Facilitating Formal Verification of Cooperative Driving Applications: Techniques and Case Study

Shou-pon Lin

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2016

ABSTRACT

Facilitating Formal Verification of Cooperative Driving Applications: Techniques and Case Study

Shou-pon Lin

The next generation of intelligent vehicles will evolve from being able to drive autonomously to ones that communicate with other vehicles and execute joint behaviors. Before allowing these vehicles on public roads, we must guarantee that they will not cause accidents. We will apply formal methods to ensure the degree of safety that cannot be assured with simulation or closed-track testing. However, there are challenges that need to be addressed when applying formal verification techniques to cooperative driving systems.

This thesis focuses on the techniques that address the following challenges: 1. Automotive applications interact with the physical world in different ways; 2. Cooperative driving systems are time-critical; 3. The problem of state explosion when we apply formal verification to systems with more participants.

First, we describe the multiple stack architecture. It combines several stacks, each of which addresses a particular way of interaction with the physical world. The layered structure in each stack makes it possible for engineers to implement cooperative driving applications without being bogged down by the details of low-level devices. Having functions arranged in a layered fashion helps us divide the verification of the whole system into smaller subproblems of independent module verification.

Secondly, we present a framework for modeling the protocol systems that uses GPS clocks for synchronization. We introduce the timing stack, which separates a process into two parts: the part modeled as an finite-state machine that controls state transitions and messages exchanges, and the part that determines the exact moment that a timed event should occur. The availability of accurate clocks at different locations allows processes to execute actions simultaneously, reducing interleaving that often arises in systems that use

multiple timers to control timed events. With accurate clocks, we create a lock protocol that resolves conflicting merge requests for driver-assisted merging.

Thirdly, we introduce stratified probabilistic verification that mitigates state explosion. It greatly improves the probability bound obtained in the original probabilistic verification algorithm. Unlike most techniques that aim at reducing state space, it is a directed state traversal, prioritizing the states that are more likely to be encountered during system execution. When state traversal stops upon depleting the memory, the unexplored states are the ones that are less likely to be reached. We construct a linear program whose solution is the upper bound for the probability of reaching those unexplored states. The stratified algorithm is particularly useful when considering a protocol system that depends on several imperfect components that may fail with small but hard-to-quantify probabilities. In that case, we adopt a compositional approach to verify a collection of components, assuming that the components have inexact probability guarantees.

Finally, we present our design of driver-assisted merging. Its design is reasonably simplified by using the multiple stack architecture and GPS clocks. We use a stratified algorithm to show that merging system fails less than once every 5×10^{13} merge attempts.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am fortunate to have wonderful people helping me along this quite unforgettable journey of Ph.D. studies and of self-exploration. It is sentimental since this thesis marks the culmination of twenty years of education since the first day I stepped into the gate of the elementary school.

First and foremost, I would like to thank my Ph.D. advisor, Professor Nicholas Maxemchuk, for his guidance and support. Without his foresight and invaluable experience this thesis would not have been possible. I am extraordinarily lucky to have had the chance to work with him, and will be forever grateful for his support and mentorship.

I am also thankful to my committee members, Prof. Debasis Mitra, Prof. Krishan Sabnani, Prof. Ümit Uyar and Prof. Gil Zussman for taking time out from their busy schedule and attending my dissertaion defense.

Special thanks go to my two college friends, Chung-heng and Tsung-hao. I am very fortunate to have them as classmates after four years of college and one year of military service. They are my family here in New York. Because of them I can endure all the setbacks. Thanks to Ming-hen for being my go-to guy whenever I have problems with programming. Thanks to Kuo-chiao to be the receiving end of my deepest worries and giving me meaningful interpretation of life and of being in New York.

I owe a dept of gratitude to my undergrad research advisor, Prof. Ping-cheng Yeh, and to my alma mater, National Taiwan University, for giving me the confidence to set out on the adventure of a Columbia University Ph.D. Program.

A very special one whom I would like to thank is Hsin-yu. She had been by my side, over the years since I landed in the United States the first time in my life. We have explored this best city in the world together, and we share numerous memories together. There are ups and downs, and we were in it all together. Should there have not been her, I could not

possibly complete this journey on my own. She is my partner, my best friend, my family.

Last but not least, I would like to express my deepest thanks to my family - my mom, my dad, my love ones, and my hometown, Taiwan.

For myself, I want to keep reminding my future self not to forget where I came from and why I embarked on this journey.

Support: This thesis was support in part by NSF grants CNS-1329593 and CNS-1035178.

To Ma, Pa, and Taochi

Chapter 1

Introduction

Sensing, communications, and vehicle control are being integrated in intelligent vehicles. There have been successful demonstrations of autonomous driving at Google and Carnegie Mellon University, and there are plans to include autonomous driving into production vehicles by Tesla, Nissan, Mercedes, and BMW.

The next generation of intelligent vehicles will be the ones that can communicate with other vehicles and execute joint maneuvers. Vehicles are evolving from autonomous vehicles that sense their environments and control their operation to vehicles that communicate and coordinate their maneuvers. For instance, vehicle platoons are being investigated in the Grand Cooperative Driving Challenges. Before allowing these cooperative vehicles on public highways, we must guarantee that they will not cause accidents, as any flaw in the system may claim human lives. We will apply formal methods to ensure the degree of safety that cannot be assured with simulation or closed-track testing.

This thesis focuses on the techniques that facilitate the developments and verifications of safe cooperative driving protocols. Specifically, we are taking the techniques that are proven to be useful in the development of communication system, such as layered architecture and formal methods, to the realm of intelligent vehicles. We use these techniques to develop a driver-assisted merge protocol as a proof-of-concept. The merge protocol not only involves cooperation between vehicles, but also includes cooperation between the driver and the vehicle, rather than being fully-automated. Semi-automated merge protocols are crucial steps toward fully automated protocols. The public should be convinced that these applications improve safety before they are asked to give up control of the vehicle.

Simulation and test tracks are the prevalent techniques that are used to study the performance of the systems and to verify that such systems are safe. However, the tolerance for failure in automobiles is much lower than can be guaranteed by these techniques. For example, recently, faulty ignition switches on several models of vehicles manufactured by General Motors resulted in a massive recall of vehicles. According to the article [Dye, 2014], there were 52 crashes that were known to be caused by the faulty switch; at least 2.6 million vehicles were recalled in the year of 2014. Supposing that each vehicle with a faulty switch spent at least 1,000 hours on public roadway from the time it left the auto plant, there would be 52 switch-related crashes in $(2.6 \times 10^8 \times 10^3)$ hours of driving, or approximately 2 crashes every 10^8 hours of driving. A failure this rare would likely go undetected by simulation and closed-track testing, whereas formal methods are proven more comprehensive.

Before we apply formal methods to prove the safety of cooperative driving systems, the following challenges need to be addressed:

- Automotive applications interact with the physical world in different ways, including sensing the surroundings of a vehicle, controlling vehicle movements, exchanging messages with other vehicles by tapping the wireless medium, and so forth. Each interaction with the physical world is susceptible to various types of failures and errors. For instance, transmitted messages may not be received in a timely manner and may be lost due to interference or sensors may measure distances inaccurately.

- Cooperative driving systems are time-critical. Vehicles travel on highways with velocities up to 70mph. All maneuvers are required to be completed within a reasonably short time slot, as the conditions on highways are constantly changing. Given that we are designing a driver-assisted merging application, the responses from drivers also affect the execution of maneuvers, since drivers may fail to respond to advisories provided by the applications.

- The complexity of model checking increases exponentially in the number of participants in the system. This complexity comes from the number of states in the composite machine, which increases exponentially with the number of participants. Although the driver-assisted merging involves only three vehicles, namely the vehicle changing lanes and the two vehicles in the adjacent lane where the target gap is located, other vehicles may interfere with the merge. On a reasonably loaded highway, this is often the case. This exponential increase in the complexity of verification is known as state explosion problem, which is a major obstacle that prevents application of model checking techniques.

This thesis introduces the techniques that address these challenges and culminates in the application of formal methods to verification of the driver-assisted merge protocol. To give an overview of the contributions of this thesis, in this chapter we first describe the successful experience of layered architecture in the field of communications (Section 1.1), introduce the accurate clocks obtained from Global Positioning System (GPS) signal (Section 1.2), and provide the background of verification of state transition systems (Section 1.3). We outline the contributions of this thesis in Section 1.4.

1.1 Driver-assisted Merge Protocol

One of the main objectives of this thesis is to design a safe driver-assisted merging application that is robust against various undesirable event is the main objective of this thesis. In this section, we describe the expected behavior of the merge protocol. We give an overview of the enabling technologies, as the available technologies influence the design of the architecture and the merge protocol itself.

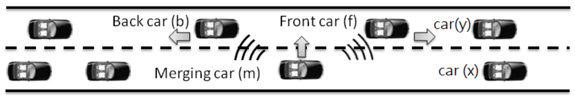


Figure 1.1: Driver-assisted merging

The envisioned driver-assisted merge protocol should assist a driver who attempts to merge between two vehicles in the adjacent lane, as shown in Fig.1.1. The merge protocol is semi-automated in the sense that it controls the longitudinal movement of the vehicle while leaving the steering control to its driver. The driver first signals his/her intention to switch into the adjacent lanes by switching on the turn signal, at which point the merge protocol is initiated. The merge protocol attempts to request for cooperation of the two vehicles surrounding the target gap. On agreeing to cooperate, the two vehicles adjust their velocities and accelerations to gradually create a gap between them. At the same time, the vehicle in which the driver issues a merge request also adjusts its velocity and acceleration so as to align with the gap being created. When the sensors on each of the three vehicles agree that the merging vehicle is fully aligned with the gap, the driver on the merging vehicle is notified, for instance, by a flashing indicator on the dashboard and/or a buzzing prompt. The driver then steers into the newly created gap and concludes the merge maneuver.

As the group of vehicles execute the merge maneuver on public highways, a wide range of undesirable events may happen, including emergency braking of the vehicle in front, an unexpected obstruction on the road, interfering third-party vehicles, and component failures such as sensor inaccuracies, message loss, communication failure, etc. When an undesirable event occurs, the merge should be aborted safely and the driver notified. The vehicle either returns the control to the driver or enters autonomous driving mode that avoids collision.

To realize the whole merge maneuver, the intelligent vehicle equipped with the merge protocol should be capable of the following tasks:

Controlling the speed and distance in relation with the preceding vehicle —- The development of Adaptive Cruise Control (ACC) [Ioannou and Chien, 1993] combines electronic control of the brakes and throttle and affordable radar technology. When given a desired headway, the system is able to control the brake and throttle accordingly to adjust headway then maintain a fixed distance from the preceding vehicle. At the time of this writing, a successful experiment [Milanés *et al.*, 2014] has been done to combine commercial ACC system with communication systems to create a Cooperative Adaptive Cruise Control (CACC) that allows a platoon of four identical vehicles to operate at a speed of 25 m/s, or 90 km/h. This is achieved by having a high-level controller issuing speed and headway settings to the commercial ACC system.

Sensing the surroundings of the vehicle —- The merge protocol is responsible for creating a gap and aligning the merging car with the created gap. This requires coordination of sensors on all three vehicles. The two neighboring vehicles in the target lane exchange sensor readings so that they can instruct their respective ACC to adjust speed and create a gap. The merging car itself should sense its surrounding environment and also receive sensor readings from the vehicles in the target lane so as to deduce the whereabouts of the gap. A wide range of techniques have been developed to merge a set of readings obtained from a single vehicle [Aeberhard *et al.*, 2012; Jo *et al.*, 2012; Li *et al.*, 2014], and also to merge readings from a group of cooperating vehicles [Li and Nashashibi, 2013; Bento *et al.*, 2012; Yuan *et al.*, 2015].

Exchanging information among a group of vehicles —- Cooperative driving, as its name suggests, involves more than one intelligent vehicle. There should be, of course, messages passed among vehicles so as to achieve cooperation. The messages being transmitted could be sensor readings measured by another vehicle at the edge of the group or a request that attempts to initiate a merge maneuver. Vehicular ad hoc networks (VANETs) that operate with no infrastructure are suitable for the merge protocol as well as other cooperative driving applications. There has already been a significant amount of research [Hartenstein and Laberteaux, 2008; Sichitiu and Kihl, 2008] on vehicular communication systems that are suitable for different applications. The most complete work on VANETs is dedicated short-range communication (DSRC), also known as wireless access in vehicular environment (WAVE) [Uzcategui and Acosta-Marum, 2009].

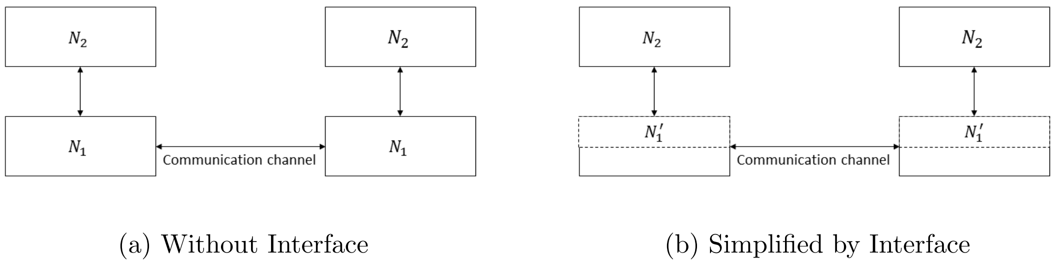
Taking measures to avoid or to alleviate crash —- On aborting the merge maneuver under emergency situations, either the control is returned to the driver or to a system handles both longitudinal control and lateral control. If the merge protocol is being used at low speed, we can hand over control to the traffic jam assistance systems introduced by Daimler [AG, 2015]. It is based on radar and stereo cameras and is designed for automated low speed driving on congested highway operating at 30 km/h. In the near future, this system is expected to operate at higher speeds. If the merge protocol aborts at high speed, the combination of lane departure warning (LDW) systems [Ishida and Gayko, 2004] combined with collision prevention systems [Kodaka *et al.*, 2003; Maurer, 2012] will steer the vehicle back to its original lane and slow it down when necessary. If a collision is unavoidable, these systems alleviate the damage caused by the crash. Evasive steering [Dang *et al.*, 2012], is also currently under development as an assistance to avoid collisions.

Based on the currently available technology, the merge protocol should be suitable for mildly to strongly congested highway, especially the toll plaza. After improving automation, we can see applications expanded in the near future to include merge maneuvers occurring at higher speeds.

1.2 Architecture for Intelligent Vehicles

An architecture is used to decompose a complex problem into smaller, more manageable pieces. Each component of the architecture has a well-defined set of inputs and provides a service in the form of a well-defined set of outputs. In this section, we briefly review the success story of communication architectures. We then review the most complete work by far on the layered architectures for vehicle control and sensor system, developed by PATH and BMW research, respectively.

Architectures in communications networks are used to isolate the physical devices that perform communications from the programs that provide the communication. Communication architectures have evolved to having layered structures that perform more sophisticated communications functions, such as routing or flow control, in different components of an implementation. The layered structure with well-defined interface makes it possible to change parts of more complex communication functions without changing others. It also makes it possible for an application programmer to implement functions without being familiar with the implementation details of how the messages are being exchanged between computers.



(a) Without Interface (b) Simplified by Interface

Figure 1.2: Service models used in verification of acknowledgment protocol

When it comes to verification, with the layered structure, each module can be verified independently by proving that each layer provides the service to the next higher layer, given that it receives the proper service from the lower layer. This is especially useful when there are more than one, say k, systems collaborating. Suppose on each system there are two modules, each contains N_1 and N_2 states, as shown in Fig.1.2a. Without the interface provided by the architecture, the composition of k systems may contain at most $(N_1 \times N_2)^k$ states. With the interface, we can replace the first module with an interface that often has much less states than the module itself. If an interface of N_1' states is obtained, like the one in Fig.1.2b, then the state space can be reduced to the one that has at most $(N_1' \times N_2)^k$ states, achieving a reduction of $(N_1/N_1')^k$.

1.2.1 Platoon control architecture

A layered architecture is used in the automated highway system (AHS). The system was developed at the University of California Partners for the Advanced Transit and Highways program (PATH) [PATH,]. The architecture controls a highway network in which automated vehicles are organized into platoons. Upon entering the highway, the driver hands control over to the automated system. The layered architecture allows design and verification to be done at distinct layers.

The architecture in Fig.1.3 consists of five layers. Each layer is self-contained and uses a model that is appropriate for the layer. The layers have interfaces that specify the data that are exchanged between adjacent layers. Starting from the bottom, the layers are called the physical layer, regulation layer, coordination layer, link layer, and network layer. The implementation of the bottom three layers resides in the vehicles, while the link layer and the network layer are located in the infrastructure.

The physical layer includes the onboard vehicle controllers of the automated vehicles, including the engine and transmission, brakes, steering control, and the various sensors. It decouples the longitudinal and lateral vehicle guidance control, which simplifies the design of the regulation layer.

The regulation layer executes the maneuvers ordered by the coordination layer when a vehicle is a platoon leader or a free vehicle. When the vehicle is a follower within a platoon, the regulation layer maintains a predetermined spacing from the preceding vehicle.

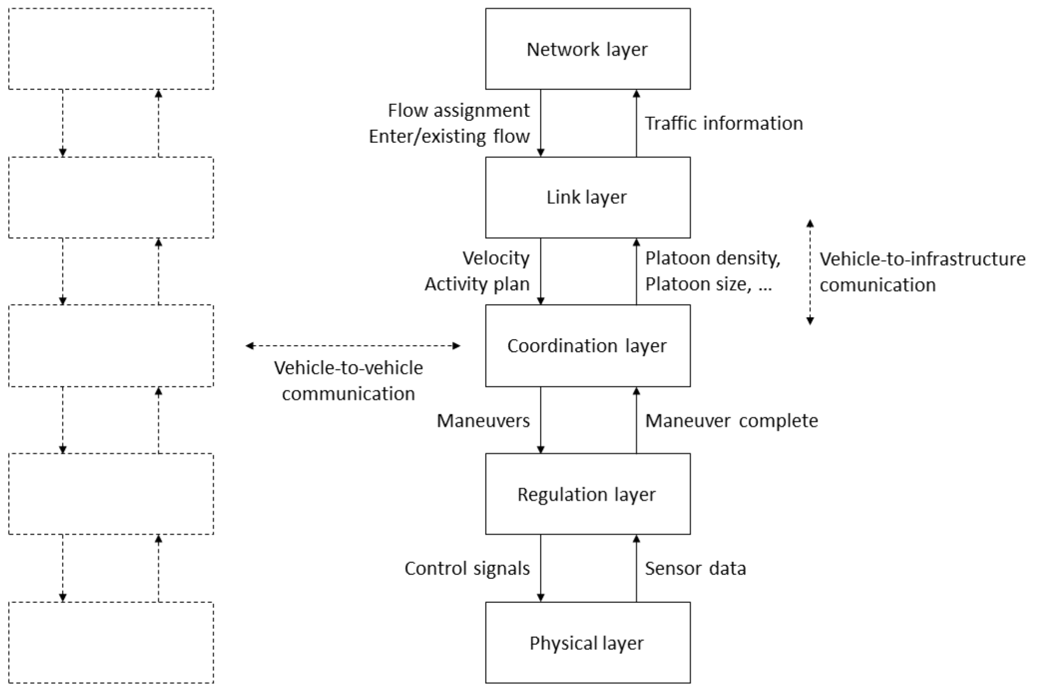


Figure 1.3: The layered architecture in the PATH project

The tasks implemented in this layer include vehicle following [Ioannou *et al.*, 1993], platoon maneuvers [Alvarez and Horowitz, 1999a; Alvarez and Horowitz, 1999b], and highway entry/exit [Godbole *et al.*, 1998].

The coordination layer determines the maneuvers that the vehicle should execute to achieve an active plan that is assigned by the link layer. The coordination layer communicates with the coordination layers in the neighboring vehicles and supervises the regulation layer.

The link layer controls the traffic flow on segments of the highway that are 0.5-5 km long. The link layer receives commands from the network layer in the form of flow assignments for the highway segment and determines the activity plan that achieves the flow assignments.

Finally, the network layer controls the traffic entering the highway and plans routes and flows to maximize the capacity or minimize the average vehicle travel time. Its objective is to reduce congestion.

Each layer is designed and verified independently and is modeled by an appropriate mathematical technique. For instance, the coordination layer is modeled as a discrete event dynamical system [Varaiya, 1993] while the regulation layer is modeled by feedback laws [Godbole *et al.*, 1998; Swaroop *et al.*, 1994]. Similar to the communication architecture, each layer is verified by proving that it provides the service to the layer above it, given that it receives the proper service from the layer below. By proceeding in a bottom-up fashion, eventually we can verify the entire system.

1.2.2 Sensor System Architecture

The layered sensor architecture in Fig.1.4 is proposed in a BMW research project on automated driver assistance systems [Aeberhard *et al.*, 2012]. The sensor system detects surrounding traffic and obstacles and creates a map of the environment. The layered architecture facilitates the design of a track-to-track fusion algorithm, allowing components to be upgraded without reimplementing the entire system.

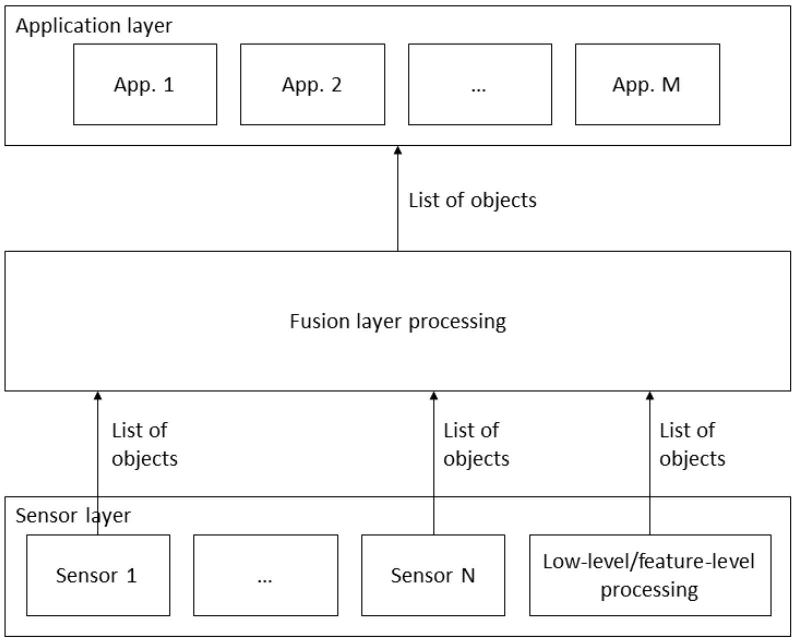


Figure 1.4: The layered sensor architecture in the BMW project

The architecture has three layers: the sensor layer, sensor fusion layer, and application layer. Each layer of the architecture produces a list of objects. Each object in the list has a state vector (which may include information on position, velocity, and acceleration), a state covariance matrix, the object existence probability, and a classification of the object.

The sensor layer processes data from each sensor and outputs a list of objects. There can be low-level or a feature-level preprocessing at this level. The fusion layer joins the data from each sensor and produces a single list of objects. First, object lists from each sensor are temporally and spatially aligned, then the objects are associated and fused together. In the application layer, there can be different driving applications that use the object list. Each application can decide to consider a subset of objects in the list.

The outputs from different sensors may be received asynchronously and may be subjected to variable delays. In addition, different sensors are likely to have different fields of view that may overlap. The fusion layer joins data from different sensors into a single map, allowing applications to be engineered without considering the properties of the individual sensors.

1.3 Coordinate Timely Actions

Incorporating time into communication protocols has been investigated extensively [Alur, 1999; Banerjea *et al.*, 1996; Fecko *et al.*, 2003; Huang *et al.*, 1996; Lin *et al.*, 1989; Nagano *et al.*, 1996; Chu and Liu, 1988; Sinha and Suri, 1999; Yannakakis and Lee, 1993]. All of the approaches use timers rather than synchronized clocks. Most determine the possible sequences that occur when timers are set in different orders among different participants.

Different sequences of timers are particularly troublesome when timers are set by a message that is transmitted over an unreliable communication channel. If two participants in a protocol set identical timers when they receive a message from a third participant, and the first recipient receives the message on the first transmission attempt but the second participant requires two transmission attempts, the first participant's timer will time out first. However, if the second participant receives the message on the first attempt, and the first participant receives the message on the second attempt, then the second participant's timer will time out first.

Our objective is to simplify timed protocols by using synchronized clocks to plan events that occur simultaneously in all of the participants. Simultaneous events reduce the number of sequences that we must consider when verifying the protocol.

A major difference between protocols that are written today and those that were written as recent as 5 or 10 years ago is the availability of inexpensive, accurate clocks. Commercial GPS not only provides positioning but also highly accurate clocks that are accurate within tens of nanoseconds [of the Secretary of Defense, 2008]. This is particularly true for automotive systems since GPS devices have become ubiquitours. Accurate crystal oscillators can maintain a clock when a GPS signal is not available for short periods of time. A short but incomplete list of applications that employ timing information obtained from GPS clocks includes the Google Spanner database [Corbett *et al.*, 2013], which depends on time-accuracy to ensure that the latest data is committed to the database, a synchronized beaconing scheme for wireless ethernet [Papadimitratos *et al.*, 2009; Scopigno and Cozzetti, 2009], and the Large Hadron Collider (LHC) [Brun *et al.*, 2003] in Switzerland.

In some circumstances, such as driving through roofed roadway like tunnel or driving in downtown district of cities where skyscrapers cause canyon effect [Misra and Enge, 2006]. In replacement, when GPS is not available for an extended period of time, the IEEE 1588-2008 standard for Precision Time Protocol (PTP) [Cooklev *et al.*, 2007; Abubakari and Sastry, 2008; Mahmood and Gaderer, 2009; Eidson and Lee, 2003] synchronizes clocks in nearby vehicles over wireless communication. These techniques provide clocks that are accurate to microseconds, which is sufficient in most cooperative driving applications. The advantage to using GPS clocks when they are available is that there is no delay when synchronizing the clocks with adjacent vehicles.

1.4 Verification of the Protocol Systems

The aim of formal methods is to establish system correctness with mathematical rigor. Formal methods are gaining traction in verification of software development of safety-critical systems. They have been actively researched, and powerful software tools have been developed such that various verification steps of software systems can be automated in a push-button fashion. They have proven to be successful in guaranteeing the robustness of systems in various fields. This list includes but is not limited to Mars Pathfinder [Jones, 1997], file system [Yang *et al.*, 2006], and several communication protocols [Edelkamp *et al.*, 2004; Musuvathi *et al.*, 2004; D'Argenio *et al.*, 1997], etc.

Model-based verification techniques are based on models describing the possible system behavior in a precise and unambiguous manner. It often turns out that, during the initial stages of model construction, accurate modeling of system reveals incompleteness, ambiguities, and inconsistencies of informal system specifications. The system models are accompanied by algorithms that systematically and automatically explore states of the system model.

Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether the property holds or not. The techniques that belong to the type of exhaustive exploration examines all possible states of the system in a brute-force manner, while there are other types of techniques explore a selected set of scenarios in the model that are representative for system behaviors in general. Finally, if the system fail to satisfy the property, a counterexample trace is produced so that the users may inspect the cause of error.

The process of model checking consists of three phases:

Model construction —- The inputs to model checking algorithms are a model of the system and a formal description of the property to be checked. Models of system describe the behavior of the system in an accurate and unambiguous manner. Possible modeling choices include finite-state automata (FA) [Baier and Katoen, 2008], timed automata (TA) [Alur, 1999], and Markov decision processes (MDPs) [Baier and Katoen, 2008], etc. These models are comprised of a finite set of states and a set of transitions.

Properties to be checked are also described in an accurate and unambiguous manner. Typical property specification languages are temporal logic [Pnueli, 1977] and its various extensions. Temporal logic allows specification of system behaviors over time and allows a wide range of properties to be specified, including functional correctness, reachability, safety, liveness, etc. The extension of temporal logic includes linear-temporal logics (LTL) [Gerth *et al.*, 1995] and computational tree logic (CTL) [Clarke and Emerson, 1982] as well as their timed and probabilistic variations, timed computation tree logic (TCTL) [Alur *et al.*, 1990] and probabilistic computation tree logic (PCTL) [Hansson and Jonsson, 1994], for specification of properties with timing constraints and probabilistic constraints, respectively.

In this thesis, the modeling choices we make are finite-state machines (FSMs) (extensions of finite-state automata that are suitable for modeling implementation with inputs and outputs interface) and a variation of MDPs. The properties we would like to check the merge protocol against are functional correctness (Can the protocol complete the merge?), reachability (Is there a deadlock?), and safety (Does protocol crash when undesirable events occur?). These properties belong to regular safety properties, which are a subset of the properties that can be represented by LTL.

Running the model checker —- There is a wide range of model checkers that are appropriate for different modeling choices: SPIN model checker [Holzmann, 1997] checks systems modeled by finite-state automata, UPPAAL [Larsen *et al.*, 1997] checks systems modeled by timed automata, and PRISM model checker [Hinton *et al.*, 2006] checks systems modeled by discrete-time and continuous-time Markov chains and Markov decision processes. These tools sometimes do not take the models directly. Rather, the inputs to these tools are described using their respective model description language. For instance, SPIN accepts input written in Promela [Holzmann, 2007], to which finite-state automata can be translated, while PRISM accepts reactive models [Alur and Henzinger, 1999], which is suitable for defining product of several Markov decision processes.

The users supply the model checkers with the system models written in appropriate model description language and the property to be checked. The model checkers then carry out the task of actual model checking automatically.

Result analysis —- Three outcomes may be produced when the model checkers terminate:

1. The property being checked holds in the given model

2. The property being checked is violated and a counterexample trace is produced

3. The model checker runs out of memory because the constructed model is too large

If a property holds, then we call it a day. If a property is violated, we study the counterexample trace and determine the cause. There can be three possible causes. First, it may be due to a modeling error, in which case the constructed model does not faithfully represent the design of the system. The users should revise the model and restart the model checker after the revision is done. Secondly, it can be property error where the property that users have described in the property specification language does not precisely represent the requirement one wished to checked. The users should alter the property accordingly then restart the model checker. Finally, it can be a design error. This type of error is the one we wish to discover before the implementation of systems takes place. It reveals the flaw in system design and prompts the designed to improve or possibly redesign the system.

A major drawback that plagues model checking is the problem of state explosion. The number of state in a protocol system can be as many as the product of the number of states of all its constituting components. That is, the state space of a system grows exponentially in the number of processes or components in the systems and in the size of the constituting processes. In this thesis, we develop a directed search over an exceedingly large state space by exploring the more likely states first and reduce the probability of reaching the states that we are not able to examine.

1.5 Summary of Contributions

This thesis focuses on the techniques that facilitate the design and verification of cooperative driving applications. We present a driver-assisted merge protocol to illustrate the usage of these techniques. In this section, we briefly summarize the contributions of this thesis.

1.5.1 A Multiple Stack Architecture for Cooperative Driving Applications

In Chapter 4, we present the multiple stack architecture for design and verification of cooperative driving applications. It combines the best of both worlds. The layered structure within each stack makes it possible for a designer of cooperative driving application to implement functions without being distracted by the details of sensing the surroundings or the low-level controller for adaptive cruise control, as long as the interface and the services provided by the lower layers remain the same. Multiple stacks address different ways of interaction with the physical world: a single sensor stack includes all the functions related to sensing the environments of a single vehicle or a group of vehicle; a single vehicle stack includes the functions for controlling a single vehicle and controlling a group of vehicles.

Having functions arranged in a layered fashion makes compositional verification possible. The modules in each layer are verified independently, proving that each layer provides service to the next higher layer, given that it receives the proper service from the lower layer. The verification of the whole cooperative driving system is thus broken down by the layered structure into smaller problems of verification of single layers, significantly reducing the complexity of verification. It also makes verification possible, as different layers differ in their logical proximity to the physical world and require different modeling strategies and different verification techniques.

