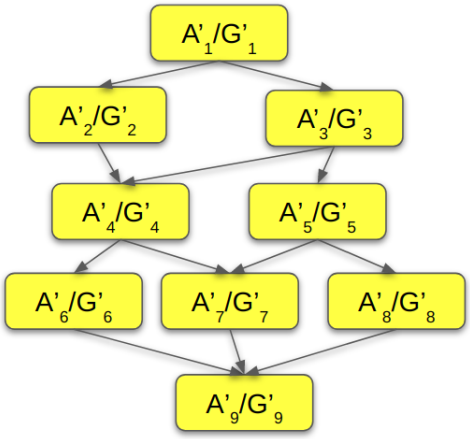


Figure 5.3: Order isomorphic assumption/guarantee lattices of Galois fixpoints computed for the 3 island scenario. See Table 5.1 for details on each A'_i/G'_i.

`button2`, is broken. We observe that $\mathcal{A}$ has 72 elements and is initially disjoint. The Galois connection calculation will reduce this relatively large number of cases to only 9 in $\mathcal{A}'$ as can be seen from Fig. 5.3.

A calculation of $\mathcal{R}$ on $\mathcal{A} \times \mathcal{G}$ (which has 72×7 elements) using `slugs`, a GR(1) synthesis tool (Ehlers and Raman, 2016), on a laptop with an Intel i7-4720HQ processor and 16GB of RAM took approximately 40 minutes (not parallelized). It took about 3 seconds to generate the lattices using a fixpoint calculation algorithm from (Outrata and Vychodil, 2012). Note also that since all actions performed by the two robots in this example are reversible, if under an assumption, two liveness guarantees can be realized separately, their conjunction will also be realizable. This observation will also be used to synthesize one of the most basic forms of reactive contracts, namely: an "optimal" reactive contract, in which the reaction R is defined to consist of all finite and infinite sequences whose elements are of the form (α, γ) where α is an assumption from a node on the assumption lattice and γ is the conjunction of the corresponding guarantee on the guarantee lattice together with all guarantees contained in its descendant nodes. The contigency Δ is defined to be the sequences obtained by projecting away γ from any sequence in R. In Fig. 5.3, if $\alpha = \square$`bridge` $\wedge$ (`r2_far` $\vee$ `r2_near`), then $\gamma = \square\Diamond$`r2_near` $\wedge \square\Diamond$`r2_far`.

Consider a scenario in which we are given the lattices $\mathcal{A}'$ and $\mathcal{G}'$ and a set $F \subseteq \mathfrak{F}$ where $\mathfrak{F}$ consists of Boolean variables encoding all impending failures (in the 3 island scenarios, $\mathfrak{F} = \{\square$`button1`$, \square$`button2`$, \square$`bridge`$\}$) along with a set $g_{\min} \subseteq \mathcal{G}'$ consisting of baseline guarantees that we would like the system to maintain. Algorithm 2 shows how one can use the assume-guarantee lattices to find an implemen-

Table 5.1: Assume-guarantee lattice data for Fig. 5.3.

i	A'_i	G'_i
1	□bridge ∧ □button2 ∧ (□button1 ∨ r1_far ∨ r1_near ∨ r2_far ∨ r2_near)	□◊box_r2_far
2	□button2∧(r1_home∨r1_near)∧(r2_home∨r2_near)∧(r2_home∨ r2_near) ∧ (□button1 ∨ r1_near ∨ r2_near)	□◊box_r2_near
3	□bridge ∧ □button1 ∧ (r1_home ∨ r2_home)	□◊box_far
4	□button1∧(r1_home∨r1_near)∧(r1_home∨r2_home)∧(r2_home∨ r2_near)	□◊box_near
5	□bridge ∧ (r2_far ∨ r2_near)	∅ (or ∧)
6	r2_home	□◊r2_home
7	r2_near	□◊r2_near
8	r2_far	□◊r2_far
9	∅ (or ∧)	∅ (or ∨)

tation that is robust against F while maintaining at least $g_{\min}$ starting from an initial configuration A_{init}. In line 2 of the algorithm, a check is performed to see if the

Algorithm 2 Find robust reactive GR(1) implementation for prioritized failures

1: **function** FINDROBUSTIMPLEMENTATION($\mathcal{A}', \mathcal{G}', A_{\text{init}}, F, g_{\min}$)
2: **if** $A_{\text{init}} \notin \cup_{\alpha \in \mathcal{A}'} \alpha$ **then**
3: error out: "initial state is not in contract!"
4: **else**
5: $\gamma \leftarrow \sup_{\leq_{\mathcal{G}'}} g_{\min}$
6: $\alpha \leftarrow$ the $\mathcal{A}'$ node corresponding to γ
7: $\alpha_{\text{inv}} \leftarrow \alpha \vee (\vee_{a \in \text{ancestors}(\alpha)} a)$ with variables in F replaced by `False` and variables in $\mathfrak{F} \setminus F$ replaced by `True`
8: **if** ISREALIZABLE($A_{\text{init}}, \gamma \wedge \alpha_{\text{inv}}$) **then**
9: $\gamma' \leftarrow$ largest guarantee such that ISREALIZABLE($A_{\text{init}}, \gamma' \wedge \alpha_{\text{inv}}$)
10: **return** GETSTRATEGY($A_{\text{init}}, \gamma' \wedge \alpha_{\text{inv}}$)
11: **else**
12: error out: "robust strategy doesn't exist!"

initial configuration A_{init} is a valid assumption; if not, an error will be thrown (line 3). In line 5, the algorithm finds the least node on $\mathcal{G}'$ such that γ and its descendants contain $g_{\min}$ (this always exists and is unique by the completeness of $\mathcal{G}'$). Line 6 defines α as the corresponding node to γ on $\mathcal{A}'$. Hence, $\alpha \vee (\vee_{a \in \text{ancestors}(\alpha)} a)$ corresponds to the most relaxed assumption for which $g_{\min}$ can be guaranteed. In other words, all configurations from which $g_{\min}$ can be satisfied are contained in it. In line 7, an invariant constraint α_{inv} is computed from $\alpha \vee (\vee_{a \in \text{ancestors}(\alpha)} a)$ by assuming that all failures in F have occurred. This represents configurations from which γ can still be guaranteed. In line 8, a check is performed to see if it is possible to satisfy γ while maintaining α_{inv}. If the answer is yes, the algorithm attempts to find a better guarantee than γ (possibly using bisection) and will return a robust

strategy that guarantees it (lines 9 – 10); otherwise it will throw an error saying that such a robust strategy does not exist (line 12).

In the 3 island example, if we start out at an initial configuration where at least one robot is not on the home island, $g_{\min}$ is anything, and $F = \{\square\texttt{button1}\}$, then algorithm 2 will return a robust strategy with $\gamma' = \mathcal{G}'$ (all guarantees) which never allows both robots to be on the "home" island at the same time (because if `button1` fails, then the robots will not be able to leave the home island). In particular, if $g_{\min} = \{\square\lozenge\texttt{box\_r2\_far}\}$, then α_{inv} will be equal to $\texttt{r1\_far} \vee \texttt{r1\_near} \vee \texttt{r2\_far} \vee \texttt{r2\_near}$.

5.6 Conclusion

In this chapter, we have developed a metatheory-compatible contract framework that focuses on specifying a system's reactions to antecedent failures and related it to a reactive synthesis setting involving GR(1) contracts. We have also looked at automating the process of synthesizing and simplifying reactive contracts by computing fixpoint pairs of a certain Galois connection between the system's assumption and guarantee sets and carried out a simulated case study that concretizes our ideas. There are potentials in 1) exploring and developing methods to automate the process of finding coarse assumptions with (Alur, Moarref, and Topcu, 2013; Chen et al., 2020) as good starting points, and 2) studying compositions of systems specified by reactive contracts as well as the propagation of failures through these compositions.

References

Alur, Rajeev, Salar Moarref, and Ufuk Topcu (2013). "Counter-strategy guided refinement of GR (1) temporal logic specifications." In: *2013 Formal Methods in Computer-Aided Design*. IEEE, pp. 26–33.

Arnold, André and Damian Niwinski (2001). *Rudiments of calculus*. Vol. 146. Elsevier.

Azzopardi, Shaun, Gordon J. Pace, and Fernando Schapachnik (2014). "Contract Automata with Reparations." In: *JURIX*. Vol. 271. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 49–54.

Bloem, Roderick (Sept. 2015). "Reactive synthesis." In: *2015 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 3–3. DOI: 10.1109/FMCAD.2015.7542241.

Censi, Andrea (2015). "A mathematical theory of co-design." In: *arXiv preprint arXiv:1512.08055*.

Chen, Yuxiao, Sumanth Dathathri, Tung Phan-Minh, and Richard M. Murray (2020). "Counter-example Guided Learning of Bounds on Environment Behavior." In: *Conference on Robot Learning*, pp. 898–909. URL: http://proceedings.mlr.press/v100/chen20b.html.

Clarke Jr., Edmund M., Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith (2018). *Model checking*. MIT press.

Dathathri, Sumanth, Scott C. Livingston, and Richard M. Murray (2017). "Enhancing tolerance to unexpected jumps in GR(1) games." In: *Proceedings of the 8th International Conference on Cyber-Physical Systems, ICCPS 2017, Pittsburgh, Pennsylvania, USA, April 18-20, 2017*. Ed. by Sonia Martínez, Eduardo Tovar, Chris Gill, and Bruno Sinopoli. ACM, pp. 37–47. DOI: 10.1145/3055004.3055014. URL: https://doi.org/10.1145/3055004.3055014.

Denecke, Klaus, Marcel Erné, and Shelly L. Wismath (2013). *Galois connections and applications*. Vol. 565. Springer Science & Business Media.

Ehlers, Rüdiger and Vasumathi Raman (2016). "Slugs: Extensible gr (1) synthesis." In: *International Conference on Computer Aided Verification*. Springer, pp. 333–339.

Henzinger, Thomas A., Peter W. Kopke, Anuj Puri, and Pravin Varaiya (1998). "What's decidable about hybrid automata?" In: *Journal of computer and system sciences* 57.1, pp. 94–124.

Kim, Eric S., Murat Arcak, and Sanjit A. Seshia (2017). "A Small Gain Theorem for Parametric Assume-Guarantee Contracts." In: *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control*. HSCC '17. Pittsburgh, Pennsylvania, USA: ACM, pp. 207–216. ISBN: 978-1-4503-4590-3. DOI: 10.1145/3049797.3049805. URL: http://doi.acm.org/10.1145/3049797.3049805.

Kim, Eric S., Sadra Sadraddini, Calin Belta, Murat Arcak, and Sanjit A. Seshia (2017). "Dynamic contracts for distributed temporal logic control of traffic networks." In: *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*. IEEE, pp. 3640–3645.

Livingston, Scott C., Pavithra Prabhakar, Alex B. Jose, and Richard M. Murray (May 2013). "Patching task-level robot controllers based on a local μ-calculus formula." In: *2013 IEEE International Conference on Robotics and Automation*, pp. 4588–4595. DOI: 10.1109/ICRA.2013.6631229.

Majumdar, Rupak, Nir Piterman, and Anne-Kathrin Schmuck (2019). "Environmentally-friendly GR(1) Synthesis." In: *arXiv preprint arXiv:1902.05629*.

Maoz, Shahar and Jan Oliver Ringert (2015). "GR (1) synthesis for LTL specification patterns." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, pp. 96–106.

Outrata, Jan and Vilem Vychodil (2012). "Fast algorithm for computing fixpoints of Galois connections induced by object-attribute relational data." In: *Information Sciences* 185.1, pp. 114–127.

Phan-Minh, Tung (2019a). *Reactive Contracts*. `https://github.com/tungminhphan/reactive_contracts`, accessed April 30 2019.

– (2019b). *TLA+ specifications for an Automated Valet Parking Garage*. `https://github.com/tungminhphan/reactive_contracts/blob/master/scenarios/AutomatedValetParking.tla`, accessed May 2 2019.

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

Chapter 6

ASSUME-GUARANTEE PROFILES[1]

6.1 Introduction

Autonomous vehicles will certainly have to function alongside humans—that is, unless or until a fully-automated transportation infrastructure can be built. The interaction between self-driving cars and humans will inevitably result in unforeseen safety and legality situations. Self-driving car manufacturers are expected to be responsible for ensuring that the behavior of their cars minimizes the risk of collision. However, the current rules are often designed heuristically on a case by case basis, lacking transparency, predictability, and performance (Paden et al., 2016; Shalev-Shwartz, Shammah, and Shashua, 2017).

If self-driving cars were to agree to restrict their state space to a region defined by some behavioral contract, there would be less uncertainty in behavior prediction, thereby making it significantly easier to choose mutually beneficial and safe actions. The process of identifying the party responsible for an accident would also be simplified. Furthermore, car manufacturers could even begin to optimize their car behavior to refine and customize driving preferences.

Formal methods offer many tools and frameworks for designing and proving specifications that guarantee high-level behaviors such as safety and liveness in complex systems like self-driving cars. However, in related approaches, specifications for self-driving cars are often scenario-specific and are formulated independently of one another (Shalev-Shwartz, Shammah, and Shashua, 2017). State space explosion also greatly limits the scalability of these methods (Baier and Katoen, 2008; Wongpiromsarn, Karaman, and Frazzoli, 2011).

A more general approach involves designing "rulebooks" for self-driving cars (Censi et al., 2019). These rulebooks order the set of rules for self-driving cars according to a hierarchy taking the form of a preorder, which intentionally leaves ambiguity in their behaviors. The authors in (Censi et al., 2019) do not consider allowing agents to make assumptions about one another. As a consequence, they do not address how to accommodate for the unpredictable and law-evasive nature of human drivers. In light of these issues, we propose a framework that can be used to:

[1] The material in this chapter comes from joint work with Karena X. Cai and Richard M. Murray.

1. Identify high-level specifications and their relationships as part of a hierarchical structure that describes a self-driving car's desirable behavior on the road.

2. Define consistency, coverage, and completeness for a set of specifications.

3. Introduce an assume-guarantee contract formalism for specification structures and notions of rationality and blame.

4. Present a basic and consistent set of axioms for self-driving cars that can be refined and built upon.

5. Demonstrate with game-theoretic examples how rational autonomous vehicle behaviors can be computed/agreed upon under the assumption that they are aware of each other's specification structures.

We ultimately want to be able to guarantee that self-driving cars will behave correctly and not be responsible for accidents. This chapter offers a step in that direction.

6.2 Overview

In a dynamic and interactive environment, the problem of providing guarantees for a single agent without making any assumption on the behaviors of other agents is ill-posed. We show the inherent coupling between the assumptions on the environment and the system's guarantees in Fig. 6.1. Our framework of assume-guarantee profiles is intended to explicitly address this issue.

Definition 6.2.1 (Assume-guarantee profile)**.** An *assume-guarantee profile* for an agent is a 2-tuple $(\mathcal{A}, \mathcal{G})$ where

- $\mathcal{A}$ is a set of behavioral preferences or characteristics that the agent assumes its environment to have.

- $\mathcal{G}$ is a set of behavioral preferences or characteristics that it is obligated to behave according to as long as its environment makes decisions in accordance with $\mathcal{A}$.

To model the sets of behavioral preferences or characteristics mentioned in Definition 6.2.1, we propose a mathematical object termed *specification structure* that describes a hierarchy on sets of what we call *dimensional properties*. A property

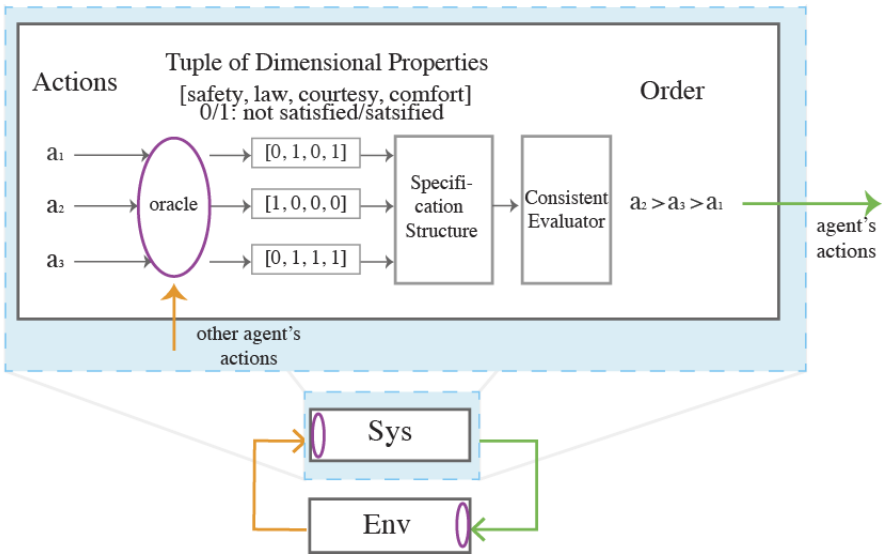


Figure 6.1: A high-level system architecture capturing the inherent coupling of the behavioral specifications for an agent and its environment is shown in the bottom figure. The details of each agent's architecture for defining its behavior are shown in the top. Each agent identifies the best action to take by using the oracle to define which specifications will be satisfied if an action is taken, and using a consistent evaluator to rank those actions based on a hierarchical ordering defined in the specification structure.

is a desirable attribute that can either be satisfied or not satisfied. A dimensional property is a property whose satisfaction is independent of the satisfaction of other properties. Examples of dimensional properties include safety, lawfulness, courtesy, and comfort. Safety does not necessarily imply comfort and vice versa. The guarantee we make in our assume-guarantee contract is that the self-driving car will act in accordance with the specification structure. Intuitively, this means of all subsets of specifications that the car can satisfy, it will choose to satisfy the one that is ranked highest in priority with respect to the specification structure. We define this more rigorously in the next section.

6.3 Evaluator and Evaluated Structure

Before presenting the formal definition of a specification structure, we make the following assumption about the predictive capabilities of a self-driving car.

Assumption 1 (Oracle)**.** We assume that each autonomous agent relies on an *oracle* (Sipser, 2012) that provides predictions to its queries about the satisfaction of dimensional properties of interest for any action or strategy that it is considering. The input to the oracle is a set of dimensional properties, a potential action or strategy

the car can choose to take, and the current world state configuration. The output of the oracle is some prediction of what specifications (dimensional properties) will be satisfied if a strategy is followed. In the simplest case, the oracle could return a valuation of a set of a Boolean variables, each indicating whether or not a property is violated.

Although many decision/optimization problems currently posed for autonomous vehicles are of high computational complexities, not to mention undecidable (Madani, Hanks, and Condon, 2003; Papadimitriou and Tsitsiklis, 1987), we expect future technology to be capable of approximating the oracle to an acceptable level of fidelity (see (Lefèvre, Vasquez, and Laugier, 2014) for a sample of related methods). If a set of dimensional properties is simply partially ordered, then there may not be enough structure to uniquely identify which action should be taken.

Example 6.3.1. Consider a set $S = \{a, b, c, d, e\}$ that is partially ordered (a poset) such that $b \prec a$, $c \prec a$, $d \prec c$, and $e \prec c$. Here, each element in the set represents a dimensional property like safety, the law, performance, etc. By this partial order, the node b cannot be compared to c or d or e. For a self-driving car, any action will result in satisfying a subset of the dimensional properties. Since b cannot be compared to c or d or e, it is ambiguous whether a self-driving car should take an action that satisfies the properties a, b, and c or an action that satisfies a, b, and d.

With only a partial ordering on the dimensional properties, there can be ambiguity in choosing the best subset of properties that can be satisfied. In order to resolve this, we introduce the idea of *consistent evaluators*, which are a class of functions that can endow some posets with a unique *weak* order on their powersets. Being weakly ordered means that all subsets are comparable, but some subsets may have equal values to each other (these are considered indistinguishable). Any of these evaluators must, in addition to basing its evaluations on the original partial order, somehow try its best to resolve any apparently missing information in a transparent and consistent way.

In a practical setting, if a self-driving car manufacturer wanted to impose a total order instead of a weak order on the powerset, they would have to face the challenging task of defining how any one set of dimensional properties is strictly better or worse than another set of dimensional properties. This is arguably not only impractical because of the exponential growth in the size of the powerset, but also because sometimes a *strict* comparison among sets of properties is simply unnecessary. A consistent

evaluator, which allows for sets in the powerset to have equal value, therefore allows for a more practical way of resolving comparisons between subsets of specfications.

We refer to chains (antichains) of partially-ordered sets in our definitions and proofs, so we present the definitions here.

Definition 6.3.1 (Chain/Antichain)**.** A chain (antichain) is a subset of a partially ordered set such that any two distinct elements in the subset are comparable (incomparable).

Definition 6.3.2 (Consistent evaluator)**.** Given a set of dimensional properties $\mathcal{P}$ and its powerset $2^{\mathcal{P}}$, we say that $f : 2^{\mathcal{P}} \to T$, where T is a totally ordered set with $\leq$ as the ordering relation, is a *consistent evaluator* for $\mathcal{P}$ if for all subsets $P_1, P_2 \subseteq \mathcal{P}$, the following hold:

1. $P_1 \neq \emptyset \Rightarrow f(\emptyset) < f(P_1)$.
2. $\forall p_1 \in P_1 :: \forall p_2 \in P_2 :: f(\{p_1\}) = f(\{p_2\}) \Rightarrow (f(P_1) \leq f(P_2) \Rightarrow f(P_1 - \{p_1\}) \leq f(P_2 - \{p_2\}))$.
3. $(\forall p_1 \in P_1 :: \forall p_2 \in P_2 :: f(\{p_1\}) \neq f(\{p_2\})) \Rightarrow (\max_{p \in P_1} f(\{p\}) < \max_{q \in P_2} f(\{q\}) \Rightarrow f(P_1) < f(P_2))$.

If $\mathcal{P}$ is partially ordered by $\preceq$ and $\mathfrak{A}_{(P,\preceq)}$ is the set of all antichains of $\mathcal{P}$, we further require that for any $p_1, p_2 \in \mathcal{P}$

4. $p_1 \prec p_2 \Rightarrow f(\{p_1\}) < f(\{p_2\})$.
5. $(\{p_1, p_2\} \in \mathfrak{A}_{(P,\preceq)} \wedge f(\{p_1\}) < f(\{p_2\})) \Rightarrow \exists s, t \in \mathcal{P} :: p_1 \prec s \wedge f(\{s\}) \leq f(\{p_2\}) \wedge f(\{p_1\}) \leq f(\{t\}) \wedge t \prec p_2$.

Intuitively, the conditions in Definition 6.3.2 mean

1. The evaluator will assign the worst value when no property is satisfied. This ensures that every property included in $\mathcal{P}$ matters to the evaluator.
2. Properties of equal value to the evaluator can be disregarded without affecting the result of the evaluation.
3. For sets that do not have properties with the same values, the one with the most highly valued property is preferable.

4. If there exists a pre-imposed hierarchy between some of the properties, then the evaluator must respect it.

5. Given a pre-imposed hierarchy on the properties, the evaluator must be impartial: it will only assign different values to two properties whose relationship is not defined in the hierarchy when they are comparable via two "proxies."

Example 6.3.2. Consider a partially ordered set Q in which p is the greatest element and all other elements belong to an antichain. Then we can define f as the function $f(\tilde{Q}) \coloneqq \mathbb{1}_{p\in\tilde{Q}}|Q| + |\tilde{Q} - \{p\}|$ for all $\tilde{Q} \subseteq Q$ where $\mathbb{1}$ is the indicator function. This function evaluates any subset with the maximal element in it as the cardinality of Q plus the dimension of the subset with the element p thrown out. It also evaluates any subset without the maximal element as the dimension of that subset. One can easily verify that f is a consistent evaluator for Q.

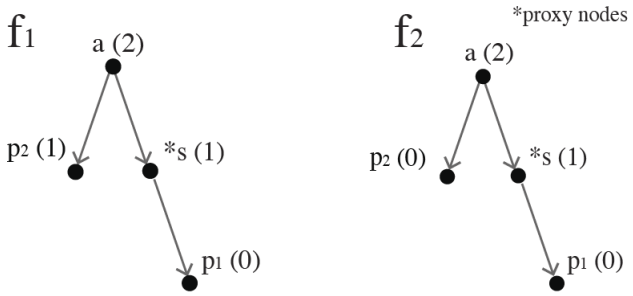


Figure 6.2: A poset that does not admit a consistent evaluator. The values in parentheses denote the value of the singleton set containing that node given by the evaluator f_i. In both cases, requirement 5 is violated.

Example 6.3.3 (Poset without consistent evaluator)**.** Consider the poset with the structure shown in Fig. 6.2 and for simplicity, assume $T = \mathbb{N}_0$. We cannot define a consistent evaluator that satisfies all five requirements on this partially-ordered set. In order to satisfy requirement 4, such that the partial order established in the poset is preserved, the consistent evaluator function f_1 on the left and f_2 on the right can, WLOG, assign all nodes on the right branch of the poset with the values shown in Fig. 6.2. To respect the partial order, by requirements 4 and 5 of consistent evaluators, p_2 must be assigned a value in $\{0, 1\}$. The left figure shows what happens if the function takes on the value 1 for the left node, i.e., $f_1(\{p_2\}) = 1$. If this happens, then $f_1(\{p_1\}) < f_1(\{p_2\})$, but there is no proxy node that is comparable to p_2 in the poset and has a value equal to $f_1(\{p_1\})$. This clearly violates requirement 5. The right figure shows what happens if the function

takes on the value 0, i.e. $f_2(\{p_1\}) = 0$. A similar violation is incurred by f_2 (see (Phan-Minh, Cai, and Murray, 2019)).

Through the previous examples, we see that not all posets admit a consistent evaluator. The natural question to ask is: what makes posets consistently evaluable? The next theorem will answer this question. Before stating the theorem, we will give one more definition. First observe that for any set of maximal antichains $\mathfrak{A}$ of a set partially ordered by $\preceq$, there exists an *induced total preorder* $\leftarrow$ (i.e., a preorder in which any two elements are comparable) defined by:

$$\forall A_1, A_2 \in \mathfrak{A} :: A_1 \leftarrow A_2 \Rightarrow \exists a_1 \in A_1 :: \exists a_2 \in A_2 :: a_1 \prec a_2.$$

Observe that $\leftarrow$ is indeed a total preorder because of the maximality assumption on the antichains.

Theorem 1. *A finite poset P of dimensional properties has a consistent evaluator if and only if it can be partitioned by a set $\mathfrak{A}$ of N maximal antichains such that*

1. *$\mathfrak{A}$'s induced total preorder is a total order.*
2. *For each dimensional property, there exists a maximal chain containing it of length N.*

Lemma 3. *Condition 1 of Theorem 1 is equivalent to the fact that the maximal antichains can be assigned ranks in such a way the partial order is respected.*

Proof (Sketch). Define the ranks in accordance with the induced total order on the antichains and verify. □

We are ready to give a proof for Theorem 1.

Proof. ($\Rightarrow$): Suppose that P is a poset of dimensional properties with the ordering relation $\preceq$ such that P has a consistent evaluator f. Since P is finite, the set $f_P \coloneqq \{f(\{p\}) \mid p \in P\}$ is also finite. Furthermore, the range of f being totally ordered implies that we can write $f_P = \{z_1, z_2, \ldots, z_n\}$ for $n = |f_P|$ such that $z_1 < z_2 < \ldots < z_n$. For each $z_i \in f_P$, let $f^{-1}(z_i) \subseteq P$ be defined by $f^{-1}(z_i) \coloneqq \{p \mid p \in P \wedge f(\{p\}) = z_i\}$. Observe that requirement 4 of Definition 6.3.2 implies that for each i, $f^{-1}(z_i)$ is an antichain. Consequently, the $f^{-1}(z_i)$'s form a partition of P by antichains. By ranking each $f^{-1}(z_i)$ by the corresponding z_i,

it also follows that the antichains respect the partial order defined by $\preceq$. To show maximality, suppose that there exists $j \in [n]$ such that $f^{-1}(z_j)$ is not a maximal antichain. This implies that there exists $k \in [n] - \{j\}$ and there is a property $q^\star \in f^{-1}(z_k)$ such that $\{q^\star\} \cup f^{-1}(z_j)$ is an antichain. WLOG, suppose $j < k$ so that $z_j < z_k$ implies $\forall q \in f^{-1}(z_j) :: f(\{q\}) < f(\{q^\star\})$. If $k = j + 1$, the existence of any $\tilde{q} \in f^{-1}(z_j)$ such that $f(\tilde{q}) < f(q^\star)$ implies, by requirement 5 of Definition 6.3.2, that there exists $q' \in f^{-1}(z_j)$ such that $q' \prec q^\star$. But this contradicts the assumption that $\{q^\star\} \cup f^{-1}(z_j)$ is an antichain. Suppose up to $k = t$ with $k + 1 \leq t < n$, there exists no $q^\star \in f^{-1}(z_k)$ such that $\{q^\star\} \cup f^{-1}(z_j)$ is an antichain. We will show that this also holds for $k = t + 1$. Indeed, by the induction hypothesis, $q^\star$ can only potentially come from $f^{-1}(z_{t+1})$. Since for any $q \in f^{-1}(z_j)$, $f(\{q\}) < f(\{q^\star\})$, by requirement 5, there is q_1 and q_2 such that $q \prec q_1$ and $q_2 \prec q^\star$ while $f(\{q_1\}) \leq f(\{q^\star\})$ and $f(\{q\}) \leq f(\{q_2\})$. By requirement 4, q_2 must belong to one of $f^{-1}(z_j), f^{-1}(z_{j+1}), \ldots, f^{-1}(z_{t+1})$. By the induction hypothesis, there is a $q' \in f^{-1}(z_j)$ such that $q' \prec q_2$. Thus $q' \prec q^\star$, a contradiction. From this, we conclude Lemma 3 and hence 1) hold.

To see that 2) holds, observe that any property $p \in f^{-1}(z_j)$, if $j \neq n$ and $n \geq 2$, then by requirement 5 and the antichain property of the $f^{-1}(z_j)$, there exists $q \in f^{-1}(z_{j+1})$ such that $p \prec q$. Similarly, if $j \neq 1$ and $n \geq 2$, there exists $r \in f^{-1}(z_{j-1})$ such that $r \prec p$. Applying this argument to r and/or q inductively yields a chain of length n that contains q. This chain is maximal by the contradiction resulting from applying the pigeonhole principle to the assignment of properties from any chain of greater length to the maximal antichains.

($\Leftarrow$): We can easily verify that requirements 1-4 of a consistent evaluator are satisfied. Now, we show that requirement 5 holds as well. Let us show this by contradiction. Consider that there exists a node p_1 and p_2 such that $f(\{p_1\}) < f(\{p_2\})$, but there does not exist a node s or t such that $p_1 \prec s$, $t \prec p_2$, and $f(\{p_2\}) = f(\{s\})$ and $f(\{p_1\}) = f(\{t\})$. WLOG, consider p_1 to be a node where there does not exist a node s such that $p_1 \prec s$ and $f(\{p_2\}) = f(\{s\})$. Since there is no node that is directly comparable to p_1 in the antichain with value equal to $f(\{p_2\})$, there exists a maximal chain containing p_1 that has length strictly less than m. This is a violation of property 2) characterizing the poset P. □

Is it possible that there may be multiple such decompositions of maximal antichains, making the ordering that is induced via the corresponding rankings non-unique and

hence the "consistent evaluator" not very consistent? Luckily, the answer is a reassuring negative.

Theorem 2. *The partition in Theorem 1 is unique.*

Proof. Suppose that $P_1, P_2, \ldots, P_m$ is also a partition of maximal antichains of P with ranks $r(P_1) < r(P_2) < \ldots < r(P_m)$ that respect the partial order. Suppose that $m \neq n$ where $n = |f_P|$. If $m > n$, then by 2) of Theorem 1, there is a chain of length m. However, assigning these m properties to the $f^{-1}(z_i)$ means by the pigeonhole principle that there are at least two properties that are assigned to the same $f^{-1}(z_j)$ for some j, implying that $f^{-1}(z_j)$ is not an antichain, namely, a contradiction. It follows that $m \leq n$. Similarly, we can argue that $m \geq n$ and therefore $m = n$. Now, we claim that $P_i = f^{-1}(z_i)$ for all $i \in \{1, 2, \ldots, n\}$. Suppose this is not the case, then there exists $p \in P_k$ such that $p \in f^{-1}(z_h)$ for $k \neq h$. Then by 2), there are two chains of length n: $p_1 \prec p_2 \prec \ldots \prec p_n$ and $f_1 \prec f_2 \prec \ldots \prec f_n$ such that $p_i \in P_i$ and $f_i \in f^{-1}(z_i)$. We also have $p_k = f_h = p$. WLOG, assume $h < k$. This implies that $p_1 \prec p_2 \prec \ldots \prec p_k = p = f_h \prec f_{h+1} \prec \ldots \prec f_n$. However, this chain has length $k + n - h > n$ since $k > h$. This contradicts the fact that P can be partitioned into n antichains. □

We have defined the necessary and sufficient properties posets need to have so they can be consistently evaluated. It turns out that not all of these posets are "intuitive" as shown in Example 6.3.4. This prompts us to introduce a class of posets that are consistently evaluable but also easier to deal with.

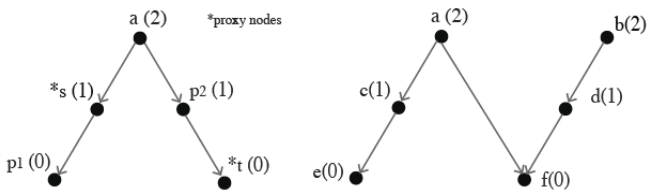


Figure 6.3: Both posets admit consistent evaluators. The value the consistent evaluator assigns on each singleton that consists of the node is written in parentheses. Note that the poset on the left has the additional graded property while the one on the right does not.

Example 6.3.4. Consider the posets in Fig. 6.3. Any evaluator f that assigns the values according to the values in the parentheses, shown in the figure, can easily

be verified to have properties 1-4 of consistent evaluation. On the left poset, we can also see that property 5 is also satisfied since for every pair of nodes such that $f(p_1) < f(p_2)$ implies that there exist proxy nodes s and t such that $f(p_1) = f(t)$, $f(p_2) = f(s)$, $p_1 \prec s$, and $p_2 \prec t$. This relation can be easily seen for the nodes p_1 and p_2 in Fig. 6.3. The same statement applies to the poset on the right.

Definition 6.3.3 (Specification structure)**.** A *specification structure* is a finite, graded, partially ordered set of dimensional properties $\mathcal{P}$. Namely, if $\preceq$ is the ordering relation for $\mathcal{P}$ and $\prec$ is the strict version thereof satisfying $x \prec y \Leftrightarrow (x \preceq y \wedge x \neq y)$, then there exists a ranking function $\rho : \mathcal{P} \to \mathbb{N}$ such that

1. $p_1 \prec p_2 \Rightarrow \rho(p_1) < \rho(p_2)$.
2. $p_1 \lessdot p_2 \Rightarrow \rho(p_2) = \rho(p_1) + 1$.
3. p is a minimal element of $\mathcal{P} \Rightarrow \rho(p) = 0$.

where $\lessdot$ denotes the *covering relation* on $\mathcal{P}$ that satisfies

$$p_1 \lessdot p_2 \Leftrightarrow p_1 \prec p_2 \wedge \forall p \in \mathcal{P} :: \neg(p_1 \prec p \wedge p \prec p_2).$$

We immediately have the following corollary.

Corollary 2. *Any specification structure can be consistently evaluated.*

Proof. This follows directly from Theorem 1 and the fact that any graded poset has properties 1) and 2) defined therein. □

The following lemma is a standard result.

Lemma 4. *A poset is graded if and only if all of its maximal chains have the same length.*

It turns out that any consistently evaluable poset can be reduced to a "canonical" form that has the graded property and the same exact consistent evaluation.

Theorem 3. *Each consistently evaluable poset can be turned into a graded poset that is equivalent under consistent evaluation.*

Proof (Sketch). This is achieved by removing all "edges" that span more than 2 levels of antichains in the unique partition of Theorem 1. One can without much difficulty verify that doing so will remove all maximal chains with length strictly less than the total number of these antichains, which by Lemma 4 implies that the resulting poset is graded. Since the other antichains are not affected by these operations, the resulting evaluation is not affected either. □

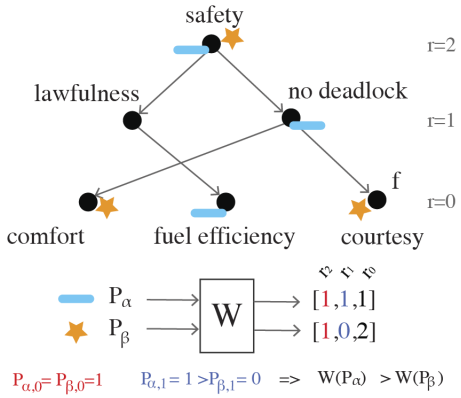


Figure 6.4: This shows how the consistent evaluator function W works on a specification structure. The function W computes a tuple for each subset, and compares the elements from most significant to least significant digits (left to right).

Example 6.3.5 (Evaluating a specification structure)**.** Let `S`, `L`, `ND`, `FE`, `Cf`, `C` be dimensional properties denoting safety, lawfulness, no deadlock, fuel efficiency, comfort, and courtesy respectively. Let $P := \{\mathtt{S}, \mathtt{L}, \mathtt{ND}, \mathtt{FE}, \mathtt{Cf}, \mathtt{C}\}$. The partial order on these dimensional properties is shown in Fig. 6.4. Given the current world configuration, we assume that the oracle can determine which subset of specifications will be satisfied by taking a given action. Let $P_\alpha := \{\mathtt{S}, \mathtt{ND}, \mathtt{L}\}$ denote the subset of specifications satisfied by taking action α. Similarly, let $P_\beta := \{\mathtt{S}, \mathtt{Cf}, \mathtt{C}\}$. To compare the actions α and β, given P_α, P_β, we use the evaluator W defined in the proof of Theorem 1 to make the comparison. $W(P_\alpha) = [1, 1, 1]$ since there is one specification from each rank that can be satisfied by taking the action α, and $W(P_\beta) = [1, 0, 2]$ since there are two properties with rank 0 and one property with rank 2 that can be satisfied by taking action β. Therefore, to evaluate their relative importance, the most significant figure corresponds to the left-most element in the tuple since that element has the highest rank. We begin our comparison there. Note that W_i represents the ith the element of the tuple. Since $W_0(P_\alpha) = 1$ and $W_0(P_\beta) = 1$, we have to keep comparing elements in the tuple to determine which one has higher ordering. We find $W_1(P_\alpha) > W_1(P_\beta)$. Therefore, P_α dominates P_β

by the weak order imposed by the W evaluator, and therefore, the action α should be chosen over β.

6.4 Consistency and Coverage

We would like to introduce the ideas of consistency and coverage of a set of specifications that are hierarchically-ordered in a specification structure. In this context, consistency is defined as the ability to be uniquely and consistently evaluable. Coverage is defined by the extent of the dimensional properties specified. A specification structure that has more dimensional properties (i.e. encompasses a broader range of specifications) therefore has more coverage.

Consistency

The notion of consistency comes from Theorem 4, which says that there is a unique weak order on the powerset of a specification structure regardless of the consistent evaluator used.

Theorem 4 (Consistency implies uniqueness)**.** *If $\mathcal{P}$ is a poset with an ordering relation $\preceq$ that can be consistently evaluated, then all consistent evaluators of $\mathcal{P}$ are equivalent. That is, for any pair of consistent evaluators f_a, f_b of $\mathcal{P}$, for all $P_1, P_2 \subseteq \mathcal{P}$, we have*

$$f_a(P_1) \leq f_a(P_2) \Leftrightarrow f_b(P_1) \leq f_b(P_2).$$

Proof. By symmetry, it is sufficient to prove the ($\Rightarrow$) direction. Suppose $f_a(P_1) \leq f_a(P_2)$. Now since $\mathcal{P}$ is consistently evaluable, by Theorems 1 and 2, it can be partitioned by a unique set of maximal antichains $\{A_r\}_{r=1}^{R}$. By requirement 4 of Definition 6.3.2, one can show that any consistent evaluator will rank these antichains the same way. Namely, any consistent evaluator f of $\mathcal{P}$ can be assumed, WLOG, to satisfy the following conditions

1. $f(\{p_1\}) < f(\{p_2\}) < \ldots < f(\{p_R\})$, for $p_r \in A_r, r \in \{1, \ldots, R\}$.
2. $f(\{q_i\}) = f(\{r_i\})$, for any $q_r, r_r \in A_r, r \in \{1, \ldots, R\}$.

Condition 2 above implies that all pairs of nodes that are of equal value to f_a are also of equal value to f_b and vice versa. So by requirement 2 of Definition 6.3.2, we can assume that P_1 and P_2 do not overlap in property values due to either f_a or f_b. If $P_1 = \varnothing$, then by requirement 1, we have $f_b(P_1) < f_b(P_2)$. Otherwise, by

requirement 3, let $p_i^\star \in P_i$ be the property that maximizes the value of f on P_i, we have $f_a(P_1) \leq f_a(P_2)$, which implies that $p_1^\star \prec p_2^\star$ since $f_a(p_1^\star) \neq f_a(p_2^\star)$ due to P_1 and P_2 not overlapping in property values and therefore $f_b(P_1) < f_b(P_2)$. □

Coverage

While the placement of the *edges* of the directed graph presenting a specification structure determines its consistency, inclusion (or exclusion) of *nodes* determines its coverage. In order to increase the coverage of an existing specification structure, we must be able to refine the graph in a consistent manner. Refinement is equivalent to adding dimensional properties (nodes) or comparisons (edges) to the specification structure in a way that preserves the gradedness property of a specification structure. We now define how to properly add a node or edge into the specification structure in a way that preserves the specification structure's mathematical properties. The following is a direct corollary of Lemma 4.

Corollary 3 (Proper mode or edge refinement)**.** *If a node (or an edge) is added to the specification structure such that its relationship to the other nodes (the comparison it makes) is defined in a way that all maximal chains have the same length, then the resulting partially ordered set is also a specification structure.*

Examples for proper (and improper) ways of adding a new node as well as examples of making minimal modifications to accommodate for an inconsistently-added node are included in (Phan-Minh, Cai, and Murray, 2019).

Example 6.4.1. Here, we give a very simple specification structure: lawfulness (L) $\prec$ no deadlock (ND) $\prec$ safety (S). We consider the consistent evaluator W (presented in the proof of Theorem 1). W will have the ordering $W(\{\mathtt{L}\}) < W(\{\mathtt{ND}\}) < W(\{\mathtt{S}\}) < W(\{\mathtt{S},\mathtt{L}\}) < W(\{\mathtt{S},\mathtt{ND}\}) < W(\{\mathtt{S},\mathtt{L},\mathtt{ND}\})$. The ordering intuitively means that a car should always prioritize taking actions that satisfy all three types of specifications. However, if there is a situation where a car cannot ensure safety without breaking the law, then it should break the law to maintain safety since $W(\{\mathtt{S}\}) > W(\{\mathtt{L}\})$. Also, this hierarchy says if there is a situation where the car is in a deadlock, it can break the law since $W(\{\mathtt{S},\mathtt{ND}\}) > W(\{\mathtt{S},\mathtt{L}\})$ as long as the action is still safe.

As long as the car chooses behaviors that respect the weak order from the consistent evaluator on the specification structure, the system will satisfy the guarantees part of the assume-guarantee contract, and therefore perform actions that are "correct."

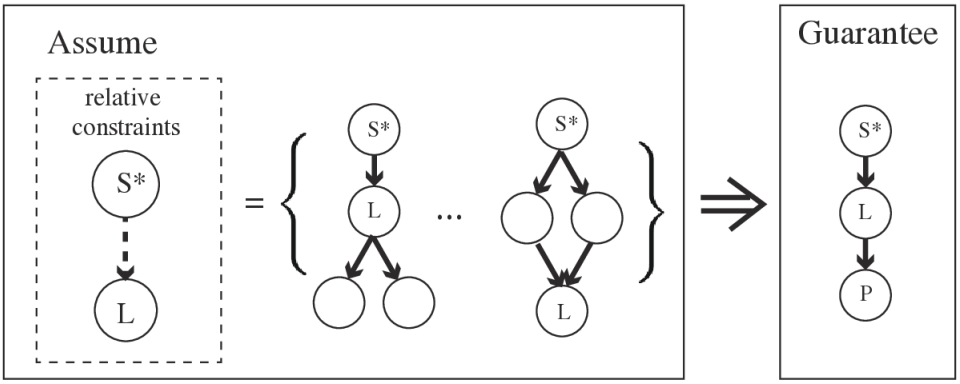


Figure 6.5: The assumptions are based on a set of specifications structures that satisfy some constraints, which is shown on the left. The guarantees are based on a single specification structure that is shown on the right.