The evolution of the multiple stack architecture over time are in [Lin and Maxemchuk, 2012; Lin *et al.*, 2014; Maxemchuk *et al.*, 2015], ordered chronologically.

1.5.2 Protocol Synchronization Based on GPS Clocks

In Chapter 5, we construct the framework for the design and verification of protocol systems that uses GPS clocks for synchronization. We separate the access and control of timing information into a standalone timing stack. The timing stack relieves the protocols of the control of timing by means of timers. From the perspective of FSMs, the timing stack maintains a list of synchronized events for the participants in a protocol. Accessing the same set of synchronized events allows them to execute synchronized actions. Having a group of participants executing synchronized actions based on clocks also reduces, although not completely eliminates, interleaving of paths that we see when timers expire at different times. There are no timers or clocks in any of the other protocols that co-located with the timing stack. The control of timing is replaced by simple messages exchanged amongst FSMs and the timing stack. This adds a small set of input and output messages to the definition of FSMs. Several compositional rules are also added to FSMs to reflect the synchronized actions induced by synchronous clocks.

Under this framework we design a lock protocol that resolves contention for merge requests issued by different vehicles on the highway. The merge protocol uses the lock protocol to create mutually-exclusive cooperating groups and to allocate a period of time for the merge maneuver until a specified deadline. The lock protocol demonstrates the advantage of synchronizing the clocks of the participants. The synchronous clocks ensure that the group persists until the specified deadline and that every participant releases the lock simultaneously. This cannot be done should the timers be used by participants that are physically separated.

The design of the lock protocol is first proposed in [Lin and Maxemchuk, 2014]. The framework for using the GPS clocks for synchronization is currently being prepared for submission.

1.5.3 Stratified Probabilistic Verification

In Chapter 6, we present the stratified probabilistic model checking algorithm for a variation of Markov decision processes, in which probabilistic choices do not have exact probability distribution but rather discretized levels of probability. It copes with state explosion problem differently by exploring part of the state space instead of reducing the problem size. The stratified approach prioritizes the state traversal on those that are more likely to be reached during system execution. Hence, at any point of state traversal, the unexplored states are the ones that are less likely to be reached during system execution than those have been traversed. If the state space is fully traversed before the memory is depleted, then the result it obtains is the same as typical explicit-state model checkers. If otherwise the complete state space cannot fit into the limited memory space and the traversal stops, and we compute an upper-bound for the probability of reaching those unexplored states. The bound answers the query if a probabilistic safety property holds.

The stratified method approach is a reincarnation of probabilistic verification in [Maxemchuk and Sabnani, 1989], but it computes a significantly tighter probabilistic bound than the original algorithm. By the time it stops state traversal, it produces a linear program whose solution is a valid probability upper-bound on the probability for the regular safety property to be violated.

The fact that the technique is developed for probabilistic system without exact probability distribution paves ways for a more flexible approach to piecewise verification of components in the architecture. Although the architecture has already divided the verification problem into smaller parts, a complete traversal of a subproblem is sometimes improbable. At best, a component may be verified to satisfy a probabilistic regular safety property with a reasonably high probability, but it can never be perfect. Stratified method can verify a system that contains partially verified components that provides required service with some inexact probability. We apply compositional methods [Clarke *et al.*, 1989] to verify a collection of components with inexact probability guarantees.

Finally, the stratified method not only applies to the variation of MDPs that contains discretized probability levels but MDPs with standard definitions. Similarly, it categorizes probabilistic choices into different discretization levels for stratified search of state space. Given the exact probability distribution, we can compute a more precise probability bounds for the regular safety property to be valid.

We summarized some of the contributions in [Lin and Maxemchuk, a; Lin and Maxemchuk, b].

1.5.4 A Driver-Assisted Merge Protocol

In Chapter 7, we focus on the design and verification of driver-assisted protocol using the techniques we developed for cooperative driving systems. The protocol is built within the multiple stack architecture constructed earlier. The simple logics of the merge protocol executes more complex execution such as controlling the speed via the simpler interfaces provided by the layered structure. The merge protocol makes use of accurate GPS clocks to keep track of gap creation and driver response. The lock protocol also uses GPS clocks and provides conflict resolution to the merge protocol.

Next, we apply stratified model checking to the whole system, including the merge protocol and all the components that provide services to the merge protocol. We check if the merge protocol assists a driver in completing a merge and aborts and notifies its driver when undesirable events occur, such as third-party vehicles interfering with the merge. For those components that cannot be verified using model-based techniques, we replace them with the finite automata that represent the service which they are expected to provide. Verification of the lock protocol is prone to state explosion, and the stratified method is able to compute a probabilistic bound for the lock protocol to provide its service. Finally, the compositional verification framework allows us to combine all service-providing components, including the lock protocol, and verify the merge protocol to ensure safety with high confidence. We have proven that the system only enters an unexplored state or fail due to component failure less than once every 5×10^{13} protocol invocations. Simulation or having vehicles driving on test tracks clearly cannot achieve this level of confidence.

The contribution presented in Chapter 7 is summarized in [Lin and Maxemchuk, 2015].

Chapter 2

Related Works

In this chapter, we provide a brief overview of the related work. Cooperative driving applications have already been researched actively, as well as their enabling architecture. However, only a small part of them uses formal methods to verify the safety. We first overview the existing architectures for various cooperative driving projects (section 2.1). Next, we review works regarding cooperative lane-change or merge (section 2.2). We then briefly review several models for time-critical systems (section 2.3). Finally, we discuss works that mitigate the problem of state explosion in verifying probabilistic system (section 2.4).

2.1 Architectures for cooperative driving systems

We survey the architectures that are being used for intelligent vehicles. Most of the architectures are modular. They consist of a number of functional boxes that are interconnected. They include: architectures that are used for cooperative cruise control systems in the 2011 Grand Cooperative Driving Challenge and in a European Commission project; two of the architectures [Caveney, 2010; HAVEit,] have a modular structure with some of the modules organized into layers.

Modular architecture is an intuitive way to organize functions so as to separate different technologies such as wireless communications, control systems, and sensors. The modules are interconnected. They pass data between one another and provide services to each other. Depending upon how the modules are interconnected, it may or may not be possible to engineer and test the modules independently. The modular functions include: wireless communication, positioning, environmental sensing, vehicle controllers, coordination controllers, human-machine interfaces. The modular architectures may emphasize the data flow between modules or the inter-dependency between modules. The modules may be organized into groups that reflect the hardware implementation.

Cooperative adaptive cruise control systems (CACCs), require communications between cooperating vehicles and the coordination of maneuvers. They are used to create platoons of vehicles that travel together on a highway.

Most cooperative driving systems assume that peer-communicating vehicles use the same control strategy and identical systems. The 2011 Grand Cooperative Driving Challenge (GCDC) [van Nunen *et al.*, 2012] is the first competition to simulate a realistic heterogeneous environment, in which passenger cars, vans, trucks, and buses are included in the same platoon, and operate on roads shared by manually driven cars. There were nine teams with different implementations of CACC systems. The architectures in these systems are implementation-specific [Nieuwenhuijze *et al.*, 2012; Mårtensson *et al.*, 2012; Guvenc *et al.*, 2012; Geiger *et al.*, 2012; Kianfar *et al.*, 2012; Lidström *et al.*, 2012]. They are tailored to reflect the design of each implementation. However, the functions of the modules are similar to the previous list. The connections between modules can either reflect data flow between functions [Geiger *et al.*, 2012; Lidström *et al.*, 2012], represent inter-dependency between modules [Kianfar *et al.*, 2012], or simply describe the connection of hardware components [Guvenc *et al.*, 2012; Mårtensson *et al.*, 2012]. Some of the modules are grouped together based on the hardware on which they are implemented [Geiger *et al.*, 2012; Guvenc *et al.*, 2012; Kianfar *et al.*, 2012; Lidström *et al.*, 2012]. Another European Commission funded project, SARTRE [SARTRE,], is also developing strategies and technologies to enable platoons of vehicles to operate on public highways. It is also developed on the basis of a modular architecture.

In the architecture proposed in [Caveney, 2010], some of the modules, or blocks, are laid out in layers which are determined by the different time constants and rates of communication that are required by the modules.

HAVEit [HAVEit,], funded by European Framework Programme 7, is developing intelligent vehicles that switch between semi-automated driving and fully automated driving, based on the driving situation. Its perception module is further subdivided into a data fusion function that combines sensor data from onboard sensors and the information gathered via inter-vehicle communication. The layers in the perception module are similar to the layered architecture for sensor systems that will be described in the next section.

Besides the PATH project, there are similar layered architecture for platoon control proposed in the Japanese Dolphin project [Tsugawa *et al.*, 2000] and the Auto21 collaborative driving system (CDS) project [Hallé *et al.*, 2004]. The projects SASPENCE (SAfe SPEed and safe distaNCE) [Bertolazzi *et al.*, 2010] and INSAFES (Integrated Safety Systems) [Amditis *et al.*, 2010], are part of the project PReVENT under the European Framework Programme 6. Both use a layered structure in their sensor system design. The perception layer provides the same world view to different applications in order to eliminate conflicts between applications. It should be noted that the notion of a single global object list, world model, or environmental mapping is becoming widely accepted [Amditis *et al.*, 2010; Broggi *et al.*, 2013; Behere *et al.*, 2013].

2.2 Highway Lane Change / Merge Solutions

Cooperative driving has become an active research area since the proposal of Automated Highway System (AHS) in 1980s, in which vehicles are fully-automated and driven in platoons. There are independent projects [PATH, ; Tsugawa *et al.*, 2000; Hallé *et al.*, 2004] that attempt to realize the concept of AHS. Among them, PATH has adopted formal approach to verify the safety of its coordination protocol. The authors of [Varaiya, 1993; Horowitz and Varaiya, 2000] have separated the coordination of a platoon of vehicles from individual vehicle control. Three protocols, namely merge platoon, split platoon, and lane change protocols, coordinate vehicle maneuvers in platoon [Hsu *et al.*, 1991], namely , each with FSM specification. Two platoon leaders use platoon merge protocol to combine two platoons into one; The platoon leader and one of the platoon follower uses platoon split protocol to split a platoon into two; a free agent vehicle uses the lane change protocol to communicate with the vehicle in the target lane and vehicles that can potentially move into the target lane so as to steer into the target lane safely. The three protocols are by far the earliest cooperative driving protocols that have been formally verified using formal language COSPAN [Hardin *et al.*, 1996].

The system in [Michaud *et al.*, 2006] has procedures that manage failure experienced by the leading vehicle of a platoon. The authors discuss several scenario that involves different levels of communication between pairs of vehicles in the system. The system adopts a two-level architecture. Behavioral level consists of a set of behavior-producing modules (BPM) control vehicle's actuators. For instance, there are a BPM that is used for collision avoidance and another BPM that is used when driving in the platoon. Which BPM to be active is determined by an FSM in the recommendation level according to the coordination scenarios. Experiments with robots are conducted to evaluate each of the scenario as proof-of-concept, while proving that the strategies to be fail-safe is not the focus of this work.

Prior to this work, there is an attempt to create a driver-assisted merge protocol of semi-automated vehicle [Kim and Maxemchuk, 2012]. It considers a set of undesirable events that may disturb the merge maneuvers. It employs an architecture composed of several components to simplify the design. Timers are explicitly used in the protocol as human drivers are more unpredictable. The authors use probabilistic verification [Maxemchuk and Sabnani, 1989] to verify the protocol.

There are approaches for platoons to respond to the interference from human drivers. The platoons in [Guo *et al.*, 2012] are able to change formation to prevent the interfering driver from splitting the platoon. The platoons in [Lam and Katupitiya, 2013] are able to overtake other platoons one vehicle at a time. The vehicles are also able to temporarily join another platoon if a slow vehicle is detected in the overtaking lane. These approaches require sophisticated coordination among multiple vehicles.

Another fairly recent approach [Segata *et al.*, 2014] on platoon maneuver considers the effect of undesirable events on the platoon join protocol. The protocol includes procedures for handling interfering vehicle during the maneuver, slow vehicle that prevents the joining vehicle from completing the maneuver, and message loss. These events are explicitly simulated to evaluate the effectiveness and the safety of the join protocol. However, there is a potential failure that is not part of the simulation scenario. During the maneuver, an interfering vehicle occupying the gap causes the maneuver to be temporarily suspended. If the interfering vehicle moves out of the gap before a timer times out, the protocol resumes the maneuver. If the same vehicle suddenly moves back into the gap at the same time, the behavior of the join protocol is undefined. While simulation is useful to demonstrate the feasibility of the protocol, it has its limitation to evaluate the safety of a system.

2.3 Modeling Real-Time Systems

Timed automata [Alur and Dill, 1994; Alur, 1999] can be seen as finite-state machines equipped with a set of clock variables that measure the time elapsed between events. A timed automaton models the behavior of a single process or module of the system. Invariants and clock guards are boolean expression. Invariants are placed on locations, which are the TA equivalent states, which restrict the values of the clock variables for the automaton to stay in the location. Guards are placed on transitions, enabling the transitions to be taken when the clock variables satisfy their boolean expression evaluate to true. A real-time system made up of multiple communicating processes is modeled as the parallel composition of timed automata.

The correctness properties can be specified in TCTL. There are several tools, including UPPAAL [Larsen *et al.*, 1997] and KRONOS [Yovine, 1997], implement model checking algorithm that checks timed automata. A system modeled in timed automata can be represented as the modeling languages of UPPAAL or KRONOS. In addition, their modeling language provide syntax for modeling cooperation and communications between processes. UPPAAL model is slightly more expressive than timed automata, as it comes with more general types of data variables such as boolean and integer variables.

Besides timed automata, there are other well-established timed models. They include TCCS [Yi, 1991], real-time Maude [Ölveczky and Meseguer, 2002], Timed Rebeca [Reynisson *et al.*, 2014], etc. These timed models are most useful when one would like to check if the system satisfy some property within timing constraints or to find the set of timing constraints for the system to satisfy timed property.

These timed models introduce a discrete or dense time domain. This additional dimension further increases the already high complexity of model checking. In chapter 5, we show that it is possible to separate the timing from the logic of state transition, thereby avoiding the more expressive yet complex timed models.

2.4 Tackling State Explosion

A rich set of methods has been proposed to overcome state explosion and to improve the scalability of model checking techniques. There are techniques that aim at reducing the size of the state space. These techniques include symbolic model checking burch1990symbolic, partial-order reduction methods [Godefroid *et al.*, 1996; Katz and Peled, 1992], symmetry reduction [Sistla and Godefroid, 2004], and bounded model checking [Clarke *et al.*, 2001], to name a few. A survey of these techniques can be found in [Clarke and Wang, 2014].

Another direction of reducing the state space is the compositional approach [Clarke *et al.*, 1989]. A system is divided into several processes to be individually verified. The environment of a process P_A is modeled by another process called an interface process P_I. The interface process is usually simpler than the full environment of P_A, which often contains several processes that interact with P_A. In [Kwiatkowska *et al.*, 2010], an assume-guarantee verification technique is developed for probabilistic system, which serves as the foundation of our approach to verify cooperative driving system within the multiple stack architecture.

There is parallel set of techniques that attack the state explosion problem differently: exploring some of the reachable states in the system. They are often based on the technique of *random sampling* of executing a random walk over the system model. They are first introduced in [West, 1989; Rudin, 1992], and they are then extended and formalized for deciding if an LTL property holds with some probability [Hérault *et al.*, 2004; Grosu and Smolka, 2005]. These methods are efficient in terms of both time and space consumption, however, the results they produce come with a confidence interval.

The stratified verification method in Chapter 6 is more a directed search rather than a randomized search on probabilistic state transition system with exceedingly large state space. The method in [Sankaranarayanan *et al.*, 2013] adopts directed search approach, although it is developed for checking computer programs with real-valued variables rather than for state transition systems. The analysis is done in two steps: first, it chooses an *adequate set of paths* within the system; next, for each path chosen, the path probability is estimated by performing a symbolic execution. The probability of each path is then aggregated to form a final probability bound on the assertion. A drawback of the approach is that it does not handle nondeterminsitic uncertainties or distribution.

Chapter 3

Preliminaries

In this chapter, we study the models of finite-state machines (FSMs) and Markov decision processes (MDPs) which will be used extensively in Chapter 5, Chapter 6, and Chapter 7. The verification of protocol systems amounts to model the protocol system as Markov decision process and to verify that the model satisfies probabilistic safety properties. In this thesis, we create two protocols, namely the lock protocol for conflict resolution and the merge protocol for driver-assisted merging as finite-state machines. The stratified probabilistic verification introduced in Chapter 6 is a model checking technique for verifying if a Markov decision process or its variation satisfies probabilistic safety properties.

A protocol system consists of one or more finite-state machines and the models that describe the services and the environment that the finite-state machines rely on and interact with. Services and environment contribute uncertainty to the protocol systems. In this thesis, they are modeled as a variation of probabilistic I/O automata that are compatible with finite-state machines. They are referred to as *service models.* We describe the composition rules of finite-state machines and services models. The result of composition is a Markov decision process that models the protocol system. Markov decision processes are suitable model for distributed system with concurrent and probabilistic behaviors. We check if a Markov decision process satisfies a probabilistic safety property by means of reachability test and solving a linear program. The reachability test finds the set of states that violate the regular safety property component of probabilistic safety property, and the solution to the linear program is the probability of reaching that set of states. The linear program constructed is polynomial in the size of the Markov decision process, whose size is exponential in the size of its constituting finite-state machines and service models.

In this chapter, we first present the finite-state machines and the service models (Section 3.1). Next, we present the standard definition of Markov decision processes and probabilistic safety properties, and we describe how to check if a Markov decision process satisfied a given probabilistic safety property. Then, we present the rules for constructing a Markov decision process from finite-state machines and the service models (Section 3.3), Finally, we present the verification of alternating-bit protocol (ABP) as an example (Section 3.5).

3.1 Constituting Components of a Protocol System

A cooperative driving system consists of various communicating modules. Some of the modules describe a protocol or a program and are deterministic. The others are the supporting functions of the protocol such as adaptive cruise control or sensor systems, and they serve as a way for a protocol or a program to interact with the physical world indirectly.

In this section, we present the definition of finite-state machines (FSMs) and the service models, which is a adaptation of probabilistic I/O automata [Wu *et al.*, 1997] that is compatible to the FSMs. FSMs are deterministic and driven purely by external input. It provides a way to specify a protocol in a formal and specific way, as compared to literal description. Service models have probabilistic behaviors. An input to a service models may lead to different state changes, and they create stimuli that drive FSMs.

3.1.1 Finite-State Machines

The model of FSMs is a variation of finite-state automata but is more suitable for protocol specification as it specifies the input and output behaviors of an implementation of the protocol. The input and output semantics makes FSM a viable target for conformance testing [Sabnani and Dahbura, 1988; Aho *et al.*, 1991; Linn Jr and Uyar, 2013]. In this section, we give its definition. To reason about systems that include multiple processes, we consider the composition of multiple FSMs. Each deterministic process in the system is modeled as a single finite-state machine. The *state* of a process being modeled is defined as a stable condition in which the process rests until a stimulus, i.e. an *input*, is received. An input triggers the module to generate an *output* and to transit to a new state, where it rests and waits for the next input.The input and output behavior of the processes are described by the syntax of communicating sequential processes (CSPs).

Definition 1 *A finite-state machine is a 4-tuple $F = (S, I, O, T)$, where*

- *S is a finite set of states; $s_0 \in S$ is the initial state, which is the state F is in immediately after being powered-up.*
- *I is the set of all permissible inputs that cause FSM F to execute a transition.*
- *O is the set of all permissible outputs that the machine generates on executing a transition. Note that ϕ, the null output, is one of the permissible outputs.*
- *$T : S \times I \cup \phi \rightarrow S \times 2^O$ is the set of edges in the FSM that move the FSM from one state to another. It can also be an edge that departs from and returns to the same state, which is called self-loop. These edges are transitions in the FSM.*

An input to FSMs has the form F_1?msg, which stands for receiving msg from another FSM called F_1.; an output of the FSM has the form of F_2!msg, which represents that the machine sending msg to another FSM with the name F_2. A transition is typically triggered by an input and it may generate zero, one, or several outputs on the transition. That is, a transition can be of the form F_x?msg0 / F_{y_1}!msg1, F_{y_2}!msg2, $\ldots$, F_{y_k}!msgk or F_x?msg0 / ϕ. Multiple outputs on the same transition are assumed to be *atomic*, in other word the outputs are expected to be dispatched to their respective destination without interleaving of other events.

A specification of an FSM is said to be *fully specified* if, for every state, every input in its permissible input set I generates an output in the set O. Otherwise, the specification is said to be *partially specified*. In a partially specified FSM, some of the inputs in set I are not enabled in some of the states. The behavior of FSM on such inputs is undefined. A partially specified FSM can be made fully specified if for any input that is not enabled in a state, the FSM simply ignores it (by generating a null output) or transits into an error state. We require that all protocols in this thesis to be fully specified.

FSMs are deterministic. Fixing for any state s, there should not be two transitions sharing the same input labels. Semantically, if an FSM F is in state s, an input message *msg* received from another process Px always cause F to transit to the same target state s' and to generate the same set of output(s).

FSMs are input-driven. An FSM F may only execute a transition, be it a self-loop or a transition to a new state, when triggered by some input.

3.1.2 Models for Services and Environments

We introduce a variation of probabilistic I/O automata that are compatible to the FSMs defined earlier. We refer to them as *service models.* They model the services depended by FSMs and the interaction between FSMs and the environment, especially the external inputs that trigger the FSMs. The change from probabilistic I/O automata to service models is the use of CSP syntax that allow service models to interact with the FSMs.

Probabilistic transitions and transitions that are not triggered by external stimuli make service models different from FSMs. When being composed with FSMs to form the model of protocol systems, these types of transitions give rises to the probabilistic choices and nondeterministic choices in MDPs. The rest of service model looks a lot like finite-state machines in terms of definition.

Definition 2 *A service model is a five-tuple $G = (S, I, O, T, \mu)$, where*

- *S is a finite set of states; $s_0 \in S$ is the initial state, which is the state G is in immediately after being powered-up.*
- *I is the set of all permissible inputs that cause the model G to change states.*
- *O is the set of all permissible outputs that the model generates on executing a transition.*
- *$T \subseteq S \times I \cup \{\phi\} \times 2^O \times S$ is the transition relation in the service model, which is the set of edges that move the service model from one state to another.*
- *$\mu : S \times I \cup \{\phi\} \times 2^O \times S \to [0, 1]$ is the transition probability function. It is required to satisfy the following conditions:*

1. $\mu(q,x,y,r) > 0$ *iff* $(q,x,y,r) \in T$ *and* $x \neq \phi$
2. $\sum_{r \in S, y \in 2^O} \mu(q,x,y,r) = 1, \forall q \in S, \forall x \neq \phi$

An input to service model has the same form as the input to FSMs: F_1?msg, which stands for receiving msg from another FSM called F_1; an output of the service model has the same form as the output of FSMs: F_2!msg, which represents that the model generates and sends msg to FSM with the name F_2.

The following are possible combinations of input x and outputs y for a transition $(q,x,y,r) \in T$:

1. $x \neq \phi$ and $|y| \geq 0$
2. $x = \phi$ and $|y| > 0$
3. $x = \phi$ and $|y| = 0$

The transitions of the first type is similar to the transitions in FSMs: it receives an input from another module and generates zero, one, or several outputs. For each state q, there can be more than one transitions with the same input label. The transition probability function μ describes the probability of choosing one transition with an input label x from state q as opposed to another transition with the same input label x.

Probabilities are not ascribed to transitions of the second type and the third type. Transitions with an output label describes an event that generates a stimulus to the destined processes. The event occurs for some reason that we do not explicitly characterize or is not relevant to the processes with which the service model interacts. Similarly, transitions without a label describes an event that causes a change of state in the service model but does not generate stimuli to any process in the system. A protocol system is modeled by a collection of FSMs and the service models that act as the supporting components or the environments. At any instance, there may be several transitions with output labels and without labels that are available. This leads to nondeterministic choices in MDPs that we will see in the next section.

Fig.3.1 shows a service model of a half-duplex channel. The model may represent a wireless channel with sophisticated contention resolution protocol or just some wired channel.

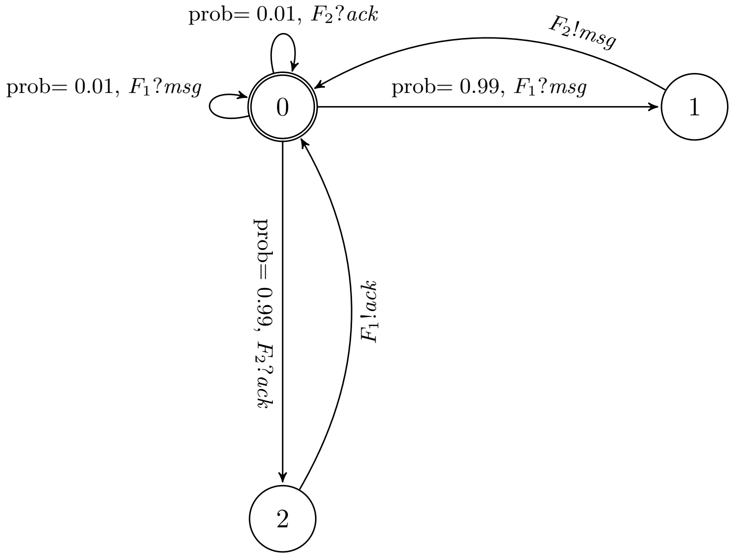


Figure 3.1: Half-duplex channel service model

It exposes a simple interface. It models a transmission of message *msg* through a physical communication channel by accepting *msg* from FSM F_1 and moving to state 1, then, when in state 1, by moving back to state 0 and generating *msg* destined for F_2. Similarly, an acknowledgement message from F_2 to F_1 is represented by a state change from state 0 to state 2 when the channel model receives *ack* from F_2 and completed by another state change from state 2 back to state 1 and generating *ack* destined for F_1 along the way.

The service model is for a communication channel with message loss rate of 0.01. Note that, at state 0, there are two transitions having the same input label $F_1?msg$. The transition probability function assigns the one that goes from state 0 to state 1 with probability 0.99, and it assigns the self-loop with probability 0.01. The transition from 0 to 1 leads to the eventual delivery of message, which is more likely, while the channel drops the message in the self-loop. The same applies to the two transitions from state 0 with label $F_2?ack$.

3.2 Modeling and Verification of Probabilistic Systems

Cooperative driving systems are subject to various phenomena with stochastic nature, such as sensor failures, erratic driver behaviors, message losses in communication channels, etc. When they operate in an unpredictable environment, guaranteeing absolute satisfaction of properties becomes difficult or even impossible.

In this section, we review the definition of Markov decision processes (MDPs), which are suitable for modeling of cooperative driving systems. In MDPs there are nondeterministic choices and probabilistic choices. As for cooperative driving protocols, the nondeterminism is appropriate to model concurrency between processes and the events whose probabilities are hard to characterized, while probabilistic choices capture events such as module failures and message losses. We then look at probabilistic safety properties, which specify the properties we would like to evaluate for cooperative driving systems. Finally, we review a standard technique that check if an MDP satisfies a given probabilistic safety property that based on depth-first search and solving a linear program in the size polynomial of the MDP.

3.2.1 Markov Decision Processes

Unlike discrete-time Markov Chain (DTMC), in which the next state is always chosen probabilistically, in any state of MDPs, a nondeterministic choice between probability distributions exists. Once a probability distribution has been chosen nondeterministically, the next state is selected according to the chosen probability distribution.

Definition 3 *A Markov decision process is a tuple $M = (S, Act, P, I_{\text{init}})$, where*

- *S is a finite set of states*
- *Act is a set of non-deterministic actions*
- *$P : S \times Act \times S \to [0, 1]$ is the transition probability function satisfying that for all states $s \in S$ and actions $\alpha \in Act$:*

$$\sum_{t \in S} P(s, a, t) = 0 \text{ or } 1 \tag{3.1}$$

- *$I_{\text{init}} : S \to [0, 1]$ is the initial distribution such that*

$$\sum_{s \in S} I_{\text{init}}(s) = 1 \tag{3.2}$$

An action α is said to be *enabled* in state s iff $P(s, a, t) > 0$ for some state $t \in S$, otherwise α is *disabled*; *Enabled*(s) denotes the set of enabled actions in s. The α-successors of s are the set of state $\alpha - succ(s) = \{t | P(s, a, t) > 0\}$, and the corresponding α-transitions from s are the set of transitions $\alpha - trans(s) = \{(s, a, t) | P(s, a, t) > 0\}$. The set $trans(s) = \bigcup_{a \in Act(s)} \alpha\text{-}trans(s)$ are the transitions from state s.

An MDP operates as follows. The starting state is decided according to the initial distribution I_{init}. A transition from state s is divided into two steps: a non-deterministic choice is made by selecting an action α from the set $Enabled(s)$. After action α is selected, one of the states in $\alpha\text{-}succ(s)$ is selected probabilistically according to the distribution $P(s, a, .)$. That is, the next state is t with probability $P(s, a, t)$.

An *adversary* resolves the nondeterministic choices in an MDP, based on the execution history of the MDP. In the literature, adversaries are also known as scheduler, policy, or strategy.

Definition 4 *An adversary σ for MDP M is a function $\sigma : S^+ \times Act \to [0, 1]$ such that $\forall s_0 s_1 \dots s_n \in S^+$*

$$\sigma(s_0 s_1 \dots s_n)(\alpha) = 0, \forall \alpha \notin Enabled(s_n) \tag{3.3}$$

and

$$\sum_{\alpha \in Act(s_n)} \sigma(s_0 s_1 \dots s_n)(\alpha) = 1 \tag{3.4}$$

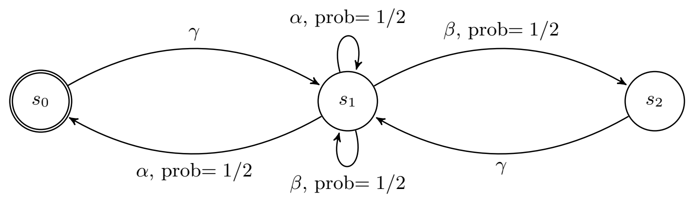
Here $s_0 s_1 \dots s_n$ is the execution history of M. It is the sequence of states traversed, starting from the initial state, during the execution. An adversary $\sigma(s_0 s_1 \dots s_n)$ maps a finite path to a probability distribution over the enabled actions in the last state of the path, sn. In this thesis, we only consider memoryless adversaries.

Definition 5 *Adversary σ is memoryless iff for each sequence $s_0 s_1 \dots s_n$ and $t_0 t_1 \dots t_m \in S+$ with $s_n = t_m$:*

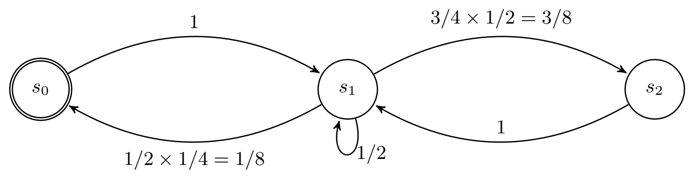
$$\sigma(s_0 s_1 \dots s_n)(\alpha) = \sigma(t_0 t_1 \dots t_m)(\alpha) = \sigma(s_n)(\alpha), \forall \alpha \in Enabled(s_n) \tag{3.5}$$

In other words, in a given state s_n, memoryless adversaries always yield the same probability distribution over the enabled actions, independent of what has happened in the execution history.

The adversary assigns probability weights to the enabled actions when there are non-deterministic choices to be made. It resolves all nondeterministic choices in an MDP and induces a Markov chain. We denote by Adv_M the set of all possible adversaries for MDP M. By $Path_M^\sigma$, we denote the set of all paths through MDP M when controlled by adversary σ. Under an adversary σ, we define a probability space Pr_M^σ over the set of paths $Path_M^\sigma$. The probability space captures the purely probabilistic behavior of M under σ.