We now introduce how assumptions can be defined with respect to the specification structure.

6.5 Assume-Guarantee Profiling

While each autonomous vehicle should only guarantee that it will behave according to a single specification structure, we want our assumptions on the environment (other agents) to accommodate for the diverse behaviors displayed by human drivers who may not follow the law all the time. This implies that other agents might choose to follow any one of a large number of possible specification structures. In order to make any guarantees about safety, we have to make some assumptions on how other agents might behave. We constrain the set of specifications structures of other agents to always prioritize safety first. Since other agents presumably follow the law most of the time, we also include a relative ordering constraint where safety is prioritized before the law. We have only defined a relative ordering between safety and law in the assumptions since we do not exactly know where other dimensional properties will fit within that agent's specification structure. Therefore, our assumptions on the environment can be defined as follows:

Definition 6.5.1 (Assumption set)**.** Let S denote the set of all specification structures. Let P be a set of dimensional properties. Let $p \in P$. The assumptions in the assume-guarantee contract is defined as:

$$A_{\text{spec}} = \{S_i \in S | (\text{safety} \in S_i) \wedge (\text{lawfulness} \in S_i) \wedge (\forall p \in S_i :: p \preceq \text{safety})\}.$$

It is the set of all specification structures that both safety and lawfulness are in-

cluded in the specification structure and that safety has the highest rank out of all dimensional properties included in the specification structure.

The following revised assume-guarantee definition of Definition 6.2.1 characterizes the set of specification structures agents in the environment can be assumed to have and the specifications that an individual self-driving car can guarantee.

Definition 6.5.2 (Assume-guarantee profiling revised)**.** An assume-guarantee contract C defined for an agent is a pair $(\mathcal{A}, \mathcal{G})$, where

1. $\mathcal{A}$ is a set of specification structures for the agent's environment that is a subset of the set generated by Definition 6.5.1.
2. $\mathcal{G}$ is the guarantee of the agent in the form of a single, pre-defined specification structure.

This assume-guarantee profiling is shown in Fig. 6.5. Let $\mathcal{J}$ be the index set for a set of agents. For $C_j = (\mathcal{A}_j, \mathcal{G}_j)$, let j be the index of an agent, $\mathcal{A}_j$ be the assumptions that agent j is making about its environment, and $\mathcal{G}_j$ be its guarantees. We say that the group of agents indexed by $\mathcal{J}$ are *compatible* if

$$\forall j \in \mathcal{J} :: \forall i \in \mathcal{J} - \{j\} :: \mathcal{G}_j \in \mathcal{A}_i.$$

This says that the guarantees of agent j must be included in the assumptions of all other agents in the compatible set. If one agent i has guarantees corresponding to a specification structure that is not included in another agent k's assumptions, then correct behavior cannot be guaranteed. Assuming that all agents' assumptions and guarantees are compatible, we can formulate the notion of a *blame-worthy* action/strategy.

Definition 6.5.3 (Blameworthy action)**.** A blameworthy action/strategy is one in which an agent violates its guarantees, thereby causing another agent's assumptions not to be satisfied and thus resulting in an unwanted situation where blame must be assigned.

In order to show an example of an assume-guarantee contract that might be legally imposed for self-driving cars, we present a *set of axioms for the road*. The specification structures defined in the assumptions and guarantees of this contract are intentionally left unrefined, since it would ultimately be up to a car-manufacturer to determine the remaining ordering of specification properties.

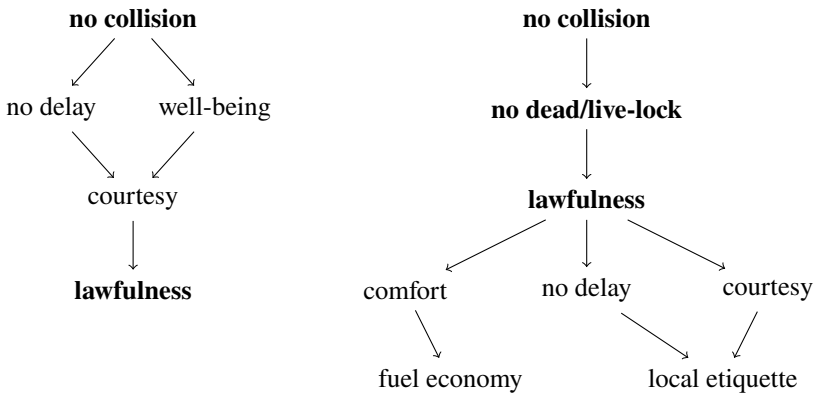


Figure 6.6: Two examples of refined assumption (left) and guarantee (right) specification structures. Dimensional properties of the root structures are in bold text. The left structure may correspond to that of an ambulance while the right structure may correspond to a civilian vehicle.

A1 *Other agents will not act such that collision is inevitable.*

A2 *Other agents will often act corresponding to traffic laws, but will not always follow them.*

G1 *An agent will take no action that makes collision inevitable.*

G2 *An agent will follow traffic laws, unless following them leads to inevitable collision.*

G3 *An agent may violate the law if that can safely get it out of a dead/live-lock situation.*

We can see from Fig. 6.6 how these axioms have a direct mapping to a specification structure. We argue that this sort of root structure might be imposed by a governing body to ensure the safe behaviors of self-driving cars.

6.6 Some Examples and the Definition of Completeness

The assume-guarantee contract over specification structures allows us to formally characterize the set of correct behaviors for self-driving cars and human-operated vehicles. In this section, we present some preliminary examples of how these types of high-level behavioral specifications might be applied in some traffic scenarios and use them to motivate the definition of completeness. Under the simplified assumption that each agent has a single specification structure (i.e. agents are not human), each agent will have a well-defined ordering of which actions have higher

value, and will therefore have a well-defined utility function over actions. Game theory provides a mathematical model of strategic interaction between rational decision-makers that have known utility functions (Fudenberg and Tirole, 1991). We can therefore use game-theoretic concepts to analyze which pair of actions will be jointly advantageous for the agents given their specification structures.

Example 6.6.1. Consider the case where there are two agents, each of whose specification structures are specified in Fig. 6.7. In this game, Player Y encounters some debris, and must choose an action. Player Y can either choose to stay in its current location, or do a passing maneuver that requires it to break the law. Player X represents a car moving in the opposite direction of Player Y. In this case, Player X can either move at its current velocity or accelerate. The move and accelerate action make Player X move one and two steps forward, respectively. The W function is the same as one in the proof of Theorem 1.

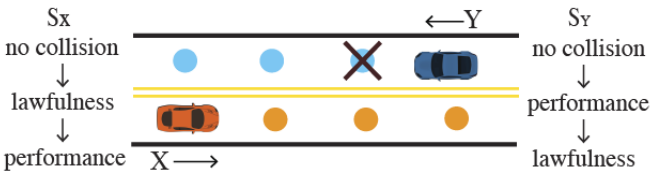


Figure 6.7: The game scenario when Player Y encounters debris on its side of the road. The specification structures of each of the agents are given by S_x and S_y.

W_x is evaluated on the specification structure S_x shown on the left side of Fig. 6.7 and W_y is evaluated on S_y. Assuming there is a competent oracle who gives the same prediction for both agents, the resulting payoff matrix according to the specification structures are given in Table 6.1 (note that an equivalent decimal conversion of the scores is given for ease of reading).

Table 6.1: Obstacle game.

playerX/playerY	Stay	Pass
Move	$W_x(1,1,0) \sim 6$ $W_y(1,0,1) \sim 3$	$W_x(1,1,0) \sim 6$ $W_y(1,1,0) \sim 6$
Accelerate	$W_x(1,1,1) \sim 7$ $W_y(1,0,1) \sim 3$	$W_x(0,0,0) \sim 0$ $W_y(0,0,0) \sim 0$

From the table, we can see that there are two Nash equilibria in this game scenario. The two equilibria are Pareto efficient, meaning there are no other outcomes where one player could be made better off without making the other player worse off.

Since there are two equilibria, there is ambiguity in determining which action each player should take in this scenario despite the fact that the specification structures are known to both players. This shows that compatibility is not enough to have completeness.

There is a whole literature on equilibrium selection (Fudenberg and Tirole, 1991). The easiest way to resolve this particular stand-off, however, would be to either 1) communicate which action the driver will take or 2) define a convention that all self-driving cars should have when such a situation occurs. In this particular scenario, however, Player X can certainly avoid accident by choosing to maintain speed while Player Y can also avoid accident by staying. Any "greedy" action of either Player X or Y may pose the risk of crashing depending on the action of the other player. This suggests a risk-averse resolution in accident-sensitive scenarios like this one. Alternatively, we can define new rules/specifications to refine the agents' specification structures so that their optimal actions either imply one of the two cases above.

Example 6.6.2. For this chapter, we have abstracted the perception system of the self-driving car to the all-knowing oracle. We first consider the case where the oracles on each of the cars are in agreement, and then consider the potential danger when the oracles of the cars differ. In this scenario, we assume that there are two cars that are entering an intersection with some positive velocity, as shown in Fig. 6.8.

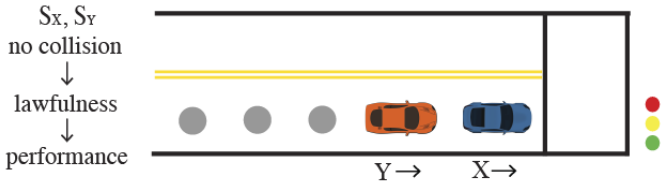


Figure 6.8: The game scenario where two cars are approaching an intersection, but have different beliefs about the state of the traffic light.

In the case where both vehicles' oracles agree on the same information, i.e. that the yellow light will remain on for long enough for both vehicles to move past the intersection, the best action for both Player X and Player Y is to move forward.

Now, consider the case where the oracles are giving incompatible beliefs about the environment, namely, the state of the traffic signal. Let X have the erroneous belief the traffic light will turn red very soon, and it assumes that Y's oracle believes the

Table 6.2: Red light game 1.

playerX/playerY	Slow	Move
Slow	$W_x(1,1,0) \sim 6$ $W_y(1,1,0) \sim 6$	$W_x(0,1,0) \sim 2$ $W_y(0,0,0) \sim 0$
Move	$W_x(1,0,1) \sim 3$ $W_y(1,1,0) \sim 6$	$W_x(1,0,1) \sim 3$ $W_y(1,0,1) \sim 3$

same thing. X's oracle gives rise to Table 6.2, according to which the conclusion that Player X will make is that both of the cars should choose to slow down.

Assume that Y has a perfect oracle that predicts that the traffic light will stay yellow for long enough such that Y would also be able to make it through the intersection. If Y assumes that X has the same information (see Table 6.3), then the best choice for both is to move forward into the intersection.

Table 6.3: Red light game 2.

playerX/playerY	Slow	Move
Slow	$W_x(1,1,0) \sim 6$ $W_y(1,1,0) \sim 6$	$W_x(0,1,0) \sim 2$ $W_y(0,0,0) \sim 0$
Move	$W_x(1,1,1) \sim 7$ $W_y(1,1,0) \sim 6$	$W_x(1,1,1) \sim 7$ $W_y(1,1,1) \sim 7$

The incompatible perception information will thus cause Player X to stop and Player Y to move forward, ultimately leading to collision. This particular collision is not defined by incorrectly-specified behavior, but is instead caused by errors in the perception system.

These examples motivate the following definition of completeness.

Definition 6.6.1 (Completeness)**.** A set of specification structures of a group of agents are *complete* if in all scenarios, assuming perfect sensor input, the optimal action returned by the oracle for each agent with respect to their respective specification structure can be chosen without compromising safety (no collision) and liveness (each agent will eventually reach its goal).

6.7 Conclusion

To summarize, we have introduced a framework that allows us to formulate specifications that govern high-level behaviors of autonomous vehicles. If specifications are hierarchically ordered into a specification structure, actions and strategies can

be compared to one another in a consistent manner. Furthermore, the coverage of specification structure can be increased by properly defining new properties and relations. We introduce the idea of having assume-guarantee contracts defined over these specification structures that serve as profiles for implicit agreement. A contract essentially says that if the environment can be assumed to behave according to some softly-constrained specification structure, then the self-driving car can guarantee that it will behave according to its own specification structure. Blame is defined as the case where a car does not act according to its assumed specification structure. Finally, we provide some examples of how cars following these specification structures might behave in game-theoretic experiment settings.

In the next chapter, we will present an application of this framework to a system of autonomous (vehicles) agents in which liveness and safety can be formally guaranteed.

References

Baier, Christel and Joost-Pieter Katoen (2008). *Principles of Model Checking*. Cambridge, Massachussetts: MIT Press.

Censi, Andrea, Konstantin Slutsky, Tichakorn Wongpiromsarn, Dmitry Yershov, Scott Pendleton, James Fu, and Emilio Frazzoli (Feb. 2019). "Liability, Ethics, and Culture-Aware Behavior Specification using Rulebooks." In: *arXiv e-prints*, arXiv:1902.09355.

Fudenberg, Drew and Jean Tirole (1991). *Game Theory*. MIT Press.

Lefèvre, Stéphanie, Dizan Vasquez, and Christian Laugier (2014). "A survey on motion prediction and risk assessment for intelligent vehicles." In: *ROBOMECH* 1.1, p. 1.

Madani, Omid, Steve Hanks, and Anne Condon (2003). "On the undecidability of probabilistic planning and related stochastic optimization problems." In: *Artificial Intelligence* 147.1-2, pp. 5–34.

Paden, Brian, Michal Cap, Sze Z. Yong, Dmitry Yershov, and Emilio Frazzoli (Mar. 2016). "A Survey of Motion Planning and Control Techniques for Self-Driving Urban Vehicles." In: *T-IV* 1.1, pp. 33–55. ISSN: 2379-8904.

Papadimitriou, Christos H. and John N. Tsitsiklis (1987). "The complexity of Markov decision processes." In: *Mathematics of operations research* 12.3, pp. 441–450.

Phan-Minh, Tung, Karena X. Cai, and Richard M. Murray (2019). *Towards Assume-Guarantee Profiles for Autonomous Vehicles*. Tech. rep. `https://www.cds.caltech.edu/\%7emurray/preprints/pcm19-acc_tr_s.pdf`, accessed March 2019. California Institute of Technology.

Shalev-Shwartz, Shai, Shaked Shammah, and Amnon Shashua (Aug. 2017). "On a Formal Model of Safe and Scalable Self-driving Cars." In: *arXiv e-prints*, arXiv:1708.06374, arXiv:1708.06374. arXiv: 1708.06374 [cs.RO].

Sipser, Michael (2012). *Introduction to the Theory of Computation*. Cengage Learning.

Wongpiromsarn, Tichakorn, Sertac Karaman, and Emilio Frazzoli (Oct. 2011). "Synthesis of provably correct controllers for autonomous vehicles in urban environments." In: *ITSC*, pp. 1168–1173.

Chapter 7

RULES OF THE ROAD[1]

7.1 Introduction

It is difficult to imagine that autonomous vehicles will ever be integrated into our society unless we can be assured of their safety and efficacy. The current prevailing methodology used for proving safety of these vehicles is simulating and test-driving these vehicles for millions of miles, which is a practice that lacks both formal verifiability and scalability.

Formal methods offers tools for designing provably correct control strategies for complex systems like autonomous vehicles that satisfy high-level behavioral specifications like safety and liveness (Baier and Katoen, 2008) for each individual vehicle. The algorithms used for synthesizing formally-correct strategies for the vehicles, however, cannot guarantee global safety since they do not make the assumptions that must hold on other vehicle behaviors explicit (Wongpiromsarn, Karaman, and Frazzoli, 2011; Censi, Slutsky, et al., 2019; Tumova et al., 2013).

Instead of reasoning about safety on the individual agent level, Shoham and Tennenholtz introduced the idea of reasoning about safety as a property of a collection of agents (Shoham and Tennenholtz, 1995). In particular, they introduce the idea of social laws, which are a set of rules imposed upon all agents in a multiagent system to ensure some desirable global behaviors like safety or progress (Shoham and Tennenholtz, 1995; van Der Hoek, Roberts, and Wooldridge, 2007). The design of social laws is intended to achieve the desirable global behavioral properties in a minimally-restrictive way (Shoham and Tennenholtz, 1995). The problem of automatically synthesizing useful social laws for a set of agents for a general state space, however, has been shown to be NP-complete (Shoham and Tennenholtz, 1995).

The Responsibility-Sensitive-Safety (RSS) framework (Shalev-Shwartz, Shammah, and Shashua, 2017) adopts a similar top-down philosophy for guaranteeing safety by providing a set of rules, which if followed by all agents, guarantees no collisions. In the case of an accident, the responsible party can be identified. This frame-

[1]The material in this chapter comes from joint work with Karena X. Cai, Soon-Jo Chung, and Richard M. Murray.

work, however, does not take into account the progress of vehicles towards their destinations.

The problem of fully guaranteeing safety and liveness of decision-making agents is especially challenging since 1) agents are often competing for the same set of resources (some region of the road network) and 2) agents must reason about highly-coupled and complex interactions with other agents. Historically, interactive partially observable Markov Decision Processes (I-POMDPs) have been proposed to model these complex interactions—but these methods lack scalability (Gmytrasiewicz and Doshi, 2005; Papadimitriou and Tsitsiklis, 1987). More recently, learning approaches have been proposed, but they require large amounts of data while failing to provide any safety or liveness guarantees (Sadigh, Sastry, et al., 2016; Fisac et al., 2019; Finn, Levine, and Abbeel, 2016; Sadigh, Dragan, et al., 2013).

The process for resolving multiple conflicting processes in a local, decentralized manner is addressed in the Drinking Philosopher problem, which provides a mechanism for resolving issues arising from synchronous decision-making (Chandy and Misra, 1984). The solution to the Drinking Philosopher problem is an algorithm that assigns precedence among a set of agents that have conflicting goals. The algorithm preserves acyclicity and fairness in the precedence graph, thereby ensuring consistency and fairness among all agents in the game. Our work is an adaptation of the Drinking Philosopher problem to the multi-agent collision-avoidance problem on a road network, where the resource agents compete for road occupancy. We design a decision-making strategy that defines how agents choose their actions. Minimal communication among agents allows each agent to consistently establish precedence and resolve conflicts in a local, decentralized manner. Unlike previous work by Sahin and Ozay (2020), our framework leverages the structure of the driving road network and takes into account the inertial properties of agents. Furthermore, we can guarantee safety and liveness if all agents operate according to the specified decision-making strategy. Lastly, our approach is closely related to the token-based conflict resolution analysis for interactions between autonomous agents that was studied in (Censi, Bolognani, et al., 2019), but does not only consider pairwise interactions.

The main contributions of this chapter are as follows:

1. The introduction of a new game paradigm, which we term the quasi-simultaneous

discrete-time multi-agent game.

2. The definition of an agent protocol that defines local rules agents must follow in two different contexts on a road network (i.e. road segments and intersections).

3. Safety and liveness proofs when all agents operate according to these local rules.

4. A simulated environment that serves as a proof of concept for the safety and liveness guarantees.

7.2 Overview

In this overview, we give a high-level presentation of the main contributions in this work. All the terms will be formalized and described in further depth in later sections of the chapter.

The Quasi-Simultaneous Discrete-Time Game

In this chapter, we introduce the quasi-simultaneous discrete-time multi-agent game. The quasi-simultaneous game is a modification of turn-based games so that turn order is defined locally and induced by the agent states as opposed to being fixed and/or predefined. This new game format leverages the structure of the game environment to assign precedence among agents. In this way, it partially constrains the set of actions an agent can choose from based on the agent environment (via the order the agent gets to take its turn in). The quasi-simultaneous game models the agent's decision-making process in a multi-agent game differently than traditional simultaneous or turn-based games found in (Gmytrasiewicz and Doshi, 2005) and (Fisac et al., 2019), and to the best of the authors' knowledge have not been introduced in the literature before.

The Agent Protocol

The Agent Protocol is defined to establish local rules agents must follow while making decisions on the road network.

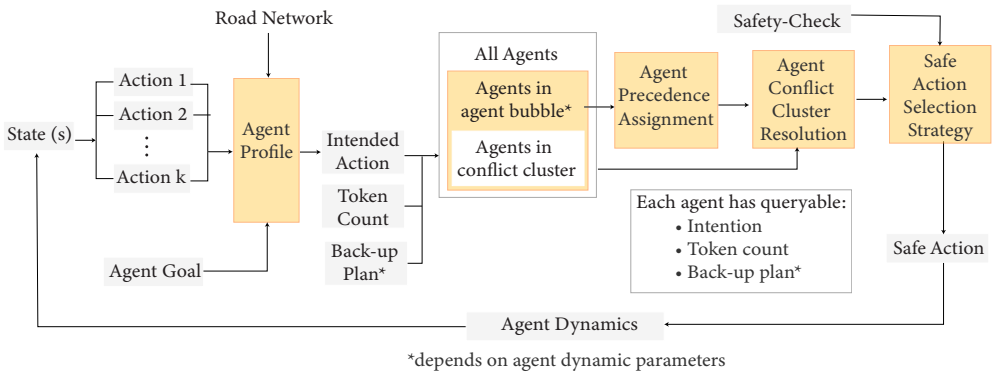


Figure 7.1: Agent Protocol Architecture.

The novelty of our work is the introduction of a single backup plan action that all agents rely on. Dependence on this backup plan action is what ultimately allows for the decoupling of agent dependencies when reasoning about one another, while still allowing for strong global guarantees on safety and liveness. The following is an overview of the Agent Protocol. For each time step of the game, each agent first assigns local precedence according to the rules described in Section 7.4, thereby establishing a consistent turn order (in a local manner). Then, each agent evaluates a set of actions and chooses the best one according to their own Agent Profile (thereby selecting their intended action), described in Section 7.4. Since the turn-order does not fully resolve ambiguity regarding which agents should be allowed to take their intended action at a given time, the action selection strategy, based on 1) what actions agents ahead of it (in turn) have taken and 2) the results of the conflict-cluster resolution described in 7.4, defines how agents should ultimately decide which action to select. The next sections of this chapter will formalize these ideas and go into greater depth of how these ideas work for a specific class of agents with a particular set of dynamics. Note that specific assumptions specified in Sections 7.5 and 7.6 must hold on the road network for the guarantees to hold.

7.3 Quasi-Simultaneous Discrete-Time Multi-Agent Game

We formalize the definition of a quasi-simultaneous discrete-time multi-agent game as follows. A *state* associated with a set of variables is an assignment of values to those variables. A game evolves by a sequence of state changes. A quasi-simultaneous game has the following two properties regarding state changes: 1) Each agent will get to take a turn in each time-step of the game and 2) Each agent must make their turn in an order that emerges from a locally-defined precedence

assignment algorithm (where locality is described in Section 7.4). Thus, the state-change is simultaneous yet locally sequential because each agent must make a state-change in a given time step, but it must "wait" for its turn according to turn order (defined based on the locally-defined precedence assignment algorithm) during this time-step. Let us define some preliminaries before formally defining the game.

We define a quasi-simultaneous game where all agents act in a local, decentralized manner as follows

$$\mathfrak{G} = \langle \mathfrak{A}, \mathcal{Y}, Act_{[\cdot]}, \rho_{[\cdot]}, \tau_{[\cdot]}, P \rangle \tag{7.1}$$

where

- $\mathfrak{A}$ is the set of all agents in the game $\mathfrak{G}$.
- $\mathcal{Y}$ is the set of all variables in the game $\mathfrak{G}$.
  - Let $\mathcal{U}$ be the set of all $\mathcal{Y}$-states, i.e. all possible assignment of values (states) of the game.
  - Given a subset $Y \subseteq \mathcal{Y}$, denote by $\mathcal{U} \mid_Y$ the projection of $\mathcal{U}$ onto Y, i.e. all possible states of the variables in the set Y.
- For each agent $\mathrm{Ag} \in \mathfrak{A}$, let:
  - S_{Ag} be the set $\mathcal{U} \mid_{\mathcal{V}_{\mathrm{Ag}}}$ that contains all possible states of Ag.
  - Act_{Ag} be the set of all possible actions Ag can take.
  - $\tau_{\mathrm{Ag}} : S_{\mathrm{Ag}} \times Act_{\mathrm{Ag}} \to S_{\mathrm{Ag}}$. τ_{Ag} be the transition function that defines the state an agent will transition to when taking an action $a \in Act_{\mathrm{Ag}}$ from a given state.
  - $\rho_{\mathrm{Ag}} : S_{\mathrm{Ag}} \to 2^{Act_{\mathrm{Ag}}}$ be a state-precondition function that defines a set of actions an agent can take at a given state.
- $P : \mathcal{U} \to \mathrm{PolyForest}(\mathfrak{A})$, is the precedence assignment function where PolyForest is an operator that maps a set X to the set of all forests (undirected graphs whose connected components are trees) defined on the set X. The polyforest defines the global turn order (of precedence) of the set of all agents $\mathfrak{A} \in \mathfrak{G}$ based on the agent states.

Note, the transition function τ_{Ag} and the state-precondition function ρ_{Ag} must be compatible for any agent Ag. In particular,

$$\forall \mathrm{Ag} \in \mathfrak{A}. \forall s \in S_{\mathrm{Ag}}. \mathrm{Domain}(\tau_{\mathrm{Ag}}(s, \cdot)) = \rho_{\mathrm{Ag}}(s).$$

Agent Game Environment

Let us introduce the game environment for the agents.

Definition 7.3.1 (Road Network)**.** A road network $\mathfrak{R}$ is a graph $\mathfrak{R} = (G, E)$ where G is the set of grid points and E is the set of edges that represent immediate adjacency in the Cartesian space among grid points. Note that each grid point $g \in G$ has a set of associated properties $\mathcal{P}$, where $\mathcal{P} = \{p, d, \texttt{lo}\}$ which denote the Cartesian coordinate, drivability of the grid point, and the set of legal orientations allowed on the grid point, respectively. Note, $p \in \mathbb{Z}^2$, $d \in \{0, 1\}$ and $\texttt{lo}$ is a set of headings ϕ_l where each $\phi_l \in \{\texttt{north}, \texttt{east}, \texttt{south}, \texttt{west}\}$.

Let us define $\mathfrak{R}|_{\text{legal}} = (G, E|_{\text{legal}})$ to be a road network where a directed edge $e = (g_1, g_2)$ implies a legal (based on legal orientations) and dynamically-feasible transition (according to ρ_{Ag}) exists between the grid points $g_1, g_2 \in G$.

Special Grid Point Sets

Grid points where specific properties hold are given special labels, which can be seen in Fig. 7.2. These labels and the associated properties are defined as follows:

- $\mathcal{S}_{\text{sources}}$, ($\mathcal{S}_{\text{sinks}}$): A set of grid points designated for Ag to enter (or leave) the road network $\mathfrak{R}$ from. Any $s \in \mathcal{S}_{\text{sources}}$ ($s \in \mathcal{S}_{\text{sinks}}$) must be a grid point with no inbound (outbound) edges in $\mathfrak{R}\ |_{\text{legal}}$.
- $\mathcal{S}_{\text{intersection}}$: A set of grid points that contains all grid points with more than one legal orientation.
- $\mathcal{S}_{\text{traffic light}}$: A set of grid points that represent the traffic light states in the vertical or horizontal direction via its color (for every intersection).

Road Network Decomposition

The road network is hierarchically decomposed into lanes and bundles, which are defined as follows:

- Lanes and Bundles: Let lane $La(g)$ define a set of grid points that contains g and all grid points that form a line going through g. Let $Bu(g)$ be a set of grid points that make up a set of lanes that are adjacent or equal to the lane containing g and have the same legal orientation.

- Road Segments RS: Each bundle can be decomposed into a set of road segments (separated by intersections). For each bundle, let us define a graph $G_{RS} = (G, E)$ where G is a set of grid points and E is a set of edges between any two grid points g_1, g_2 where $g_1, g_2 \notin \mathcal{S}_{\text{intersection}}$ and $Bu(g_1) = Bu(g_2)$. Road segments $Rs(g)$ is the set of all grid points that are in the same connected component as the grid point g in the graph G_{RS}.

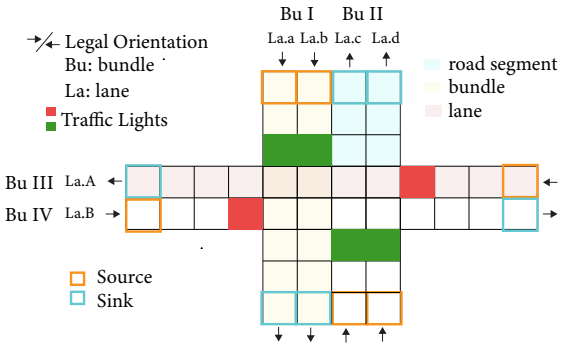


Figure 7.2: Road network decomposition where each box represents a grid point.

We introduce the following graph since it will be used in the liveness proof:

Definition 7.3.2 (Road Network Dependency Graph)**.** We define a dependency graph $G_{\text{dep}} = (RS, E)$, RS is the set of road segments in $\mathfrak{R}$, and a directed edge $e : (r_1, r_2)$ for $r_1, r_2 \in RS$ implies that an agent Ag whose $x - y$ position is $(s.x_{Ag}, s.y_{Ag}) \in r_1$ depends on the clearance of some agent Ag′ whose $x - y$ position $(s.x_{Ag'}, s.y_{Ag'}) \in r_2$, where $s.x_{Ag}$ and $s.y_{Ag}$ are the x and y positions of agent Ag and are defined formally in Section 7.4.

For clarity of the road network decomposition, refer to Fig. 7.2.

7.4 Agent Protocol

In the following section, we present the set of attributes agents must have and the set of rules agents must adhere to in order to satisfy our proposed Agent Protocol.

Agent Attributes

Each *agent* Ag is characterized by a set of variables $\mathcal{V}_{\text{Ag}}$ such that

$$\{\texttt{Id}_{\text{Ag}}, \texttt{Tc}_{\text{Ag}}, \texttt{Goal}_{\text{Ag}}\} \subseteq \mathcal{V}_{\text{Ag}}$$

where $\texttt{Id}_{\text{Ag}}$, $\texttt{Tc}_{\text{Ag}}$, and $\texttt{Goal}_{\text{Ag}}$ are the agent's ID number, token count, and goal, respectively, where the token count and ID are used in the conflict-cluster resolution defined in Section 7.4.

In this chapter, we only consider *car* agents such that if $\text{Ag} \in \mathfrak{A}$, then $\mathcal{V}_{\text{Ag}}$ includes x_{Ag}, y_{Ag}, θ_{Ag}, v_{Ag}, namely its absolute coordinates, heading, and velocity. $\mathcal{V}_{\text{Ag}}$ also has parameters:

$$a_{\text{minAg}} \in \mathbb{Z}, a_{\text{maxAg}} \in \mathbb{Z}, v_{\text{minAg}} \in \mathbb{Z} \text{ and } v_{\text{maxAg}} \in \mathbb{Z}$$

which define the minimum and maximum accelerations and velocities, respectively.

The agent control actions are defined by two parameters: 1) an acceleration value acc_{Ag} between a_{minAg} and a_{maxAg}, and 2) a steer maneuver $\gamma_{\text{Ag}} \in$ {`left-turn`, `right-turn`, `left-lane change`, `right-lane change`, `straight`}.

The discrete agent dynamics works as follows. At a given state s at time t, for a given control action $(\text{acc}_{Ag}, \gamma_{Ag})$, the agent first applies the acceleration to update its velocity $s.v_{Ag,t+1} = s.v_{Ag,t} + \text{acc}_{\text{Ag}}$. Once the velocity is applied, the steer maneuver (if at the proper velocity) is taken and the agent occupies a set of grid-points, specified in Fig. 7.3, while taking its maneuver.

Agent grid point occupancy is defined as follows. Note that agents occupy a single grid point at a given time, but when taking an action, they may occupy a set of grid points. More formally:

Definition 7.4.1 (Grid Point Occupancy)**.** The notion of grid point occupancy is captured by the definitions of the following maps for each $\text{Ag} \in \mathfrak{A}$. To define the grid point an agent is occupying at a given time, we use the map: $\mathcal{G}_{\text{Ag},t} : S_{\text{Ag}} \to 2^G$, mapping each agent to the single grid point the agent occupies. By a slight abuse of notation, we let $\mathcal{G}_{Ag,t} : S_{\text{Ag}} \times Act_{\text{Ag}} \to 2^G$ be a function that maps each $s \in S_{\text{Ag}}$ and $a \in \rho_{\text{Ag}}(s)$ to denote the set of all nodes that are occupied by the agent Ag when it takes an allowable action a from state s at the time-step t.

The occupancy grids associated with each of the maneuvers allowed for the agents, and the velocity that the agent has to be at to take the maneuver are shown in Fig. 7.3.

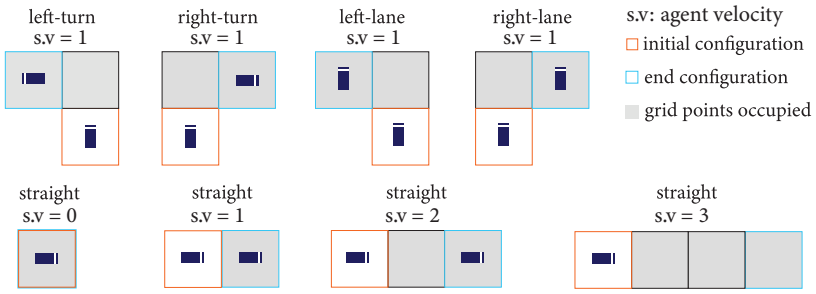


Figure 7.3: Shows different grid point occupancy associated with different discrete agent maneuvers.

Note, the safety and liveness guarantees will hold for any choice of agent dynamic parameters (i.e. $a_{min}, a_{max}, v_{min}, v_{max}$), but only under the condition that all agents have the same set of dynamic parameters. The maneuvers must be the ones specified above.

With slight abuse of notation, we let $La(\text{Ag})$ refer to the lane ID associated with the grid point $(s.x_{\text{Ag}}, s.y_{\text{Ag}})$, and $Bu(\text{Ag})$ mean the bundle ID associated with the lane $La(\text{Ag})$.

Motion-Planning Algorithm

We assume that any graph planning algorithm can be used to specify an agent's motion plan. The motion plan must be divided into a set of critical points along the graph that the agent must reach in order to get to its destination, but should not specify the exact route agents must take to get to these critical points. It should be noted that the liveness guarantees rely on the assumption that rerouting of the agent's motion plan is not supported.

Agent Backup Plan Action

A *backup plan* is a reserved set of actions an agent is *entitled* to execute at any time while being immune to being at fault for a collision if one occurs. In other words, an agent will always be able to safely take its backup plan action. We show that if each agent can maintain the ability to safely execute its own backup plan (i.e. keep a far enough distance behind a lead agent), the safety of the collective system safety is guaranteed.

The default backup plan adopted here is that of applying maximal deceleration until a complete stop is achieved, which is defined as:

Definition 7.4.2 (Backup Plan Action)**.** The backup plan action a_{bp} is a control action where $a = \max(a_{\min}, -s.v_{Ag})$, and $\gamma_{\text{Ag}} = \texttt{straight}$. Note, the max is because applying the deceleration ($a_{\min}$) should not push the car velocity below 0. Note $a_{\min}$ is less than 0.

Note, it may take multiple time-steps for an agent to come to a complete stop because of the inertial dynamics of the agent.

Limits on Agent Perception

In real-life, agents make decisions based on local information. We model this locality by defining a region of grid points around which agents have access to the full (state and intention) information of the other agents.

Road Segments

For road segments, the region around which agents make decisions cannot be arbitrarily defined. In fact, an agent's bubble must depend on its state, and the agent attributes and dynamics of all agents in the game. In particular, the bubble can be defined as follows:

Definition 7.4.3 (Bubble)**.** Let Ag with state $s_0 \in \mathcal{S}_{\text{Ag}}$. Let agent Ag$'$ be another agent. Then the bubble of Ag with respect to agents of the same type as Ag$'$ is given by $\mathcal{B}_{\text{Ag/Ag}'}(s_0)$. The bubble is the minimal region of space (set of grid points) agents need to have full information over to guarantee they can make a decision that will preserve safety under the defined protocol. Since all Ag considered in this chapter have the same attributes, for ease of notation, we refer to the bubble of Ag as $\mathcal{B}_{\text{Ag}}$.

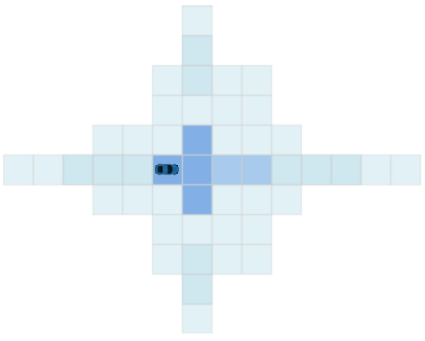


Figure 7.4: Bubble if all Ag $\in \mathfrak{A}$ have the Agent Dynamics specified in Section 7.4.

For our protocol, the bubble contains any grid points in which another agent Ag' occupying those grid points can interfere with at least one of Ag's next possible actions and the backup plan it would use if it were to take any one of those next actions. With a slight abuse of notation, we say $\text{Ag}' \in \mathcal{B}_{\text{Ag}}(s)$ if $(s.x_{\text{Ag}'}, s.y_{\text{Ag}'})$ is on a grid point in the set $\mathcal{B}_{\text{Ag}}(s)$.

Intersections

The locality of information agents are restricted to is relaxed at intersections because agents can presumably see across the intersection when making decisions about crossing the intersection. More precisely, any Ag must be able to know about any $\text{Ag}' \in \mathfrak{A}$ that is in the lanes of oncoming traffic (when performing an unprotected left-turn). The computation of the exact region of perception necessary depends on the agent dynamics. Locality for the local-precedence assignment algorithm is also extended to this larger region at intersections as well.

Precedence Rules

The definition of the quasi-simultaneous game requires agents to locally assign precedence, i.e. have a set of rules to define how to establish which agents have higher, lower, equal, or incomparable precedence to it. Our precedence assignment algorithm is motivated by capturing how precedence among agents is generally established in real-life scenarios on a road network. In particular, since agents are designed to move in the forward direction, we aim to capture the natural inclination of agents to react to the actions of agents visibly ahead of it.

Before presenting the precedence assignment rules, we must introduce a few definitions. Let us define: $\text{proj}^{B}_{\text{long}} : \mathfrak{A} \to \mathbb{Z} \cup \{\varnothing\}$ as $\text{proj}^{B}_{\text{long}}(\text{Ag})$ = the projection of the Ag's state onto the bundle B if Ag is in B and otherwise $\text{proj}^{B}_{\text{long}}(\text{Ag}) = \varnothing$. In other words, $\text{proj}^{B}_{\text{long}}(Ag)$ is the mapping from an agent to its scalar projection onto the longitudinal axis of the bundle B the agent Ag is in. If $\text{proj}^{B}_{\text{long}}(\text{Ag}') < \text{proj}^{B}_{\text{long}}(\text{Ag})$, then the agent Ag' is behind Ag in B.

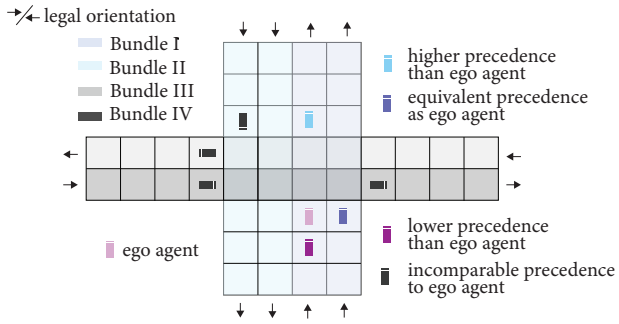


Figure 7.5: Rules for precedence assignment.

For every agent Ag′, the agent Ag defines the precedence relation between Ag and Ag′ using the following set of precedence rules:

Local Precedence Assignment Rules

1. If $\text{proj}^B_{\text{long}}(\text{Ag}') < \text{proj}^B_{\text{long}}(\text{Ag})$ and $Bu(\text{Ag}') = Bu(\text{Ag})$, then $\text{Ag}' \prec \text{Ag}$, i.e. if agents are in the same bundle and Ag is longitudinally ahead of Ag′, Ag has higher precedence than Ag′.

2. If $\text{proj}^B_{\text{long}}(\text{Ag}') = \text{proj}^B_{\text{long}}(\text{Ag})$ and $Bu(\text{Ag}') = Bu(\text{Ag})$, then $\text{Ag} \sim Ag'$, and we say that Ag and Ag′ are equivalent in precedence.

3. If Ag′ and Ag are not in the same bundle, then the two agents are incomparable.

Each agent $\text{Ag} \in \mathfrak{A}$ only assigns precedence according to the above rules locally to agents within its perception region (e.g. bubble on road segments and a slightly larger region at intersections, defined in Section 7.4) when making a decision of which action to take. Thus, we must show if all agents locally assign precedence according to these rules, a globally-consistent turn precedence among all agents is established. In particular, we need to prove the following lemma.

Lemma 5. *If all agents assign precedence according to the local precedence assignment rules to agents in their respective bubbles, then precedence relations will induce a polyforest on $\mathfrak{A}/\sim$ (the quotient set of S by $\sim$).*

Proof. Suppose there is a cycle C in $\mathfrak{A}/\sim$. For each of the equivalent classes in C (C must have at least 2 to be a cycle), choose a representative from $\mathfrak{A}$ to form a set R_C. Let $\text{Ag} \in R_C$ be one of these representatives. Applying the second local

precedence assignment rule inductively, we can see that all agents in R_C must be from Ag's bundle. By the first local precedence assignment rule, any C edge must be from an agent with lower projected value to one with a higher projected value in this bundle. Since these values are totally ordered (being integers), they must be the same. This implies that C only has one equivalence class, a contradiction. □

The acyclicity of the polyforest structure implies the consistency of local agent precedence assignments. Note, the local precedence assignment algorithm establishes the order in which agents are taking turns. Even when this order is established, it is ambiguous what agents should do either when 1) agents of equal precedence have conflicting intentions, since they select their actions at the same time or 2) an agent's intended action is a lane-change action and requires agents of lower or equivalent precedence to change their behavior so the lane-change action is safe. The additional set of rules introduced to resolve this ambiguity is what we refer to as conflict cluster resolution, defined in Section 7.4.

Assume-Guarantee Profile

An assume-guarantee profile, introduced in Chapter 6, is a mechanism for ordering agent specifications into a specific hierarchy so the process for choosing an agent's preferred action is transparent and safe. The concept is related to the concepts of minimum-violation planning (Tumova et al., 2013; Censi, Slutsky, et al., 2019).

In this work, each agent uses an assume-guarantee profile to *propose* an intended action. From Chapter 6, the assume-guarantee profile requires defining a set of agent specifications. For completeness, we define how specifications are evaluated in the game. Let $r \in R$ denote a specification for an agent and $\text{Ag} \in \mathfrak{A}$. For the specification r, an oracle evaluates whether an agent taking an action in the current joint state game configuration will satisfy the specification. More formally, the oracle is defined as follows $O_{\text{Ag},t} : R \times S_{\text{Ag}} \times Act_{\text{Ag}} \times \mathcal{U} \to \mathbb{B}$ where $\mathbb{B} = \{\mathtt{T}, \mathtt{F}\}$ and the subscript t denotes the time-step the oracle is evaluated. These evaluations can easily be refactored to accommodate specifications that are more continuous in nature.

In this work, each agent has a total of ten different specifications, three of whose oracles are defined as follows:

1. $O_{Ag,t,\textbf{dynamic safety}}(s, a, u)$ returns $\mathtt{T}$ when the action a from state s will not cause Ag to either collide with another agent or end up in a state where the

agent's safety backup plan a_{bp} is no longer safe with respect to other agents (assuming other agents are not simultaneously taking an action).

2. $O_{Ag,t,\textbf{unprotected left-turn safety}}(s, a, u)$ returns `T` when the action a from the state s will result in the complete execution of a safe, unprotected left-turn (invariant to agent precedence). Note, an unprotected left turn spans over multiple time-steps. The oracle will return `T` if Ag has been waiting to take a left-turn (while traffic light is green), traffic light turns red, and no agents in oncoming lanes.

3. $O_{Ag,t,\textbf{reachability preservation progress}}(s, a, u)$ returns `T` if the action a from the state s will allow Ag$'s$ planned path to remain reachable.

4. $O_{\textbf{static safety}}(s, a, u)$ returns `T` when the action a from state s will not cause the agent to collide with a static obstacle or end up in a state where the agent's safety backup plan a_{bp} with respect to the static obstacle is no longer safe.