(a) Markov decision process



(b) Markov chain induced by the adversary

Figure 3.2: An Adversary Resolves an MDP into a Markov Chain

Fig.3.2 is an example of having an adversary transform an MDP into a Markov chain. Fig.3.2a is a simple MDP with three states. In the initial state s_0 and state s_2, γ is the only enabled action. In state s_1, α and β are enabled. When α is selected, the MDP stays in s_1 and moves to s_0 both with probability 1/2, while when β is selected, the MDP stays in s_1 and moves to s_2 both with probability 1/2. We consider the following memoryless adversary

$$\begin{aligned}
\sigma(s_0)(\gamma) &= 1 \\
\sigma(s_1)(\alpha) &= 1/4 \\
\sigma(s_1)(\beta) &= 3/4 \\
\sigma(s_1)(\gamma) &= 1
\end{aligned} \tag{3.6}$$

The memoryless adversary determines a probability distribution over enabled action only based on the state the MDP is in. In s_0 and s_2, the probability distribution is trivial as there is only one enabled action. In s_1, probability weight of 1/4 is assigned to action α and the rest 3/4 is assigned to action β. Fig.3.2b is the Markov chain induced by this memoryless adversary.

3.2.2 Probabilistic Safety Properties

A probabilistic safety property combines a regular safety property and a rational probability bound. Regular safety properties are a subset of safety properties that can be represented by deterministic finite automata (DFA). Checking a regular safety property P_{safe} on a finite-transition system such as an MDP M can be reduced to reachability analysis on the product of M and the DFA that represented the *bad prefixes* of P_{safe}.

A regular safety property P_{safe} represents a set of infinite words, denoted by $\mathcal{L}(P_{\text{safe}})$, that is characterized by a regular language of bad prefixed, which are finite words of which any extension is not in $\mathcal{L}(P_{\text{safe}})$. Typical regular safety properties in the context of cooperative driving include "the lock protocol indicates successful cooperating group creation only when all participants agree to cooperate" and "if an undesirable event occurs, the driver should not be told to proceed to change lanes". A regular safety property P_{safe} can be defined more precisely by a complete DFA $A^* = (Q, q_0, \alpha_{A^*}, \delta_{A^*}, F_{A^*})$, where

- Q is the set of states
- q_0 is the initial state of A^*
- α_{A^*} is the set of alphabets of A^*
- $\delta_{A^*} : S \times \alpha_{A^*} \to S$ is the transition function
- $F_{A^*} \subseteq S$ is the set of accepting states

The language of the regular safety property is then the set of all words of the alphabet α_{A^*} whose prefix is an accepting run of A^*.

Probabilistic safety properties are similar to regular safety properties but are less absolute. It does not require a properties to hold in 'all' instances but to hold within some probability bound. For instance, "the maximum probability of the lock protocol indicating successful cooperating group creation without all participants agreeing to cooperate is at most 10^{-10}" and "the maximum probability of the driver being told to proceed to merge when an undesirable event is present is at most 10^{-14}" are both probabilistic safety properties.

Definition 6 *A probabilistic safety property $\langle A\rangle_{\geq p}$ constitutes of a regular safety property A and a probability bound p. An MDP M satisfies $\langle A\rangle_{\geq p}$ if the probability of satisfying A is at least p for any adversary σ:*

$$M \models \langle A\rangle_{\geq p} \Leftrightarrow \forall \sigma \in Adv_M \cdot Pr_M^{\sigma}(A) \geq p \Leftrightarrow Pr_M^{\min}(A) \geq p \tag{3.7}$$

The superscript *min* in $Pr_M^{\min}(A)$ means that the minimum is taken over all possible adversaries.

3.2.3 Verification of Probabilistic Safety Properties for MDPs

Verifying if an MDP satisfies a probabilistic safety property consists of three steps. First, we construct the product of the MDP and the DFA that defines the bad prefixes of the probabilistic safety property. Next, a reachability test is conducted on the MDP-DFA product to find the set of states that violate the regular safety property part of the probabilistic safety property. The last step is to solve a linear program whose solution is a probability bound of reaching the states that violate the property. The result is then compared to the probability bound in the probability safety property to determine if the property holds.

The goal of verification is to decide if $Pr_M^{\sigma}(P_{safe}) \geq p$ for all possible adversaries σ of M. Finding all the paths that satisfy P_{safe} and aggregating their probability is often difficult. Checking regular safety properties in deterministic models such as finite-state automata [Baier and Katoen, 2008] is reduced from checking path properties to reachability test by constructing the product of finite-state automata and the DFA for the bad prefixes of the regular safety properties. This technique is adapted to the probabilistic case. This involves constructing the product of MDP and DFA.

Definition 7 *The product of an MDP $M = (S, Act, P, I_{init})$ and DFA $A^* = (Q, q_0, \alpha_{A^*}, \delta_{A^*}, F_{A^*})$ with $\alpha_{A^*} \subseteq Act$ is given by the MDP $M \otimes A^* = (S \times Q, Act, P', I'_{init})$ where:*

- $$P'(\langle s,q\rangle, a, \langle s',q'\rangle)) = \begin{cases} P(s,a,s'), & \text{if } a \in \alpha_{A^*} \wedge a \in \mathit{Enabled}(s) \wedge q' = \delta_{A^*}(q,a) \\ P(s,a,s'), & \text{if } a \notin \alpha_{A^*} \wedge a \in \mathit{Enabled}(s) \wedge q' = q \\ 0, & \mathit{otherwise.} \end{cases}$$

- $$I'_{init}(\langle s,q\rangle) = \begin{cases} I_{init}(s), & \text{if } q = q_0 \\ 0, & \mathit{otherwise.} \end{cases}$$

The product $M \otimes A^*$ unholds M by having the current state of A^* encodes the path fragment taken by M so far. For each path fragment $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \dots$ of M, there is a corresponding run $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \dots$. Of course we still have $q_{i+1} = q_i$ if $a_i \notin \alpha_{A^*}$. We have a path fragment $\pi^+ = \langle s_0, q_0\rangle \xrightarrow{a_0} \langle s_1, q_1\rangle \xrightarrow{a_1} \langle s_2, q_2\rangle \xrightarrow{a_2} \dots$. Therefore, for a path fragment of M that violates P_{safe}, the corresponding run of A^* ends up in its accepting state. The DFA does not affect the probabilities.

Finding the paths that violate the regular safety property P_{safe} in M is then reduced to finding the paths in $M \otimes A^*$ that reach the set of states $T = \{\langle s,q\rangle | q \in F\}$. Using the temporal logic operator 'eventually' $\Diamond$, we let $s \models \Diamond T$ if there exists a path from s to the set T. We have

$$Pr^{\sigma}_{M}(A) = 1 - \sum_{s \in S} \eta_{init}(s) Pr^{\sigma}_{M}(s \models \Diamond T) \tag{3.8}$$

Set T represents the states that the paths end up in if they violate the regular safety property. It can be decided by having a state traversal algorithm starts from set of initial states. The following is a typical state traversal algorithm that is based on depth-first search (DFS).

DFS is applicable to models that can be represented as graph. The product $M \otimes A^*$ is such. It visits every state that is reachable from any of the initial states. It stores all the states that have been examined in the container *Explored*, and the states whose DFA component belongs to F in the container *Target*. Both *Explored* and *Target* can be implemented efficiently as hash tables. Algorithm 1 here takes $M' = M \otimes A^*$ as input parameter.

It starts the subroutine DFS-visit from each initial state (line 2-7). For each state s, it first checks if s is a deadlock or represents an error state (line 10). It then proceeds to

Algorithm 1 Depth-first search

```
1: procedure DFS($M'$)
2:     for all $\langle s, q\rangle \in I$ do
3:         if $\langle s, q\rangle \notin \mathit{Explored}$ then
4:             Insert $s$ into $\mathit{Explored}$
5:             DFS-visit($M', \langle s, q\rangle$)
6:         end if
7:     end for
8: end procedure
9: procedure DFS-visit($M', \langle s, q\rangle$)
10:     if $A(\langle s, q\rangle) = \phi$ then
11:         Report deadlock
12:     end if
13:     if $q \in F$ then
14:         Add $q$ to set $\mathit{Target}$
15:     end if
16:     for all $(\langle s, q\rangle, \alpha, \langle t, r\rangle) \in \mathit{trans}(\langle s, q\rangle)$ do
17:         if $t \notin \mathit{Explored}$ then
18:             Insert $t$ into $\mathit{Explored}$
19:             DFS-visit($M', \langle t, r\rangle$)
20:         end if
21:     end for
22: end procedure
```

examine every transitions in $trans(s)$ (line 16). If a successor state t is not yet traversed, i.e., not stored in *Explored*, DFS-visit is called recursively to continue exploring the successors of t (line 19). By the time the algorithm terminates, all states in M' should be traversed and $\mathit{Target} = T$.

Recall that for a probabilistic safety property to hold, it requires that for all adversaries, the aggregated probability of all paths that satisfy regular safety property P_{safe} to be greater than p. That is, we determine if the aggregated probabiltiy of all paths that satisfy P_{safe} minimized over all adversaries is greater than p. Or, we maximize the aggregated probability of reaching states in T,

$$Pr_M^{\min}(A) = 1 - \sum_{s \in S} I_{\text{init}}(s) Pr_M^{\max}(s \models \Diamond T) \tag{3.9}$$

The maximal probability $Pr_M^{\max}(s \models \Diamond T)$ can be computed by solving a linear program. Specifically, the vector $(x_s)_{s \in S}$ with $x_s = Pr^{\max}(s \models \Diamond T)$ yields the unique solution of the linear program in (3.10).

$$\begin{aligned}
&\mathbf{min} \sum_{s\in S} x_s \\
&\mathbf{s.\ t.}\ x_s = 1,\ \text{If } s \in T \\
&\qquad x_s = 0,\ \text{If there is no path from } s \text{ to } T \\
&\qquad x_s \geq 0,\ \forall s \in S \\
&\qquad x_s \leq 1,\ \forall s \in S \\
&\qquad x_s \geq \sum_{s'\in S} P(s,\alpha,s')x_{s'},\ \text{If } s \notin T \text{ and } \exists \text{ path(s) from } s \text{ to } T,\ \forall \alpha \in \mathit{Enabled}(s)
\end{aligned} \tag{3.10}$$

Simple inspection shows that the number of constraints is roughly the total number of enabled actions in all states: $O(\sum_{s\in S} |\mathit{Enabled}(s)|)$, or polynomial in the size of Markov decision process. The proof that the solution to this linear program is the maximal probability can be found in [Baier and Katoen, 2008].

3.3 From Finite-State Machines and Service Models to Markov Decision Process

In verification, we are interested in the behavior of a protocol system as a whole rather to individual components. To reason about the property of a protocol system, we construct the model of protocol system as a Markov decision process from its constituting components, which are specified in FSMs or modeled as service models. In this section, we present the composition rules that describe the interaction between FSM and FSM or between FSM and service model. The system operate by having its constituting components interact with each other. The composition rules lead to the transitions in the constructed MDP, which stands for the executions of the system being modeled. The number of states in the constructed MDP is exponential in the number of constituting modules.

As a protocol system comprises of the modules that are either modeled as FSMs or service models, a state of the protocol system is a tuple of the states of the constituing FSMs and service models. Suppose $F_1, \ldots, F_n$ are the FSMs and $G_1, \ldots, G_m$ are the service models, a state of the protocol system takes the following form:

$$\langle F_1 = s_1, \ldots, F_n = s_n, G_1 = t_1, \ldots, G_m = t_m \rangle \tag{3.11}$$

where $s_1, \ldots, s_n$ and $t_1, \ldots, t_m$ are the states of FSMs $F_1, \ldots, F_n$ and service models $G_1, \ldots, G_m$, respectively. When the FSMs and the service models are clear from the context, it is shortened to $\langle s_1, \ldots, s_n, t_1, \ldots, t_m \rangle$.

A change in the system's configuration is represented as an edge in the state transition graph. It is also called a transition or a step from one state to another in the state transition graph. A transition is a result of interaction between two modules, be them two FSMs, two service models, or an FSM and a service model. This interaction is called *handshaking*, because two processes interact by both participating in this action at the same time.

The handshaking is suitable for communication between modules that located at the same physical location, allowing them to be connected reliably by a wires. Two modules that are not co-located should not use handshaking. Instead, service models of communication channel should be used to describe their communication.

The following transition rule is a handshaking action between two modules M_i and M_j, with M_i being the sender of the synchronization message *msg* and M_j being the receiver of *msg*:

$$\frac{M_i : s_i \xrightarrow{x/M_j!msg} s'_i \wedge M_j : s_j \xrightarrow{M_i?msg/y} s'_j}{\langle s_1, \ldots, s_i, \ldots, s_j, \ldots, s_n, t_1, \ldots, t_m \rangle \to \langle s_1, \ldots, s'_i, \ldots, s'_j, \ldots, s_n, t_1, \ldots, t_m \rangle} \tag{3.12}$$

, where x is the input that triggers M_i to move from s_i to s'_i and y contains zero, one, or more output that M_j generates on receiving *msg* from M_i when in state s_j.

Recall that in the definition of FSMs and service models, a transition come with zero, one, more several atomic output labels. A state transition of the protocol system is always initiated by a service model executing a transition with no input label but some output labels. The output of the service model triggers a transition $s \xrightarrow{x/y} s'$ at the destination module. If the transition has no output in y, then the state change of the protocol system is completely captured by the rule in (3.12). If otherwise y contains one or more outputs, then the outputs will then trigger transitions in other modules.

Let us first consider that case in which y contains one output. Suppose we have service model G_i having a transition $t_i \xrightarrow{\phi/F_j!msg_1} t'_i$, FSM F_j having a transition $s_j \xrightarrow{G_i?msg_1/F_k!msg_2}$

s'_j, and FSM F_k having a transition $s_k \xrightarrow{F_j?msg_2/\phi} s'_k$. These lead to two state changes of the protocol system:

$$\frac{G_i : t_i \xrightarrow{\phi/F_j!msg_1} t'_i \wedge F_j : s_j \xrightarrow{G_i?msg_1/F_k!msg_2} s'_j \wedge F_k : s_k \xrightarrow{F_j?msg_2/\phi} s'_k}{\langle \ldots, s_j, \ldots, s_k, \ldots, t_i, \ldots \rangle \to \langle \ldots, s'_j, \ldots, s_k, \ldots, t'_i, \ldots \rangle \to \langle \ldots, s'_j, \ldots, s'_k, \ldots, t'_i, \ldots \rangle} \tag{3.13}$$

The first state change is a handshaking between G_i and F_j, which both change states. The second state change is a handshaking between F_j and F_k, and only F_k changes state since the state change of F_j has been accounted for. Sequence of state changes initiated by the stimulus input generated by a service model does not interleave with another sequence of state changes. That is, the sequence of transitions is atomic. It represents a collection of atomic executions within the protocol systems. The only possible state change after $\langle \ldots, s_j, \ldots, s_k, \ldots, t_i, \ldots \rangle \to \langle \ldots, s'_j, \ldots, s_k, \ldots, t'_i, \ldots \rangle$ is $\langle \ldots, s_j, \ldots, s_k, \ldots, t_i, \ldots \rangle \to \langle \ldots, s'_j, \ldots, s'_k, \ldots, t'_i, \ldots \rangle$. This can be generalized to include several FSMs being triggered in sequences:

$$\frac{G_i : t_i \xrightarrow{\phi/F_j!msg_1} t'_i \wedge F_{j_1} : s_{j_1} \xrightarrow{G_i?msg_1/F_{j_2}!msg_2} s'_{j_1} \wedge \ldots \wedge F_{j_r} : s_{j_r} \xrightarrow{F_{j_{r-1}}?msg_2/\phi} s'_{j_r}}{\langle \ldots, s_{j_1}, \ldots, s_{j_r}, \ldots, t_i, \ldots \rangle \to \langle \ldots, s'_{j_1}, \ldots, s_{j_r}, \ldots, t'_i, \ldots \rangle \to \ldots \to \langle \ldots, s'_{j_1}, \ldots, s'_{j_r}, \ldots, t'_i, \ldots \rangle} \tag{3.14}$$

The chain of transitions alway start with the service model generating a stimulus input to an FSM or another service model, and it always ends by not generating another output, either in a FSM transition without an output, or a service model transition with input label. Again, the sequence does not interleave with other sequences.

In fact, we have to distinguish states within a chain of transitions. Given a chain of states and transitions, the states that are neither the first nor the last are *intermediate* states. Intermediate state $\langle \ldots, s'_{j_1}, \ldots, s_{j_r}, \ldots, t'_i, \ldots \rangle$ only has a restricted enabled action, while typical $\langle \ldots, s'_{j_1}, \ldots, s_{j_r}, \ldots, t'_i, \ldots \rangle$ state often has different set of enabled actions, depending upon the ϕ-input transitions that the service models have given the states they are in. It is more appropriate to append the intermediate states with the only enabled actions they have. For example, (3.14) becomes

$$\begin{aligned}&\langle \ldots, s_{j_1}, \ldots, s_{j_r}, \ldots, t_i, \ldots \rangle \to \langle \ldots, s'_{j_1}, \ldots, s_{j_r}, \ldots, t'_i, \ldots \rangle \cdot \langle F_{j_1}, msg_2, F_{j_2} \rangle \to \ldots \\ &\qquad \to \langle \ldots, s'_{j_1}, \ldots, s_{j_r}, \ldots, t'_i, \ldots \rangle \cdot \langle F_{j_{r-1}}, msg_r, F_{j_r} \rangle \to \langle \ldots, s'_{j_1}, \ldots, s'_{j_r}, \ldots, t'_i, \ldots \rangle\end{aligned} \tag{3.15}$$

We now consider that some FSM transitions contain more than one outputs. The outputs on the same transition are atomic, thus we would expect the sequence it leads to does not interleave with other sequences, either. Again suppose a service model G_i has a transition $t_i \xrightarrow{\phi/F_j!msg_1} t'_i$, but this time in the transition of FSM F_j there are two outputs $s_j \xrightarrow{G_i?msg_1/F_k!msg_2 \cdot F_l!msg_3} s'_j$. The last receiving FSMs F_k have transitions $s_k \xrightarrow{F_j?msg_2/\phi} s'_k$ and $s_l \xrightarrow{F_j?msg_3/\phi} s'_l$, respectively. The state changes are summarized as below:

$$\frac{\ldots \wedge F_j : s_j \xrightarrow{x/F_k!msg_2 \cdot F_l!msg_3} s'_j \wedge F_k : s_k \xrightarrow{F_j?msg_2/\phi} s'_k \wedge F_l : s_l \xrightarrow{F_j?msg_3/\phi} s'_l}{\ldots \to \langle \ldots, s'_j, \ldots, s_k, \ldots, s_l, \ldots \rangle \to \langle \ldots, s'_j, \ldots, s'_k, \ldots, s_l, \ldots \rangle \to \langle \ldots, s'_j, \ldots, s'_k, \ldots, s'_l, \ldots \rangle} \tag{3.16}$$

In this case, the state change of F_k does not have to come before the state change of F_l. It is expected that the order of transitions does not affect the last state in the sequence, so that in verification one ordering of transitions is sufficient to represent the behavior of the system. This is particularly important if the F_k and F_l in this example do generate outputs. If however the ordering does matter, then it is not appropriate to model the interactions using handshaking. Similarly, if within a chain a transitions there are transitions that have more than one outputs, the set of enabled actions at intermediate states are also restricted. We should append these intermediate states with the set of enabled actions as well. For example, (5.5) becomes

$$\begin{aligned}
\ldots &\to \langle \ldots, s'_j, \ldots, s_k, \ldots, s_l, \ldots \rangle \cdot \langle F_j, msg_2, F_k \rangle \cdot \langle F_j, msg_3, F_l \rangle \\
&\to \langle \ldots, s'_j, \ldots, s'_k, \ldots, s_l, \ldots \rangle \cdot \langle F_j, msg_3, F_l \rangle \\
&\to \langle \ldots, s'_j, \ldots, s'_k, \ldots, s'_l, \ldots \rangle
\end{aligned} \tag{3.17}$$

The rules discussed above combines a collection of FSMs and service models into an MDP $M_p = (S, Act, P, I_{\text{init}})$. A state of the protocol system $\langle F_1 = s_1, \ldots, F_n = s_n, G_1 = t_1, \ldots, G_m = t_m \rangle$ is a state of Markov decision process.

$$\langle s_1, \ldots, s_n, t_1, \ldots, t_m \rangle \in M_p.S,\ \forall\ s_1 \in F_1.S, \ldots, s_n \in F_n.S, t_1 \in G_1.S, \ldots t_m \in G_m.S \tag{3.18}$$

Given a state that is not part of a sequence of transitions initiated by a service model, there may be one or several service models that are in a state that has one or more transition with output labels. These output labels are exactly the ones that initiate sequences of transitions. The union of these transitions with output labels is the set of nondeterministic choices. Choosing a nondeterministic choice is selecting an output transition among service models, and it leads to a sequence of transitions which are discussed earlier. Let the service model whose output transition is chosen be G_o, the receiving FSM or service model be F_i or G_i, and the handshaking message be m, the action that corresponds to such nondeterministic choice is

$$\langle G_o, m, F_i\rangle \in M_p.Act \text{ or } \langle G_o, m, G_i\rangle \in M_p.Act \tag{3.19}$$

A sequence of transitions contains intermediate states. In these intermediate states, the only nondeterminsitic choice is to continue to the next transition. The only enabled action is

$$\langle F_o, m, F_i\rangle \in M_p.Act \tag{3.20}$$

where F_o is the message generating FSM, F_i is the receiving FSM, and m is the handshaking message.

Singleton probabilistic choice arises if the receiving module is an FSM. In such case, the only probability choice has distribution:

$$P(\langle \ldots, s_i, \ldots\rangle, \langle F_o, m, F_i\rangle, \langle \ldots, s_i', \ldots\rangle) = 1 \tag{3.21}$$

if FSM F_o is the one generates message m, and

$$P(\langle \ldots, s_i, \ldots\rangle, \langle G_o, m, F_i\rangle, \langle \ldots, s_i', \ldots\rangle) = 1 \tag{3.22}$$

if service model G_o is the one that generates message m.

If the last of a chain of transitions is a transition of service model, there can be more than one probabilistic choices. Given a service model G_i in state t_i, each transition with the same input label, $F_o?m$ if the handshaking message m comes from an FSM F_o or $G_o?m$ if the handshaking message m comes from another service model G_o, leads to a probabilistic choice. The probabilistic choices have the following probability distribution:

$$P(\langle \ldots, t_i, \ldots\rangle, \langle F_o, m, G_i\rangle, \langle \ldots, t_i', \ldots\rangle) = \mu(t_i, G_i?m, \phi, t_i') \tag{3.23}$$

or

$$P(\langle \ldots, t_i, \ldots\rangle, \langle G_o, m, G_i\rangle, \langle \ldots, t_i', \ldots\rangle) = \mu(t_i, G_i?m, \phi, t_i') \tag{3.24}$$

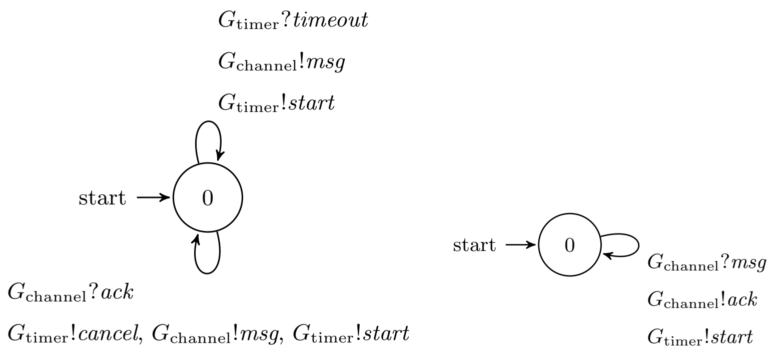
where μ is its transition probability function of G_i.

The constructed MDP can potentially has up to

$$\prod_{1 \le i \le n} |F_i(S)| \cdot \prod_{1 \le j \le m} |G_j(S)| \tag{3.25}$$

states. The number of states grows *polynomially* in the number of states of each constituting modules and *exponentially* in the number of modules being considered. This exponential growth is known as *state explosion* problem.

During verification, reachability analysis, often based on DFS, is applied to the MDP. In practice, it is not necessary to construct the complete MDP before state traversal. Using the rules described in this section, the non-deterministic choices and the probabilistic choices entailed can be determined on-the-fly by examing the state of MDP.



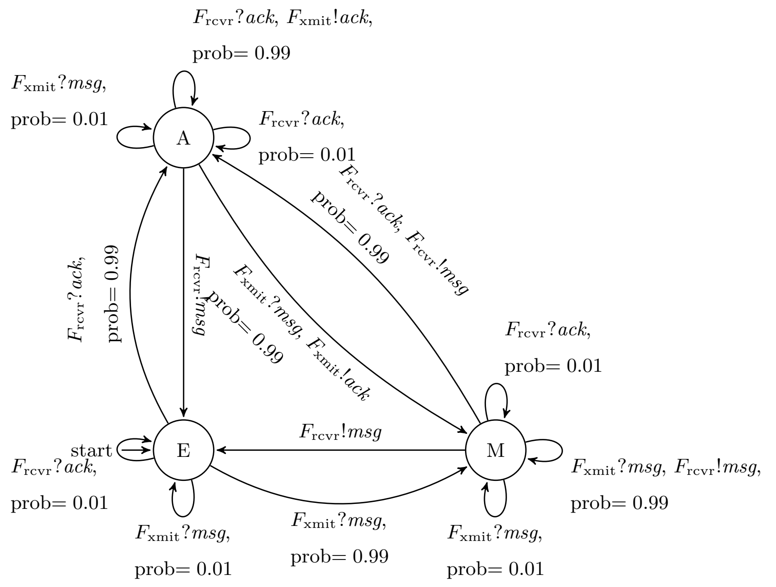
(a) Transmitter FSM F_{xmit} (b) Receiver FSM F_{rcvr}

Figure 3.3: Specification of acknowledgment protocol

3.4 Example: Verifying Acknowledgement Protocol

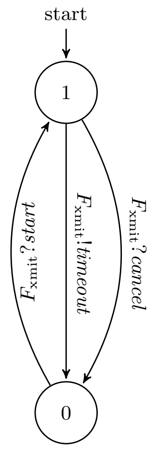
As an example, we verify a simple acknowledgment protocol using the models and methods discussed in the previous sections. The modules in the protocol systems, namely the transmitter, receiver, communication channel, and timers, are specified and modeled as FSMs and service models described in Section 3.1. The FSMs and the service models are then combined to form an MDP using the rules in Section 3.3. The states in MDP is constructed on-the-fly during state traversal. Only a portion of MDP is explored in this example. We compare the state traversal of the MDP and the traversal of the product of the MDP and the DFA that represents the regular safety property part of the probabilistic safety property we wish to check in Section 3.2.

The acknowledgment protocol is specified as two FSMs. They are the transmitter F_{xmit} in Fig.3.3a and the receiver F_{rcvr} in Fig.3.3b. Both FSMs are completely specified, as there is only a single state in both the transmitter and the receiver.



$F_{\text{rcvr}}?ack$, $F_{\text{xmit}}!ack$, prob= 0.99
$F_{\text{xmit}}?msg$, prob= 0.01
$F_{\text{rcvr}}?ack$, prob= 0.01
A
$F_{\text{rcvr}}?ack$, $F_{\text{rcvr}}!msg$ prob= 0.99
$F_{\text{rcvr}}?ack$, prob= 0.99
$F_{\text{rcvr}}!msg$
$F_{\text{xmit}}?msg$, $F_{\text{xmit}}!ack$ prob= 0.99
$F_{\text{rcvr}}?ack$, prob= 0.01
start
E
$F_{\text{rcvr}}!msg$
M
$F_{\text{rcvr}}?ack$, prob= 0.01
$F_{\text{xmit}}?msg$, $F_{\text{rcvr}}!msg$, prob= 0.99
$F_{\text{xmit}}?msg$, prob= 0.01
$F_{\text{xmit}}?msg$, prob= 0.99
$F_{\text{xmit}}?msg$, prob= 0.01

(a) Service model of channel G_{channel}



(b) G_{timer}

Figure 3.4: Service models used in verification of acknowledgment protocol

The two FSMs communicate through a half-duplex communication channel. The channel is modeled as a service model as in Fig.3.4a. When a message is put into the channel by either transmitter or receiver, a two-point probability distribution is at work. With probability 0.99 the channel moves out of the empty state 0, while with probability 0.01 the channel stays in the empty state, meaning that the message is lost during transmission. The channel is bounded, that is, it only allows one message being in transit at any time, be it a transmitter message *msg* or a receiver message *ack*. If the transmitter or the receiver attempts to put another message in the channel when there is already one, the message that is already in transit is 'pushed out' of the channel to its destination, should the new message does not get lost.

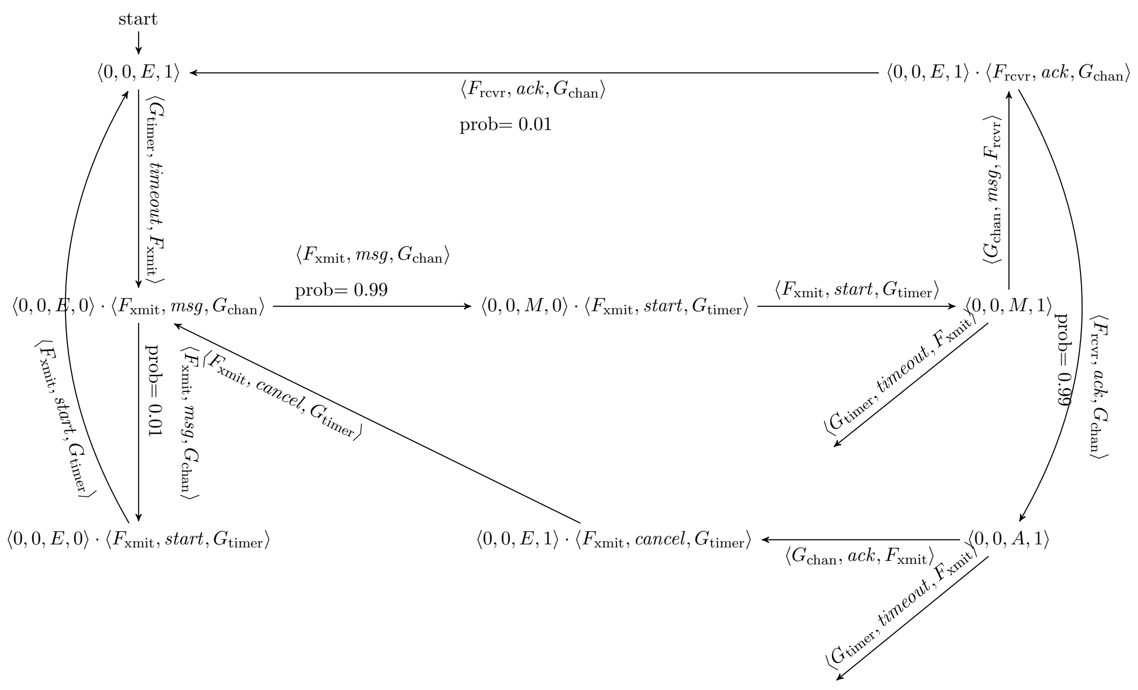

start
⟨0, 0, E, 1⟩
⟨F_rcvr, ack, G_chan⟩
prob= 0.01
⟨0, 0, E, 1⟩ · ⟨F_rcvr, ack, G_chan⟩
⟨G_timer, timeout, F_xmit⟩
⟨G_chan, msg, F_rcvr⟩
⟨F_xmit, msg, G_chan⟩
prob= 0.99
⟨0, 0, E, 0⟩ · ⟨F_xmit, msg, G_chan⟩
⟨0, 0, M, 0⟩ · ⟨F_xmit, start, G_timer⟩
⟨F_xmit, start, G_timer⟩
⟨0, 0, M, 1⟩
prob= 0.99
⟨F_rcvr, ack, G_chan⟩
⟨F_xmit, start, G_timer⟩
prob= 0.01
⟨F_xmit, msg, G_chan⟩
⟨F_xmit, cancel, G_timer⟩
⟨G_timer, timeout, F_xmit⟩
⟨0, 0, E, 0⟩ · ⟨F_xmit, start, G_timer⟩
⟨0, 0, E, 1⟩ · ⟨F_xmit, cancel, G_timer⟩
⟨G_chan, ack, F_xmit⟩
⟨0, 0, A, 1⟩
⟨G_timer, timeout, F_xmit⟩

Figure 3.5: A Portion of Constructed MDP

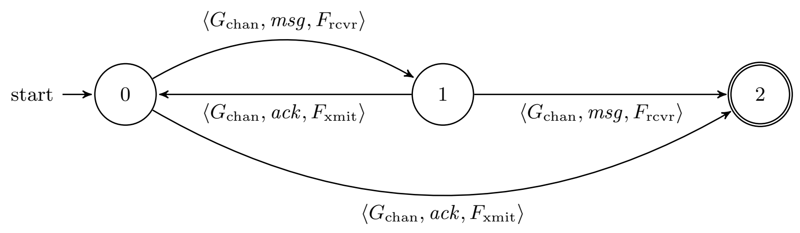
The transmitter also uses a timer to regulate the retransmission of messages. The timer is also modeled as a service model as in Fig.3.4b. Note that the initial state of the timer is in state 1, meaning that the timer is ‘set’. This is for the sake of convenience, as timer expiring prompts the transmitter to retransmit a message to start the system.