5. $O_{\textbf{traffic light}}(s, a, u)$ returns `T` if the action a from the state s satisfies the traffic light laws (not crossing into intersection when red. It also requires that Ag be able to take a_{bp} from $s' = \tau_{\text{Ag}}(s, a)$ and not violate the traffic-light law.

6. $O_{\textbf{legal orientation}}(s, a, u)$ returns `T` if the action a from the state s follows the legal road orientation.

7. $O_{\textbf{traffic intersection clearance}}(s, a, u)$ returns `T` if the action causes the agent to enter the intersection and leave it when the traffic light turns red and if the action causes the agent to end up in a state where if it performs its backup plan action, it will still be able to leave the intersection.

8. $O_{\textbf{traffic intersection lane change}}(s, a, u)$ returns `T` if the action is such that γ_{Ag} = {`left-lane change`, `right-lane change`} and the agent either begins in an intersection or ends up in the intersection after taking the action.

9. $O_{\textbf{maintains progress}}(s, a, u)$ returns `T` if the action a from the state s stays the same distance to its goal.

10. $O_{\textbf{forward progress}}(s, a, u)$ returns `T` if the action a from the state s will improve the agent's progress towards the goal.

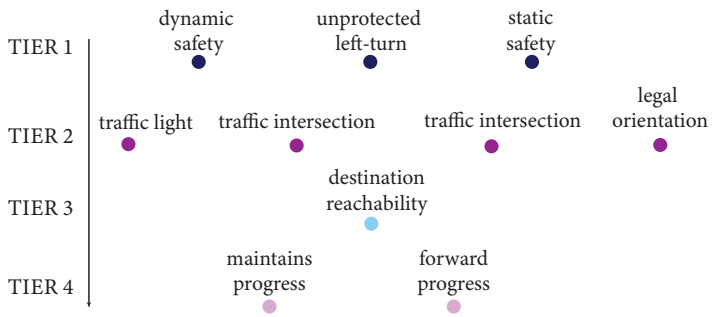


Figure 7.6: Assume-guarantee profile that shows ordering of specifications, where specifications on the same tier are incomparable to one another.

The ordering of the specifications that each agent must follow is shown in Fig. 7.6. In Chapter 6, the consistent-evaluating function W evaluates each action based on the number of specifications it satisfies in each tier, where actions satisfying more specifications in the higher tiers are valued more highly. The action with the highest value is then selected as the action the agent takes. For this work, the agent profile is used to define the agent's intended action a_i and the agent's best straight action a_{st} which is defined in Section 7.4. Note, the dynamic safety oracle is not included in the selection of the intended action a_i–otherwise an agent might never propose a lane-change action (since it would require other agents to yield in order for the lane-change action to be safe).

Conflict-Cluster Resolution

At every time-step t, each agent will know when to take its turn based on its local precedence assignment algorithm. Before taking its turn, the agent will have selected an intended action a_i using the Agent profile. When it is the agent's turn to select an action, it must choose whether or not to take its intended action a_i. When the intended actions of multiple agents conflict, the conflict-cluster resolution is a token-based querying method used to help agents determine which agent should get to take its action.

Under the assumption that agents have access to the intentions of other agents within a local region as defined in Section 7.4, agents can use the following criteria to define when it conflicts with another agent.

Definition 7.4.4 (Agent-Action Conflict)**.** Let us consider a $\mathcal{Y}$-state in which an agent Ag currently at state $s \in S_{\text{Ag}}$ and wants to take action $a \in \rho_{Ag}$ and an agent Ag$'$ at state $s' \in S_{\text{Ag}'}$ that wants to take action $a' \in \rho_{Ag'}$. We say that an agent-action

conflict between Ag and Ag′ for a and a' occurs and write $(\text{Ag}, s, a) \dagger (\text{Ag}', s', a')$ if either of the following conditions holds:

- $\mathcal{G}_{\text{Ag},t}(s, a) \cap \mathcal{G}_{\text{Ag'},t}(s', a') \neq \emptyset$,
- Let $s_{t+1} = \tau_{\text{Ag}}(s, a)$ and $s'_{t+1} = \tau_{\text{Ag}'}(s', a')$:

  If $La(s_{t+1}) = La(s'_{t+1})$ and $d(s_{t+1}, s'_{t+1}) \leq gap_{\text{req}}$,

where $d(s_{t+1}, s'_{t+1})$ defines the l_2 distance between two states in the same lane and gap_{req} is the minimum distance between two agents in the same lane so that if the agent in front applies their backup plan, the agent behind will be able to apply their own backup plan without colliding with the former. gap_{req} can easily be computed depending on the agent dynamics.

In the case that an agent's action does conflict with another agent, the agent must send a conflict request that ultimately serves as a bid the agent is making to take its intended action. It cannot, however, send requests to any agent (e.g. agents in front of it). The following criteria are used to determine the properties that must hold in order for an agent Ag to send a conflict request to agent Ag′:

Criteria that Must Hold for Agent Ag to Send Conflict Request to Agent Ag'

- Ag's intended action a_i is a lane-change action.
- $\text{Ag}' \in \mathcal{B}_{Ag}(s)$, i.e. Ag′ is in agent Ag's bubble.
- $\text{Ag}' \precsim Ag$, i.e. Ag has equal or higher precedence than Ag′.
- $s.\theta_{Ag} = s.\theta_{Ag'}$, i.e. the agents have the same heading.
- $(Ag, a_i) \dagger (Ag', a'_i)$: agents' intended actions are in conflict with one another.
- $\mathcal{F}_{\text{Ag}}(u, a_i) = \texttt{F}$, where $\mathcal{F}_{\text{Ag}}(u, a_i)$ is the max-yielding-not-enough flag and is defined below.

Definition 7.4.5 (maximum-yielding-not-enough flag)**.** The maximum-yielding-not-enough flag $\mathcal{F}_{\text{Ag}} : \mathcal{U} \times Act_{\text{Ag}} \to \mathbb{B}$ that is defined as follows:

$$\mathcal{F}_{\text{Ag}}(u, a_i) = \begin{cases} \texttt{T} & \text{if } \exists \text{Ag}' \in \mathcal{B}_{Ag}(s) s.t. ((\text{Ag}, a_i) \dagger (\text{Ag}', a_{\text{bp}})) \\ \texttt{F} & \text{otherwise.} \end{cases}$$

When the flag is set, it indicates a configuration in which even if Ag′ maximally yielded to Ag, if Ag did a lane-change, it would violate the safety of Ag′'s backup plan action.

We note that if $\mathcal{F}_{\text{Ag}}(u, a_i)$ is set, Ag cannot send a conflict request by the last condition. Even though Ag does not send a request, it must use the information that the flag has been set in the agent's Action Selection Strategy.

After a complete exchange of conflict requests, each agent will be a part of a cluster of agents that define the set of agents it is ultimately bidding for its priority (to take its intended action) over. These clusters of agents are defined as follows:

Definition 7.4.6 (Conflict Cluster)**.** A conflict cluster for an agent Ag is defined as $C_{Ag} = \{Ag' \in \mathfrak{A} \mid Ag \ \texttt{send} \ \text{Ag}' \text{ or } Ag' \ \texttt{send} \ Ag\}$, where Ag `send` Ag' implies Ag has sent a conflict request to Ag′. An agents' conflict cluster defines the set of agents in its bubble that an agent is in conflict with.

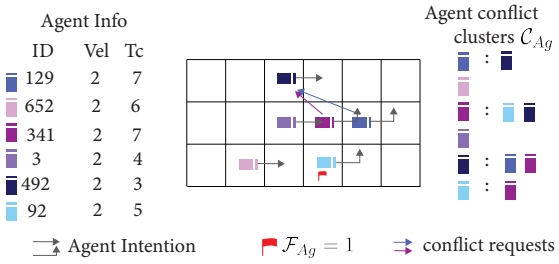


Figure 7.7: Conflict clusters

Once the conflict requests have been sent and an agent can thereby identify the other agents in its conflict cluster, it needs to establish whether or not the conflict resolution has resolved in its favor, as shown in Fig. 7.7.

Token Resolution

The conflict resolution strategy must be fair, meaning each agent always eventually wins a conflict resolution. The resolution is based on the agents' token counts `Tc`, which is updated by agents to represent how many times an agent has been unable to take a forward progress action.

Given the assumption that all agents can query the token counts of all other agents, let us define the conflict resolution strategy. For each $\text{Ag} \in \mathfrak{A}$, let $\mathcal{W}_{\text{Ag}} \in \mathbb{B}$ be an

indicator variable for whether or not the agent has won in its conflict cluster. Let $\texttt{Tc}_{Ag}$ represent the token count of the agent when it has sent its request. Let $\texttt{Id}_{Ag}$ represent a unique ID number of an agent. The conflict cluster resolution indicator variable $\mathcal{W}_{\text{Ag}}$ is determined as follows:

$$\mathcal{W}_{\text{Ag}} \coloneqq \forall \text{Ag}' \in \mathcal{B}_{\text{Ag}}(s) :: (\texttt{Tc}_{\text{Ag}'} < \texttt{Tc}_{\text{Ag}}) \vee ((\texttt{Tc}_{\text{Ag}'} = \texttt{Tc}_{\text{Ag}}) \wedge \texttt{Id}_{\text{Ag}'} < \texttt{Id}_{\text{Ag}}).$$

The agent with the highest token count is defined as the winner of the agents' conflict cluster and any ties are broken via an agent ID comparison.

The following lemma, which comes from the definition of the conflict-cluster resolution scheme, is helpful for proving safety of the agent protocol.

Lemma 6. *At most one agent will win in each agent's conflict cluster.*

Proof (Sketch). This follows from the definition of conflict clusters (i.e. all agents will only send and receive requests from agents within their bubble), and the winner-takes-all conflict-cluster resolution. □

The next section defines how each agent uses information from the conflict cluster resolution to ultimately select an action.

Action Selection Strategy

The purpose of the agent Action Selection Strategy is to define whether or not an agent is allowed to take its intended action a_i and if it is not, which alternative action it should take. The action-selection strategy is defined to coordinate agents so that lane-change maneuvers can be performed safely.

In the case where an agent is not allowed to take a_i, the agent is restricted to take either: the best straight action a_{st}, which is defined in 7.4.7, or its backup plan action a_{bp}. The action-selection process that determines which of the three actions an agent Ag will choose is determined by the following five conditions:

1. a_i, the agent's and other agents' (in its bubble) intended actions, which have been selected via the agent profile and the consistent evaluating function defined in Section 7.4.

2. Ag's role in conflict request cluster being:

    - A conflict request sender ($\exists \text{Ag}' \in \mathcal{B}_{\text{Ag}}(s)$: Ag `send` Ag′).

- A conflict request receiver ($\exists \text{Ag}' \in \mathcal{B}_{\text{Ag}}(s) : \text{Ag}'$ `send` Ag).
- Both a sender and a receiver of conflict requests.
- Neither a conflict request sender or receiver.

3. The agent's conflict cluster resolution $\mathcal{W}_{\text{Ag}}$.
4. Evaluation of $O_{\text{Ag},t,\text{dynamic safety}}(s, a_i, u)$.
5. $\mathcal{F}_{\text{Ag}}(u, a_i)$ for Ag is raised, where $\mathcal{F}_{\text{Ag}}(u, a_i)$ is the maximal-yielding-not-enough flag defined in Section 7.4.

The Action Selection Strategy decision tree, shown in Fig. 7.8, defines how agents should select which action to take based on the five different conditions.

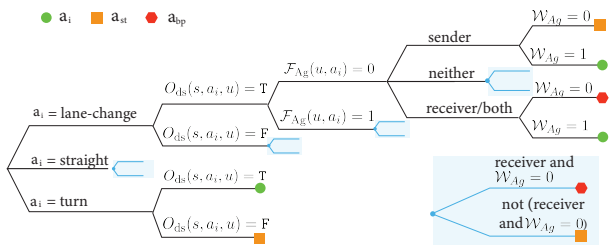


Figure 7.8: Agent action selection strategy.

The best straight action a_{st}, one of the three allowable actions the agent can take according to the action-selection strategy, is defined as follows:

Definition 7.4.7 (Best Straight Action)**.** Let us consider Ag and its associated action set $\rho_{\text{Ag}}(s)$. Let us define $\rho_{Ag}(s) \mid_{\text{st}} := \{a \in \rho_{\text{Ag}}(s) \mid \gamma_{\text{Ag}} = \texttt{straight}\}$, i.e. the set of all allowable straight actions of Ag. The best straight action $a_{st} = \arg\max_{a \in \rho_{\text{Ag}}(s)\mid_{\text{st}}} W_{\text{Ag}}(a)$, where W is the consistent evaluating function defined with respect to the agent profile in Fig. 7.6, with the dynamic safety oracle ($O_{\text{Ag},t,\text{dynamic safety}}(s, a, u)$) included.

When an agent is both a receiver of a conflict request and a loser in its conflict cluster, it must yield to the agent that it received the conflict request from. The Action Selection Strategy requires the agent to take its backup plan action a_{bp} in these scenarios, where the backup plan action a_{bp} is the control action defined in Definition 7.4.2.

Token Count Update

The token count updates according to the agent's chosen action, in particular, if Ag selects action a:

$$\mathtt{Tc}_{\mathrm{Ag}} = \begin{cases} \mathtt{Tc}_{\mathrm{Ag}} + 1 & \text{if } O_{\text{forward progress}}(s, a, u) = \mathtt{F} \\ 0 & \text{otherwise.} \end{cases}$$

.

7.5 Safety Guarantees

Safety is guaranteed when agents do not collide with one another. An agent causes collision when it takes an action that satisfies the following conditions.

Definition 7.5.1 (Collision)**.** An agent Ag that takes an action $a \in Act_{\mathrm{Ag}}$ will cause collision if either of the following conditions hold:

1. $\mathcal{G}_{\mathrm{Ag},t}(s, a) \cap (\cup_{\mathrm{Ag}'} \mathcal{G}_{\mathrm{Ag}'}(s', a')) \neq \emptyset$.
2. $\mathcal{G}_{\mathrm{Ag},t}(s, a) \cap O_{\mathrm{st}}) \neq \emptyset$, where $O_{\mathrm{st}} = \{g \in G \mid g.d = 0\}$, i.e. the set of all undrivable grid points.

In other words, when Ag's action a causes it to overlap in the occupancy grid with another agent's occupancy grid or a static obstacle. Note when the agent Ag′ is not simultaneously taking an action with Ag, the occupancy set $\mathcal{G}_{\mathrm{Ag}'}(s', \cdot)$ is the singleton set for the agent Ag′, representing the single grid point Ag' is occupying.

A strategy where agents simply take actions that avoid collision in the current time-step is insufficient for guaranteeing safety because of the inertial properties of the agent dynamics. The Agent Protocol is thus also defined to avoid violating the safety of its own and any other agent's backup plan action a_{bp} defined in Section 7.4. An agent's backup plan action a_{bp} is evaluated to be safe when the following conditions hold:

Definition 7.5.2. [Safety of a Backup Plan Action] Let us define the safety of an agent's backup plan action $S_{\mathrm{Ag},bp} : \mathcal{U} = \mathbb{B}$, where $\mathbb{B} = \{\mathtt{T}, \mathtt{F}\}$ is an indicator variable that determines whether an agent's backup plan action is safe or not. It is defined as follows:

$$S_{Ag,bp}(u) = \wedge_{o \in O} o(s, a_{bp}, u),$$

where the set O is the set of all oracles in the top three tiers of the agent profile defined in Section 7.4.

An agent Ag takes an action $a \in Act_{\text{Ag}}$ that violates the safety backup plan action of another agent Ag$'$ when the following conditions hold:

Definition 7.5.3 (Safety Backup Plan Violation Action)**.** Let us consider an agent Ag that is taking an action $a \in Act_{\text{Ag}}$, and another agent Ag$'$. The action $(\text{Ag}, a) \perp \text{Ag}'$, i.e. agent Ag violates the safety backup plan of an agent Ag$'$ when by taking an action a where u' is the state of the game after Ag has taken its action, then $S_{\text{Ag}',bp}(u') = \texttt{F}$.

Note, when Ag $\neq$ Ag$'$, then Ag can only violate the backup plan action of the agent Ag$'$ with its action a if the following conditions hold:

1. $\mathcal{G}_{\text{Ag}}(s, a) \cap \mathcal{G}_{\text{Ag}'}(s', a') \neq \emptyset$,
2. Let $s_{t+1} = \tau_{Ag}(s, a)$ and $s'_{t+1} = \tau_{Ag'}(s', a')$: If $\mathcal{G}_{\text{Ag}}(s_{t+1}), \mathcal{G}_{\text{Ag}'}(s_{t+1})$ are in the same lane and if $d(s_{t+1}, s'_{t+1}) < gap_{\text{req}}$,

where $d(s_{t+1}, s'_{t+1})$ and gap_{req} are the same as defined in Section 7.4.4.

The safety proof is based on the premise that all agents only take actions that do not collide with other agents and maintain the invariance of the safety of their own **and** other agents' safety backup plan actions. The safety theorem statement and the proof sketch are as follows.

We can treat the quasi-simultaneous game as a program, where each of the agents are separate concurrent processes. A safety property for a program has the form $P \Rightarrow \Box Q$, where P and Q are immediate assertions. This means if the program starts with P true, then Q is always true throughout its execution (Owicki and Lamport, 1982).

Theorem 5 (Safety Guarantee)**.** *If all agents Ag $\in \mathfrak{A}$ in the quasi-simultaneous game select actions in accordance to the Agent Protocol specified in Section 7.4, then we can show the safety property $P \Rightarrow \Box Q$, where the assertion P is an assertion that the state of the game is such that $\forall Ag, S_{Ag,bp}(s, u) = \texttt{T}$, i.e. each agent has a backup plan action that is safe, as defined in Section 7.5.2. We denote P_t as the assertion over the state of the game at the beginning of the time-step t, before agents take their respective actions. Q is the assertion that the agents never occupy the same grid point in the same time-step (e.g. collision never occurs when agents take their respective actions during that time-step). We denote Q_t as the assertion for the agent states/actions taken at time-step t.*

Proof. To prove an assertion of this form, we need to find an invariant assertion I for which i) $P \Rightarrow I$, ii) $I \Rightarrow \Box I$, and iii) $I \Rightarrow Q$ hold. We define I to be the assertion that holds on the actions that agents select to take at a time-step. We denote I_t to be the assertion on the actions agents take at time t such that $\forall Ag$, Ag takes $a \in Act_{\text{Ag}}$ where 1) it does not collide with other agents, and 2) $\forall$Ag, $S_{\text{Ag},bp}(u') = \texttt{T}$ where $s' = \tau_{\text{Ag}}(s, a)$, and u' is the corresponding global state of the game after each Ag has taken its respective action a.

It suffices to assume:

1. Each Ag $\in \mathfrak{A}$ has access to the traffic light states.
2. There is no communication error in the conflict requests, token count queries, and the agent intention signals.
3. All intersections in the road network R are governed by traffic lights.
4. The traffic lights are designed to coordinate traffic such that if agents respect the traffic light rules, they will not collide.
5. Agents follow the agent dynamics defined in Section 7.4.
6. For $t = 0$, $\forall$Ag $\in \mathfrak{A}$ in the quasi-simultaneous game is initialized to:
   - Be located on a distinct grid point on the road network.
   - Have a safe backup plan action a_{bp} such that $S_{\text{Ag},bp}(s, u) = \texttt{T}$.

We can prove $P \Rightarrow \Box Q$ by showing the following:

1. $P_t \Rightarrow I_t$. This is equivalent to showing that if all agents are in a state where P is satisfied at time t, then all agents will take actions at time t where the I holds. This can be proven using arguments based on the Agent Protocol showing each agent will always take actions that 1) do not collide with other agents and 2) will not violate the safety of its own or other agents' backup plan action.
2. $I \Rightarrow \Box I$. If agents take actions at time t such that the assertion I_t holds, then by the definition of the assertion I, agents will end up in a state where at time t+1, assertion P holds, meaning $I_t \Rightarrow P_{t+1}$. Since $P_{t+1} \Rightarrow I_{t+1}$ from 1, we get $I \Rightarrow \Box I$.

3. $I \Rightarrow Q$. This is equivalent to showing that if all agents take actions according to the assertions in I, then collisions will not occur. This follows in the immediate time-step from Condition 1, and the fact that all Ag have a safe backup plan action a_{bp} to choose from when Condition 2 holds, and will always be able to (and will) take an action from which it can avoid collision in future time steps.

□

Proof of safety alone is not sufficient reason to argue for the effectiveness of the protocol, as all agents could simply stop for all time and safety would be guaranteed. A liveness guarantee, i.e. proof that all agents will eventually make it to their final destination, is critical. In the following section, we present liveness guarantees.

7.6 Liveness Guarantees

Note, we introduce the definition of liveness, from (Owicki and Lamport, 1982), as follows:

Definition 7.6.1 (Liveness)**.** A liveness property asserts that program execution eventually reaches some desirable state.

Here, the eventual desirable state for each agent is to reach is its respective final destination. Proving fairness, as described in (Owicki and Lamport, 1982), is proving that each action will always terminate, and is fundamental for proving liveness. Additionally for liveness, the absence of 1) deadlocks and 2) collisions also need to be proved. Deadlock occurs when agents indefinitely wait for resources held by other agents (Reveliotis and Roszkowska, 2010). Since the Manhattan grid road network has loops, agents can enter a configuration in which each agent in the loop is indefinitely waiting for a resource held by another agent. When the density of agents in the road network is high enough, deadlocks along these loops will occur. We can therefore guarantee liveness only when certain assumptions hold on the density of the road network.

Definition 7.6.2 (Sparse Traffic Conditions)**.** Let M denote the number of grid points in the smallest loop (defined by legal orientation) of the road network, not including grid points $g \in \mathcal{S}_{\text{intersections}}$. The sparsity condition must be such that $N < M - 1$, where N is the number of agents in the road network. Note, these sparsity conditions

are conservative because it is a bound defined by the worst possible assignment of agents and their destinations (i.e. where all agents enter the smallest loop).

Now, we introduce the liveness guarantees under these sparse traffic conditions. The proof of liveness is based on the fact that 1) agent profile includes progress specifications and 2) conflict precedence is resolved by giving priority to the agent that has waited the longest time (a quantity that is reflected by token counts).