Before constructing the MDP, we assume that at the transmitter side, there is a buffer that contains unlimited supply of message from the application. Whenever the transmitter is ready to transmit the next message, there is always a message in the buffer waiting to be transmitted to the application on the receiver side.

The construction of a part of the MDP is shown in Fig.3.5. The initial state of the MDP is $\langle 0, 0, E, 1\rangle$. In the initial state, only the timer has a transition without input that is able to cause state change in the transmitter, so $\langle G_{\text{timer}}, \mathit{timeout}, F_{\text{xmit}}\rangle$ is the only nondeterministic choice available. This leads to the next state $\langle 0, 0, E, 0\rangle \cdot \langle F_{\text{xmit}}, \mathit{msg}, G_{\text{chan}}\rangle$. The trailing $\langle F_{\text{xmit}}, \mathit{msg}, G_{\text{chan}}\rangle$ indicates that the state is restricted to have single enabled action as it is part of a chain of transitions.

In $\langle 0, 0, E, 0\rangle \cdot \langle F_{\text{xmit}}, \mathit{msg}, G_{\text{chan}}\rangle$, $\langle F_{\text{xmit}}, \mathit{msg}, G_{\text{chan}}\rangle$ is the only choice, which yields two probabilistic choices with probability weights 0.99 and 0.01 according to the channel's service model. The choice with probability 0.01 leads to state $\langle 0, 0, E, 0\rangle \cdot \langle F_{\text{xmit}}, \textit{start}, G_{\text{timer}}\rangle$. Note that without the trailing $\langle F_{\text{xmit}}, \textit{start}, G_{\text{timer}}\rangle$ we cannot distinguish the new state from the last. The next state after this is the end of the chain of transitions and returns to the already explored $\langle 0, 0, E, 0\rangle$. Back to the branch of probabilistic choices, the choice of 0.99 probability leads to $\langle 0, 0, M, 0\rangle \cdot \langle F_{\text{xmit}}, \textit{start}, G_{\text{timer}}\rangle$, whose only enabled action starts the timer and arrives at the end of the chain of transitions $\langle 0, 0, M, 1\rangle$.

In $\langle 0, 0, M, 1\rangle$ both channel and timer have transition without input stimulus. This leads to two nondeterministic transitions: $\langle G_{\text{chan}}, \textit{msg}, F_{\text{rcvr}}\rangle$ and $\langle G_{\text{timer}}, \textit{timeout}, F_{\text{xmit}}\rangle$. As we are using DFS, we may opt to explore the states by choosing the former choice, and return to the later choices after all states after the former choice have been explored. The explored states shown in Fig.3.5 are all states explored after choosing $\langle G_{\text{chan}}, \textit{msg}, F_{\text{rcvr}}\rangle$ here, and $\langle G_{\text{chan}}, \textit{ack}, F_{\text{xmit}}\rangle$ when facing another two nondeterministic choices.



$\langle G_{\text{chan}}, \textit{msg}, F_{\text{rcvr}}\rangle$
start
0
1
2
$\langle G_{\text{chan}}, \textit{ack}, F_{\text{xmit}}\rangle$
$\langle G_{\text{chan}}, \textit{msg}, F_{\text{rcvr}}\rangle$
$\langle G_{\text{chan}}, \textit{ack}, F_{\text{xmit}}\rangle$

Figure 3.6: Bad prefix DFA of the regular safety property

Constructing MDP alone is fine when checking state invariant properties. When checking MDP against regular safety property, the MDP-DFA product is being constructed when state traversal is at work. For the acknowledgment protocol, a correct message delivery from the transmitting side to the receiving side would be alternating between retrieving a message from the buffer and delivery the message to the receiving application. The DFA in Fig.3.6 accepts the sequence that do not obey the expected behavior. $\langle G_{\text{chan}}, \textit{ack}, F_{\text{rcvr}}\rangle$ corresponds to the retrieval of message from the buffer, this happens every time an acknowledgment is received at the transmitter, which means successful delivery of the previous message. $\langle G_{\text{chan}}, \textit{msg}, F_{\text{rcvr}}\rangle$ corresponds to the delivery of message to the receiving application, and this happen whenever a message is received from the channel. Note that a correct alternating sequence starts with label $\langle G_{\text{chan}}, msg, F_{\text{rcvr}}\rangle$. This is also for the sake of expediency, because the protocol system starts in a state where timer is 'set' and a new message is put into the channel to be received by the receiver when timer expires.

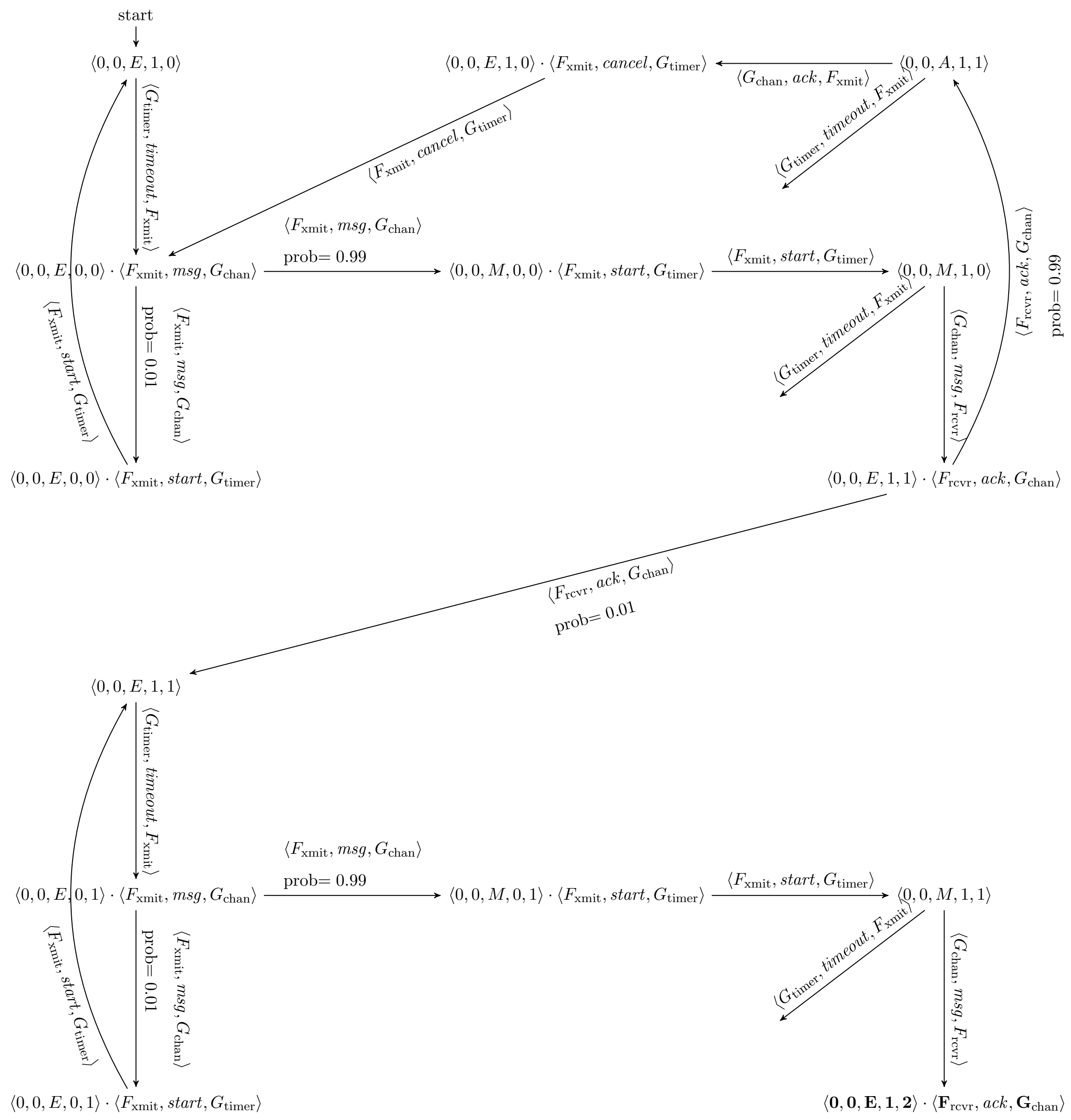

start
$\langle 0,0,E,1,0\rangle$
$\langle G_{\text{timer}}, timeout, F_{\text{xmit}}\rangle$
$\langle 0,0,E,1,0\rangle \cdot \langle F_{\text{xmit}}, cancel, G_{\text{timer}}\rangle$
$\langle G_{\text{chan}}, ack, F_{\text{xmit}}\rangle$
$\langle 0,0,A,1,1\rangle$
$\langle F_{\text{xmit}}, cancel, G_{\text{timer}}\rangle$
$\langle G_{\text{timer}}, timeout, F_{\text{xmit}}\rangle$
$\langle F_{\text{xmit}}, msg, G_{\text{chan}}\rangle$
prob= 0.99
$\langle 0,0,E,0,0\rangle \cdot \langle F_{\text{xmit}}, msg, G_{\text{chan}}\rangle$
$\langle 0,0,M,0,0\rangle \cdot \langle F_{\text{xmit}}, start, G_{\text{timer}}\rangle$
$\langle F_{\text{xmit}}, start, G_{\text{timer}}\rangle$
$\langle 0,0,M,1,0\rangle$
$\langle F_{\text{rcvr}}, ack, G_{\text{chan}}\rangle$
prob= 0.99
$\langle F_{\text{xmit}}, start, G_{\text{timer}}\rangle$
prob= 0.01
$\langle F_{\text{xmit}}, msg, G_{\text{chan}}\rangle$
$\langle G_{\text{timer}}, timeout, F_{\text{xmit}}\rangle$
$\langle G_{\text{chan}}, msg, F_{\text{rcvr}}\rangle$
$\langle 0,0,E,0,0\rangle \cdot \langle F_{\text{xmit}}, start, G_{\text{timer}}\rangle$
$\langle 0,0,E,1,1\rangle \cdot \langle F_{\text{rcvr}}, ack, G_{\text{chan}}\rangle$
$\langle F_{\text{rcvr}}, ack, G_{\text{chan}}\rangle$
prob= 0.01
$\langle 0,0,E,1,1\rangle$
$\langle G_{\text{timer}}, timeout, F_{\text{xmit}}\rangle$
$\langle F_{\text{xmit}}, msg, G_{\text{chan}}\rangle$
prob= 0.99
$\langle 0,0,E,0,1\rangle \cdot \langle F_{\text{xmit}}, msg, G_{\text{chan}}\rangle$
$\langle 0,0,M,0,1\rangle \cdot \langle F_{\text{xmit}}, start, G_{\text{timer}}\rangle$
$\langle F_{\text{xmit}}, start, G_{\text{timer}}\rangle$
$\langle 0,0,M,1,1\rangle$
$\langle F_{\text{xmit}}, start, G_{\text{timer}}\rangle$
prob= 0.01
$\langle F_{\text{xmit}}, msg, G_{\text{chan}}\rangle$
$\langle G_{\text{timer}}, timeout, F_{\text{xmit}}\rangle$
$\langle G_{\text{chan}}, msg, F_{\text{rcvr}}\rangle$
$\langle 0,0,E,0,1\rangle \cdot \langle F_{\text{xmit}}, start, G_{\text{timer}}\rangle$
$\langle \mathbf{0,0,E,1,2}\rangle \cdot \langle \mathbf{F}_{\text{rcvr}}, ack, \mathbf{G}_{\text{chan}}\rangle$

Figure 3.7: Portion of MDP-DFA product

Now we are ready to explore the MDP-DFA product. Fig.3.7 shows the state traversal of a portion of MDP-DFA product. The state traversal starts from the initial state of the product, $\langle 0,0,E,1,0\rangle$, with the fifth element in the tuple being the state of the

DFA. States and transitions are pretty much the same until state $\langle 0, 0, M, 1, 0\rangle$. It has two nondeterministic choices, and we choose $\langle G_{\text{chan}}, \mathit{msg}, F_{\text{rcvr}}\rangle$ like what we did in the exploration of MDP. In current discussion, we always choose to explore the choice other than $\langle G_{\text{timer}}, \mathit{timeout}, F_{\text{xmit}}\rangle$ whenever we face nondeterministic choice. After the transition MDP has state $\langle 0, 0, E, 1, 0\rangle \cdot \langle F_{\text{rcvr}}, \mathit{ack}, G_{\text{chan}}\rangle$. The state of the DFA changes to 1, as per the definition.

At this point the traversal faces two probabilistic choices. The choice with probability 0.99 leads to a loop of state and transitions back to $\langle 0, 0, E, 1, 0\rangle \cdot \langle F_{\text{rcvr}}, \mathit{ack}, G_{\text{chan}}\rangle$. This loop is the correct operation of the protocol. The system alternates between retrieval of message and delivery of message. The choice with probability 0.01 eventually leads to state $\langle 0, 0, E, 1, 2\rangle \cdot \langle F_{\text{rcvr}}, \mathit{ack}, G_{\text{chan}}\rangle$ (the only boldfaced state in the diagram). The DFA then moves into state 2, meaning that a bad prefix is found. Following the series of transitions we see that action $\langle G_{\text{chan}}, \mathit{msg}, F_{\text{rcvr}}\rangle$, or delivery of message, appears twice in a row without $\langle G_{\text{chan}}, \mathit{ack}, F_{\text{xmit}}\rangle$, or retrieval of new message from the buffer. On reaching state 2 of DFA, the state traversal algorithm puts such state in the set T and stop continuing along such path. The algorithm continues from the previous unexplored branches, for instance, choice $\langle G_{\text{timer}}, \mathit{timeout}, F_{\text{xmit}}\rangle$.

By the time the algorithm terminates, we construct the linear program to compute the probability of reaching T from the initial state. This is the probability for the acknowledgment protocol to violate the property, which is then compared to the part of probability bound in the probabilistic safety property to determine whether the probabilistic safety property holds or not.

3.5 Conclusions

In this chapter, we provided definitions of the formal modeling construct for modules in a protocol systems. FSMs model those protocol instances which are input-driven and deterministic; service models describe the modules that interact with the physical world directly and indirectly and thus have probabilistic behaviors. Next, we described MDPs for modeling probabilistic systems and probabilistic safety properties. It was then shown that an

MDP modeling the protocol system can be constructed from FSMs and service models using the given composition rules.

In later chapters, we use FSMs to specify the lock protocol (Chapter 5) and the driver-assisted merge protocol (Chapter 7). The environment where these protocols reside in, including communcation channels, sensors, safe spacing system, and the timing stack are modeled as service model. The stratified verification technique considers a discretized variation of MDPs (Chapter 6).

Chapter 4

Multiple Stack Architecture

This chapter introduces a multiple stack architecture for cooperative driving applications. Each stack in the architecture addresses a particular aspect of vehicle's interaction with the physical world. Within each stack the modules are arranged in layers, which allows new technogies to be introduced to existing architecture and verification to be done in a layered fashion.

Intelligent vehicles are similar to communication networks, whether they cooperate using communication channels or not. They interact with the physical world using devices that are rapidly evolving. The design of intelligent vehicle architectures involves the integration of new features into existing vehicle architectures under the guarantee of robustness. The problem is defined as what functional characteristics should be added to make vehicles intelligent, and how to do the integration. Different intelligent vehicle projects have their own designed architectures to meet the specific objectives, but in common, they inherit general characteristics of all automotive systems. However, architectures with a single stack cannot compose the whole picture of intelligent vehilces. Different from communications networks, intelligent vehicles interact with the physical world in several ways, rather than one, and because many of the interactions are time critical. While it is necessary to accommodate changes in all of the technologies, a simple layered architecture is not sufficient, which motivates the adoption of multiple stacks.

In this chapter, we describe the functions of layers in each stack (section 4.1), then we discuss the implication of layered structure to the verification of cooperative driving protocol (section 4.2).

4.1 Multiple Interactions with the Physical World

Intelligent driving systems interact with the physical world in several ways. In addition to communications, intelligent vehicles have sensors that detect the surroundings, measuring devices that monitor the operation of the vehicle, and devices that control the operation of the vehicle. Each type of the layered architectures described above only deals with one aspect of the physical domain. We organize the interactions into separate stacks, so that we can isolate the hardware that controls or monitors the physical world from the software that uses the information or issues commands, as shown in Fig.4.1. A precursor of this architecture is described in [Lin and Maxemchuk, 2012].

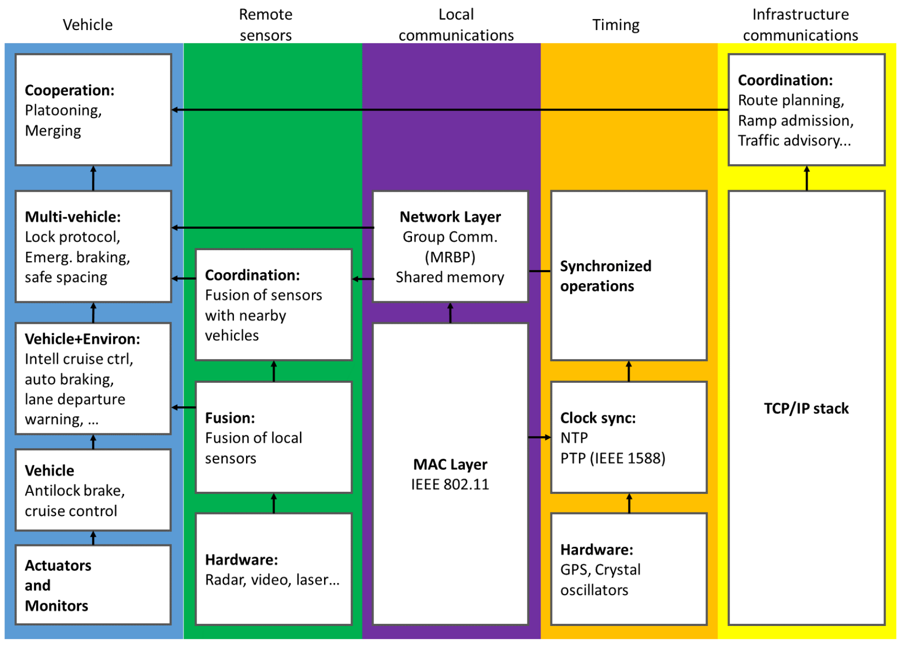

Vehicle
Remote sensors
Local communications
Timing
Infrastructure communications
Cooperation: Platooning, Merging
Multi-vehicle: Lock protocol, Emerg. braking, safe spacing
Vehicle+Environ: Intell cruise ctrl, auto braking, lane departure warning, ...
Vehicle Antilock brake, cruise control
Actuators and Monitors
Coordination: Fusion of sensors with nearby vehicles
Fusion: Fusion of local sensors
Hardware: Radar, video, laser...
Network Layer Group Comm. (MRBP) Shared memory
MAC Layer IEEE 802.11
Synchronized operations
Clock sync: NTP PTP (IEEE 1588)
Hardware: GPS, Crystal oscillators
Coordination: Route planning, Ramp admission, Traffic advisory...
TCP/IP stack

Figure 4.1: Multiple stack layered architecture for cooperative driving system

The arrows represent service provide/use relationship between modules. For example, in each stack there are arrows pointing upward between layers, which indicates that module(s) in a lower layer provides services to the module(s) that resides one layer higher than it. To state it in another way, module(s) in a higher layer uses on the services provided by the module(s) one layer beneath them.

Besides providing services to the layer directly above, modules in a layer can also provide services to components residing in other stacks. For instance, the coordination layer in the sensor stack requires group communication service from the local communications stack to exchange sensor information with nearby vehicle, and clock synchronization layer also requires communications. A collaborative driving application may also depend on the services in different stacks, as long as the dependencies do not constitute to directed cycles.

4.1.1 The Vehicle Stack

The vehicle stack consists of applications that control the vehicle and interface to the vehicle's hardware. The organization of this stack is similar to the stack in the PATH project. The applications are organized to allow more complex applications to use the services provided by more basic applications, so that complicated applications can be designed and tested more easily.

The vehicle's hardware is the physical layer for the applications, consisting of devices that monitor the operation of the vehicle (monitors) and devices that control the movement of the vehicle (actuators). The monitors include speedometer, sensors measuring tire rotation, sensors within the engine, and etc. The actuators control the brakes, throttles, and steering.

The applications in the vehicle layer only use information within the vehicle. For instance, an antilock braking system monitors tire rotation and individually actuates the brakes to avoid skids. Cruise control systems monitor the vehicle speed and control the throttle, and possibly the brakes. The implementations of applications in the vehicle layer are governed by feedback laws: the antilock braking system applies the brakes depending upon the measurements, which changes the measurements, and changes the command from the antilock braking. The objective is to stop the vehicle as quickly as possible without skidding or losing control of the vehicle. The design of an antilock braking system is dependent on the physics of motion and control theory. The implementation depends on the characteristics of the vehicle.

The applications in the vehicle stack "hide" the implementation details from the driver and the applications in one layer above, the vehicle plus environment layer. The antilock brakes present an interface to the higher layer that is independent of the physical layer. A driver presses the brakes harder if she/he wants to stop more quickly, and the antilock braking system applies the brakes in order to stop safely. The cruise control system allows the driver to set the desired speed, and it is good to go. The driver retains longitudinal control by stepping on the brake or by pressing a button on the steering wheel. No matter how the antilock brakes and the cruise control are implemented, the services provided across the vehicle layer interface remain the same.

The vehicle plus environment layer uses information from local sensors that detect adjacent vehicles. And automatic braking system monitors the distance to the lead vehicle to actuate the antilock brakes in the event that a rear end collision will occur. An adaptive cruise control system takes a driver-specified headway and speed, and it uses the cruise control to maintain the speed when there is no vehicle in front and reduces the speed set by cruise control to maintain the desired gap when the the preceding vehicle is detected. A lane departure warning system uses camera to detect the line markings on the highway and notifies the driver when the vehicle deviates. A park assist system detects parked cars and the marking of parking spot and controls the throttle, brakes, and steering.

The multi-vehicle layer uses sensors and monitors from adjacent vehicles to improve the operation of the single vehicle system. Using sensors from adjacent vehicles provides a map of vehicles beyond the range of the local sensors, improves the distance estaimates from the local sensors by making multiple measurements of the same gaps, and detects potentially defective sensors. In addition, communications provides a notification that the lead vehicle is braking, which allows an earlier response for an emergency braking system than detecting the change in the relative position of that vehicle. In addition, measurements on the recent stopping distance for the vehicle as well as its neighboring vehicles, under the current road conditions and load in the vehicles, allow for a safer and more accurate calculation of the safe space between vehicles. The authors of [Tientrakool *et al.*, 2011] show that this will significantly increase the highway capacity in comparison with an automatic braking protocol that uses insurance industry standards. The lock protocol is described in Chapter 5.

The cooperative driving layer uses communication between nearby vehicles to coordinate their operation. Platooning systems create convoys of cars and trucks on highways. The driver-assisted merge protocol, which is one of the main objective of this thesis, is described in more detail in Chapter 7.

4.1.2 The Communication Stacks

There are two communication stacks in the architecture, a local communication stack that is used to communicate with nearby vehicles and and an infrastructure communication stack that is used for all other communications. The TCP/IP stack, used in the Internet, is currently the dominant architecture and is recommended for the infrastructure stack.

The Internet is a “best effort” network. There are no delay guarantees. If a message is lost in transit, it may be retransmitted several times and can arrive at the destination tens of seconds later. Cooperative driving applications the coordinate the maneuvers of adjacent vehicles have severe constraints on delay. While it is possible to construct a local network within the framework of the TCP/IP protocols, the network must provide service guarantees that are different from the Internet.

The infrastructure communication stack accounts for vehicle-to-infrastructure (V2I) communications. It is used to gather traffic reports for route planning and incident reports for the roadway that is beyond a driver’s view. In a more centralized system, such as the one envisioned by the PATH project, the routing commands given to platoons of vehicles are computed in a centralized location then dispatched to the vehicles using V2I communications.

The local communication stack accounts for vehicle-to-vehicle (V2V) communications. The media access control (MAC) layer for V2V communication has been under active research. This layer contains DSRC, which has several layers of its own or similar channel sharing protocols.

The group communications layer provides reliable communication between collaborating vehicles, for instance, the three participants in a merge protocol, or the members of a platoon. The Mobile Reliable Broadcast Protocol (MRBP) [Maxemchuk *et al.*, 2007] and Timed Reliable Broadcast Protocol (TRBP) [Maxemchuk and Shur, 2001] are group communication protocols that provide a common list of messages to all of the receivers within a deadline.

MRBP and TRBP are scheduled token passing protocols. Each member of the group is scheduled to transmit information, a list of the messages it has not received or recovered, and a vote on the messages that it is certain that other receivers have received, at specified times. The time that each user is scheduled to transmit is rotated in a round-robin order. If a group member does not receive a scheduled transmission, it starts the recovery process immediately after the message is scheduled, instead of waiting for the source to retransmit a message that is not acknowledged. After one cycle of the token plus the time allowed for message recovery, all of the members of the group have reported any messages that they have not received and begin a distributed voting procedure to decide which messages to include in the common list of messages. Any receivers that are uncertain about the outcome of the vote leave the group and stop transmitting.

The Fail-Safe Broadcast Protocol (FSBP) [Gu *et al.*, 2015] is also a group communication protocol. It is based synchronous clocks described in Chapter 5, and it ensures a message is received by everyone in the group within a deadline, otherwise a failure is detected within a time constraint.

In [Kim and Maxemchuk, 2012], an early version of the driver-assisted protocol in this thesis survived communication failure or a communicating vehicle moving out of range. This earlier version of the merge protocol was made fail-safe by requiring an unanimous vote of the messages in the common group and aborting the merge when unanimity for any message could not be guaranteed.

4.1.3 The Timing Stack

The purpose of the timing stack is to supply the cooperative driving application within an intelligent vehicle with accurate clocks with respective to the timing information supplied by GPS. With applications on different vehicles having accessed to this fairly accurate clocks, one may assume that the applications are operating according to the same clock. This is particularly true since GPS provides the timing information that is accurate within hundreds of nanoseconds, which is more than sufficient for intelligent vehicle.

The timing stack has three layers: 1) The hardware layer, 2) clock synchronization layer, and 3) synchronized operations layer. All timed events are initiated from the routines in synchronized operations layer, thus there are no timers or clocks in any of the other protocols. By removing time from the other protocols, it is possible to use verification and conformance testing techniques that have been developed for communications protocols to prove that these protocols will operate properly.

The hardware stack uses GPS receivers and crystal oscillators to maintain a local clock. A possible arrangement is having a main oscillator circuit that is constantly adjusted according to the GPS signal received. This circuit is called *GPS disciplined crystal oscillator*. The other circuits provides redundancy. They are adjusted periodically to the reading of the main circuit and can help detect errors in the main circuit.

The clock synchronization layer provides additional synchronization for a group of vehicles within the same neighborhood or the vehicles that are engaging in a cooperative maneuver by using PTP over wireless communication. When GPS signal is available, this layer does not have effect. When GPS signal is temporarily unavailable, the crystal oscillator runs freely and its accuracy with respect to UTC deteriorate. When the oscillator is not adjusted for an extended period of time, synchronization over wireless comes into effect to ensure that clocks on a group of vehicles are still synchronized within the order of microseconds, which is still sufficient for most cooperative driving applications.

The synchronized operations layer maintains a list of timed events for the modules in the intelligent vehicle. A module in the system sets one or more timestamps for the event it expects to occur in the future, and the layer generates a message that indicates the occurrence of the event when the clock reads its corresponding timestamp. Its use is elaborated with more details in Chapter 5.

4.1.4 The Sensor Stack

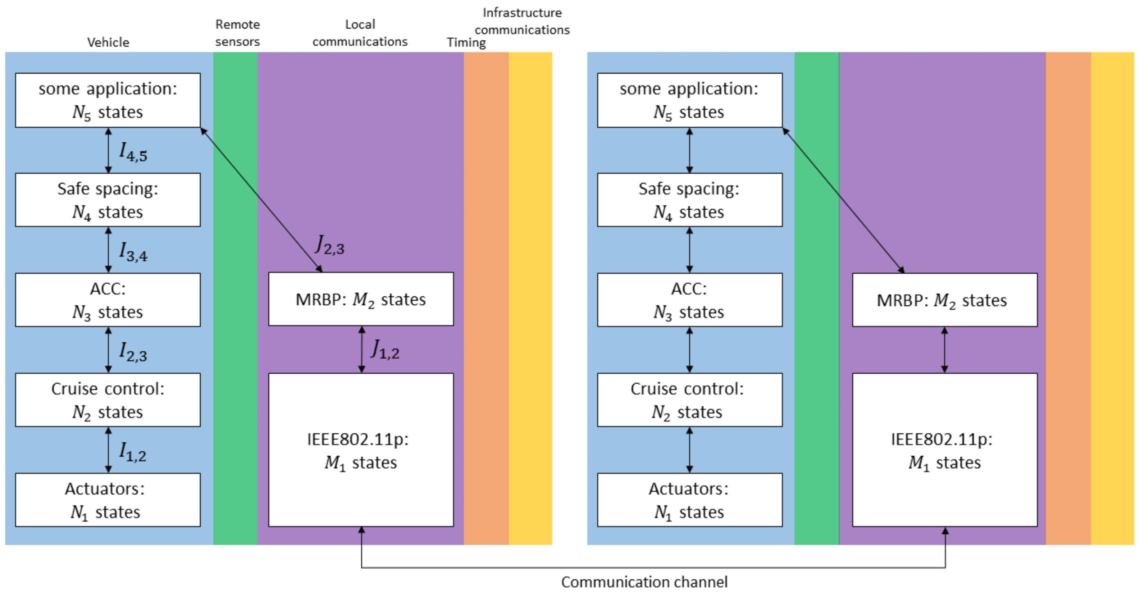
The sensor stack in this architecture is similar to the one developed by BMW research [Aeberhard *et al.*, 2012]. There is a fusion layer that combines readings from all of the sensors in the vehicle to create a map of the vehicles and obstacles surrounding the vehicles. The hardware and fusion layer are separate so that new sensor technologies can be developed without changing the algorithm in the fusion layer.

The additional layer in the sensor stack is a coordination layer. This layer considers the sensor readings from different vehicles, and it guarantees that all of the participants are using identical maps. For instance, if the distance that a vehicle measures to an adjacent vehicle differs from the distance that vehicle measures, both vehicles can use the safer distance estimate or they can decide not to cooperate. Exchanging measurements with other vehicles detects possible failures or inaccuracies in sensors, or malicious users who have inserted erroneous measurements. The design of a secure sensor fusion algorithm for the coordination layer is, however, beyond the scope of this thesis.

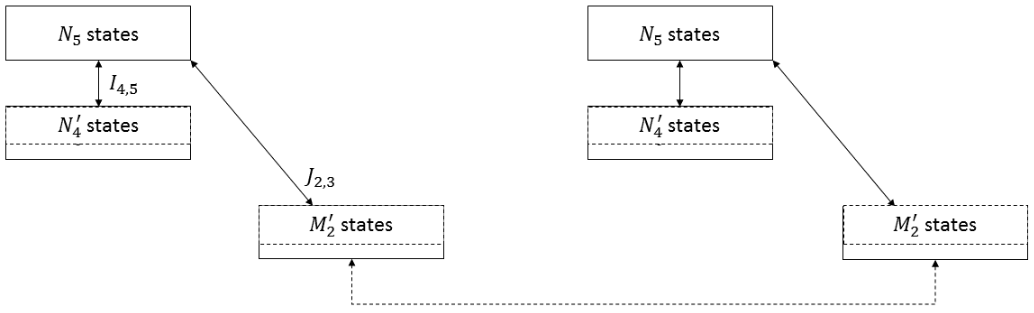
4.2 Layered Architecture and Verification

The multiple stack architecture organizes a cooperative driving system into multiple interacting modules, which are arranged as layers in different stack. Using layered architectures, rather than the more general, modular architectures that are common in computer programs and several cooperative driving projects, also simplifies the problem of verifying that a system operates correctly. The benefits are twofold: First, layers can be verified separately. A single layer is much smaller compared to the complete system, which reduces the complexity of verification and mitigates state explosion problem. Secondly, it makes holistic verification of a cooperative driving application possible as different layers are better characterized by different mathematical models, which require different types of verification techniques.

In a layered architecture we can independently verify each module, by proving that each layer provides the service to the next higher layer, given that it receives the proper service from the lower layer. Consider an application on the topmost layer of the vehicle stack as shown in Fig.4.2a. Suppose that it depends on a service at the top layer of the communication stack as well as the service in the layer directedly below it. When we verify the module separately, the composition of the two systems may consist of up to $(N_1N_2N_3N_4N_5M_1M_2)^2$ states. Furthermore, while FSMs are suitable for modeling the coordination function of driver-assisted merge protocol, but it cannot characterize the cruise control (in the second layer of the vehicle stack) or the MAC/PHY layer of communication stack (the bottom layer) properly. The modeling and verification of these layers requires different techniques and math models.



(a) An application that involves two vehicles and two stacks



(b) Verifying the application based on simplified interface

Figure 4.2: Simplifying verification

The layered structure allows us to verify the system layer-by-layer, with appropriate models and techniques. First the actuator and monitor layer is verified that it satisfies an interface $I_{1,2}$, then the cruise control is verified to satisfy another interface $I_{2,3}$ by assuming $I_{1,2}$, and so on. In the communication layer, the MAC layer is first verified to provide an interface $J_{1,2}$ by considering the parties on both vehicle, then MRBP is verified basing on interface $J_{1,2}$. The process continues until that M_3 satisfies an interface $J_{1,2}$ on both sides of parties. Finally, N_5, an application involves two vehicles, is verified assuming the service provided through interface $I_{4,5}$ by the applications in the stack below it and the service provided through interface $J_{2,3}$, as illustrated in Fig.4.2b. The number of states of verification is thus reduced to $(N_5 N'_4 M'_2)^2$, where N'_4 and M'_2 are the number of states of the interfaces $I_{4,5}$ and $J_{2,3}$ which are less than N_4 and M_2.

Note that in Fig.4.1, the directed edges that represent service provide/use relationship between modules do not contain cycles. This is deliberately done so that the modules can be verified in a layered fashion even there are dependency relationship that appear between stacks. In a more general, modular architecture, in which the output from a module directly, or indirectly, feeds back to a module that it receives data from, which comprises a circular dependency, we must simultaneously prove that all of the modules operate correctly.

4.3 Conclusions

A multiple stack architecture for cooperative driving systems is presented in this chapter. The layered structure in each stacks makes it possible for an system engineer to develop collaborative driving applications without being familiar with the details of physical interactions such as the feedback rules that govern the vehicle control, or the sensor fusion algorithms that produce mapping of objects in the surrounding environment. With multiple stacks, which extend the modular approaches in other architectures, an application can interact with physical world in different ways with sufficient abstraction. From a high level perspective, the stacks with well-defined interfaces between layers simplify verification of a cooperative driving system by dividing it into verification of individual layers. However, it is not trivial to verify a system with some of its components being verified only within a probability bound. Compositional verification cannot applied directly. Chapter 6 describes compositional verification for probabilistic systems such as cooperative driving systems.

Chapter 5

Synchronous Clocks

In this chapter, we present a framework for the protocols that are specified as FSMs to take advantage for the accurate clocks provided by the Global Positioning System (GPS). It allows protocols to operate like human making appointments — by indicating a future time at which event should take place. Both the driver-assisted merging and the lock protocol that resolves conflicting requests execute synchronized operations using these accurate clocks to synchronize actions instead of using timers. Having actions executed simultaneously at physically-separated locations eliminates the interleaving of actions that are caused by communication delay and multiple timers.

With the widely available commercial GPS receiver, fairly accurate clock readings with respect to Coordinated Universal Time (UTC) can be obtained everywhere in a system. Processes on different locations may coordinate their actions based on the accurate clocks and execute simultaneous transitions as if they were situated closely and connected by wires. This reduces interleaving of actions that arises when processes communicate over delay-prone communication channels and register future events with timers. It leads to an entirely new class of protocols that take advantage of these accurate clocks by interacting with the timing stack, in which GPS-disciplined clocks reside. We start off by considering a novel design of the lock protocol that resolves conflicting merge requests for driver-assisted merging. We conceive of a simplified interface of the accurate clocks, i.e., a framework consisting of the timing stack and a modification of FSMs. Within the framework protocols may take advantage of the accurate clocks without much consideration about timed events and timing constraints, which is the main objective timed automata. The lock protocol can then be specified using the modified FSM. The framework also comes with the service model for the timing stack and an additional set of compositional rules that allow the timing stack to be included when constructing the MDP for verification. The FSM specification of the lock protocol will serve as a running example for the composition rules.

In this chapter, we motivate the use of GPS clocks by considering our expectation and design of the lock protocol (section 5.1). Next, we describe the interface of timing stack and present a small addition to the FSM defined in Chapter 3, and we specify the lock protocol using the modified FSMs (section 5.2). The service model representation of the timing stack is then explained (section 5.3), which leads to an additional set of composition rules (section 5.4).

5.1 The Lock Protocol

In a toll plaza, or a reasonably loaded section of public highway, it is highly probable that more than one vehicles attempt to change lanes and merge between the other two vehicles in the target lane. Without an infrastructure to coordinate maneuvers from a more global perspective, it is difficult for a vehicle to cooperate in separate merge at the same time. A conflict resolution mechanism must be in place.

The lock protocol presented in this section is a simple solution that resolves conflicting request to common resource. In driver-assisted merging, the resource is the exclusive cooperation of an intelligent vehicle on a merge maneuver. In Chapter 7, we will see that throughout a merge maneuver, driver-assisted merging has three phases, each phase is demarcated by an attempt of the lock protocol. That is, driver-assisted merging uses the lock protocol three times during a successful merge.

We first describe the design of the lock protocol informally, in English and in diagrams. Then we attempt to formalize the specification of the lock protocol in timed automata, or more specifically, in the model specification language of UPPAAL, which is based on timed automata.

5.1.1 Real World Analogy

The goal of the lock protocol is to allow the driver-assisted protocol on the merging vehicle to solicit cooperation from the neighboring vehicles in the target lane for a given amount of time. The lock protocol notifies the merge protocol if both vehicles in the target lane agrees to cooperate. It should ensures that, within the given time, both of the cooperating vehicles will continue to cooperate and only cooperate with the merging vehicle.

A straightforward solution is to have each vehicle equipped with a timer. Suppose the merging vehicle estimates that the merge maneuver should take 30 seconds to complete. It first starts a timer of 40 seconds, to be conservative on time usage, and sends out requests to both of the neighboring vehicles. On receiving the request, each of the neighboring vehicles also sets a timer that lasts 40 seconds and responds to the merging vehicle by a *granted* message. Before the timer expires, it no longer responds to the requests from other vehicles. If the merging vehicle receives both granted messages, then it knows that it has both the neighboring vehicle cooperating before its timer expires. However, the timers on the merging vehicle and the neighboring vehicles always expire at different points. This create interleaving sequences that significantly increase the complexity of verification, especially when we consider an already-complex system that includes up to 7 vehicles within a neighborhood, as shown in the figure. Also when the timer on the merging vehicle expires, it is unnecessary for any of the neighboring vehicles not to respond to a new merge request.

Synchronized clocks are often used to coordinate actions among humans and this results in an unambiguous sequence of operation. Consider three battalions led by a general and his lieutenants. The general dispatches messengers to give orders to his lieutenants. The time taken by the messengers to deliver the orders may vary due to road condition. If the general orders each battalion to attack a certain time after the messenger arrives, there will be a range of attack times for the two remote battalions, which result in a range of differences in the time between attacks. If the objective is to have all the armies attack simultaneously, and the battalions attack when the messengers arrive, there will be size different possible sequences of attacks, with the probability of simultaneous attacks approaching zero. However, if the general and lieutenants synchronize their watches, and the general orders them to attack at a specific time, that is greater than the arrival times of the messengers, then the attacks will occur simultaneously. We still need a protocol to guarantee that the messages have been received or to initiate an alternative plan when there is uncertainty, say, the messengers being intercepted by the enemy.

Inspired by the way us human arrange timed events, we redesign the operations by replacing the timers with reference to synchronized clocks. Instead of starting a timer that expires 40 seconds after, the merging vehicle check current time t and sends out requests that are appended with a parameter $t + 40$. On receiving the request, each of the neighboring vehicles take note of the time $t + 40$ and responds to the merging vehicle by a *granted message*. Before the clock reads $t + 40$, it no longer responds to additional requests from other vehicles. If the merging vehicle receives both granted messages, then it knows that it has both the neighboring vehicle cooperating before the clock reads $t + 40$. When the clock reads $t+40$, the cooperating group is dismissed. The merging vehicle stops the merge maneuver and both neighboring vehicles return to a state in which they respond to merge requests.

5.1.2 The Operation

The lock protocol hides conflict resolution from the merge protocol and replace it with a simpler interface. The merge protocol simply issues to the lock protocol an *attempt* command including the identifier of the two vehicles it wishes to cooperate with and a deadline before which it expects the merge maneuver to complete. If both the two vehicles agree to cooperate, the lock protocol ensures the mutual exclusiveness of the group until the deadline and notifies the merge protocol of success. The participants that are agreed to cooperate returns to the initial state on reaching the deadline when there is a message loss or either one or both of the vehicles cannot cooperate.

On every merge-enabled vehicle there is an instance of the lock protocol. If the merge vehicle attempts to execute merge maneuver, it uses the lock protocol to create a cooperating group and that instance of the lock protocol assumes the role of *master*. When a vehicle that is not participating in any group receives a request to cooperate, it agrees and assumes the role of *slave*.

Fig.5.1 illustrates the operation of the lock protocol. Suppose the driver of vehicles

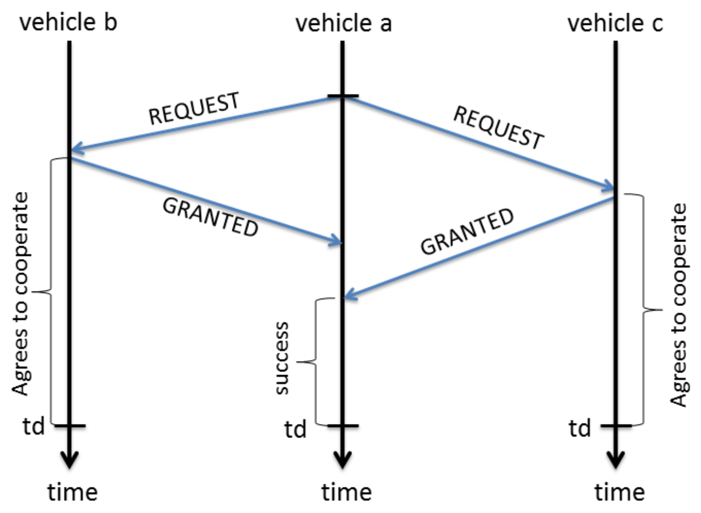


Figure 5.1: Normal Operation of the Lock Protocol

signals his/her intention to merge between vehicles b and c. The merge protocol issues command *attempt*(b, c, t_d), where b and c are the identifiers of the lock protocol's process on vehicles b and c and t_d is the expected time instant when the merge maneuver should finish. The lock protocol on vehicle a becomes master and sends out *request* message with parameter t_d to the instances of lock protocol on vehicles b and c through communication channel. It records t_d to be the deadline when the cooperating group is being dismissed. After becoming master, the instance of lock protocol ignores all *request* message it receives before time t_d.

If vehicles b and c are not being in any cooperating group, they become *slave* locks when receiving *request*. Each of them records parameter t_d that comes along with *request* message, then it responds to *master* lock with message *granted*. The slave lock initiate necessary function of the driver-assisted merge protocol for cooperation. After becoming slave, the instance of lock protocol also ignores all *request* message it receives before time t_d.

The master expects *granted* messages from both of the slaves after it sends out *request* messages. After successfully collecting *granted* from the two slaves, it knows that both vehicles b and c are cooperating until t_d. It now notifies the merge protocol on vehicle a that the cooperating group is formed and the necessary functions of the merge protocol on vehicles b and c are initiated.

If any message is lost during a transmission, the master will not be able to collect *granted* messages from both slaves. When this occurs, the lock protocol instance on either vehicle b or c may or may not become slave. The use of deadline t_d ensures that, if an instance becomes slave but the group is not formed successfully, every participant returns to the initial state after t_d.

5.1.3 Specification in Timed Automata

Time-critical systems are often modeled by timed automata. Here we attempt to specify the behavior of the lock protocol using UPPAAL model. The lock protocol modeled as an UPPAAL model is shown in Fig.5.2. The diagram here represents an instance the lock protocol on a single vehicle. The complete system contain the model of communication channels in addition to the model of lock protocol instance. An input of verification contains the UPPAAL models of each component in the system. The toolbox carries out verification by composing the component models and check the system against specified properties.

It is worth mentioning the two variables in the model, namely x and td. Clock variable x is initialized to zero, which denotes the accurate clock that the lock protocol instance can access. Parameter td specifies the time by when driver-assisted merge protocol expects to complete the merge maneuver. The cooperating group should persist until the specified time td.

The edges of the automaton are decorated with three types of labels.

1. *Guard* expresses a condition on the values of clock variable that must be satisfied in order for the edge to be taken. The edge from *wait_both* state to the *start* state has guard $x > td$, meaning that the edge is taken only when the clock variable x reads a value greater than td.

2. *Synchronization action* is performed when the edge is taken. Synchronization actions the one appended with either an exclamation mark or a question mark. Synchronization actions pair up in a way similar to handshaking: a transmitting action with a exclamation mark ending occurs simultaneously as a receiving action of the same

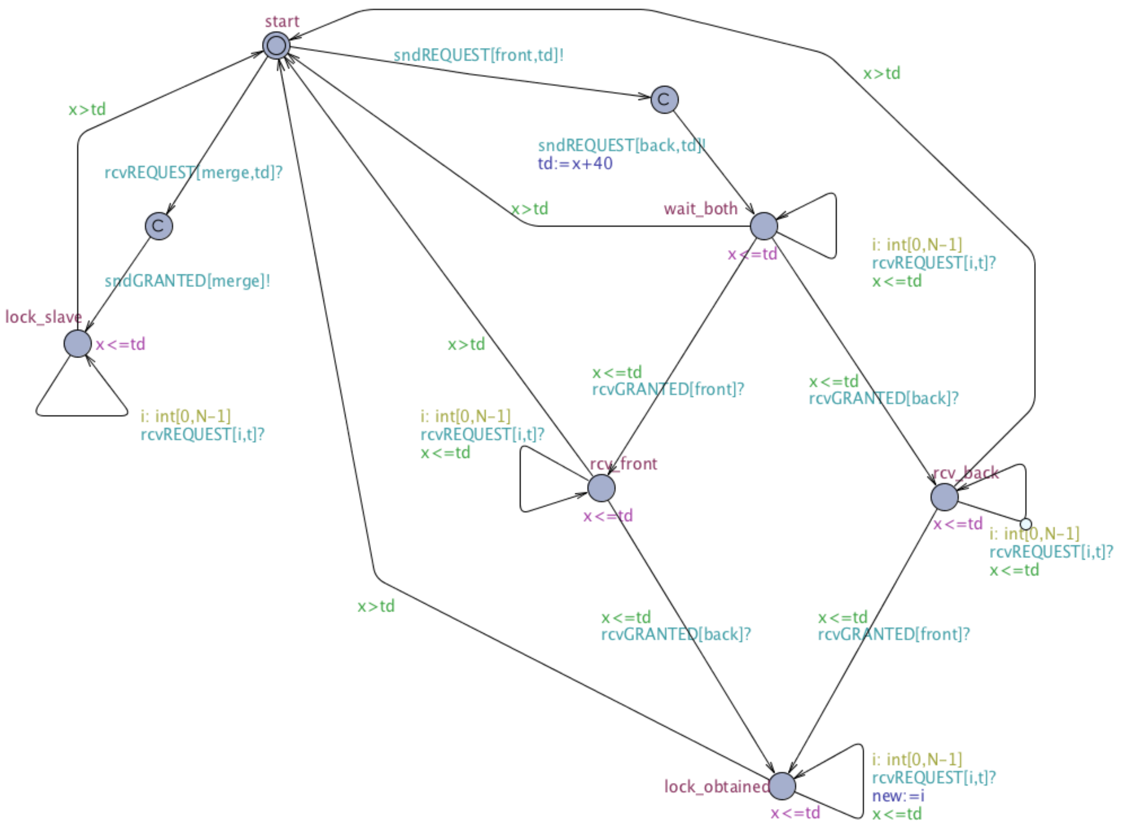


Figure 5.2: Specification of the Lock Protocol in UPPAAL model

name but with a question mark ending. When combining several UPPAAL models, a pair of synchronization actions cause transitions of two models together.

3. A number of *clock resets* and *clock assignments* to clock variable. The only type integer reset/assignment appears within the lock protocol model is $td := x + 40$, which sets the value of deadline t_d to 40 seconds later than current time x. Some of the states of the automaton come with *invariants*, which are conditions expressing constraints on the clock values in order for control to remain in a particular state. The combination of an invariant $x <= td$ and a guard $x > td$ on an edge captures the event of returning to the initial state when the clock reads t_d.

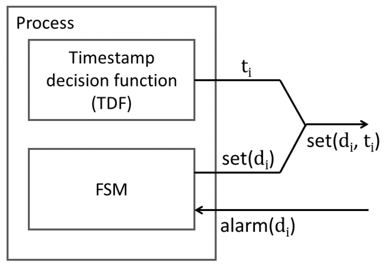
Two states with latin letter ‘C’ on them are *committed locations*. Committed location is a syntax used by UPPAAL modeling language that ensure that atomicity of several transitions. If a system component arrives at some committed location, on the next transition it must leave that committed location. Other component cannot interfere. This is similar to the chain of transitions when we consider the composition of FSMs and service models in Chapter 3. The committed location on the right ensures that *request* messages are put into the channel toward the two neighboring vehicles atomically, and the one on the left ensures that the instance of the lock protocol responds with *granted* message on receiving *request* message.

In general, *td* may take arbitrary value, as driver-assisted merge protocol needs different amount of time to complete a merge maneuver for different road condition, vehicle speed, etc. Accounting for the behavior of a protocol when different *td*'s are supplied requires the use of timed automata. That is, timed automata account for different behaviors of a system when different timing constraints are provided.

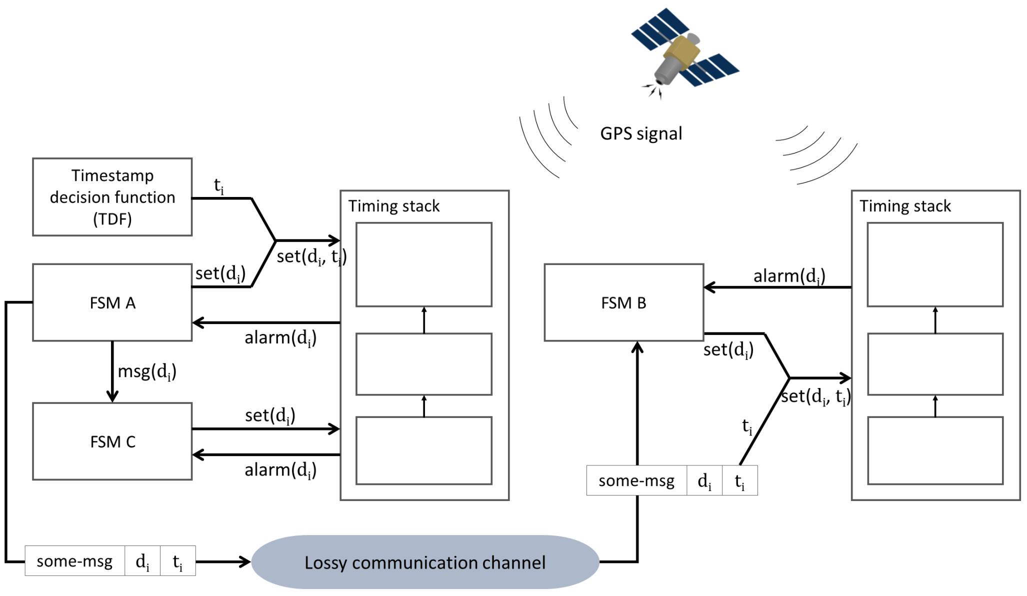
In the next section, we introduce an interface for the timing stack, with which we separate the timing rules to a different process from the logic of message exchange between instances of lock protocol. The logic stays the same without having to account for the timing. Lock protocol, or any protocol that requires timing, may operate according to the accurate clocks provided by the timing stack. We shall see shortly that this simplifies protocols, allows them to be specified using a simpler and less expressive model — FSMs.

5.2 Revisit the Timing Stack

In this section, we revisit the timing stack that has been introduced in Chapter 4. This time we focus on the interface of timing stack as seen from the perspective of FSMs, and its implication. First of all, as shown in Fig.5.3a, we separate a process into two parts: the part that is modeled as an FSM controls state transitions and messages exchanges, and the part called timestamp decision function (TDF) determines the time instant that a timed event should occur. The setup of timed events are replaced by sending a message from the FSM to the timing stack. The occurrence of a timed event is triggered by an 'alarm' message generated by the timing stack. FSMs are adequate for modeling systems with synchronized clocks with a small modification. The lock protocol can then be specified with this newly conceived model.



(a) Divide a process into TDF and FSM



(b) Schedule timed event that occurs at different locations

Figure 5.3: Interacting with the Timing Stack

5.2.1 Interface of Timing Stack

In Chapter 4 we have seen that with GPS, crystal oscillators, and synchronization protocols, a reasonably accurate clock is maintained within the timing stack. On each physical entity, e.g. an intelligent vehicle, there is a timing stack. It ensures that this accurate clock is readily available to all modules co-located with the timing stack.

For the ease of use of this accurate clock, the timing stack provides interface from which instances protocol access timing information. Recall that in the acknowledgment protocol example in Chapter 3, the transmitter controls timing by sending command to the timer. Similarly, suppose a protocol wishes an event to take place at some future time t, it submits timestamp t to the timing stack. By the time the clock reads t, the timing stack dispatches a message to the protocol instance, which executes the action accordingly.

The timing stack maintains a list of pending timestamps and remembers which module in an intelligent vehicle submitted each timestamp. Whenever the clock reads the value of a timestamps it stores, it notifies the *registrant* of the timestamps, that is, the module which submitted the timestamp in the first place.

There are two messages reserved for timing stack. Message $set(d_i, t_i)$ is a message that a protocol instance uses to submit a timestamp of value t_i and to assign the timestamp with identifier d_i. Message $alarm(d_i)$ is generated by the timing stack indicating that the clock now reads the value of the timestamp of identifier d_i.

5.2.2 Finite-State Machines with Timestamps

A separate timing stack relieves a process of the responsibility of handling timing. The timing stack should be implemented on the same physical location as the process itself. As long as we are only interested in the temporal properties rather than timed properties, we distinguish different timestamps by their identifiers instead of their values. Here we introduce a small modification to the original FSM definition. Unlike timed automata, in modified FSMs the time of occurrence of events is no longer a concern.

There are two ways to submit timestamps to the timing stack. The first being submitting a value determined by a TDF. For instance, in driver-assisted merge protocol, TDF decides how long the merge maneuver is expected to complete by considering weather, road condition, highway congestion, and etc. Another way is to receive a value through handshaking or communication channel from the process whose TDF has determined that value, and submit that value to the timing stack.

Several FSMs may coordinate simultaneous actions using synchronous clocks and timestamps. An example is shown in Fig.5.3b. First, FSM A wishes to schedule an events that should take place simultaneously at different locations. It submits a timestamp d_i to the timing stack, with its value t_i supplied by its accompanying TDF. The value of the timestamp is distributed to FSM B that is located at a different location, through a message that includes the identifier d_i and the value t_i. On receiving the timestamp, FSM B submits the timestamp d_i to its respective timing stack, with its value t_i obtained from the message. FSM C is also prompted to submit the same timestamp by handshaking, however, it does not have to submit the value because it shares the same timing stack with FSM A. When the clock reads t_i, the timing stacks on different physical entities will send *alarm* messages to FSMs A, B, and C at the same time. When FSMs A, B, and C receive the alarm messages, they execute the actions that have been associated with timestamp d_i.

After a the value of a timestamp is determined, it never changes. Therefore, when constructing the model of protocol systems, the timestamps are distinguished by its identifier. The same value is used by the FSM whose TDF has decided its value and the FSMs that receive this timestamp by means of handshaking or communication channels. The actual implementation of the FSM should provide a mechanism to pass the values of timestamps around. That is, embedding d_i in a message being sent through a communication channel requires its value t_i to be included in the same message.

We aim for minimal modification to the FSM model. The timestamp may take arbitrary value that represents some instance since a specified epoch (January 1, 1970 for UNIX-based system). Reasoning about the system by consider the identifier of the timestamp rather than its value preserves the finiteness of FSM model.

The following messages are added to the set of inputs and the set of outputs of a FSM. They stand for interaction with the timing stack.

- Inputs: there are *alarm* messages generated by the timing stack. They are of the form $G_{\text{Timing}}?alarm(d_y)$, which represents the alarm message sent from the timing stack that indicates that the clock reads the value of timestamp dy

- Inputs: there are messages that have identifier(s) of timestamp(s) embedded as their parameter. They are of the form $F_j?msg(d_y)$ and $F_j?msg(d_{y,\text{passed}})$. The former stands for receiving a timestamp that has not passed yet, while the latter stands for receiving an outdated timestamp.

- Outputs: there are messages used to set up timestamps that expire at some later time. They are of the form $G_{\text{Timing}}!set(d_y)$, which register the FSM as a recipient of alarm message for dy.
- Outputs: there are messages that have identifier(s) of timestamp(s) embedded as their parameter. They are of the form $F_i!msg(d_y)$

The output labels $G_{\text{Timing}}!set(d_y)$ should comply to the two ways of submitting values of timestamps to the timing stack. If the FSM has an accompanying TDF that supplies the value for the timestamp, then the timing stack should require that the identifier d_y is either used the first time or that the clock has passed the last instance of d_y.

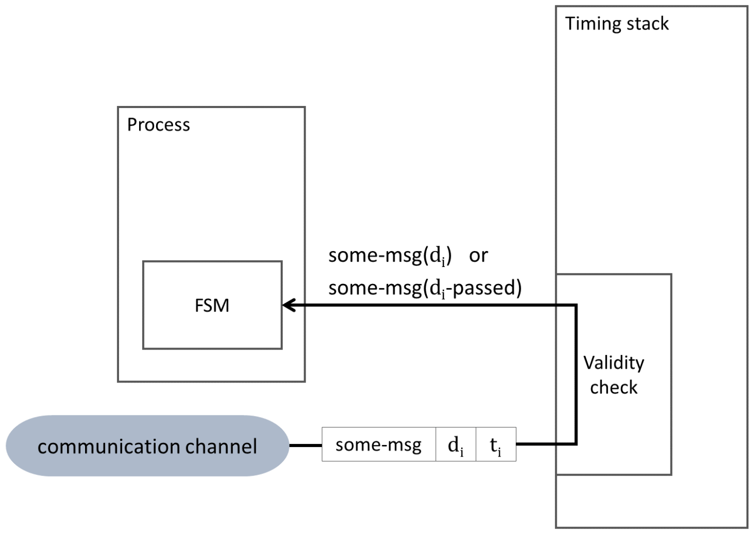


Figure 5.4: Validity Check when Receving Messages Containing Timestamp(s)

If on a transition the implementation of the FSM is expected to receive the timestamp from others then submit that timestamp to the timing stack, the input label of the transition that the label $G_{\text{Timing}}!set(d_y)$ is on should be a label that receives a message containing the identifier dy, e.g. $G_{\text{channel}}?msg(d_y)$ or $F_j!msg(d_y)$. We require that, in implementation of a FSM with timestamp, the timestamp delivered by a message sent through a delay-prone communication channel is first checked for its validity, as shown in Fig.5.4. That is, its value is compared to the current clock reading to decide if it has an outdated value. This is because a message that delivers the value of some timestamp di may arrive after di has already expired. The validity check is part of the timing stack's functions and it lets the implementation of a FSM knows whether a timestamp is valid (value less than current time) or invalid (value greater than or equal to current time). In every state of a fully-specified FSM, it is required that the input labels that receives an outdated timestamp, e.g. $G_{\text{channel}}?msg(d_{y,\text{passed}})$, are specified.

The definition circumvents around the complexity of considering actual timestamp values, which requires models to be constructed using timed automata or other timed models. The mechanism for determining the timestamps is transparent to the protocol designer if we only look at the FSMs. The FSMs would be sufficient in modeling the system as long as we are interested in the sequences of events rather than the timing of the events.

5.2.3 Specification of Lock Protocol

The system includes processes of lock protocol that exchange messages through communication channels. On each intelligent vehicle there can be more than one process of lock protocol, with each process controlling the access of a specific resource. We do not place any assumption on the communication channel, and a minimal best-effort communication protocol would suffice.

The diagram in Fig.5.5 is the specification of a lock protocol process with identifier i, or lock i. Any input that is not in the diagram is ignored by the FSM, that is, has an empty output. A process becomes a master lock when it moves from s_0 to s_2, and during its stay in states s_2, s_3, s_3' and s_4; it becomes a slave lock when it is in state s_1. Being in state s_0 means that this instance is not cooperate with anyone nor has attempted to create a group.