Theorem 6 (Liveness Under Sparse Traffic Conditions)**.** *Under the Sparse Traffic Assumption given by Definition 7.6.2 and given all agents* $Ag \in \mathfrak{A}$ *in the quasi-simultaneous game select actions in accordance to the Agent Protocol specified in Section 7.4, liveness is guaranteed, i.e. all* $Ag \in \mathfrak{A}$ *will always eventually reach their respective goals.*

Proof (Sketch). It suffices to assume:

1. $\forall \mathrm{Ag} \in \mathfrak{A}$, $\forall \mathrm{Ag}' \in \mathbb{B}_{Ag}$ in road segments, and $\forall Ag'$ within a local region around the agent as defined in Section 7.4 at intersections, Ag has access to other agents' state and intended action.
2. Each $\mathrm{Ag} \in \mathfrak{A}$ has access to the traffic light states.
3. There is no communication error in the conflict requests, token count queries, and the agent intention signals.
4. For $t = 0$, $\forall \mathrm{Ag} \in \mathfrak{A}$ in the quasi-simultaneous game is initialized to:
   - Be located on a distinct grid point on the road network.
   - Have a safe backup plan action a_{bp} such that $S_{\mathrm{Ag},bp}(u) = \mathtt{T}$.
5. The traffic lights are red for a window of time Δt_{tl} such that $t_{\min} < \Delta t_{\mathrm{tl}} < \infty$, and $t_{\min}$ is defined so that agents are slowed down sufficiently long such that an agent waiting to make a lane-change to a critical tile is such that its max-yielding-flag is not always set to $\mathtt{T}$.
6. The static obstacles are not on any grid point g where $g.d = 1$.
7. Each Ag treats its respective goal $\mathrm{Ag}.\mathtt{g}$ as a static obstacle.
8. Bundles in the road network $\mathfrak{R}$ have no more than 2 lanes.

9. All intersections in the road network $\mathfrak{R}$ are governed by traffic lights.

and prove:

1. The invariance of a no-deadlock state follows from the sparsity assumption and the invariance of safety (no collision) follows from the safety proof.

2. Inductive arguments related to control flow are used to show that all Ag will always eventually take $a \in Act_{\text{Ag}}$ where $O_{\text{forward progress}}(s, a, u) = \texttt{T}$.

    a) Let us consider a road segment $r \in RS$ that contains grid point(s) $g \in \mathcal{S}_{\text{sinks}}$, i.e. the road segment contains grid points with sink nodes. Inductive arguments based on the agents' longitudinal distance to destination grid points are used to show that every $\text{Ag} \in r$ will be able to always eventually take $a \in Act_{\text{Ag}}$ for which $O_{\text{forward progress}}(s, a, u) = \texttt{T}$.

    b) Let us consider a road segment $rs \in RS$. Let us assume $\forall rs \in RS, \exists (rs, rs') \in G_{\text{dep}}$, i.e. the clearance of rs depends on the clearance of all rs'. Inductive arguments based on agents' longitudinal distance to the front of the intersection are used to show that any Ag on rs will always eventually take $a \in Act_{\text{Ag}}$ where $O_{\text{forward progress}}(s, a, u) = \texttt{T}$.

    c) For any $\mathfrak{R}$ where the dependency graph G_{dep} (as defined in Section 7.3.2) is a directed-acyclic-graph (DAG), inductive arguments based on the linear ordering of road segments $rs \in G_{\text{dep}}$ are used to prove all $\text{Ag} \in \mathfrak{A}$ will always eventually take $a \in Act_{\text{Ag}}$ for which $O_{\text{forward progress}}(s, a, u) = \texttt{T}$.

    d) When the graph G_{dep} is cyclic, the Sparsity Assumption 7.6.2 breaks the cyclic dependency and allows for the similar induction arguments in 2c to apply.

3. By the above inductive arguments and the definition of the forward progress oracle $O_{\text{forward progress}}(s, a, u)$, all Ag will always eventually take actions that allow them to make progress towards their respective destinations.

□

Features of the Agent Protocol, like fairness from the conflict-cluster resolution and eventual satisfaction of all oracles in the agent profile are used for the arguments in the proof.

7.7 Simulation

In order to streamline discrete-time multi-agent simulations, we have built a traffic game simulation platform called Road Scenario Emulator (RoSE). This emulator offers an easy-to-use, simple, and modular interface. We use RoSE to generate different game scenarios and simulate how agents will all behave if they each follow the agent strategy protocol introduced in this chapter. We simulate the game with randomized initialization of spawning agents at the source nodes for three different road network environments: 1) the straight road segment, 2) small city blocks grid, and 3) large city blocks grid. A snapshot of a small city blocks grid simulation is shown in Fig. 7.9. The agent attributes are as follows: $v_{\min} = 0$, $v_{\max} = 3$, $a_{\min} = -1$, and $a_{\max} = 1$. For each road network environment, we simulate the game 100 times for $t = 250$ time-steps.

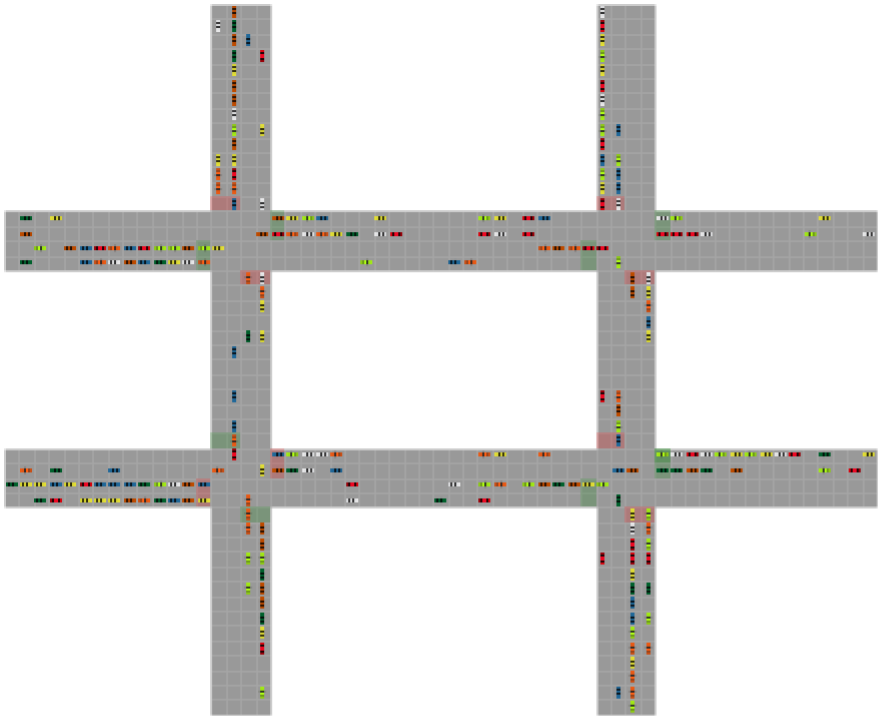


Figure 7.9: Simulation.

For all simulation trials collision does not occur. Although liveness is only guaranteed in sparse traffic conditions, we simulate for a number of agents $N > M - 1$

specified in the sparsity condition and deadlock does not occur. In particular, for the straight road segment, on average 77%, 36%, and 43% made it to their respective destinations on the respective maps by the end of the 250 time-steps.

7.8 Conclusion

In this chapter, we have proposed a novel paradigm for designing safety-critical decision-making modules for agents whose behavior is extremely complex and highly-coupled with other agents. The main distinction of our proposed architecture from the existing literature is the shift from thinking of each agent as separate, individual entities, to agents as a collective where all *all* agents adopt a *common* local, decentralized protocol (where additional customization can be built in later). The protocol defines the agent attributes, the region it must reason over (i.e. the bubble), how the agent chooses its intended action, and how it ultimately selects which action to take. With this protocol, we are able to formally guarantee specifications safety and liveness (under sparse traffic conditions) for all agents. We validate the safety and liveness guarantees in a randomized simulation environment.

The current work still lacks 1) liveness guarantees in all scenarios, 2) robustness to imperfect sensory information and 3) does not account for other agent types like pedestrians and cyclists. Future work may involve modifying the agent strategy architecture to prevent the occurrence of the loop deadlock introduced in Section 7.6 from occurring. In addition to providing stronger liveness guarantees, the architecture must be modified in a way to effectively accommodate impartial and imperfect information. We also hope to accommodate a diverse, heterogenous set of car agents and also other agent types like pedestrians and cyclists. Although the work needs to be extended to make it more applicable to real-life systems, we believe this work is a first step towards defining a comprehensive method for guaranteeing safety and liveness for all agents in an extremely dynamic and complex environment.

References

Baier, Christel and Joost-Pieter Katoen (2008). *Principles of Model Checking*. Cambridge, Massachussetts: MIT Press.

Censi, Andrea, Saverio Bolognani, Julian G. Zilly, Shima S. Mousavi, and Emilio Frazzoli (2019). "Today Me, Tomorrow Thee: Efficient Resource Allocation in Competitive Settings using Karma Games." In: *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*. Auckland, New Zealand: IEEE, pp. 686–693.

Censi, Andrea, Konstantin Slutsky, Tichakorn Wongpiromsarn, Dmitry Yershov, Scott Pendleton, James Fu, and Emilio Frazzoli (Feb. 2019). "Liability, Ethics, and Culture-Aware Behavior Specification using Rulebooks." In: *arXiv e-prints*, arXiv:1902.09355.

Chandy, K. Mani and Jayadev Misra (1984). "The drinking philosophers problem." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 6.4, pp. 632–646.

Finn, Chelsea, Sergey Levine, and Pieter Abbeel (2016). "Guided cost learning: Deep inverse optimal control via policy optimization." In: *International conference on machine learning*. New York, New York, USA: JMLR, pp. 49–58.

Fisac, Jaime F., Eli Bronstein, Elis Stefansson, Dorsa Sadigh, Shankar Sastry, and Anca D. Dragan (2019). "Hierarchical game-theoretic planning for autonomous vehicles." In: *2019 International Conference on Robotics and Automation (ICRA)*. Montreal, Canada: IEEE, pp. 9590–9596.

Gmytrasiewicz, Piotr J. and Prashant Doshi (2005). "A framework for sequential planning in multi-agent settings." In: *Journal of Artificial Intelligence Research* 24, pp. 49–79.

Owicki, Susan and Leslie Lamport (1982). "Proving liveness properties of concurrent programs." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3, pp. 455–495.

Papadimitriou, Christos H. and John N. Tsitsiklis (1987). "The complexity of Markov decision processes." In: *Mathematics of operations research* 12.3, pp. 441–450.

Reveliotis, Spyros A. and Elzbieta Roszkowska (2010). "On the Complexity of Maximally Permissive Deadlock Avoidance in Multi-Vehicle Traffic Systems." In: *IEEE Transactions on Automatic Control* 55.7, pp. 1646–1651.

Sadigh, Dorsa, Anca D. Dragan, Shankar Sastry, and Sanjit A. Seshia (2013). "Active Preference-Based Learning of Reward Functions." In: *Robotics: Science and Systems (RSS)*. Berlin, Germany.

Sadigh, Dorsa, Shankar Sastry, Sanjit A. Seshia, and Anca D. Dragan (2016). "Planning for autonomous cars that leverage effects on human actions." In: *Robotics: Science and Systems (RSS)*. Vol. 2. Ann Arbor, MI, USA.

Sahin, Yunus E. and Necmiye Ozay (2020). "From Drinking Philosophers to Wandering Robots." In: *arXiv preprint arXiv:2001.00440*.

Shalev-Shwartz, Shai, Shaked Shammah, and Amnon Shashua (Aug. 2017). "On a Formal Model of Safe and Scalable Self-driving Cars." In: *arXiv e-prints*, arXiv:1708.06374, arXiv:1708.06374. arXiv: 1708.06374 [cs.RO].

Shoham, Yoav and Moshe Tennenholtz (1995). "On social laws for artificial agent societies: off-line design." In: *Artificial intelligence* 73.1-2, pp. 231–252.

Tumova, Jana, Gavin C. Hall, Sertac Karaman, Emilio Frazzoli, and Daniela L. Rus (2013). "Least-violating control strategy synthesis with safety rules." In: *Proceedings of the 16th international conference on Hybrid systems: computation and control*. Philadelphia, Pennsylvania, USA: ACM, pp. 1–10.

van Der Hoek, Wiebe, Mark Roberts, and Michael Wooldridge (2007). "Social laws in alternating time: Effectiveness, feasibility, and synthesis." In: *Synthese* 156.1, pp. 1–19.

Wongpiromsarn, Tichakorn, Sertac Karaman, and Emilio Frazzoli (Oct. 2011). "Synthesis of provably correct controllers for autonomous vehicles in urban environments." In: *ITSC*, pp. 1168–1173.

Chapter 8

MULTI-MODAL TRAJECTORY PREDICTION: A SET COVER APPROACH[1]

8.1 Introduction

Previous chapters have discussed contract frameworks and algorithms that can provide provable guarantees to a set of simulated systems of autonomous vehicles. A key assumption that we often use in these examples is that at anytime, the set of available actions for each agent is available to us. In the real world, where the environments are dynamic, interactive, and uncertain, this assumption needs to be addressed. Therefore, in this chapter, we focus on the problem of predicting the behavior of vehicles navigating in an urban environment, where the road must be shared with a diverse set of other agents, including other vehicles, bicyclists, and pedestrians. In this context, reasoning about the possible future states of agents is critical for safe and confident operation. Effective prediction of future agent states depends on both road context (e.g., lanes geometry, crosswalks, traffic lights) and the recent behavior of other agents.

We treat the *behavior prediction* problem effectively as predicting future trajectories of a vehicle. Trajectory prediction is inherently challenging due to a wide distribution of agent preferences (e.g., a cautious vs. aggressive driver) and intents (e.g., turn right vs. go straight). Useful predictions must represent multiple possibilities and their associated likelihoods. Furthermore, we expect that predicted trajectories are physically realizable.

Multimodal regression models appear naturally suited for this task, but may degenerate during training into a single mode. Careful considerations are required to avoid this "mode collapse" (Cui, Radosavljevic, et al., 2019; Chai et al., 2019; Hong, Sapp, and Philbin, 2019). Additionally, most state-of-the-art methods predict unconstrained positions (Cui, Radosavljevic, et al., 2019; Chai et al., 2019; Hong, Sapp, and Philbin, 2019; Rhinehart, McAllister, et al., 2019), which may result in trajectories that are not physically possible for the agent to execute ((Cui, Nguyen, et al., 2019) is a recent exception). Our main insights leverage domain-specific

[1]The material in this chapter comes from joint work with Elena Corina Grigore, Freddy A. Boulton, Oscar Beijbom, and Eric M. Wolff.

knowledge to effectively structure the output representation to address these concerns. Our first insight is that there are relatively few *distinct* actions that can be taken over a *reasonable* time horizon. Dynamic constraints considerably limit the set of reachable states over a standard six-second prediction horizon, and the inherent uncertainty in agent behavior renders small approximation errors inconsequential. We exploit this insight to formulate the multimodal, probabilistic trajectory prediction problem as classification over a trajectory set. This formulation avoids mode collapse and lets the user design the trajectory set to meet specific requirements (e.g., dynamically feasible, coverage guarantees).

Our second insight is that predicted trajectories should be consistent with the current dynamic state. Thus, we formulate our output as motions relative to our initial state (e.g., turn slightly right, accelerate). When integrated with a dynamics model, the output is converted to an appropriate sequence of positions. Beyond helping to ensure physically valid trajectories, this *dynamic* output representation ensures that the outputs are diverse in the control space across a wide range of speeds. While (Cui, Nguyen, et al., 2019) exploit a similar insight for regression, we extend the use of a dynamic representation to classification and anchor-box regression.

We now summarize our main contributions on multimodal, probabilistic trajectory prediction with CoverNet:

- introduce the notion of trajectory sets for multimodal trajectory prediction, and show how to generate them in both a fixed and dynamic manner;
- compare state-of-the-art methods on nuScenes (Caesar et al., 2019), a public, real-world urban driving benchmark;
- empirically show the benefits of classification on trajectory sets over multimodal regression.

8.2 Related Work

We focus on trajectory prediction approaches based on deep learning, and refer the reader to (Lefèvre, Vasquez, and Laugier, 2014) for a survey of more classical approaches. The approaches below typically use convolutional neural networks (CNN) to combine agent history with scene context, and vary significantly in their output representations. Depending on the method, the scene context will include everything from the past states of a single agent to the past states of all agents along with high-fidelity map information.

Stochastic approaches encode choice over multiple possibilities via sampling random variables. One of the earliest works on motion forecasting frames the problem as learning stochastic 1-step policies (Kitani et al., 2012). R2P2 (Rhinehart, Kitani, and Vernaza, 2018) improves sample coverage for such policies via a symmetric Kullback–Leibler divergence (KL) loss. Recent work has considered the multi-agent setting (Rhinehart, McAllister, et al., 2019) and uncertainty in the model itself (Henaff, LeCun, and Canziani, 2019). Other methods generate samples using correlated variational auto-encoders (CVAEs) (Hong, Sapp, and Philbin, 2019; Lee et al., 2017; Bhattacharyya, Schiele, and Fritz, 2018; Ivanovic et al., 2018) or generative adversarial networks (GANs) (Sadeghian et al., 2019; Gupta et al., 2018; Zhao et al., 2019). Stochastic approaches can be computationally expensive due to a) repeated 1-step rollouts (in the 1-step policy approach), or b) due to requiring a large number of samples for acceptable performance (often hard to determine in practice).

Unimodal approaches output a single future trajectory per agent (Luo, Yang, and Urtasun, 2018; Casas, Luo, and Urtasun, 2018; Djuric et al., 2018; Alahi et al., 2016). A single trajectory is often unable to adequately capture the possibilities that exist in complex scenarios, even when predicting Gaussian uncertainty. Furthermore, these approaches typically average over behaviors, which may result in nonsensical trajectories (e.g., halfway between making a right turn and going straight).

Multimodal approaches output either a distribution over multiple trajectories (Chai et al., 2019; Cui, Radosavljevic, et al., 2019; Hong, Sapp, and Philbin, 2019; Deo and Trivedi, 2018) or a spatial-temporal occupancy map (Hong, Sapp, and Philbin, 2019; Bansal, Krizhevsky, and Ogale, 2019; Zeng et al., 2019). Spatial-temporal occupancy maps flexibly capture multiple outcomes, but often have large memory requirements to grid the output space at a reasonable resolution. Additionally, sampling trajectories from a spatial-temporal occupancy map is a) not well defined, and b) adds additional compute at the inference stage. Multimodal regression approaches can easily suffer from "mode collapse" to a single mode, leading (Chai et al., 2019) to use a fixed set of anchor boxes to mitigate the issue.

Most trajectory prediction algorithms do not explicitly encode constraints on object motion, and may predict trajectories that are physically infeasible (a recent exception is (Cui, Nguyen, et al., 2019)). By careful choice of our output representation, we are able to exclude all trajectories that would be physically impossible to execute.

Graph search is a classic approach to motion planning (LaValle, 2006), and often

used in urban driving applications (Buehler, Iagnemma, and Singh, 2009). A motion planner grows a compact graph (or tree) of possible motions, and computes the best trajectory from this set (e.g., max clearance from obstacles). Since we do not know the other agent's goals or preferences, we cannot directly plan over the trajectory set. Instead, we implicitly estimate these features and directly classify over the set of possible trajectories. There is a fundamental tension between the size of the trajectory set and the coverage of all potential motions (Branicky, Knepper, and Kuffner, 2008). Since we are only trying to predict the motions of other vehicles well enough to drive, we can easily accept small errors over moderate time horizons (3 to 6 seconds).

Comparing results on trajectory prediction for self-driving cars in urban environments is challenging, largely due to the wide variety of datasets. Numerous papers are evaluated purely on internal datasets (Cui, Radosavljevic, et al., 2019; Zeng et al., 2019; Casas, Luo, and Urtasun, 2018; Bansal, Krizhevsky, and Ogale, 2019), as common public datasets are either relatively small (Geiger, Lenz, and Urtasun, 2012), purely focused on highway driving (Colyar and Halkias, 2007), and/or are tangentially related to driving (Robicquet et al., 2016). While there are encouraging new developments in public datasets for trajectory prediction (Chang et al., 2019; Hong, Sapp, and Philbin, 2019), there is no standard. To help provide clear and open results, we evaluate our models on nuScenes (Caesar et al., 2019), a recent public self-driving car dataset focused on urban driving.

8.3 Method

CoverNet computes a multimodal, probabilistic prediction of the future states of a given vehicle using i) the current and past states of all agents (e.g., vehicles, pedestrians, bicyclists) and ii) a high-definition map.

We assume access to the state outputs of an object detection and tracking system of sufficient quality for self-driving. We denote the set of agents that a self-driving car interacts with at time t by $\mathcal{I}_t$ and s_t^i the state of agent $i \in \mathcal{I}_t$ at time t. Let $\mathbf{s}_{m:n}^i = \left[s_m^i, \ldots, s_n^i\right]$ where $m < n$ and $i \in \mathcal{I}_t$ denote the discrete-time trajectory of agent i for times $t = m, \ldots, n$. Without loss of generality, denote the current state of agent i by s_0^i, its past trajectory $\mathbf{s}_{m:0}^i$ (where $m < 0$), and its future trajectory $\mathbf{s}_{1:N}^i$ (where $N \geq 1$). In the remainder, we drop the agent index when referring to the agent for which we are currently making a prediction.

Furthermore, we assume access to a high-definition map $\mathcal{M}$. The map $\mathcal{M}$ includes

lane geometry, crosswalks, drivable area, and other relevant information.

Let $\mathbf{x} = \{\bigcup_i s^i_{m:0}; \mathcal{M}\}$ denote the scene context over the past m steps (i.e., map and partial history of all agents).

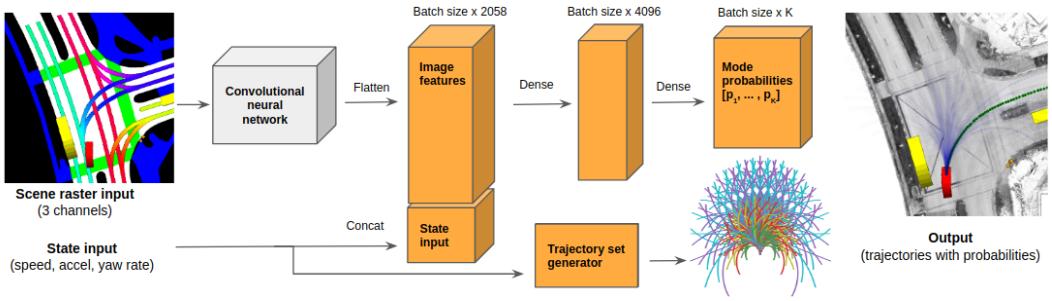


Figure 8.1: CoverNet overview. We generate a trajectory set (fixed or dynamic based on current state) that we classify over. The input and backbone follow (Cui, Radosavljevic, et al., 2019).