Note that the diagram is a compact representation of the FSM $F_{\text{lock,i}}$. An edge from s_0 to s_1 represents multiple transitions. Here, m is a variable, and there are distinct transitions triggered by $G_{\text{channel,m}}$ for any process identifier m. When receiving *request*(d_m) from $F_{\text{lock,m}}$, it submits timestamp d_m and responds by sending back *granted*(d_m). The process then notifies the necessary function of driver-assisted merging application that it should start cooperate until timestamp d_m ($F_{\text{front}}!$*cooperate*(d_m)). The FSM does not leave state s_1 until the timing stack delivers *alarm*(d_m). It is important to keep in mind that if

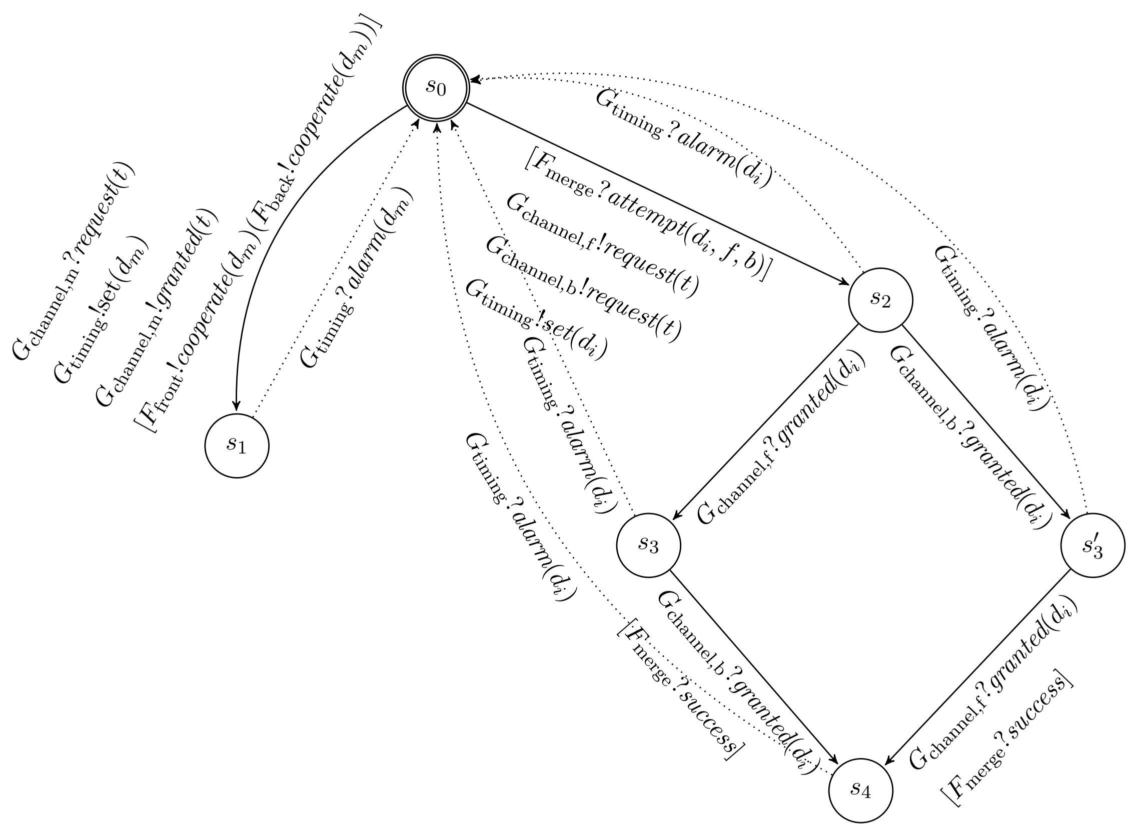

s_0
s_1
s_2
s_3
s'_3
s_4
$G_{\text{channel,m}}?request(t)$
$G_{\text{timing}}!set(d_m)$
$G_{\text{channel,m}}!granted(t)$
$[F_{\text{front}}!cooperate(d_m)(F_{\text{back}}!cooperate(d_m))]$
$G_{\text{timing}}?alarm(d_m)$
$G_{\text{timing}}?alarm(d_i)$
$[F_{\text{merge}}?attempt(d_i, f, b)]$
$G_{\text{channel,f}}!request(t)$
$G_{\text{channel,b}}!request(t)$
$G_{\text{timing}}!set(d_i)$
$G_{\text{timing}}?alarm(d_i)$
$G_{\text{timing}}?alarm(d_i)$
$G_{\text{timing}}?alarm(d_i)$
$G_{\text{channel,f}}?granted(d_i)$
$G_{\text{channel,b}}?granted(d_i)$
$G_{\text{channel,b}}?granted(d_i)$
$[F_{\text{merge}}?success]$
$G_{\text{channel,f}}?granted(d_i)$
$[F_{\text{merge}}?success]$

Figure 5.5: Specification of the lock protocol, id= i ($F_{\text{lock,i}}$)

any message with an identifier d_m in its parameter is sent through communication channel, the implementation should include its value t_m in the message being transmitted.

The FSM moves from state s_0 to s_2 if driver-assisted merge protocol attempt to create a cooperating group with the front car and the back car ($F_{\text{merge}}?attempt(d_i, f, b)$). Parameter d_i refers to the timestamp whose value denotes the time until when the cooperating group should last. Parameters f and b are the identifiers of the lock process on the front car and the back car, respectively. On the transition it sends out *request*s toward processes f and b, and submits timestamp d_i. The identifier i of the process will be used by the slave locks on the front car and the back car to decides which process they are currently cooperating with.

In states s_2, s_3 and s'_3, it collects *granted*(d_i) message from processes f and b and moves

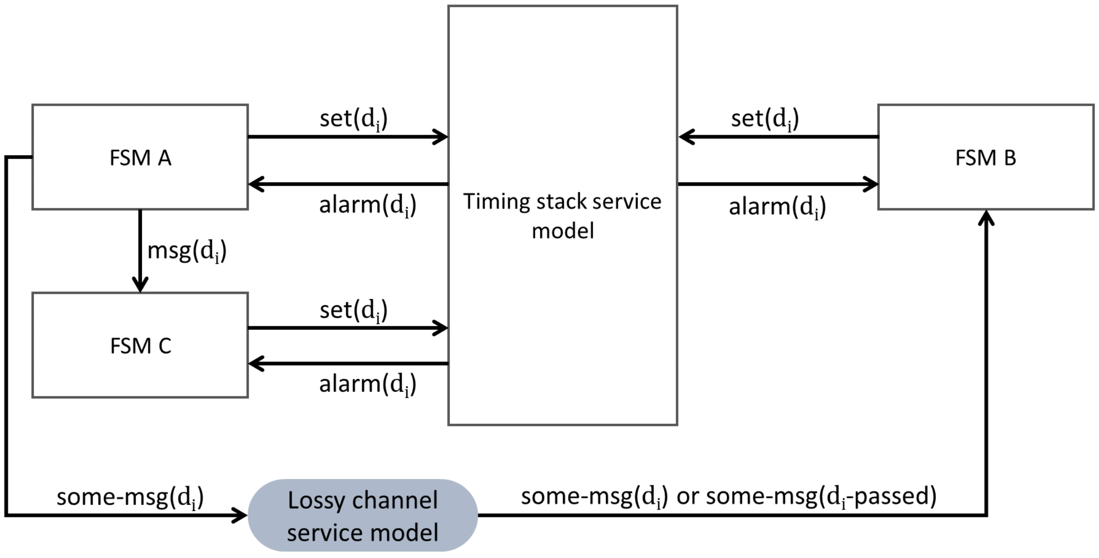

FSM A
set(d_i)
alarm(d_i)
Timing stack service model
set(d_i)
alarm(d_i)
FSM B
msg(d_i)
FSM C
set(d_i)
alarm(d_i)
some-msg(d_i)
Lossy channel service model
some-msg(d_i) or some-msg(d_i-passed)

Figure 5.6: The System Appears to Have a Single Stack

into s_4 after collecting both. If it fails to collect messages from both f and b before d_i expires, it returns to the initial state. Otherwise it proceeds to s_4 and notifies the higher level function that the cooperating group is formed ($F_{\text{merge}}!success$). The FSM stays in state s_4 until d_i expires.

5.3 Modeling of Protocol Systems with Timing Stacks

In this section, we present the service model of the timing stacks so that they are included in the MDP model construction of the protocol system. Although in a system there are multiple timing stacks at different physical locations, in the models of protocol systems the timing stack appears as a single service model as a result of synchronization of the clock readings.

The fact that every timing stack is synchronized to the GPS clocks and that the alarm messages are delivered to all FSMs that have submitted the value of the same timestamp creates an 'illusion' that a single timing stack distributes this alarm message to every FSM in the system. Logically, the system appears as illustrated in Fig.5.6.

The timing stack model, which is the logical illusion of multiple timing stacks at separate locations, maintains a list of identifiers of pending timestamps. The physical implementation of timing stack is responsible for storing their values. However, the timing stack model only keeps track of the set of FSMs that have submitted the value of the same timestamp. For example, FSM A is the first FSM that submits a value for timestamp d_i. FSM B and FSM C submits the same value for timestamp d_i after they receive timestamps from FSM A. The timing stack model should associate identifier d_i with FSMs A, B, and C so that it is able to distribute alarm messages to the three FSMs on the same transition.

Suppose that there are at most k distinct timestamps in the system. For each timestamp d_i, the state of the timing stack model should keep track of which FSMs that have submitted d_i, even though in effect it is the TDF on one of them that decides its value. Consequently, the state of the timing stack model could be written as a set of k key-value pairs with each key corresponding to a timestamp and the value being a set of FSMs that are its registrants. A state of the timing stack model takes the form of

$$[d_1 : D_1, d_2 : D_2, \ldots, d_k : D_k] \tag{5.1}$$

where $D_1, D_2, \ldots, D_k$ are the set of FSMs that have been registered with timestamps $d_1, d_2, \ldots, d_k$, respectively. Initially, $D_j = \phi$ for all $1 \leq j \leq k$. Set D_j being empty means that there has not been anyone submitted timestamp d_j. In F, initially the state of the timing stack is $[d_i : \{\}]$. When FSM A submits timestamp d_i, its accompanying TDF determines the value of d_i. FSM A then passes $msg(d_i)$ to FSM C by handshaking, which prompts FSM C to submit d_i as well. Now the timing stack model is in the state $[d_i : \{A, C\}]$.

As we remove timing information, there is no relationship between different deadlines, such as time difference, ordering, etc.. Set D_j being non-empty simply implies that timestamp d_j and its value have been submitted sometime earlier. Timestamp d_j expiring is then a valid event, and it incurs sending alarm message to all the FSMs in the set D_j. If both D_j and D'_j are non-empty, both d_j expiring and d'_j expiring are valid events as we did not place any restrictions on the values of d_j and d'_j. If there are more than one sets among $D_1, \ldots, D_k$ being non-empty, then expirations of their associated timestamps are valid events.

Assume that, in G, the timing stack model is in state $[d_i : \{A, C\}]$ when *somemsg*(d_i) is in transit from FSM A toward FSM B. In this case d_i expiring is valid. In such event, *alarm*(d_i) is sent to FSM B and FSM C on the same transition while FSM B has not received timestamp d_i yet. When *somemsg*(d_i) is delivered, timestamp d_i is checked for validity. FSM B then knows whether the received timestamp d_i has expired.

5.4 Transition Rules

The timestamps allow physically-separated entities to execute actions at the same time. This is stronger than the two-way handshaking in the composition rules described in Chapter 3. The timing stack model adds additional rules that are directly related to the setup of timestamps and the expirations of timestamp. In this section, we discuss the compatibility of FSMs with timestamps, then introduce three more composition rules to constructing the model of protocol systems. Finally, execution sequences of a 4-process system of the lock protocol are used as examples of transition rules at work.

5.4.1 Compatibility of FSMs with Timestamps

Compatibility of FSMs with timestamps require that each timestamp in the composite system is determined uniquely by a single TDF accompanying one of the constituting FSMs. This makes it impossible for a timestamp to have more than one value.

Suppose on the contrary that there are two FSMs, F_1 and F_2, and each of them has TDF that computes the value for the same timestamp d_i. Suppose F_1 submits t_1 to be the value of d_i, while F_2 submits t_2 to be the value of d_i at the same time. Both F_1 and F_2 pass on their determined values of d_i to a third FSM F_3, while F_3 submits the value of d_i accordingly when it receives the assigned value of d_i from either F_1 or F_2. This creates an ambiguity: we do not know the value of timestamp submitted by F_3 can be either t_1 or t_2.

To solve this problem, we define ownership of timestamps.

Definition 8 *Timestamp d_i is said to be owned by a FSM F iff*

1. *F is accompanied by a TDF that determines the value for d_i.*

2. *F does not have any receiving input labels for messages that contain d_i in its parameter.*

3. *F has an utput label of $G_{\text{timing}}!set(d_i)$ on at least one of its transitions*

We define $D(F)$ to be the set of timestamps owned by FSM F.

With ownership of timestamp, we may then consider the compatibility of FSMs.

Definition 9 *FSMs $F_1, \ldots, F_n$ with timestamp are said to be compatible if*

$$D(F_i) \cap D(F_j) = \phi, \ \forall \ i \neq j \tag{5.2}$$

If all FSMs are compatible, every timestamp has a single owner FSM. The timestamps in a transition system that consists of compatible FSMs are only assigned values by their respective owners. Hence, any timestamp will not be assigned different values before it expires.

5.4.2 Transition rules imposed by the timing stack model

A timestamp d_i become relevant when is is first submitted to the timing stackby the FSM that owns d_i. Its value is determined by the TDF. At this point, no other FSM has registered itself with d_i, otherwise the FSMs in the system model must be incompatible. Hence we have the first rule:

> The first FSM that execute a transition containing output label $G_{\text{Timing}}!set(d_i)$ should be the FSM which owns d_i.

The FSMs that do not own d_i may submit d_i to the timing stack only after it receives d_i in a message that can be attributed to the FSM that owns it. Submission of d_i only takes place on a transition that represents reception of d_i. Here goes the second rule:

> FSMs that do not own d_j may register itself with d_j only after the owner of d_j has its TDF determine and submit the value of d_j; moreover, it may only do so on the transition on which it receives identifier d_j contained in some message

If on a transition an outdated value assignment is received, e.g. $F_x?msg(d_{j,\text{passed}})$, then the value is behind current time and the FSM should not submit it to the timing stack.

As seen in the previous section, when considering a single timestamp d_j, several timing stacks operate as a logically single entity and deliver $alarm(d_j)$ to the FSMs that have submitted the value of d_j. We have also seen that it is possible for the message bearing timestamps to be delayed after the clock has passed the timestamp. When a protocol implementation receives a message that has a timestamp among its parameters, it always does a validity check on the timestamp before execute any transitions. An input to an FSM is always an identifier of a timestamp, e.g. d_j, or an identifier of an outdated timestamp, e.g. $d_{j,\text{passed}}$. Equivalently, when a timestamp expires while there are timestamp-bearing messages in transit, we mark the parameter in these messages as *passed* when the message is in transit. We summarize the discussion above as the third rule:

> The expiration of timestamp d_j is a valid event after timestamp d_j is submitted, i.e., D_j is non-empty. All messages in communication channel that have identifier d_j in its parameter are marked as passed after the transition

Let $t_s = [d_1 : D_1, \ldots, d_k : D_k]$ be the state of the timing stack model. We consider the states of channels $c_1, c_2, \ldots, c_m$ separately from the states of FSMs and service models $s_1, \ldots, s_n$. Rule 1 becomes

$$\frac{d_i \in D(F_x) \wedge D_i = \phi \wedge F_x : s_x \xrightarrow{a/b} s'_x \wedge G_{\text{timing}}!set(d_i) \in b}{\langle s_1 \ldots, s_x, \ldots, s_n, c1 \ldots, c_m, t_s\rangle \rightarrow \langle s_1 \ldots, s'_x, \ldots, s_n, c1 \ldots, c_m, t'_s\rangle} \tag{5.3}$$

where $t'_s = [d_1 : D'_1, \ldots, d_k : D'_k]$ and $D'_j = D_j$ for all $j \neq i$, $D'_i = \{F_x\}$.

If the owner FSM of d_i is executing some transition whose output label contains $G_{\text{timing}}!set(d_i)$ while D_i is non-empty, this implies that the owner FSM is attempting to overwrite the timestamp of d_i, which has been determined earlier. This is considered a design flaw and should be fixed.

Rule 2 becomes

$$\frac{d_i \notin D(F_x) \wedge D_i \neq \phi \wedge F_x : s_x \xrightarrow{G_{\text{channel,j}}?msg(d_i)/b} s'_x \wedge G_{\text{timing}}!set(d_i) \in b}{\langle s_1 \ldots, s_x, \ldots, s_n, c1 \ldots c_j, \ldots, c_m, t_s\rangle \rightarrow \langle s_1 \ldots, s'_x, \ldots, s_n, c1 \ldots, c'_j, \ldots, c_m, t'_s\rangle} \tag{5.4}$$

where $t_s = [d_1 : D_1, \ldots, d_k : D_k]$ and $D_j = D'_j$ for all $j \neq i$, $D'_i = D_i \cup \{F_x\}$, and $c'_j = c_j \setminus msg(d_i)$.

Rule 3 becomes

$$\frac{D_i \neq \phi \wedge \forall F_x \in D_i \cdot F_x : s_x \xrightarrow{G_{\text{timing}}?alarm(d_i)/b} s'_x}{\langle s_1 \ldots, s_x, \ldots, s_n, c1 \ldots c'_m, t_s\rangle \rightarrow \langle s_1 \ldots, s'_x, \ldots, s_n, c1', \ldots, c_m, t'_s\rangle} \tag{5.5}$$

where $D_j = D'_j$ for all $j \neq i$, $D'_i = \phi$. $s'_y = s_y \forall y \notin D_i$ and F_x changes state from s_x to s'_x if $x \in D_i$. If there is any message in transit that have d_i among its parameters, the parameter d_i should be marked as *passed*: if $msg(d_i) \in c_y$ for some y and any msg, then $c'_y = (c_y - msg(d_i) \cup msg(d_{i,\text{passed}})$; if otherwise c_y does not contain any message with parameter d_i, $c'_y = c_y$.

5.5 Example: Execution Sequences of the Lock Protocol

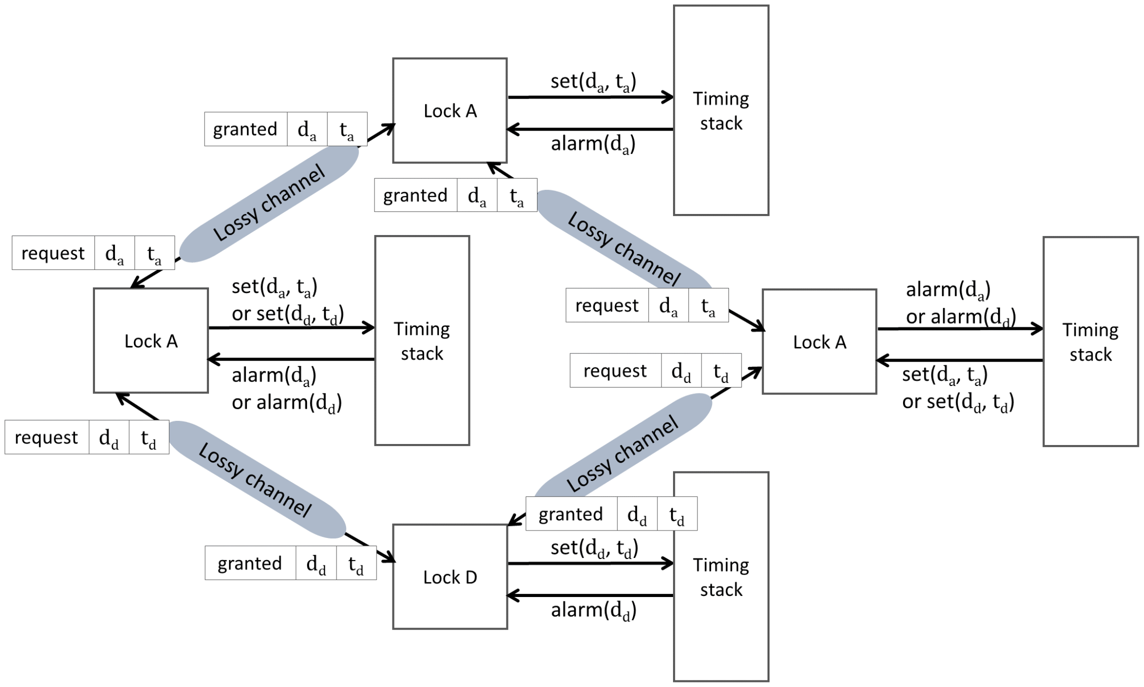
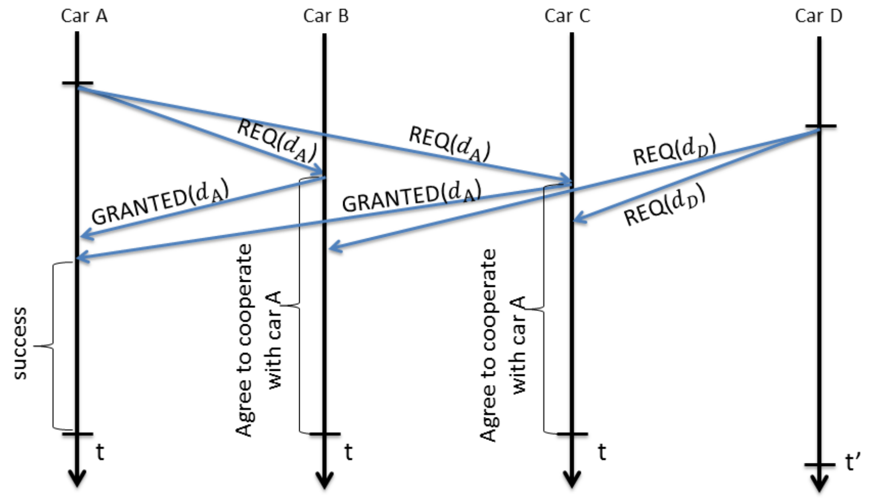
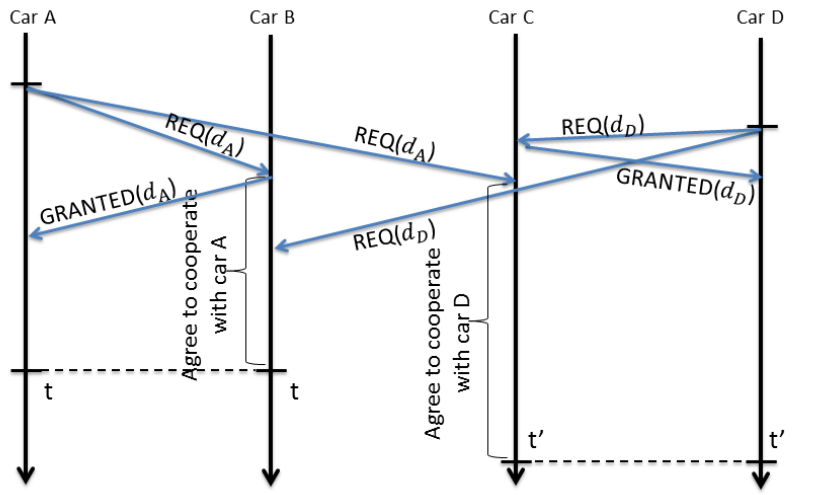


Figure 5.7: Four lock protocol processes on four vehicles

We see the transition rules in action through the following example. There are four different vehicles in Fig.5.7, and each of them is equipped with an implementation of the lock protocol and a timing stack. Two among the lock protocol implementation, namely FSM A and FSM D, issue competing requests contains different timing requirements. FSM A attempts to create a group by soliciting cooperation from FSM B and FSM C, while FSM D also attempts to create a group by soliciting cooperation from FSM B and FSM C as well. We immediately see that at most one of FSM A and FSM D and it cannot be both.



(a) Car A succeeds



(b) No one succeeds

Figure 5.8: Competing Requests of Lock Protocol

FSM A owns timestamp d_A. Its accompanying TDF determines its value. The value of d_A is then piggybacked on the request messages sent toward FSM B and FSM C. The same goes to FSM D, which owns timestamp d_D.

The TDF can be part of the higher level function that relies on the lock protocol to solve conflicting request and create mutual exclusive group. The value of the timestamp may be decided based on purpose of the higher level function. The value of d_A and the value of d_D do not relate to each other. We should consider the case where the value of d_A could be less than the value of d_D, and vice versa.

If the requests from FSM A reach FSM B and FSM C before the requests from FSM D do, FSM B and FSM C choose to operate using the timestamp d_A. FSM A succeeded to create a group, while FSM D failed and had to wait. It is also possible that none of FSM A and FSM D succeeds: if the request from FSM A reaches FSM B first and the request from FSM D reaches FSM C first, FSM B operate according to timestamp d_A while FSM C uses timestamp d_C. The timeline inFig.5.8 depicts such scenario:

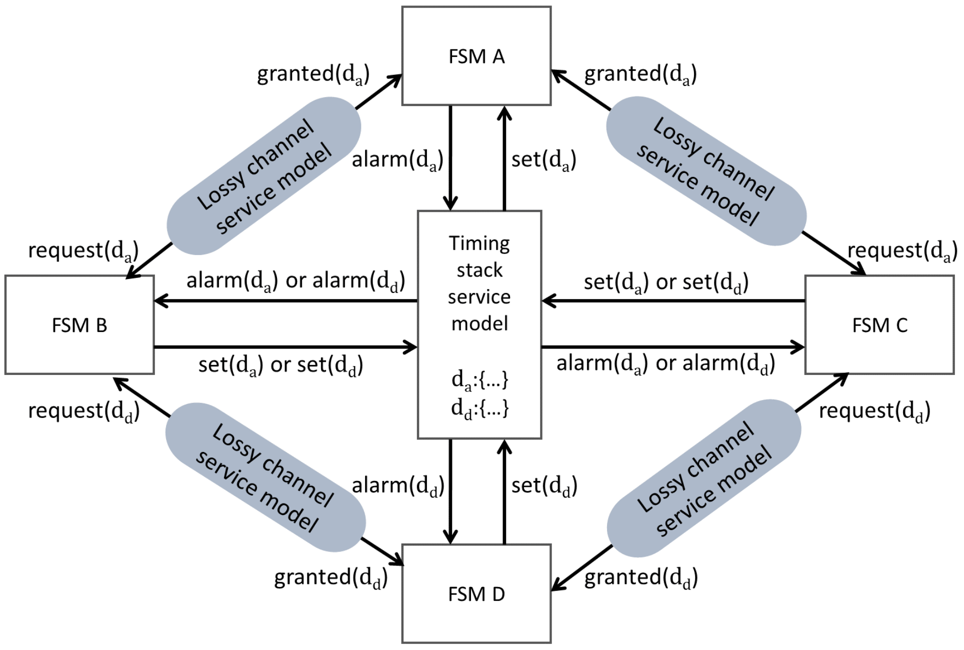


Figure 5.9: Logical View of the Four-Instance System

To construct the composite machine for verification, we consider the logical view (Fig.5.9) of the system that consists of four FSMs, communication channels connecting the FSMs, and a single timing stack model. They are the constituting modules of the composite machine. The execution sequences of the composite machine are the possible evolutions of the whole system.

To see the transition rules in action, we consider the sequence that reflects the timeline we just saw. As lock B agrees to cooperate with lock A, it uses timestamp d_A. Lock C agrees to cooperate with lock D, so it becomes registered with timestamp d_D. The sets of registrants keyed by d_A and d_D list the FSMs to which the timing stack should send alarm messages when d_A and d_D expires.

$$
\begin{aligned}
&\langle F_A, F_B, F_C, F_D, G_{A\to B}, G_{A\to C}, G_{B\to A}, G_{C\to A}, G_{D\to B}, G_{D\to C}, G_{B\to D}, G_{C\to D}, G_{\text{timing}}\rangle :\\
&\langle s_0, s_0, s_0, s_0, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, [d_A : \{\}, d_D : \{\}\rangle\\
&\quad \to \langle s_2, s_0, s_0, s_0, \mathit{req}(d_A), \mathit{req}(d_A), \phi, \phi, \phi, \phi, \phi, \phi, [d_A : \{F_A\}, d_D : \{\}\rangle\\
&\quad \to \langle s_2, s_0, s_0, s_2, \mathit{req}(d_A), \mathit{req}(d_A), \phi, \phi, \mathit{req}(d_D), \mathit{req}(d_D), \phi, \phi, [d_A : \{F_A\}, d_D : \{F_D\}\rangle\\
&\quad \to \langle s_2, s_1, s_0, s_2, \phi, \mathit{req}(d_A), \mathit{gra}(d_A), \phi, \mathit{req}(d_D), \mathit{req}(d_D), \phi, \phi, [d_A : \{F_A, F_B\}, d_D : \{F_D\}\rangle\\
&\quad \to \langle s_2, s_1, s_1, s_2, \phi, \mathit{req}(d_A), \mathit{gra}(d_A), \phi, \mathit{req}(d_D), \phi, \phi, \mathit{gra}(d_D),\\
&\qquad\quad [d_A : \{F_A, F_B\}, d_D : \{F_C, F_D\}\rangle\\
&\quad \to \langle s_2, s_1, s_1, s_2, \phi, \phi, \mathit{gra}(d_A), \phi, \mathit{req}(d_D), \phi, \phi, \mathit{gra}(d_D), [d_A : \{F_A, F_B\}, d_D : \{F_C, F_D\}\rangle\\
&\quad \to \langle s_2, s_1, s_1, s_2, \phi, \phi, \mathit{gra}(d_A), \phi, \phi, \phi, \phi, \mathit{gra}(d_D), [d_A : \{F_A, F_B\}, d_D : \{F_C, F_D\}\rangle\\
&\quad \to \langle s_3, s_1, s_1, s_2, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \mathit{gra}(d_D), [d_A : \{F_A, F_B\}, d_D : \{F_C, F_D\}\rangle\\
&\quad \to \langle s_3, s_1, s_1, s_2, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, [d_A : \{F_A, F_B\}, d_D : \{F_C, F_D\}\rangle\\
&\quad \to \langle s_0, s_0, s_1, s_2, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, [d_A : \{\}, d_D : \{F_C, F_D\}\rangle\\
&\quad \to \langle s_0, s_0, s_0, s_0, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, [d_A : \{\}, d_D : \{\}\rangle
\end{aligned} \tag{5.6}
$$

Suppose that *request*(d_A) and *request*(d_D) suffer delay in channel from FSM A to FSM C and the channel from FSM D to FSM C, respectively, transition rule 3 requires that the message to be marked ‘passed’ when deadlines expire:

$$
\begin{aligned}
&\langle F_A, F_B, F_C, F_D, G_{A\to B}, G_{A\to C}, G_{B\to A}, G_{C\to A}, G_{D\to B}, G_{D\to C}, G_{B\to D}, G_{C\to D}, G_{\text{timing}}\rangle : \\
&\langle s_0, s_0, s_0, s_0, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, [d_A : \{\}, d_D : \{\}\rangle \\
&\quad \to \langle s_2, s_0, s_0, s_0, \mathit{req}(d_A), \mathit{req}(d_A), \phi, \phi, \phi, \phi, \phi, \phi, [d_A : \{F_A\}, d_D : \{\}\rangle \\
&\quad \to \langle s_2, s_0, s_0, s_2, \mathit{req}(d_A), \mathit{req}(d_A), \phi, \phi, \mathit{req}(d_D), \mathit{req}(d_D), \phi, \phi, [d_A : \{F_A\}, d_D : \{F_D\}\rangle \\
&\quad \to \langle s_2, s_1, s_0, s_2, \phi, \mathit{req}(d_A), \mathit{gra}(d_A), \phi, \mathit{req}(d_D), \mathit{req}(d_D), \phi, \phi, [d_A : \{F_A, F_B\}, d_D : \{F_D\}\rangle \\
&\quad \to \langle s_3, s_1, s_0, s_2, \phi, \mathit{req}(d_A), \phi, \phi, \mathit{req}(d_D), \phi, \phi, \mathit{gra}(d_D), [d_A : \{F_A, F_B\}, d_D : \{F_D\}\rangle \\
&\quad \to \langle s_0, s_0, s_0, s_2, \phi, \mathit{req}(d_{A,\text{passed}}), \phi, \phi, \mathit{req}(d_D), \phi, \phi, \mathit{gra}(d_d), [d_A : \{\}, d_D : \{F_D\}\rangle \\
&\quad \to \langle s_0, s_0, s_0, s_2, \phi, \phi, \phi, \phi, \mathit{req}(d_D), \phi, \phi, \mathit{gra}(d_d), [d_A : \{\}, d_D : \{F_D\}\rangle \\
&\quad \to \langle s_0, s_0, s_0, s_0, \phi, \phi, \phi, \phi, \mathit{req}(d_{D,\text{passed}}), \phi, \phi, \mathit{gra}(d_{D,\text{passed}}), [d_A : \{\}, d_D : \{\}\rangle \\
&\quad \to \langle s_0, s_0, s_0, s_0, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \mathit{gra}(d_{D,\text{passed}}), [d_A : \{\}, d_D : \{\}\rangle \\
&\quad \to \langle s_0, s_0, s_0, s_0, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, [d_A : \{\}, d_D : \{\}\rangle
\end{aligned}
\tag{5.7}
$$

5.6 Conclusions

In this chapter, we presented a framework for the protocols to use GPS clocks as a way to coordinate synchronous actions. It is possible to separate timing from the logic of state transitions by relegating timing management to a separate timing stack, which also maintains an accurate clock. Without considering timing constraint, the logic can be described by using only FSMs. Furthermore, having actions executed synchronously even when they are separated by distance eliminates the interleaving of actions that are caused by communication delay and multiple timers.

We presented the service model of the timing stack, which interacts with FSMs by receiving messages that set up future events at a particular time and dispatching messages that indicate such events when the time comes. In a cooperative driving system, the model of timing stack still appears as a single entity from the perspective of modules, even if the modules may located within different vehicles. The additional composition rules introduced in this chapter ensure that the transitions reflect the synchronous actions executed by modules.

Additionally, we design a lock protocol that resolves conflicting requests issued by different vehicles which attempt to merge. The lock protocol uses GPS clock to keep track of a specific time until which a vehicle needs to remain cooperative with the merging vehicle.

Chapter 6

Stratified Probabilistic Verification

This chapter introduces stratified probabilistic verification technique that conducts reachability analysis and checks probabilistic safety properties on Markov decision processes with very large state spaces. It derives from the probabilistic verification technique presented in [Maxemchuk and Sabnani, 1989]. By considering probabilistic choices with discretized levels of distribution, the technique prioritizes the traversal of the states that are more likely to be encountered during system execution. As the algorithm proceeds, the unexplored states are less significant in terms of reachability probability. When the algorithm stops when using up available memory, it computes the probability bound of reaching the error states by solving a linear program.

In cooperative driving systems, the problem of state explosion is prominent. Besides the cooperating vehicles, we often have to consider the behavior of multiple vehicles within their neighborhood. Furthermore, there are probabilistic events whose distributions are difficult to obtain. The stratified probabilistic verification technique addresses both problems. Discretized levels of probability distribution not only allows state traversal to prioritize the more probable states but also accepts events whose probability weights are characterized by discretized bounds rather than exact weights. By the time the memory used to store the explored states exceeds memory limit, the procedure constructs a linear program whose solution is an upper bound of the probability for a regular safety property to hold, even though the probabilities in the system model are not exact. In most cases it is able to determine whether a probabilistic safety property holds. Linear programs yield a much tighter bound than the one obtained by the precursor probabilistic verification technique.

The technique is well-suited for the multiple stack architecture or other modular architectures. It verifies those modules whose state spaces are too large to be tractable using conventional techniques and determines if they satisfy given probabilistic guarantees. Modules are then replaced with simpler interfaces constructed from the deterministic finite automata that represent the guarantees. This divides the problem of verifying a large composite system into subproblems of verifying individual modules.

In this chapter, we start by introducing the discretized levels of distribution for probabilistic choices and related definitions (section 6.1). We then present the stratified state traversal algorithm, the construction of linear program, and their proofs, and we compare its verification results with those of the predecessor technique and PRISM model checker (section 6.2). Next, we describe how verification of a complex system is divided into verification of subsystems (section 6.3). Finally, we show that the stratified technique also applies to standard MDPs (section 6.4).

6.1 Discretized Probability Levels

In probabilistic systems modeled as MDPs, the probabilistic choices may serve to model and quantify the possible outcomes of randomized actions with specific probabilistic distributions. In case where probabilistic distribution is unavailable, nondeterministic choices are often used. However, in some systems there are events that occur most of the time, while the other events are much less often, although without a specific distribution. Using nondeterministic choices to model those events overestimates their probabilities.

In this section, a variation of MDP, namely, discretized-probability Markov decision process (DMDP) is defined. The probabilistic choices are no longer associated with exact distribution, but are labeled as high probability choices or different levels of low probability choices. For instance, in communication protocol, most of the time messages are received correctly, which are high probability events; infrequent message losses are low probability events. Events that are labeled as high probability are more common than the events labeled as low probability. There are different levels of low probability choices: a message loss on a channel can be infrequent, while the complete failure of error recovery protocol is even rarer.

Even though exact distributions are not available, with the discretization levels we may still calculate upper bounds on the probability of a sequence of states and transitions. This leads to equivalent classes of reachable states. Finally, we define errors that occur in the model in addition to the functional errors characterized by DFA.

6.1.1 Discretized-probability Markov Decision Processes

Definition 10 *A discretized-probability Markov decision process (DMDP) is a tuple $M = (S, Act, \hat{p}, \pi, \iota_{\text{init}})$, where*

- *S is a countable set of states*
- *Act is a set of non-deterministic actions*
- *$0 < \hat{p} < 1$ is the discretization parameter, which stands for the maximum value of the probability of infrequent events*
- *$\pi : S \times A \times S \to \{0\} \cup \mathbb{Z}^+$ is a partial function such that for all states $s \in S$ and actions $\alpha \in Act$,*
  - *$\pi(s, \alpha, t) = i$ implies that $0 < P(s, \alpha, t) \leq \hat{p}^i$.*
  - *$\pi(s, \alpha, t)$ is undefined if $P(s, \alpha, t) = 0$.*
- *$\iota_{\text{init}} : S \to \{0\} \cup \mathbb{Z}^+$ is a partial function such that for all states $s \in S$:*
  - *$\iota_{\text{init}}(s) = i$ implies that $I_{\text{init}}(s)$ has a non-zero initial probability weight $I_{\text{init}}(s) \leq \hat{p}^i$.*
  - *$\iota_{\text{init}}(s)$ is undefined if $I_{\text{init}}(s) = 0$.*

The definition of an action being enabled or disabled follows from the standard definitions of MDPs given in Chapter 3. An action α is said to be *enabled* in s if and only if $\pi(s, \alpha, t)$ is defined for some state $t \in S$, otherwise α is *disabled*; $Act(s)$ denotes the set of enabled actions in s. The *α-successors* of s are the set of states α-$succ(s)$ =

$\{t|\pi(s,\alpha,t)$ is defined$\}$, and the corresponding *α-transitions* from s are the set of transitions $\alpha\text{-}trans(s) = \{(s,\alpha,t)|\pi(s,\alpha,s')$ is defined$\}$. The set $trans(s) = \bigcup_{\alpha \in Act(s)} \alpha\text{-}trans(s)$ are the transitions from state s.

The high probability choices lead to transitions with $\pi(.) = 0$. Those leading to transitions with $\pi(.) > 0$ are low probability choices, and there can be multiple levels of low probability choices. Different levels of low probability choices differ greatly in their frequencies. A level-1 low probability choice that leads to a transition with $\pi(.) = 1$ occurs much more often than a level-2 low probability choice leading to a transition with $\pi(.) = 2$, and so forth.

Low probability probabilistic choices can often be used to model message losses in communication systems, sensor inaccuracies or mechanical malfunction in intelligent vehicles.

The adversaries in DMDP are also discretized. A discretized adversary is a partial function that maps an execution history to a discretization level rather than a probability weight over the set of enabled actions.

Definition 11 *A discretized adversary σ_D for DMDP M is a function $\sigma_D : S^+ \rightarrow \{0\} \cup \mathbb{Z}^+$ such that $\forall s_0 s_1 \ldots s_n \in S^+$*

- *$\sigma_D(s_0 s_1 \ldots s_n)(\alpha) = i$ implies that the actual probability weight $\sigma(s_0 s_1 \ldots s_n)(\alpha) \leq \hat{p}^i$.*
- *$\sigma_D(s_0 s_1 \ldots s_n)(\alpha)$ is undefined if $\alpha \notin Act(s_n)$.*

$\sigma(s_0 s_1 \ldots s_n)(\alpha)$ can be undefined for some $\alpha \in Act(s_n)$ but not all, because an adversary may not choose an action if there are alternative choices. In the discretized model, only memoryless adversaries are considered in this thesis as well.

6.1.2 Execution sequence and structure of reachable graph

An execution sequence of protocol systems corresponds to a sequence of states and transitions in the Markov decision process. It originates at some initial state s_0, where $\iota_{\text{init}}(s_0)$ is defined. An execution sequence can be either finite or infinite. We start by considering finite execution sequences in protocol systems, and we shall see later a desirable property of a system is that a system should not stay in a cycle indefinitely.

Consider a finite execution sequence $\psi = s_0\alpha_0s_1\alpha_1s_2\dots\alpha_{n-1}s_n$, where s_i's are states and $\alpha_i \in Act(s_i)$. We omit the action from the representation of ψ wherever it is not relevant to the context. We use ψ_0 to denote the first state in ψ and ψ_{-1} to denote the last state. To characterize the probability of a given sequence, we use the following definition.

Definition 12 *$L(\psi)$ is the sum of the labels $\pi(.)$ of all transitions in ψ,*

$$L(\psi) = \sum_{(u,\alpha,v)\in\psi} \pi(u,\alpha,v) \tag{6.1}$$

Similar to paths in Markov chain, execution sequences in DMDPs can be associated with probabilities. Although we cannot find the exact probability of each possible sequence without exact probability distribution, a sequence $\psi = s_0\alpha_0s_1\alpha_1s_2\dots\alpha_{n-1}s_n$ can be associated with a probability bound

$$P(\psi) = P(s_0,\alpha_0,s_1)\cdot P(s_1,\alpha_1,s_2)\dots P(s_{n-1},\alpha_{n-1},s_n) \le \hat{p}^{L(\psi)} \tag{6.2}$$

A sequence may contain zero or several low probability transitions. If the transitions in this sequence are all high probability transitions, i.e., $L(\psi) = 0$, we refer to it as a *high probability sequence*. It has a probability bound

$$P(\psi) \le \hat{p}^{L(\psi)} = 1 \tag{6.3}$$

If a sequence $\psi = s_0\alpha_0s_1\alpha_1s_2\dots\alpha_{n-1}s_n$ contains at least one low probability transition, then $L(\psi) > 0$. For instance, suppose that ψ contains three low probability transitions, specifically $\pi(s_0,\alpha_0,s_1) = \pi(s_2,\alpha_2,s3) = 2$, and $\pi(s_{n-1},\alpha_{n-1},s_n) = 1$, has $L(\psi) = 5$ and $P(\sigma) \le \hat{p}^{L(\psi)} = \hat{p}^5$.

The labeling function π transforms the structure of the reachable graph into layers, or equivalent classes. We can now define equivalence classes in the transition system.

Definition 13 *A state s belongs to $\mathcal{C}_k$ (equivalence class k) if s is reachable from one of the initial states via an execution sequence ψ s.t. $L(\psi) = k$, and there does not exist any execution sequence ψ from any of the initial states to s with $L(\psi) < k$.*

Intuitively speaking, given a state s in equivalence class $\mathcal{C}_k$, there can be multiple execution sequences that reach s. The execution sequence which is most likely to be taken to reach state s has a probability no greater than $\hat{p}^k$.

The equivalence classes have the following properties:

Property 1 $\mathcal{C}_i \cap \mathcal{C}_j = \phi,\ \forall\ i \neq j$

Property 2 $\bigcup_{i \geq 0} \mathcal{C}_i =$ *All the reachable states*

The two properties imply that we can explore equivalence classes in the increasing order and eventually we traverse all the reachable states. They motivate the stratified algorithm in the next section.

6.1.3 Correctness properties

Beside state invariants and functional correctness properties which are checked by taking product of the DMDP and a DFA, we also look for deadlocks and high probability cycles in a system. The definition of deadlocks is standard, while we find high probability cycles instead of livelocks in discretized model.

Definition 14 *A state s is considered a deadlock if $Act(s) = \phi$, i.e., there is no enabled action at state s.*

If this is the case, then state s has no successor. A deadlock is an undesirable system state. The system cannot proceed further once it arrives at a deadlock.

Definition 15 *A cycle $\psi = s_0\alpha_0 s_1\alpha_1 s_2\alpha_2 \ldots s_n\alpha_n s_0$ is a livelock if all the transitions are deterministic and all the states on the cycle are neither initial states nor states that indicate progress.*

A livelock is a loop in which the system does not perform useful work, and it is considered a system failure. Once entering the loop, the system stays in it forever. A system has infinite sequence if it has a livelock. In stratified model, we extend the definition of livelock and look for *high probability cycles.*

Definition 16 *A cycle $\psi = s_0\alpha_0 s_1\alpha_1 s_2\alpha_2 \ldots s_n\alpha_n s_0$ is a high probability cycle if all the transitions are high probability transitions, that is, $L(\psi) = 0$, and all states on the cycle are neither initial states nor states that indicate progress.*

Starting at state s_0, the probability of traversing a high probability cycle and returning to s_0 is upper-bounded by $\hat{p}^{L(\sigma)} = 1$. Systems that have high probability cycles tend to have a large average number of steps in their execution sequences without performing useful work, which leads to inefficient protocols.

For a system that does not have high probability cycle, all of its cycles (if any) must contain some low probability transition. The probability of traversing a cycle containing low probability transitions and returning to the starting point must be less than $\hat{p}$. The system stays in such a cycle with small probability. A desirable property of a system is not to possess high probability cycles, for which we do not have to consider infinite execution sequences.

6.2 Stratified State Traversal Algorithm

We present Stratified State Traversal Algorithm, or Stratified Algorithm for short. It is a composition of depth-first search and probabilistic search. It prioritizes traversal of the states that are more likely to be encountered during system executions, and it attempts to heuristically reduce the probability for the system to reach those states that have not been examined. By the time the algorithm terminates, it constructs a linear program to upper bound the probability for a given regular safety property to be violated.

In this section, we demonstrate how to check if a DMDP M satisfies a probabilistic safety property $\langle A \rangle_{\geq p}$ using stratified algorithm. The check is done in two steps. First, the algorithm traverses as many states as possible, roughly in the order of how likely a state is encountered. Next, we construct a linear program to compute an upper bound of the probability for the regular safety property A to be falsified.

We introduce stratified algorithm by looking at the pseudocode and a small example. The algorithm is then proved to traverse all the states in increasing order of equivalent classes, and errors are identified during the state traversal. We then show how to construct a linear program whose optimal solution yields the upper bound we wish to find. Finally, we show that the bounds obtained by solving the linear program is significantly tighter than the bounds obtained by the predecessor algorithm, and we also show that stratified algorithm is able to obtain a probability bound when being constrained in memory while the explicit engine of PRISM cannot produce a result before using up all the memory.

6.2.1 Integrate Probabilistic Search into Depth-First Search

The stratified algorithm is a result of introducing probabilistic search into standard DFS. It explores states in the decreasing order of how likely they are to appear during system executions. The algorithm attempts to examine all states in equivalence class $\mathcal{C}_0$, then the states in $\mathcal{C}_1$, etc. It terminates when all the states in the DMDP are explored, or when it uses up all the memory on the computer. We prove that after the algorithm finishes its k-th iteration, all states in $\mathcal{C}_k$ have been examined during that iteration.

Stratified DFS consists of two procedures described in Algorithm 2. STRATIFIED-VERIFY is the main procedure, which takes a DMDP $M' = M$ or the product of a DMDP and a DFA $M' = M \otimes A^*$, and a parameter *lim*. A state s in the algorithm is a state of DMDP if M' is a DMDP, or it is a product state $\langle s, q\rangle$ if M' is a product of DMDP and DFA. In the latter case, $s.q$ refers to the DFA state of $\langle s, q\rangle$.

The algorithm contains a loop (line 3 - 10). We use iteration 0 to refer to the first pass of the loop, iteration 1 refers to the second pass of the loop, etc. The subroutine STRATIFIED-DFS-VISIT implements the stratified DFS. It takes M' and two parameters s and k. The former is the starting point of DFS traversal, and the latter refers to the iteration that the procedure is currently in. The subroutine recursively traverse all the states that are reachable from s without going through any low probability transition.

Class[.] and Entry[.] are two data structures that serve as containers for reachable states. Class[i] is used to store the states that have been traversed in iteration i, and Entry[i] stores the states from which we start the stratified DFS in iteration i. Both containers can be implemented efficiently using hash table and state fingerprinting [Holzmann, 1998].

Target is yet another container. Its purpose is to store the states that are considered errors. They include deadlocks, states whose DFA components are acceptance states, and states on high probability cycles.

At the beginning of iteration 0, Entry[0] is initialized to the set of initial states. The initial states are the states whose $\iota_{\text{init}}(\cdot)$ are defined. STRATIFIED-DFS-VISIT is used to

Algorithm 2 Stratified Verification

```
1: procedure STRATIFIED-VERIFY($M'$, lim)
2:     Entry[0] $\leftarrow$ initial states
3:     for $k \leftarrow [0 \ldots \mathrm{lim}]$ do                                    ▷ iteration [0 . . . lim]
4:         for all $s \in$ Entry[$k$] do
5:             if $s \notin$ Class[$i$], $\forall i \leq k$ then
6:                 Insert $s$ into Class[$k$]
7:                 STRATIFIED-DFS-VISIT($M^*, s, k$)
8:             end if
9:         end for
10:    end for
11: end procedure
12: procedure STRATIFIED-DFS-VISIT($M', s, k$)
13:    Push $s$ onto DFS-stack
14:    if $Act(s) = \phi \vee s.q \in F$ then
15:        Insert $s$ into $Target$                            ▷ deadlock or acceptance state
16:    end if
17:    for all $(s, \alpha, t) \in trans(s)$ do
18:        if $\pi(s, \alpha, t) > 0$ then
19:            Insert $t$ into Entry[$k + \pi(s, \alpha, t)$]
20:        else
21:            if $t \in$ DFS-stack then
22:                Cycle found. Insert the states on cycle into $Target$        ▷ high prob. cycle
23:            else if $t \notin$ Class[$i$], $\forall i \leq k$ then
24:                Insert $t$ into Class[$k$]
25:                STRATIFIED-DFS-VISIT($M', t, k$)
26:            end if
27:        end if
28:    end for
29:    Pop DFS-stack
30: end procedure
```

examine all the states that are reachable from every state $s \in$ Entry[0] without traversing any low probability transition. The states that have been examined are being placed in Class[0]. All the states that are discovered by traversing a low probability transition are placed in Entry[j] for some $j > 0$, depending on the level of low probability transition taken to reach them.

In iteration 1, the algorithm starts the stratified DFS from the states in Entry[1]. Again, all the states that are reachable without traversing any low probability transition are placed in the container Class[1], while those discovered after a low probability transition are placed in Entry[j] for some $j > 1$.

Next we take a closer look at Stratified-DFS-visit. On entering the subroutine, the state s is push to a stack (line 13). When leaving the subroutine, s is popped from the stack (line 18). There are checks for deadlock and error before the transitions from s are examined (line 14). Note that we do not need to enumerate all the states in M before the algorithm starts. The set $trans(s)$ can be generated on-the-fly as the procedure proceeds by examining the state s.

The key difference between the stratified DFS and the standard DFS is that whenever a state is reached by traversing a low probability transition, unlike standard DFS, we do not continue the recursive traversal on such state. Rather, the state is inserted into Entry[$k + \pi(.)$] (line 19), and will be traversed in later iteration of the algorithm (line 5). When Stratified-DFS-visit subroutine terminates, it would have traversed all the states to which there exists at least one high probability path from the state where it was originally started.

If a transition (s, α, t) is not a low probability transition, the stratified DFS behaves the same as DFS. It first check if t already belongs to the DFS stack (line 22). If yes, then (s, α, t) is a backward edge, and the content in the stack constitutes a high probability cycle, which is a livelock according to our extended definition; otherwise, we then check if t is already stored in Class[m] for some $0 \leq m \leq k$ (line 23), if t is not explored yet, then t is placed into Class[k] and Stratified-DFS-visit is called recursively to continue exploring the successors of t (line 24-25).

Sometimes when discovering a deadlock, an error state, or a high probability cycle is found, we are interested in the sequence of events that lead to these states. To obtain a such counterexample, we could start typical DFS again from the initial state that traverses the states that have been stored in Class[.]. On reaching the error state, the counterexample is the content of the recursion stack.

A small example is shown in Fig.6.1.

- Iteration 0: Mod-DFS-visit starts from the initial states. x_0, x_1, x_2, y_0, y_1 are put into Class[0], x_3, y_2 are put into Entry[1], and x_4, y_3 are put into Entry[2].

- Iteration 1: Mod-DFS-visit starts from states in Entry[1]. x_3, y_2 are put into Class[1],

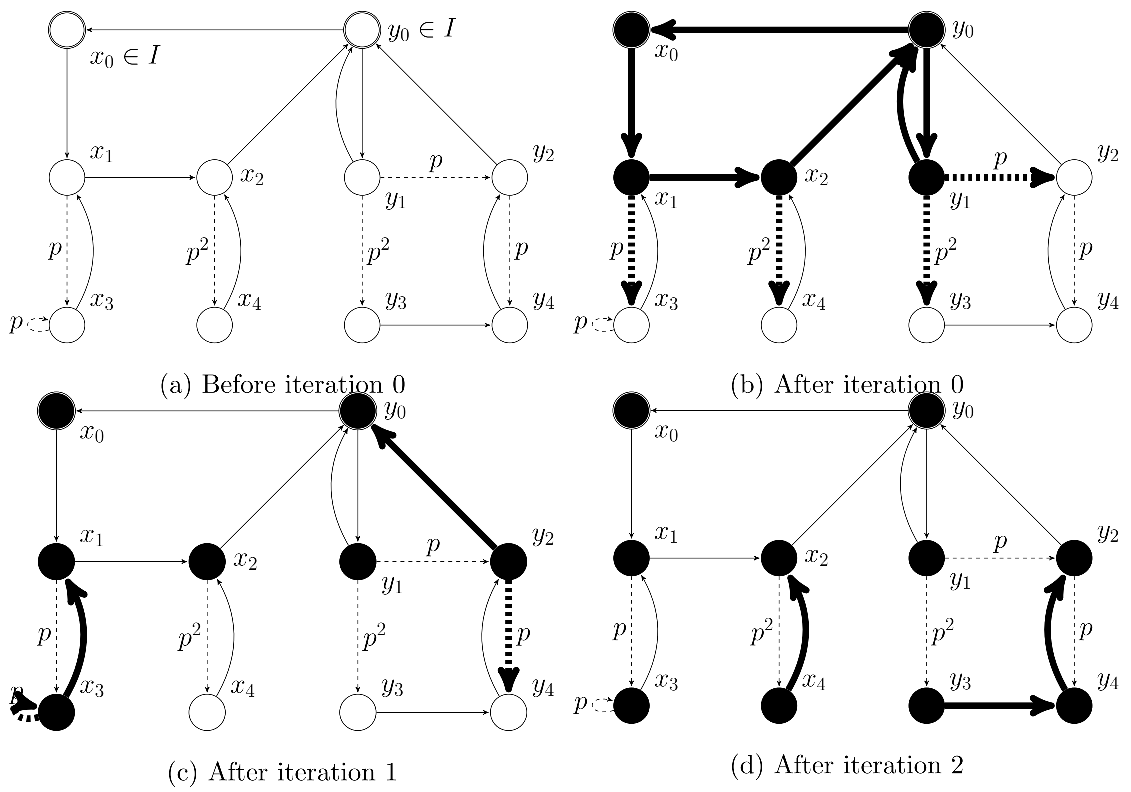


Figure 6.1: An example of verification

and y_4 is put into Entry[2].

- Iteration 2: Mod-DFS-visit starts from states in Entry[2]. x_4, y_3, y_4 are put into Class[2], while Entry[i]$= \phi$, $i > 2$.

The iterations are repeated until one of the following terminating condition occurs:

1. On finishing iteration k, there are no states in Entry[i] for all $i > k$, in which case the state space of the transition system is fully searched.
2. The computer on which the algorithm executes is running out of memory to store the states in containers Class[.] and Entry[.].
3. The algorithm finishes iteration *lim*, where *lim* is a parameter to the algorithm that specifies the upper bound of equivalence class to be examined.

6.2.2 Proof of Correctness

When the verification algorithm terminates, we would want to make sure that all the explored states in Class[.] have been checked. The following theorem and its corollaries assert that when iteration k terminates, it follows that among all the execution paths that contain states from classes $\mathcal{C}_0, \ldots, \mathcal{C}_k$, all the error states among $\mathcal{C}_0, \ldots, \mathcal{C}_k$ have been found and stored in *Target*. As a result, the first terminating condition implies that we have conducted a complete search on all the reachable states, and even though the second and the third terminating conditions leave the transition system to be partially explored, all the error states are identified.

Theorem 3 *When iteration k terminates, Class[k]* $= \mathcal{C}_k, \ \forall k \geq 0$

This theorem implies that when iteration k finishes, all states in $\mathcal{C}_k$ have been checked for errors and placed into container Class[k]. Furthermore, deadlock states, states $\langle s, q \rangle$ that have DFA state $q \in F$, and the states on high probability cycles have been placed into container *Target*. The algorithm starts by traversing the states that are more likely to occur, which are states in equivalence class $\mathcal{C}_0$. As the iteration repeats, the equivalence classes are explored one after another, with the probability of the occurrence of states becoming smaller.

We shall prove the theorem by arguing that all the states in $\mathcal{C}_k$ can be reached from at least one state in Entry[k] via a high probability path. Hence, all states in $\mathcal{C}_k$ will be reached by Stratified-DFS starting from some state in Entry[k] and be placed in Class[k] during iteration k.

Proof. We prove the theorem by mathematical induction. First start with the case $k = 0$. In iteration 0, the stratified DFS starts from the initial states. Let

$$R_0 = \{q | \exists \psi \text{ s. t. } \ \iota_{\text{init}}(\psi_0) \geq 0 \wedge \psi_{-1} = q \wedge L(\psi) = 0\} \tag{6.4}$$

Note that as there exists at least one high probability path from s_0 to q, the stratified DFS is guaranteed to encounter q when started from one of the initial states. When iteration 0 terminates, all states in R_0 will be placed into Class[0]. Recall the definition of equivalence class, $R_0 = \mathcal{C}_0$ and hence Class[0] $= \mathcal{C}_0$.

For the induction hypothesis, suppose that Class[k]$= \mathcal{C}_k$, $\forall k \leq m$. At the beginning of iteration $m+1$, line 19 of Algorithm 2 implies that

$$\text{Entry}[m+1] = \{s' | \exists (s, \alpha, s') \text{ s. t. } s \in \text{Class}[n], \pi(s, \alpha, s') = m+1-n\} \tag{6.5}$$

Note that not all states in Entry[$m+1$] belong to $\mathcal{C}_{m+1}$. The check on line 5 rules out those have already been reached earlier and were placed in Class[i] for some $i \leq m$. Let Entry[$m+1$]* be the set of states that survive the check, and Entry[$m+1$]$^* \subseteq \mathcal{C}_{m+1}$, as they can only be reached via an execution sequence ψ such that $L(\psi) \geq m+1$. For all $s \in$ Entry[$m+1$]*, there exists at least one execution sequence ψ such that $L(\psi) = m+1$ and $\psi_{-1} = s$. Stratified DFS only starts from the states in Entry[$m+1$]*.

By definition, a state t in $\mathcal{C}_{m+1}$ is reachable from one of the initial states through an execution sequence ψ such that $L(\psi) = m+1$, while there is no path such that $L(\psi) < m+1$. Let $\psi = s_0\alpha_0 s_1 \alpha_1 \dots \alpha_{n-1} s_n \alpha_n \dots s_{n+j-1}\alpha_{n+j-1}s_{n+j}$, where $s_{n+j} = t$, $\pi(s_{n-1}, \alpha_{n-1}, s_n) > 0$, and $\pi(s_n, \alpha_n, s_{n+1}) = \pi(s_{n+1}, \alpha_{n+1}, s_{n+2}) = \dots = \pi(s_{n+j-1}, \alpha_{n+j-1}, s_{n+j}) = 0$. That is, the last low probability transition on ψ is $(s_{n-1}, \alpha_{n-1}, s_n)$. We have a length-$j$ suffix $\psi_{n,n+j-1} = s_n\alpha_n \dots s_{n+j-1}\alpha_{n+j-1}s_{n+j}$ being a high probability sequence, and s_n must belong to Entry[$m+1$]*.

The existence of high probability sequence $\psi_{n,n+j-1}$ implies that for all states t' in $\mathcal{C}_{m+1}$, there exists a high probability path to t' from some state(s) in Entry[$m+1$]*. Stratified DFS can reached t' recursively in iteration $m+1$ without putting t' into yet another container Entry[m'], $m' > m+1$. Therefore, every state t' in $\mathcal{C}_{m+1}$ and only the states in $\mathcal{C}_{m+1}$ will be examined and be placed into container Class[$m+1$] during iteration $m+1$. Therefore, when iteration $m+1$ terminates without being interrupted by discovering a state that represents system error, Class[$m+1$] $= \mathcal{C}_{m+1}$. $\square$

Since the algorithm fills the container Class[0], Class[1], up to Class[k] in that order, we obtain the following corollary immediately

Corollary 4 *On finishing the k-th iteration, all states in $\mathcal{C}_0, \mathcal{C}_1, \dots, \mathcal{C}_k$ have been traversed and are checked for errors.*

At this point, we know that all states in $\bigcup_{0 \leq i \leq k} \mathcal{C}_i$ have been checked for deadlocks, functional errors, which are indicated by entering an acceptance state of the DFA, and high probability cycles containing the states entirely within the same equivalent class. The error states are stored in container *Target*. The following corollary shows that $\bigcup_{0\leq i\leq k}\mathcal{C}_i$ is free of high probability cycle if there is not any in $\mathcal{C}_i$ for $0 \leq i \leq k$.

Corollary 5 *If the k-th iteration has terminated without finding any high probability cycle, then there is no high probability cycle among states in $\bigcup_{0\leq i\leq k}\mathcal{C}_i$*

Proof. First we know that within each equivalence class there is no high probability cycle, hence a cycle must contain states from different equivalence classes. Let the cycle be $\psi = s_1 s_2 \ldots s_n s_1$ and let s_i and s_j be the states on the cycle that are from different equivalence classes, specifically s_i in $\mathcal{C}_i$ and s_j in $\mathcal{C}_j$. Assume without losing generality that $j > i$. Then the path from s_i to s_j must contain at least one low probability transition. Otherwise, if the path is high probability path, s_i and s_j cannot be of different equivalence classes. Therefore, there cannot be any high probability cycle among states in the union of $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_k$. $\square$

Corollary 6 *On finishing the k-th iteration, if Entry[i] is empty for all $i > k$, then the union of $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_k$ is the complete transition system.*

Finally, the last corollary states that if the algorithm terminates on condition 1, then the whole system is traversed and error states are stored in container *Target*.

Proof. Since Entry[i] is empty for all $i > k$, Class[i] is empty for all $i > k$, and hence $\mathcal{C}_i$ is empty for all $i > k$. $\square$

6.2.3 Bound Computation

We cannot determine the exact probability for a system to violate a given safety property if not all the reachable states have been explored. Even if all the reachable states are traversed, as long as there are error states, the exact probability still cannot be determined because we are given with a discretized model in which the probability distributions of some probability choices cannot be obtained. We may conclude that the safety property holds with probability one only if there are no error states found after all the reachable states explored.

Still, it is possible to upper-bound the probability for the system to reach some error states. When verifying a system, we are interested in whether the system, when it starts from one of the initial states or states that indicate progress, will

1. return to any of the initial states or states that indicate progress
2. arrive at an error state
3. arrive at any state that has not been explored by the algorithm, which can possibly lead to an error.

We compute the upper-bound of the probability for the system to take some execution paths that lead to the second and the third cases.