Figure 8.1 overviews our model architecture. It largely follows (Cui, Radosavljevic, et al., 2019), with the key difference in the output representation (see Section 8.3). We use ResNet-50 (He et al., 2015) given its effectiveness in this domain (Cui, Radosavljevic, et al., 2019; Chai et al., 2019).

While our network only computes a prediction for a single agent at a time, our approach can be extended to simultaneously predict for all agents in a similar manner as (Chai et al., 2019). We focus on single agent predictions (as in (Cui, Radosavljevic, et al., 2019)) both to simplify the presentation and focus on our main contributions.

The following sections detail our input and output representations. Our core innovations are in our output representations, specifically the *dynamic* encoding of trajectories, and in treating the problem as classification over a diverse set of trajectories.

Input representation

Similar to (Djuric et al., 2018), we rasterize the scene for each agent as an RGB image. We start with a blank image of size (H, W, 3) and draw the driveable area, crosswalks, pedestrian crossings, and walk ways using a distinct color for each semantic category.

We rotate the image so that the agent's heading faces up, and place the agent on pixel (l, w), measured from the top-left of the image. We assign a different color

to vehicles and pedestrians and choose a different color for the agent so that it is distinguishable. In our experiments, we use a resolution of 0.1 meters per pixel and choose $l = 400$ and $w = 250$. Thus, the model can "see" 40 meters ahead, 10 meters behind, and 25 meters on each side of the agent.

We represent the sequence of past observations for each agent as faded bounding boxes of the same color as the agent's current bounding box. We fade colors by linearly decreasing saturation (in HSV space) as a function of time.

Although we have only used one input representation in these experiments, our novel output representation can work with the input representations of (Bansal, Krizhevsky, and Ogale, 2019; Zeng et al., 2019).

Output representation

Due to the relatively short trajectory prediction horizons (up to 6 seconds) and inherent uncertainty in agent behavior, we approximate all possible motions with a set of trajectories that gives sufficient coverage of the space.

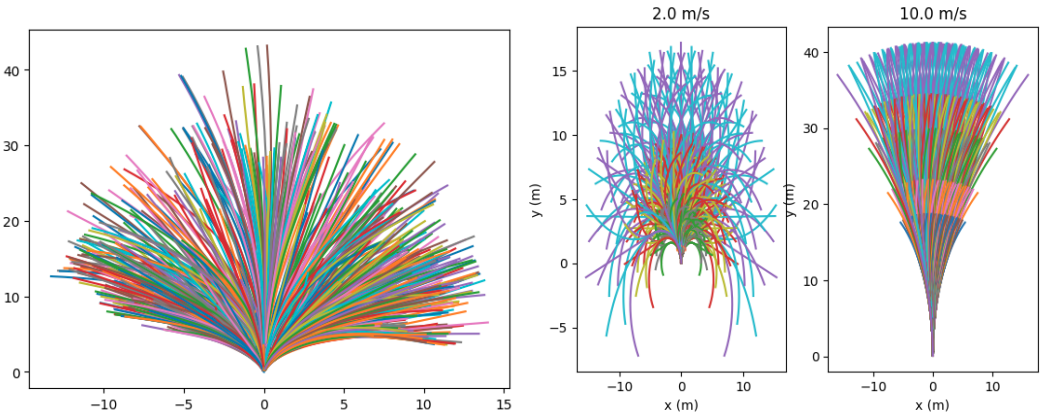
Let $\mathcal{R}_N(s_0)$ be the set of all states that can be reached by an agent with current state s_0 in N seconds (purely based on physical capabilities). We approximate this set by a finite number of trajectories $\mathcal{S}_{1:N} = \{s^k_{1:N}\}_{k=1}^K$, to define a trajectory set (k is over a single agent). Define a *dynamic* trajectory set generator as a function $f_N : s_0 \rightarrow \mathcal{S}_{1:N}$, which allows the trajectory set to be consistent with the current dynamics. This is in contrast to a *fixed* generator which does not use information about the current state, and thus returns the same trajectories for each instance. We discuss construction trajectory sets in Section 8.4.

We encode multimodal, probabilistic trajectory predictions by classifying over the appropriate trajectory set given an agent of interest and the scene context $\mathbf{x}$. As is common in the classification literature, we use the softmax distribution. Concretely, the probability of the k-th trajectory is given as $p(s^k_{1:N}|\mathbf{x}) = \frac{\exp f_k(\mathbf{x})}{\sum_i \exp f_i(\mathbf{x})}$, where $f_i(\mathbf{x}) \in \mathbb{R}$ is the output of the network's penultimate layer.

In contrast to previous work (Cui, Radosavljevic, et al., 2019; Chai et al., 2019), we choose not to learn an uncertainty distribution over the space. While it is straightforward to add Gaussian uncertainty along each trajectory in a similar manner to (Chai et al., 2019), the density of our trajectory sets reduces its benefit compared to the case when there are a only a handful of modes.

8.4 Trajectory Sets

In this section, we outline the main contribution of the chapter: a novel method for trajectory set generation. An ideal trajectory set always contains a trajectory that is close to the ground truth. We consider two broad categories of trajectory set generation functions: fixed and dynamic (see Figure 8.2). In both cases, we normalize the current state to be at the origin, with the heading oriented upwards (see Section 8.3).



(a) fixed (b) dynamic

Figure 8.2: Overview of trajectory set generation approaches.

Fixed trajectory sets

We consider a trajectory set to be "fixed" if the trajectories that it contains do not change as a function of the agent's current dynamic state or environment. Intuitively, this makes it easy to classify over since it allows for a fixed enumeration over the set, but may result in many trajectories that are poor matches for the current situation.

Given a set of representative trajectory data, the problem of finding the best fixed trajectory set of size $|\mathcal{K}|$ can be cast as an instance of the NP-hard set cover problem (Cormen et al., 2009). Approximating a dense trajectory set by a sparse trajectory set that still maintains good coverage and diversity has been studied in the context of robot motion planning (Branicky, Knepper, and Kuffner, 2008). In this work, we use a coverage metric δ defined as the maximum point-wise Euclidean distance between trajectories. Specifically, given two trajectories $s_{1:N}$ and $s'_{1:N}$, $\delta(s_{1:N}, s'_{1:N}) := \max_{k=1}^{N} \|s_k - s'_k\|$ where the norm is ℓ^2. Thus, we will also be referring to this metric as the maximum point-wise ℓ^2 distance. Our trajectory set construction procedure starts with subsampling a reasonably large set $\mathcal{K}$ of trajec-

tories (ours have size 20,000) from the training set. Selecting an acceptable error tolerance ε, we proceed to find the solution to

$$\begin{aligned} \underset{\mathcal{S}_{1:N}}{\text{argmin}} \quad & |\mathcal{S}_{1:N}| \\ \text{subject to} \quad & \mathcal{S}_{1:N} \subseteq \mathcal{K}, \\ & \forall k \in \mathcal{K}. \exists l \in \mathcal{S}_{1:N}. \delta(k, l) \leq \varepsilon. \end{aligned} \tag{8.1}$$

We employ a simple greedy approximation algorithm to solve (8.1), which we refer to as the "bagging" algorithm. In this procedure, we cherrypick the best among candidate trajectories to place in a bag of trajectories that will be used as the covering set. We propose two variants of this algorithm based on how we choose candidate trajectories. The first version repeatedly considers as candidates those trajectories that have not yet been covered and chooses the one that covers the most uncovered trajectories (ties are broken arbitrarily). The second variant makes a weighted random choice based on how many uncovered trajectories the candidates are covering until there are no more trajectories to cover. The latter (random) version can be repeated many times to obtain multiple bags, and we can choose the cover set based on the smallest number of elements. For simplicity, we use the former (deterministic) version in this work. Standard results (without using the specialized structure of the data) show that the deterministic greedy algorithm is suboptimal by a factor of at most $\log(|\mathcal{K}|)$ (see Chapter 35.3 (Cormen et al., 2009)). In our experiments, we were able to obtain decent coverage (specifically, under 2 meters in maximum point-wise ℓ^2 distance for 6 second trajectories) with fewer than 2,000 elements in the covering set.

Dynamic trajectory sets

We consider a trajectory set to be *dynamic* if the trajectories that it contains change as a function of the agent's current dynamic state. This construction guarantees that all trajectories in the set are dynamically feasible.

We now describe a simple approach to constructing such a dynamic trajectory set, focused on predicting vehicle motion. We use a standard vehicle dynamical model (LaValle, 2006) as similar models are effective for planning at urban (non-highway) driving speeds (Kong et al., 2015). Our approach, however, is not limited

to vehicles or any specific model. The dynamical model we use is:

$$\begin{aligned}
\dot{x} &= v\cos\theta \\
\dot{y} &= v\sin\theta \\
\dot{\theta} &= \frac{v}{L}\tan(u_{steer}) \\
\dot{v} &= u_{accel}
\end{aligned}$$

with states: x, y (position), v (speed), θ (yaw); controls: u_{steer} (steering angle), u_{accel} (longitudinal acceleration); and parameter: L (wheelbase).

The dynamics model, controls sequence, and current state determine a trajectory $s_{1:N}$ by forward integration. We create a dynamic trajectory set $S_{1:N}$ based on the current state s_0 by integrating forward with our dynamic model over diverse control sequences. Such a dynamic trajectory set has the possibility of being sparser than a fixed set for the same coverage, as each control sequence maps to multiple trajectories (as a function of the current state).

We parameterize the controls (output space) by a diverse set of constant lateral and longitudinal accelerations over the prediction horizon. Using lateral acceleration instead of steering angle is a way of normalizing the output over a range of speeds (a desired lateral acceleration will correspond to different steering angles as a function of speed). We convert the lateral acceleration into a steering angle assuming instantaneous circular motion $a_{lat} = v^2\kappa$ with curvature $\kappa = \tan(u_{steer})/L$. This conversion is ill-defined when the speed is near zero, so we use $\max(v, 1)$ in place of v. Note that it is straightforward to expand the controls (output space) to include multiple lateral and longitudinal accelerations over a non-uniform prediction horizon.

We can further prune the dynamic trajectory set construction in a similar manner to how we handled the fixed trajectory sets in Section8.4. The main difference is that the covering set here is constructed from the set of control input profiles as opposed to elements of $\mathcal{K}$ itself. Namely, we use an analogous greedy procedure to cover the set of sample trajectories with a subset of control profiles (e.g., lateral and longitudinal accelerations as a function of time). Note that unlike the case of fixed trajectories, the synthetic nature of the dynamic profile may not guarantee 100% coverage of $\mathcal{K}$. To counter this problem, we can also create a *hybrid* trajectory set by combining a fixed and dynamic set. Particularly, we find a covering subset for the elements of $\mathcal{K}$ that cannot be covered by the dynamic choices, and combine this

subset with the dynamic choices. When the dynamic set is well-constructed, this can result in a smaller covering set as may be seen from Figure 8.3.

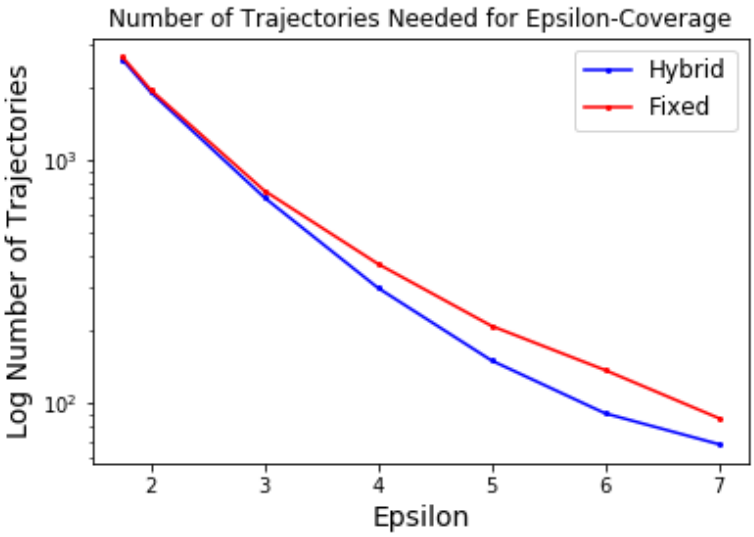


Figure 8.3: Number of trajectories needed for ε coverage (in meters, see Section 8.4)

8.5 Experiments

We now present our empirical results on trajectory prediction of other vehicles in urban environments. The following sections describe the baselines, metrics, and urban self-driving datasets that we considered. We used the same input representation and model architecture across our models and baselines.

Baselines

Physics oracle. We introduce a simple and interpretable model that extends classic physics-based models. We use the track's current velocity, acceleration, and yaw rate to compute the following predictions: i) constant velocity and yaw, ii) constant velocity and yaw rate, iii) constant acceleration and yaw, and iv) constant acceleration and yaw rate. The *oracle* is the minimum average point-wise Euclidean distance over the four models.

Regression baselines and extensions. We compare our contribution to state-of-the-art methods by implementing two main types of regression models: multimodal regression to coordinates (Cui, Radosavljevic, et al., 2019) and multimodal regression to residuals from a set of anchors (Chai et al., 2019) (ordinal regression). We overview these methods for completeness and to provide context for novel variations that we introduce.

Multimodal regression to coordinate

Our implementation follows the details of Multiple-Trajectory Prediction (MTP) (Cui, Radosavljevic, et al., 2019), adapted for the datasets we use. This model predicts a fixed number of trajectories (modes) and their associated probabilities. The per-agent loss (agent i at time t) is defined as:

$$\mathcal{L}_{it}^{MTP} = \sum_{k=1}^{|\mathcal{K}|} \mathbb{1}_{k=\hat{k}}[-\log\, p_{ik} + \alpha L(s_{1:N}^{it}, \hat{s}_{1:N}^{it})], \tag{8.2}$$

where $\mathbb{1}(\cdot)$ is the indicator function that equals 1 only for the "best matching" mode, k represents a mode, L is the regression loss, and α is a hyper-parameter used to trade off between classification and regression. The original implementation (Cui, Radosavljevic, et al., 2019) uses a heuristic based on the relative angle between each mode and the ground truth. We select a mode uniformly at random when there are no modes with an angle below the threshold.

Multimodal regression to anchor residuals

Our implementation follows the details of MultiPath (MP) (Chai et al., 2019). This model implements ordinal regression by first choosing among a fixed set of anchors (computed a priori) and then regressing to residuals from the chosen anchor. The proposed per-agent loss is (8.2) where $\alpha = 1$ and the k-th trajectory is the sum of the corresponding anchor and predicted residual. To remain true to the implementation in (Chai et al., 2019), we choose our best matching anchor by minimizing the average displacement to the ground truth.

Fixed anchors. We compute the set of fixed anchors by employing the same mechanism described in Section 8.4. Note that this set of trajectories is the same for all agents in the dataset. We then regress to the residuals from the chosen anchor.

Dynamic anchors. We compute dynamic anchors by employing a similar approach as described in Section 8.4. The set of anchors is thus a function of the agent's current speed, which helps ensure that anchors are dynamically feasible. We then regress to the residuals from the chosen anchor.

Our models

CoverNet (fixed). One of our classification approaches where there is a set of $|\mathcal{K}|$ fixed trajectories.

CoverNet (dynamic). One of our classification approaches where the set of $|\mathcal{K}|$ trajectories is a function of the initial state of the agent.

MultiPath with dynamic anchors. This is the MultiPath (sec. 8.5), extended to utilize dynamic anchors.

Implementation details

Our implementation setup follows (Cui, Radosavljevic, et al., 2019) and (Chai et al., 2019), with key differences highlighted below.

We implemented our models using ResNet-50 (He et al., 2015) as our backbone, with pre-trained ImageNet (Russakovsky et al., 2015) weights downloaded from (Contributors, 2019). We read the ResNet "conv5" feature map and apply a global pooling layer. We then concatenate the result with an agent state vector (including speed, acceleration, yaw rate), as detailed in (Cui, Radosavljevic, et al., 2019). We then add a fully connected layer, with dimension 4096.

The output dimension of CoverNet is equal to the number of modes, namely $|\mathcal{K}|$.

For the regression models, our outputs have dimension $|\mathcal{K}| \times (|\vec{x}| \times N + 1)$, where $|\mathcal{K}|$ represents the total number of predicted modes, $|\vec{x}|$ represents the number of features we are predicting per point, N represents the number of points in our predictions, and the extra output per mode is the probability associated with each mode. For our implementations, $N = H \times F$, where H represents the length of the prediction horizon, and F represents the sampling frequency. For each point, we predict (x, y) coordinates, so $|\vec{x}| = 2$.

The comparative datasets have $F = 10\,Hz$, while the publicly available nuScenes is sampled at $F = 2\,Hz$. We include results on two different prediction horizon lengths, namely $H = 3$ seconds and $H = 6$ seconds.

The loss functions we use are the same across all of our implementations: for any classification losses, we utilize cross-entropy with positive samples determined by the element in the trajectory set closest to the actual ground truth in minimum average of point-wise Euclidean distances, and for any regression losses, we utilize smooth ℓ^1. For our MTP implementation, we place equal weighting between the classification and regression components of the loss, setting $\alpha = 1$, similar to (Cui, Radosavljevic, et al., 2019).

For our classification models, we utilize a fixed learning rate of 1e−4. For our regression models, we utilize a learning rate of 1e−4, with a drop by 0.1 after epoch

31 for models with lower numbers of modes (one and three modes), and after epoch 7 for models with higher number of modes (16 and 64).

Metrics

There are multiple ways of evaluating multimodal trajectory prediction. Common measures include log-likelihood (Chai et al., 2019; Rhinehart, McAllister, et al., 2019), average displacement error, and hit rate (Hong, Sapp, and Philbin, 2019). We focus on the a) displacement error, and b) hit rate, both computed over a subset of the most likely modes.

For insight into trajectory prediction performance in scenarios where there are multiple plausible actions, we use the minimum average displacement error (ADE). The minADE_k is $\min_{k \in K} \frac{1}{T} \sum_{t=1}^{T} ||s_t - s_t^k||$, where K is the set of k most likely modes. We also analyze the final displacement error (FDE), which is $||s_t - s_T^*||$, where s^* is the most likely mode.

In the context of planning for a self-driving vehicle, the above metrics may be hard to interpret. We use the notion of a *hit rate* (see (Hong, Sapp, and Philbin, 2019)) to simplify interpretation of whether or not a prediction was "close enough." We define a $\text{Hit}_{k,\delta}$ for a single instance (agent at a given time) as 1 if $\min_{k \in K} \max_{t=1}^{T} ||s_t - s_t^k|| \leq \delta$, and 0 otherwise. When averaged over all instances, we refer to it as the $\text{HitRate}_{k,\delta}$.

Large comparative self-driving dataset

For comparison purposes, we will include results from (Phan-Minh et al., 2020) on a dataset obtained by collecting 60 hours of real-world, urban driving data in Singapore. Raw sensor data is collected by a car outfitted with cameras, lidars, and radars. A highly-optimized object detection and tracking system filters the raw sensor data to produce tracks at a 10 Hz rate. Each track includes information regarding its type (e.g., car, pedestrian, bicycle, unknown), pose, physical extent, and speed, with quality sufficient for fully-autonomous driving. Additionally, access is to high-definition maps with semantic labels of the road such as the drivable area, lane geometry, and crosswalks is available.

Each ego vehicle location at a given timestamp is considered a data point. They do not predict on any tracks that are stationary over the entire prediction horizon. This large compartive dataset contains around 11 million usable data points but for this analysis we created train, validation, and test sets with 1 million, 300,000, and 300,000 data points, respectively.

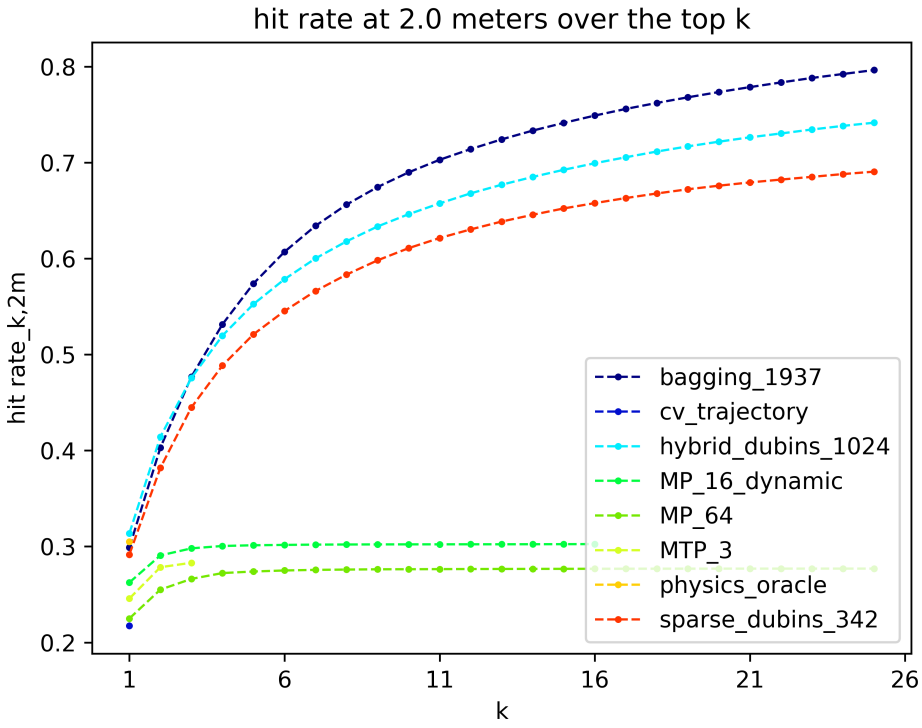


Figure 8.4: Best models of each type on the comparative dataset (6-second horizon). CoverNet models significantly outperform others.

nuScenes

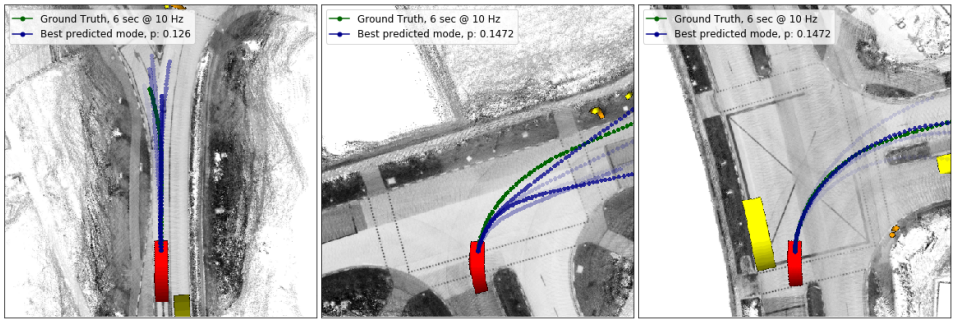
We also report results on nuScenes (Caesar et al., 2019), a public self-driving car dataset. nuScenes consists of 1000 scenes, each 20 seconds in length. Scenes are taken from urban driving in Boston, USA, and Singapore. Each scene includes hand-annotated tracks and high-definition maps. Tracks have 3D ground truth annotations, and are published at 2 Hz. We processed each of the vehicles in the nuScenes train-val split to train, validation, and test sets such that the number of vehicles operating in left-hand and right-hand driving locations was equal. We removed vehicles that are stationary. This leaves us with 37,714 observations in the train set, 8,064 observations in the validation set, and 7,790 observations in the test set. We use this split to make both three-second and six-second predictions.

Ablation study: Matching ground truth

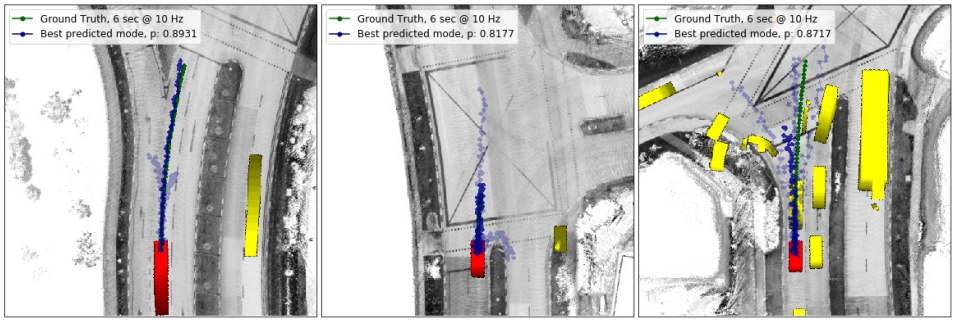
We analyzed different methods for matching the ground truth to the most suitable trajectory in the trajectory set. Table 8.1 compares performance using the max, average, and root-mean-square of the point-wise error vector of Euclidean distances

Method	minADE_1	minADE_5	minADE_{10}	minADE_{15}
max ℓ^2	1.0	0.67	0.64	0.64
average ℓ^2	0.96	0.66	0.64	0.64
RMS of ℓ^2	0.96	0.66	0.64	0.63

Table 8.1: Ground truth matching for fixed trajectory set (150 modes) on comparative dataset (3 sec horizon).



(a) Cover (fixed $\varepsilon = 2$) (b) Cover (hybrid) (c) Cover (dynamic $\varepsilon = 3$)



(d) MTP (3 modes) (e) MP (dynamic templates, 16 modes) (f) MP (fixed templates, 64 modes)

Figure 8.5: Examples of predicted trajectories. The top row includes our CoverNet models, ranging from fixed to dynamic. The bottom row includes the baselines we compare against, as well as our dynamic templates variation.

for matching ground truth to the "best" trajectory in a fixed trajectory set of size 150. Performance is relatively consistent across all three choices, so we picked the average point-wise ℓ^2 norm to better align with related regression approaches (Chai et al., 2019).

Discussion

The main results are summarized in Table 8.2. Qualitative results are shown in Figure 8.5.

Method	Modes	$\text{minADE}_1 \downarrow$	$\text{minADE}_5 \downarrow$	$\text{minADE}_{10} \downarrow$	$\text{minADE}_{15} \downarrow$	FDE ↓	$\text{HitRate}_{5,\,2\text{m}} \uparrow$
Const. vel. & yaw	N/A	3.75 (3.63)	3.75 (3.63)	3.75 (3.63)	3.75 (3.63)	9.44 (9.86)	0.19 (0.22)
Physics oracle	N/A	**2.79** (**1.88**)	2.79 (1.88)	2.79 (1.88)	2.79 (1.88)	**7.18** (5.72)	0.23 (0.31)
MTP (Cui, Radosavljevic, et al., 2019)	1	4.37 (**1.88**)	4.37 (1.88)	4.37 (1.88)	4.37 (1.88)	9.84 (**5.22**)	0.10 (0.24)
MTP (Cui, Radosavljevic, et al., 2019)	3	4.98 (2.01)	3.73 (1.73)	3.73 (1.73)	3.73 (1.73)	10.91 (5.45)	0.18 (0.28)
MTP (Cui, Radosavljevic, et al., 2019)	16	5.38 (3.15)	4.01 (2.48)	3.90 (2.43)	3.86 (2.42)	11.46 (7.79)	0.21 (0.25)
MTP (Cui, Radosavljevic, et al., 2019)	64	5.47 (3.21)	3.97 (2.63)	3.83 (2.51)	3.77 (2.47)	11.47 (7.74)	0.21 (0.27)
MultiPath (Chai et al., 2019)	16	4.76 (2.34)	2.66 (1.71)	2.61 (1.71)	2.60 (1.70)	10.91 (5.83)	0.17 (0.24)
MultiPath (Chai et al., 2019)	64	4.84 (2.30)	**2.22** (1.42)	1.96 (1.36)	1.89 (1.34)	10.21 (5.63)	0.22 (0.27)
MultiPath (Chai et al., 2019) (dyn.)	16	4.24 (2.06)	2.46 (1.47)	2.40 (1.46)	2.38 (1.46)	9.80 (5.76)	0.28 (0.30)
MultiPath (Chai et al., 2019) (dyn.)	64	4.20 (2.23)	2.38 (1.53)	2.19 (1.46)	2.12 (1.44)	9.67 (6.17)	0.28 (0.28)
Cover, fixed, ε=8	64 (64)	4.71 (2.77)	2.41 (1.98)	2.13 (1.93)	2.07 (1.93)	10.16 (6.65)	0.05 (0.06)
Cover, fixed, ε=5	232 (208)	4.98 (2.32)	2.31 (1.35)	1.85 (1.25)	1.69 (1.22)	10.71 (5.67)	0.13 (0.31)
Cover, fixed, ε=4	415 (374)	4.87 (2.27)	2.37 (1.29)	1.84 (1.15)	1.65 (1.10)	10.38 (5.85)	0.30 (0.35)
Cover, fixed, ε=3	844 (747)	5.29 (2.28)	2.61 (1.32)	1.94 (1.13)	1.69 (1.07)	11.20 (5.92)	0.33 (0.33)
Cover, fixed, ε=2	2206 (1937)	5.88 (2.16)	3.28 (**1.16**)	2.49 (**0.93**)	2.14 (**0.84**)	12.13 (5.53)	0.34 (**0.57**)
Cover, dyn., ε=3	357 (342)	4.89 (2.06)	2.70 (1.17)	2.18 (0.97)	1.93 (0.88)	11.68 (5.90)	0.35 (0.52)
Cover, dyn., ε=1.75	n/a (313)	n/a (2.03)	n/a (1.26)	n/a (1.04)	n/a (0.94)	n/a (5.70)	n/a (0.52)
Cover, hybrid	774 (1024)	4.71 (2.18)	2.42 (1.24)	**1.80** (0.99)	**1.52** (0.88)	10.68 (5.84)	**0.38** (0.55)

Table 8.2: nuScenes and comparative datasets (6-second horizon). Results listed as nuScenes (comparative). Smaller minADE_k and FDE is better. Larger $\text{HitRate}_{5,\,2\text{m}}$ is better. Dyn. = dynamic, vel. = velocity, const. = constant, ε is given in meters.

Across the 6 metrics and the 2 datasets, CoverNet outperforms previous methods and baselines in 7 out of 12 cases. However, there are big differences in method ranking depending which metric is considered.

CoverNet method represents a significant improvement on the $\text{HitRate}_{5,\,2\text{m}}$ metric achieving 38% on nuScenes using the hybrid trajectory set. The next best model is MultiPath, where our dynamic grid extension represents a large improvement over the fixed grid used by the authors (28% vs. 22%). MTP performs worse achieving 18% $\text{HitRate}_{5,\,2\text{m}}$, slightly lower than the constant velocity baseline.

A similar pattern is seen on the comparative dataset with CoverNet outperforming previous methods and baselines. Here, the fixed set with 1937 modes performs best (57%), closely followed by the hybrid set (55%). Among previous methods, again MultiPath with dynamic set works the best at (30%) $\text{HitRate}_{5,\,2\text{m}}$. Figure 8.4 shows that CoverNet significantly outperforms previous methods as the hit rate is expanded over more modes.

CoverNet also performs well according the Average Displace Error minADE_k metrics. In particular for $k \in \{5, 10, 15\}$. For minADE_{15}, the hybrid CoverNet with fixed set and 2206 modes performs best with minADE_{15} =0.84, 4x better than the constant velocity baseline and 2x better than the MTP and MultiPath. For lower k, such as minADE_1 the regression methods performed the best. This is not surprising since for low k, it is more important to have one trajectory very close to the ground truth, a metric paradigm that favors regression over classification.

A notable difference between nuScenes and comparative is that the $\text{HitRate}_{5,\,2\text{m}}$ and minADE_k continues to improve for larger sets, while it plateaus, or even decreases at around 500-1000 modes on nuScenes. We hypothesize that this is due to the relatively limited size of nuScenes.

8.6 Conclusion

We introduced CoverNet, a novel method for multimodal, probabilistic trajectory prediction in real-world, urban driving scenarios. By framing this problem as classification over a diverse set of trajectories, we were able to a) ensure a desired level of coverage of the state space, b) eliminate dynamically infeasible trajectories, and c) avoid the issue of mode collapse. We showed that the size of our trajectory sets remains manageable over realistic prediction horizons. Dynamically generating trajectory sets based on the agent's current state further improved performance. We compared our results to multiple state-of-the-art methods on real-world self-driving

datasets, and showed that it outperforms similar methods.

References

Alahi, Alexandre, Kratarth Goel, Vignesh Ramanathan, Alexandre Robicquet, Li Fei-Fei, and Silvio Savarese (June 2016). "Social LSTM: Human Trajectory Prediction in Crowded Spaces." In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Bansal, Mayank, Alex Krizhevsky, and Abhijit Ogale (June 2019). "ChauffeurNet: Learning to Drive by Imitating the Best and Synthesizing the Worst." In: *Robotics: Science and Systems (RSS)*.

Bhattacharyya, Apratim, Bernt Schiele, and Mario Fritz (June 2018). "Accurate and Diverse Sampling of Sequences Based on a "Best of Many" Sample Objective." In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Branicky, Michael S., Ross A. Knepper, and James J. Kuffner (May 2008). "Path and trajectory diversity: Theory and algorithms." In: *2008 IEEE International Conference on Robotics and Automation*.

Buehler, Martin, Karl Iagnemma, and Sanjiv Singh (2009). *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic*. 1st. Springer Publishing Company.

Caesar, Holger, Varun Bankiti, Alex H. Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom (2019). "nuScenes: A multimodal dataset for autonomous driving." In: *arXiv preprint arXiv:1903.11027*.

Casas, Sergio, Wenjie Luo, and Raquel Urtasun (Oct. 2018). "IntentNet: Learning to Predict Intention from Raw Sensor Data." In: *Proceedings of The 2nd Conference on Robot Learning*, pp. 947–956.

Chai, Yuning, Benjamin Sapp, Mayank Bansal, and Dragomir Anguelov (2019). "MultiPath: Multiple Probabilistic Anchor Trajectory Hypotheses for Behavior Prediction." In: *3rd Conference on Robot Learning (CoRL)*. Osaka, Japan.

Chang, Ming-Fang et al. (June 2019). "Argoverse: 3D Tracking and Forecasting With Rich Maps." In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Colyar, James and John Halkias (2007). *US highway 101 dataset*.

Contributors, Torch (2019). *TORCHVISION.MODELS*. URL: `https://pytorch.org/docs/stable/torchvision/models.html`.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (2009). *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press.

Cui, Henggang, Thi Nguyen, Fang-Chieh Chou, Tsung-Han Lin, Jeff Schneider, David Bradley, and Nemanja Djuric (2019). *Deep Kinematic Models for Physically Realistic Prediction of Vehicle Trajectories*. arXiv: `1908.00219 [cs.RO]`.

Cui, Henggang, Vladan Radosavljevic, Fang-Chieh Chou, Tsung-Han Lin, Thi Nguyen, Tzu-Kuo Huang, Jeff Schneider, and Nemanja Djuric (2019). "Multimodal Trajectory Predictions for Autonomous Driving using Deep Convolutional Networks." In: *2019 International Conference on Robotics and Automation (ICRA)*, pp. 2090–2096.

Deo, Nachiket and Mohan Trivedi (June 2018). "Convolutional Social Pooling for Vehicle Trajectory Prediction." In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, pp. 1549–15498.

Djuric, Nemanja, Vladan Radosavljevic, Henggang Cui, Thi Nguyen, Fang-Chieh Chou, Tsung-Han Lin, and Jeff Schneider (2018). *Short-term Motion Prediction of Traffic Actors for Autonomous Driving using Deep Convolutional Networks*. arXiv: 1808.05819.

Geiger, Andreas, Philip Lenz, and Raquel Urtasun (2012). "Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite." In: *Conference on Computer Vision and Pattern Recognition (CVPR)*.

Gupta, Agrim, Justin Johnson, Li Fei-Fei, Silvio Savarese, and Alexandre Alahi (June 2018). "Social GAN: Socially Acceptable Trajectories With Generative Adversarial Networks." In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2015). "Deep Residual Learning for Image Recognition." In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778.

Henaff, Mikael, Yann LeCun, and Alfredo Canziani (2019). "Model-predictive policy learning with uncertainty regularization for driving in dense traffic." In: *7th International Conference on Learning Representations (ICLR)*.

Hong, Joey, Benjamin Sapp, and James Philbin (June 2019). "Rules of the Road: Predicting Driving Behavior With a Convolutional Model of Semantic Interactions." In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Ivanovic, Boris, Edward Schmerling, Karen Leung, and Marco Pavone (Oct. 2018). "Generative modeling of multimodal multi-human behavior." In: *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*.

Kitani, Kris M., Brian D. Ziebart, J. Andrew Bagnell, and Martial Hebert (2012). "Activity Forecasting." In: *The European Conference on Computer Vision (ECCV)*.

Kong, Jason, Mark Pfeiffer, Georg Schildbach, and Francesco Borrelli (June 2015). "Kinematic and dynamic vehicle models for autonomous driving control design." In: *2015 IEEE Intelligent Vehicles Symposium (IV)*.

LaValle, Steven M. (2006). *Planning Algorithms*. New York, NY, USA: Cambridge University Press. ISBN: 0521862051.

Lee, Namhoon, Wongun Choi, Paul Vernaza, Chris Choy, Philip Torr, and Manmohan Chandraker (July 2017). "DESIRE: Distant Future Prediction in Dynamic Scenes with Interacting Agents." In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2165–2174.

Lefèvre, Stéphanie, Dizan Vasquez, and Christian Laugier (2014). "A survey on motion prediction and risk assessment for intelligent vehicles." In: *ROBOMECH* 1.1, p. 1.

Luo, Wenjie, Bin Yang, and Raquel Urtasun (2018). "Fast and Furious: Real Time End-to-End 3D Detection, Tracking and Motion Forecasting with a Single Convolutional Net." In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Phan-Minh, Tung, Elena Corina Grigore, Freddy A. Boulton, Oscar Beijbom, and Eric M. Wolff (2020). "Covernet: Multimodal Behavior Prediction using Trajectory Sets." In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14074–14083. DOI: 10.1109/CVPR42600.2020.01408.

Rhinehart, Nicholas, Kris M. Kitani, and Paul Vernaza (Sept. 2018). "R2P2: A ReparameteRized Pushforward Policy for Diverse, Precise Generative Path Forecasting." In: *The European Conference on Computer Vision (ECCV)*.

Rhinehart, Nicholas, Rowan McAllister, Kris Kitani, and Sergey Levine (2019). "PRECOG: PREdiction Conditioned On Goals in Visual Multi-Agent Settings." In: *International Conference on Computer Vision*.

Robicquet, Alexandre, Amir Sadeghian, Alexandre Alahi, and Silvio Savarese (2016). "Learning Social Etiquette: Human Trajectory Prediction In Crowded Scenes." In: *European Conference on Computer Vision (ECCV)*.

Russakovsky, Olga et al. (Dec. 2015). "ImageNet Large Scale Visual Recognition Challenge." In: *Int. J. Comput. Vision* 115.3, pp. 211–252.

Sadeghian, Amir, Vineet Kosaraju, Ali Sadeghian, Noriaki Hirose, Hamid Rezatofighi, and Silvio Savarese (June 2019). "SoPhie: An Attentive GAN for Predicting Paths Compliant to Social and Physical Constraints." In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Zeng, Wenyuan, Wenjie Luo, Simon Suo, Abbas Sadat, Bin Yang, Sergio Casas, and Raquel Urtasun (June 2019). "End-To-End Interpretable Neural Motion Planner." In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Zhao, Tianyang, Yifei Xu, Mathew Monfort, Wongun Choi, Chris Baker, Yibiao Zhao, Yizhou Wang, and Ying Nian Wu (2019). "Multi-Agent Tensor Fusion for Contextual Trajectory Prediction." In: *CVPR*.

Chapter 9

CONCLUSIONS

9.1 Summary

A variety of theoretical frameworks and cyber-physical system applications of the contract-based design paradigm were considered in this thesis.

First, we constructed a metatheory-compliant contract theory for modal interface input/output automata with guards, and presented an application of the framework in the design of a traffic intersection control system. Then, we developed an assume-guarantee contract framework to model the directive-response architecture of components of an automated valet parking system to capture communication between subsystems with a vertical hierarchy. To restrict implementations from being "opportunistic" in the presence of a potential environment failure while not sacrificing performance, we introduced the theory of reactive contracts and provided contract synthesis algorithms involving meta-specifications at the assume-guarantee level. These led to a more fine-grain adaptive control of the specifications.

In the second half of thesis, we formulated a behavior contract framework for autonomous agents and showed how to apply it to the development of autonomous driving technologies, specifically the distributed control of autonomous vehicles. We gave a theoretical framework for transparent distributed multi-agent decision making and behavior prediction using "assume-guarantee profiles." A provably correct distributed conflict resolution algorithm for quasi-simultaneous games was provided to guarantee safety and liveness for a composed system of autonomous vehicle agents. Finally, we proposed a neural-network based probabilistic multi-modal behavior prediction technique as a step toward extending our contract-based frameworks to real-world engineering systems.

9.2 Future Directions

Within the context of cyber-physical systems, there remain many exciting opportunities and open challenges for contract-based design. A limiting/ultimate challenge is achieving a unified formal specification language for the implementations of engineering systems and their contracts. Though expressive frameworks like TLA$^+$ (Lamport, 2015) and supporting tools are available, the sheer variety of ap-

plications means that mapping and translating these contracts either by hand or in an automated manner between different domains is and will remain difficult even if we have a good understanding of the underlying systems. The following is an outline of potential directions for future investigation.

- **Specialized contract theories**: a gradual approach to address the above problem involves first "decomposing" it into specialized contract theory instances corresponding to settings where domain knowledge is abundant. The benefits of contract-based reasoning will be most strongly felt when we are to identify and categorize classes of systems for which there are known efficient decision algorithms for contract realizability and compliance. We can further take advantage of these specialized structures to develop contract-based verification and synthesis methods such as automated refinement to obtain realizability certificate, as has been done with SMT solvers in (Gacek et al., 2015).
- **Learning contracts and implementations from data**: when domain knowledge is incomplete or difficult to obtain, it also makes sense to try learning and verifying contracts and implementations from data. This can be done in the form of learning assumptions as in our previous work (Chen et al., 2020), guarantees as in (DeCastro et al., 2018), or (more generally) input-output relations (Seshia et al., 2018)

With regard to the *Rules of the Road* portion of this work, we propose the following.

- **Pushing for a complete set of road rules**: the completeness of our framework is predicated upon including continuous behaviors, more road structures and agent types like bicyclists and pedestrians. It is also important to evaluate its robustness in the presence of failures and uncertainty and address how communication assumptions may be relaxed while still guaranteeing liveness. The completeness of specification structures depends on whether we can incorporate all edge cases. One interesting direction involves leveraging learning approaches (You et al., 2019).

References

Chen, Yuxiao, Sumanth Dathathri, Tung Phan-Minh, and Richard M. Murray (2020). "Counter-example Guided Learning of Bounds on Environment Behavior." In: *Conference on Robot Learning*, pp. 898–909. URL: http://proceedings.mlr.press/v100/chen20b.html.

DeCastro, Jonathan, Lucas Liebenwein, Cristian-Ioan Vasile, Russ Tedrake, Sertac Karaman, and Daniela Rus (2018). "Counterexample-guided safety contracts for autonomous driving." In: *International Workshop on the Algorithmic Foundations of Robotics*. Springer, pp. 939–955.

Gacek, Andrew, Andreas Katis, Michael W. Whalen, John Backes, and Darren Cofer (2015). "Towards realizability checking of contracts using theories." In: *NASA Formal Methods Symposium*. Springer, pp. 173–187.

Lamport, Leslie (2015). *The TLA+ Hyperbook*. `http://lamport.azurewebsites.net/tla/hyperbook.html`.

Seshia, Sanjit A., Ankush Desai, Tommaso Dreossi, Daniel J Fremont, Shromona Ghosh, Edward Kim, Sumukh Shivakumar, Marcell Vazquez-Chanlatte, and Xiangyu Yue (2018). "Formal specification for deep neural networks." In: *International Symposium on Automated Technology for Verification and Analysis*. Springer, pp. 20–34.

You, Changxi, Jianbo Lu, Dimitar Filev, and Panagiotis Tsiotras (2019). "Advanced planning for autonomous vehicles using reinforcement learning and deep inverse reinforcement learning." In: *Robotics and Autonomous Systems* 114, pp. 1–18.