In the predecessor algorithm, the algorithm stops when discovering an error state. When error states cannot be found and the memory is running out, the algorithm computes the probability bound of reaching the unexplored states, assuming that they possibly lead to error. The bound is simply the product of the number of distinct execution sequences having the same number of low probability transitions, and the probability of taking such execution sequence.

In the stratified search, the algorithm no longer stops when discovering an error state. Furthermore, it improves bound computation by combining those execution sequences induced by probabilistic choices that have probabilities sum to one. The bound is improved significantly by solving a linear program, whose constraints are obtained by examining the transitions of each explored state. We look at the linear program and justify that its optimal solution is indeed the desired probability bound.

Our objective is to bound the probability for the system to end up in the set of error states or unexplored states when starting from the initial states or states that indicate progress. Let the set of the explored error states be T, and it contains exactly the states in *Target*, according to Corollary 4 and Corollary 5.

Suppose when the verification algorithm terminates, all states in class[0], …, class[k], or equivalently, states in $\mathcal{C}_0, \ldots, \mathcal{C}_k$, have been examined, while the states in class[$k+1$], class[$k+2$], …, have not been examined. The reachable states of the transition system can be divided into two disjoint sets: the set of explored states $R = \bigcup_{0\leq i\leq k} \mathcal{C}_i$, and the set of unexplored states $U = \bigcup_{i>k} \mathcal{C}_i$.

Let $\hat{P}_e(s)$ be the maximum probability for the system to start the execution from state s and arrive at some state in either T or U, instead of returning to any of the initial states or states that indicate progress. That is, we have

$$\hat{P}_e(s) = P^{\max}_{M'}(s \models \Diamond T) + P^{\max}_{M'}(s \models \Diamond U) \tag{6.6}$$

It is an upper bound of the error state, assuming that unexplored states lead to errors. Then the probability bound for a system to possibly end up in error is

$$\begin{aligned} P_e &\leq \sum\{\hat{P}_e(s) \cdot I_{\text{init}}(s) | s \in R, I_{\text{init}}(s) > 0\} \\ &\leq \max\{\hat{P}_e(s) | s \in R, \iota_{\text{init}}(s) \text{ is defined}\} \end{aligned} \tag{6.7}$$

It is worth noting that if $T = \phi$, the probability for the regular safety property to be violated is upper-bounded by the probability of reaching U.

In the following theorem, we shall see that the solution to a linear program yields an upper-bound to P_e.

Theorem 7 *Let M be a partially explored DMDP, with T being the explored error states of M and U being the unexplored states of M. Also, let J be the union of initial states and the states that indicate progress. The optimal solution vector $(x^*_s)_{s\in R}$ to the following linear program yields tha maximal reachabilibty probabilities with $\hat{P}_e(s) = P^{\max}_{M^*}(s \models T \cup U) = x^*_s$.*

$$\begin{aligned} &\textbf{min} \sum_{x_s \in R\cup U} x_s \\ &\textbf{s. t. } x_s = 1, \ \forall s \in T \cup U \\ &\qquad x_s = 0, \ \textit{if there is no path from } s \textit{ to } T \cup U \\ &\qquad x_s \geq 0, \ \forall s \in R \setminus T \\ &\qquad x_s \leq 1, \ \forall s \in R \setminus T \\ &\qquad x_s \geq \sum_{t \in \alpha\text{-}succ(s)\cap R} q_{(s,\alpha,t)} \cdot x_t + \sum_{u \in \alpha\text{-}succ(s)\cap U} q_{(s,\alpha,u)} + x_b, \\ &\qquad\qquad \forall\ b \in R \wedge \pi(s,\alpha,b) = 0, \ \forall s \in R, \alpha \in Act(s) \end{aligned} \tag{6.8}$$

where

$$q_{(v,\alpha,w)} = \begin{cases} 0 & \textit{if } \pi(v,\alpha,w) = 0 \vee w \in J \\ \hat{p}^{\pi(v,\alpha,w)} & \textit{if } \pi(v,\alpha,w) > 0 \end{cases} \tag{6.9}$$

Proof. The main idea of the proof is that, if the algorithm is unable to fully explore the state space, which leaves $U \neq \phi$, then we assume that all states in U lead to the set of error states or acceptance states T. This leads to the first constraint. The second constraint considers the states such that, for any adversary, all paths leaving the states do not reach T or U, that is, they eventually return to some state in J.The third set and the fourth set of constraints ensure that x_s is a probability for all $s \in R \setminus T$.

The fifth set of constraints account for the probability relation between transitions for a given state. First consider $\hat{P}_e(s)$ in an MDP with explicit probability distribution for all probabilistic choices. The semantic of MDP implies that the transition from some state s can be separated into selecting a non-deterministic choice and selecting a probabilistic choice according to probability distribution. Given any memoryless adversary σ,

$$\hat{P}_e(s) = \sum_{\alpha \in Act(s)} \sigma(s)(\alpha) \cdot \hat{P}_e(s|\alpha) \tag{6.10}$$

For any adversary $\sum_{\alpha \in Act(s)} \sigma(s)(\alpha) = 1$. Then (6.10) becomes

$$\begin{aligned} \hat{P}_e(s) &\leq \left[\sum_{\alpha \in Act(s)} \sigma(s)(\alpha) \right] \cdot \max_{\alpha \in Act(s)} \{\hat{P}_e(s|\alpha)\} \\ &\leq \max_{\alpha \in Act(s)} \{\hat{P}_e(s|\alpha)\} \end{aligned} \tag{6.11}$$

Suppose that the transitions have explicit probability distribution. Given that a non-deterministic action $\alpha \in Act(s)$ is selected, we have

$$\hat{P}_e(s|\alpha) = P(s,\alpha,t_0)\hat{P}_e(t_0) + P(s,\alpha,t_1)\hat{P}_e(t_1) \ldots + P(s,\alpha,t_n)\hat{P}_e(t_n) \tag{6.12}$$

, where $t_0, \ldots, t_n$ are the α-successors of s with states in J removed. The paths toward set J do not contribute to the probability of reaching unexplored state. If t is an unexplored state, that is, $t \in U$, then $\hat{P}_e(t) = 1$.

Since DMDPs do not come with explicit probability distribution for all probabilistic choices, (6.12) does not apply anymore. Fortunately, we can still find the largest possible

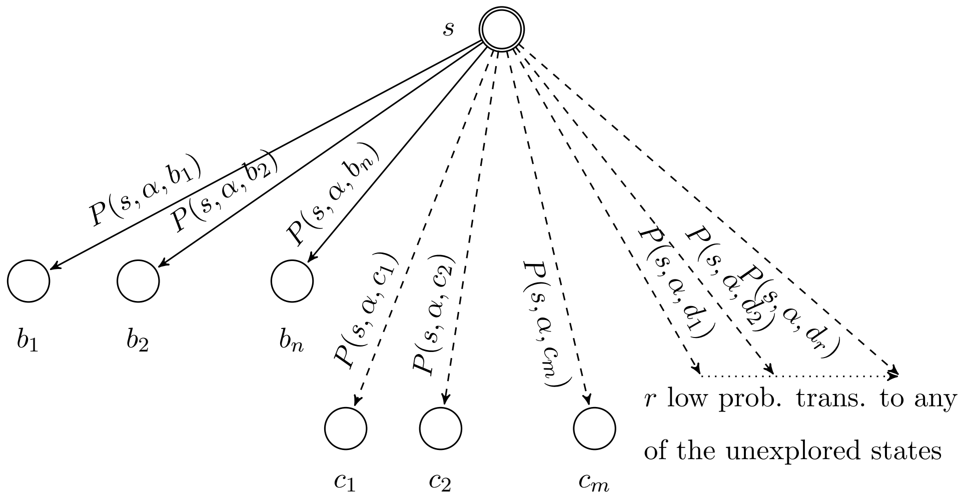


Figure 6.2: Possible transitions from a given state s

values of $\hat{P}_e(s|\alpha)$ and $\hat{P}_e(s)$. Consider possible transitions from state s after an action α is selected in Fig.6.2. The transitions to the states in set J are eliminated as they do not contribute to the probability of reaching an unexplored state. Given that α is selected, state s has n high probability transitions that lead to state $b_1, \ldots, b_n$, m low probability transitions with various levels leading to state $c_1, \ldots, c_m$, and r low probability transitions with various levels leading to some unexplored states not shown in the figure. Note that $b_1, \ldots, b_n, c_1, \ldots, c_m$ must be distinct. By definition, we have $P(v, \alpha, w) \le \hat{p}^{\pi(v,\alpha,w)}$, and $\sum_{w \in \alpha\text{-succ}(v)} P(v, \alpha, w) = 1$, so we can upper-bound $\hat{P}_e(s|\alpha)$:

$$
\begin{aligned}
&\hat{P}_e(s|\alpha) \\
&= P(s,\alpha,b_1)\hat{P}_e(b_1) + P(s,\alpha,b_2)\hat{P}_e(b_2) + \ldots P(s,\alpha,b_n)\hat{P}_e(b_n) \\
&\quad + P(s,\alpha,c_1)\hat{P}_e(c_1) + P(s,\alpha,c_2)\hat{P}_e(c_2) + \ldots + P(s,\alpha,c_m)\hat{P}_e(c_m) \\
&\quad + P(s,\alpha,d_1) + P(s,\alpha,d_2) + \ldots + P(s,\alpha,d_r) \\
&\le \max\{\hat{P}_e(b_1), \ldots, \hat{P}_e(b_n)\} \cdot \Big[\hat{P}_e(b_1) + \ldots + \hat{P}_e(b_n)\Big] \\
&\quad + P(s,\alpha,c_1)\hat{P}_e(c_1) + P(s,\alpha,c_2)\hat{P}_e(c_2) + \ldots + P(s,\alpha,c_m)\hat{P}_e(c_m) \\
&\quad + P(s,\alpha,d_1) + P(s,\alpha,d_2) + \ldots + P(s,\alpha,d_r) \\
&\le \ \max\{\hat{P}_e(b_1), \ldots, \hat{P}_e(b_n)\} \\
&\quad + q_{(s,\alpha,c_1)}\hat{P}_e(c_1) + \ldots + q_{(s,\alpha,c_m)}\hat{P}_e(c_m) \\
&\quad + q_{(s,\alpha,d_1)} + \ldots + q_{(s,\alpha,d_r)}
\end{aligned}
\tag{6.13}
$$

Substituting (6.13) into (6.11) yields the set of inequalities for each state $s \in T$

$$\begin{aligned}\hat{P}_e(s) &\leq \max_{\alpha \in Act(s)} \begin{bmatrix} \max\{\hat{P}_e(b_1), \hat{P}_e(b_2), \ldots, \hat{P}_e(b_n)\} \\ +q_{(s,\alpha,c_1)}\hat{P}_e(c_1) + \ldots + q_{(s,\alpha,c_m)}\hat{P}_e(c_m) \\ +q_{(s,\alpha,d_1)} + q_{(s,\alpha,d_2)} + \ldots + q_{(s,\alpha,d_r)} \end{bmatrix} \\ &\leq \max_{\alpha \in Act(s)} \left\{ \max_{b \in R, \pi(s,\alpha,b)=0} \begin{bmatrix} \hat{P}_e(b) \\ +q_{(s,\alpha,c_1)}\hat{P}_e(c_1) + \ldots + q_{(s,\alpha,c_m)}\hat{P}_e(c_m) \\ +q_{(s,\alpha,d_1)} + q_{(s,\alpha,d_2)} + \ldots + q_{(s,\alpha,d_r)} \end{bmatrix} \right\}\end{aligned} \tag{6.14}$$

Thus we have

$$x_s^* \geq \begin{bmatrix} x_b^* \\ +q_{(s,\alpha,c_1)}x_{c_1}^* + \ldots + q_{(s,\alpha,c_m)}x_{c_m}^* \\ +q_{(s,\alpha,d_1)} + \ldots + q_{(s,\alpha,d_r)} \end{bmatrix} \geq \hat{P}_e(s) \tag{6.15}$$

$\square$.

Finally, we can use linear program solver to find the optimal solution, then using (6.7) we find the tightest bound for P_e. Moreover, the set of constraints are reasonably sparse, which gives ways for the solver to employ more sophisticated algorithms.

A probability bound obtained by solving a linear optimization problem yields a bound that is as tight as possible without violating the constraints reflected by the information we have on probabilistic choices. In the predecessor of probabilistic verification, a bound is obtained by counting the number of distinct paths originated at the initial states and end up in the unexplored state. This number can potentially be exponential in the number of explored states.

6.2.4 Stratified Technique versus Original Probabilistic Verification

The stratified algorithm produces a significantly tighter bound for the probability of reaching unexplored states than that produced by the original probabilistic verification algorithm [Maxemchuk and Sabnani, 1989]. We apply the two algorithms to the lock protocol system shown in Fig.6.3, where the lock protocol resolves merging requests from car 4 and car 6, and the driver-assisted merging that will be described in the next chapter. Next, we compare the bounds obtained by solving linear programs and by counting the number of distinct paths that lead to unexplored states.

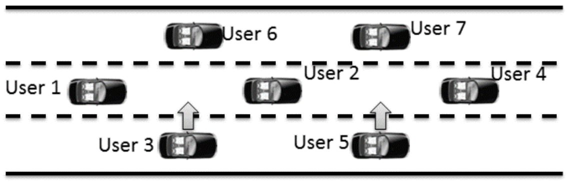


Figure 6.3: Lock protocol system resolving two conflicting requests

	Lock with 2 conflicting reqs		Driver-assisted merging	
	Original	Stratified	Original	Stratified
Sequences w/o low prob. edge	$4.44 \times 10^6 p$	$24.0012p$	$12837p$	$0.0015p$
Up to sequences bounded by p	$2.68 \times 10^7 p^2$	$105.008p^2$	$3.78 \times 10^6 p^2$	$13.0002p^2$
Up to sequences bounded by p^2	$7.91 \times 10^7 p^3$	$62.0082p^3$	$1.11 \times 10^9 p^3$	$2.0013p^3$
Up to sequences bounded by p^3	$1.39 \times 10^8 p^4$	$4p^4$	TLE	$15p^4$

Table 6.1: Comparison of probability bounds obtained by the original probabilistic verification and by using linear programming

Table 6.1 summarizes the bounds obtained using two methods. When verifying systems using the original probabilistic verification, we evaluate all sequences that have probability upper-bounded by p, p^2, and p^3. We stop examining those sequences with a lower probability bound. The bound for probability of reaching unexplored states is obtained by counting the number of distinct paths that are upper-bounded by p^4 or lower. When using stratified algorithm, we examine all reachable states up to equivalent class $\mathcal{C}_3$. We solve the linear program so as to compute the bound for probability of reaching unexplored states.

It can be immediately seen that the bounds obtained by solving linear programs are significantly tighter than the bounds obtained by counting paths. The reason is twofold. First, linear programs combine the transitions that have probabilities sum to less than one, which is done in (6.13), even though we do not have exact probability distribution. Secondly, the states in both the lock protocol and driver-assisted merging often have several nondeterministic transitions. This leads to an exponential increase in the number of distinct paths. This is particularly evident when sequences with probability upper-bounded by p^3 are considered, the number of distinct paths grows to a point that cannot be accounted for

	Lock with 5 conflicting reqs	
	PRISM	Stratified
75MB	out of memory	$40.0312p$
100MB	out of memory	$406.118p^2$
150MB	out of memory	$983.204p^3$

Table 6.2: Comparison of PRISM and stratified algorithm

within an hour of running time.

6.2.5 Stratified Technique versus PRISM Model Checker

When the system model has a large number of reachable states that cannot fit into limited amount of memory, the stratified algorithm traverses the more probable states and computes probability bound of reaching those unexplored ones. Typical model checkers that require completely traversal of the state space run out of memory when being supplied with such system. In this experiment, we apply the stratified algorithm and PRISM model checker to the lock protocol system shown in Fig.6.4. We compare their performance by fixing the amount of memory at disposal.

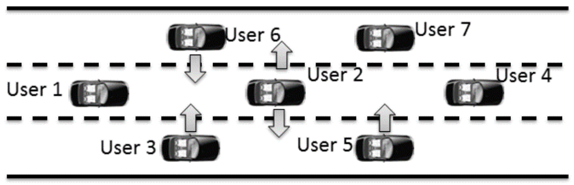


Figure 6.4: Lock protocol system resolving five conflicting requests

It should be noted that PRISM is run using explicit engine. That is, it does not use symbolic data structure in model construction. We deliberately use this setting to avoid using any optimizing options in PRISM so as to make the comparison meaningful. We would like to emphasize the advantage of directed search over state space over complete traversal when memory is constrained.

Table 6.2 summarizes the result of verification. We see that PRISM model checker cannot complete verification as it uses up all the available memory to store the explored states. On the other hand, stratified algorithm prioritizes state traversal on those states that are encountered more often during system execution. When there are only 75MB of memory available, it traverses all states in class $\mathcal{C}_0$ and computes the probability of reaching those states beyond $\mathcal{C}_0$. When the available amount of memory is increased to 100MB, it is able to traverse the states in $\mathcal{C}_1$ in addition to the states in $\mathcal{C}_0$. It then computes the probability of reaching the unexplored states. As there are more available memory, the stratified algorithm is able to traverse more reachable states, and the probability of reaching the unexplored states decreases.

6.3 Compositional Verification with Stratified Technique

In this section, we describe a compositional verification technique for probabilistic systems with discretized distributions. The multiple stack architecture introduced in Chapter 4 divides a cooperative driving system into multiple interacting modules. Verification of the whole system can be done by examining each module in isolation.

The compositional verification technique is based on the assume-guarantee approach in [Kwiatkowska *et al.*, 2010], in which both the assumptions made about system modules and the guarantees that they provide are regular safety properties, which can be represented as DFAs. The previous work reduces compositional verification to multi-objective model checking [Etessami *et al.*, 2007]. We approach the problem differently as the systems we considered have discretized probability distribution. Instead, by introducing an additional acceptance state to the DFA and considering a restricted set of adversaries, the compositional verification problem is reduced to reachability test that can be done by stratified algorithm.

First, we consider *probabilistic assume-guarantee triples* of the form $\langle A\rangle_{\geq p_A} M \langle G\rangle_{\geq p_G}$, where $\langle A\rangle_{\geq p_A}$ and $\langle G\rangle_{\geq p_G}$ are probabilistic safety properties, M is an MDP or a DMDP. The triple means that whenever M is part of a system that satisfies A with probability at least p_A, then M satisfies G with probability at least p_G. Defining this formally:

Definition 17 *If $\langle A\rangle_{\geq p_A}$ and $\langle G\rangle_{\geq p_G}$ are probabilistic safety properties, M is an MDP or* *a DMDP and* $\alpha_G \subseteq \alpha_A \cup M.Act$, *then*

$$\langle A\rangle_{\geq p_A} M \langle G\rangle_{\geq p_G} \Leftrightarrow \forall \sigma \in Adv \cdot [Pr^{\sigma}_{M}(A) \geq p_A \Rightarrow Pr^{\sigma}_{M}(G) \geq p_G] \tag{6.16}$$

We write $\langle true\rangle$ M $\langle G\rangle_{\geq p_G}$ to denote the absence of any assumption. This is equivalent to $M \models \langle G\rangle_{\geq p_G}$, which can be determined by standard probabilistic model checking, if M is an MDP, and by stratified probabilistic verifcation, if M is a DMDP.

The definitions above come with the following assume-guarantee proof rules to allow compositional verification. Their proofs are available in [Kwiatkowska *et al.*, 2010].

1. Given an appropriate assumption $\langle A\rangle_{\geq p_A}$, we can check the correctness of a probabilistic safety property $\langle G\rangle_{\geq p_G}$ on the composition $M_1 \parallel M_2$, without constructing and checking the whole model:

$$\langle true\rangle M_1 \langle A\rangle_{\geq p_A} \wedge \langle A\rangle_{\geq p_A} M_2 \langle G\rangle_{\geq p_G} \Rightarrow \langle true\rangle M_1 \parallel M_2 \langle G\rangle_{\geq p_G} \tag{6.17}$$

2. Using $\langle A_1, \ldots, A_k\rangle_{\geq p_1,\ldots,p_k}$ to denote the conjunction of probabilistic safety properties $\langle A_1\rangle_{\geq p_1}$ for $i = 1, \ldots, k$, we have the first rule extended to k assumptions:

$$\frac{\begin{array}{c}\langle true\rangle M_1 \langle A_1, \ldots, A_k\rangle_{\geq p_1,\ldots,p_k} \\ \langle A_1, \ldots, A_k\rangle_{\geq p_1,\ldots,p_k} M_2 \langle G\rangle_{\geq p_G}\end{array}}{\langle true\rangle M_1 \parallel M_2 \langle G\rangle_{\geq p_G}} \tag{6.18}$$

3. By having the first rules repeatedly applied, one obtains

$$\frac{\begin{array}{r}\langle true\rangle M_1 \langle A_1\rangle_{\geq p_1} \\ \langle A_1\rangle M_2 \langle A_2\rangle_{\geq p_2} \\ \ldots \\ \langle A_{k-1}\rangle M_{k-1} \langle A_k\rangle_{\geq p_k} \\ \langle A_k\rangle M_k \langle G\rangle_{\geq p_G}\end{array}}{\langle true\rangle M_1 \parallel M_2 \parallel \ldots \parallel M_{k-1} \parallel M_k \langle G\rangle_{\geq p_G}} \tag{6.19}$$

When given a standard MDP, multi-objective model checking technique in [Etessami *et al.*, 2007] is used to determine if (6.16) holds. This is not the case for DMDPs. However, verification of $\langle A\rangle_{\geq p_A} M \langle G\rangle_{\geq p_G}$ is reducible to a problem that can be approached by stratified verification by considering a restricted set of adversaries Adv^A that includes all adversary σ such that

$$Adv^A = \{\sigma \in Adv | \sigma(\langle s, q\rangle)(\alpha) \leq 1 - p_A \text{ if } \alpha \in Act(s) \wedge \delta_{A^*}(q, \alpha) \in F\} \tag{6.20}$$

Intuitively, with probability less than $1 - p_A$, adversaries in Adv^A chooses a nondeterministic action α that moves the system into the acceptance state from the current state in one step.

It can be shown that for any adversary σ' such that $P^{\sigma}_{M}(A) \geq p_A$, $\sigma' \in Adv^A$. If we can show that

$$\forall \sigma' \in Adv^A \cdot P^{\sigma'}_{M}(G) \geq p_G \tag{6.21}$$

, we have the following

$$\forall \sigma \in Adv \cdot [P^{\sigma}_{M}(A) \geq p_A \Rightarrow P^{\sigma}_{M}(G) \geq p_G] \tag{6.22}$$

This can be generalized to the verification of $\langle A_1, \ldots, A_k\rangle_{\geq p_1, \ldots, p_k} M \langle G\rangle_{\geq p_G}$. For each regular safety property A_i, there is a corresponding set of adversaries Adv^{A_i}. For any adversary σ' such that $P^{\sigma}_{M}(A_1) \geq p_1 \wedge \ldots \wedge P^{\sigma}_{M}(A_k) \geq p_k$, $\sigma' \in Adv^{A_1} \cap \ldots \cap Adv^{A_k}$. Similarly, showing that

$$\forall \sigma' \in Adv^{A_1} \cap \ldots \cap Adv^{A_k} \cdot P^{\sigma'}_{M}(G) \geq p_G \tag{6.23}$$

implies

$$\forall \sigma \in Adv \cdot [P^{\sigma}_{M}(A_1) \geq p_1 \wedge \ldots \wedge P^{\sigma}_{M}(A_k) \geq p_k \Rightarrow P^{\sigma}_{M}(G) \geq p_G] \tag{6.24}$$

With a small modification, we can use stratified verification to determine if (6.21) and (6.23) hold. First and foremost, the adversaries being considered must be discretized. Fixing a discretization parameter $\hat{p}$, for a given probability bound p_A we find the largest integer c such that $\hat{p}^c \geq 1 - p_A$. We include more adversaries into the restricted set by raising the criterion from $1 - p_A$ to $\hat{p}^c$:

$$Adv^{A'} = \{\sigma \in Adv | \sigma(\langle s, q\rangle)(\alpha) \leq \hat{p}^c \text{ if } \alpha \in Act(s) \wedge \delta_{A^*}(q, \alpha) \in F\} \tag{6.25}$$

Since $Adv^{A'} \supseteq Adv^A$, proving that $\forall \sigma \in Adv^{A'} \cdot P^{\sigma}_{M}(G) \geq p_G$ also implies (6.21).

The set of discretized adversaries that corresponds to $Adv^{A'}$ is

$$Adv_D^A = \left\{ \sigma_D \in Adv_D \;\middle|\; \sigma_D(\langle s, q\rangle)(\alpha) = \begin{array}{l} c \equiv \lfloor \log_{\hat{p}}(1 - p_A) \rfloor \text{ if } \alpha \in Act(s) \wedge \delta_{A^*}(q, \alpha) \in F, \\ 0, \text{ otherwise} \end{array} \right\} \tag{6.26}$$

Algorithm 3 and Theorem 8 are the appropriate modifications to the stratified algorithm and the linear program that only account for restricted sets of discretized adversaries.

The changes to Stratified-DFS-visit only appear on line 7 and line 8. When deciding whether to defer a state to later iteration, the algorithm used to consider all possible adversaries, assuming conservatively that $\sigma(s)(\alpha) \leq 1$. Now given the restricted set, for some action α we have $\sigma_D(s)(\alpha) = c > 0$. As a result, the state t is reached by selecting action α with probability $\leq \hat{p}^c$ then taking probability choice with probability $\leq \hat{p}^{\pi(s,\alpha,t)}$.

Algorithm 3 Stratified Verification on Restricted Set of Adversaries

```
1: procedure STRATIFIED-DFS-VISIT($M^*, s, k$)
2:     Push $s$ onto DFS-stack
3:     if $Act(s) = \phi \vee s.q \in F$ then
4:         Insert $s$ into $Target$                              ▷ deadlock or acceptance state
5:     end if
6:     for all $(s, \alpha, t) \in trans(s)$ do
7:         if $\pi(s, \alpha, t) + \sigma_D(s)(\alpha) > 0$ then
8:             Insert $t$ into Entry[$k + \pi(s, \alpha, t) + \sigma_D(s)(\alpha)$]
9:         else
10:            if $t \in$ DFS-stack then
11:                Cycle found. Insert the states on cycle into $Target$     ▷ high prob. cycle
12:            else if $t \notin$ Class[$i$], $\forall i \leq k$ then
13:                Insert $t$ into Class[$k$]
14:                STRATIFIED-DFS-VISIT($M^*, t, k$)
15:            end if
16:        end if
17:    end for
18:    Pop DFS-stack
19: end procedure
```

Change in Theorem 8 is the definition of $q_{(v,\alpha,w)}$. The probability of a transition is a product of the weight assigned by the adversary and the weight of the probabilistic choice. Previously no restriction on adversaries are placed, and the algorithm simply assumes that adversaries assign to each nondeterministic action with weight ≤ 1. Therefore the transition probability is only determined by the weight of probabilistic choice. The given restricted set now specifies that some actions are only selected with probability $\leq \hat{p}^c$. The probability of a transition becomes upper-bounded by $\hat{p}^{\pi(v,\alpha,w)} \cdot \hat{p}^{\sigma_D(v,\alpha)} = \hat{p}^{\pi(v,\alpha,w)+\sigma_D(v,\alpha)}$.

Theorem 8 *Let M be a partially explored DMDP, with T being the explored error states of M and U being the unexplored states of M. The optimal solution vector $(x_s^*)_{s\in R}$ to the following linear program yields tha maximal reachabilibty probabilities with $\hat{P}_e(s) = \max_{\sigma\in Adv^A} P_{M^*}(s \models T \cup U) = x_s^*$.*

$$
\begin{aligned}
&\mathbf{min} \sum_{x_s \in R\cup U} x_s \\
&\mathbf{s.\ t.}\ x_s = 1,\ \forall s \in T \cup U \\
&\qquad x_s = 0,\ \textit{if there is no path from } s \textit{ to } T \cup U \\
&\qquad x_s \geq 0,\ \forall s \in R \setminus T \\
&\qquad x_s \leq 1,\ \forall s \in R \setminus T \\
&\qquad x_s \geq \sum_{t\in\alpha\text{-}succ(s)\cap R} q_{(s,\alpha,t)} \cdot x_t + \sum_{u\in\alpha\text{-}succ(s)\cap U} q_{(s,\alpha,u)} + x_b, \\
&\qquad\qquad\qquad \forall\ b \in R \wedge \pi(s,\alpha,b) = 0,\ \forall s \in R, \alpha \in Act(s)
\end{aligned}
\tag{6.27}
$$

where

$$
q_{(v,\alpha,w)} = \begin{cases} 0 & \textit{if } \pi(v,\alpha,w) + \sigma_D(v)(\alpha) = 0 \vee w \in J \\ \hat{p}^{\pi(v,\alpha,w)+\sigma_D(v)(\alpha)} & \textit{if } \pi(v,\alpha,w) + \sigma_D(v) > 0 \end{cases}
\tag{6.28}
$$

6.4 Stratified Algorithm for Standard Markov Decision Processes

Although the stratified technique is initially developed for DMDP, it also applies to MDP with exact probability distribution. It explores states in roughly the descending order of how likely they appear in a system. As the states are generated on-the-fly, they are put into layers (the same notion as equivalent classes in DMDP), the higher of which contains the states that are more likely to be encountered. When the memory is no longer able to store the discovered states, the search stops. We then compute an estimate of the target probability by solving linear programs.

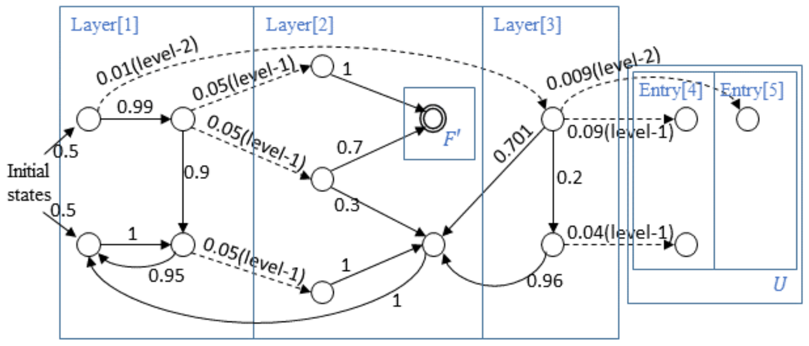


Figure 6.5: Example: layering parameter $\hat{p} = 0.1$

Stratified DFS breaks the state space of an MDP-DFA product $M \otimes A^{err}$ into layers by classifying transitions according to their probability, as shown in Fig.6.5. Fixing a layering parameter $\hat{p}$, the transitions are assigned to different discretized levels, demarcated by powers of $\hat{p}$. Transitions with probability $> \hat{p}$ are high probability transitions, otherwise they are low probability transitions. Any states discovered via low probability transitions are less likely to appear throughout system execution, so they are put into Entry[.] further from the current layer according to the discretization level of transition probability and will be explored later. States in Layer[1] are those reachable from the initial states by traversing only high probability transitions, and they are examined first. The states in Layer[2] are examined next, then the states in Layer[3], and so forth. In the example, the algorithm just finished examining states in Layer[3], and states in Entry[4] and Entry[5] are bounded to be examined next. They are the "frontier states" through which we may discover more states.

Algorithm 4 describes stratified DFS as two procedures. Stratified-DFS is the main one that takes an MDP-DFA product $M' = M \otimes A^*$, and a layering parameter $\hat{p}$. Layer[.] and Entry[.] are two data structures that serve as containers for reachable states. They can be implemented as hashed tables or priority queues. In each iteration of the while loop (line 4), the subroutine Stratified-DFS-visit starts a modified version of DFS from the states $s \in$ Entry$[k]$. It traverses the state space recursively and put the traversed states in Layer$[k]$ as does typical DFS, with one exception: it does not continue recursion when discovering a new state via a low probability transition, instead, the new state is put in

Entry[.] for it to be examined later (line 25).

Algorithm 4 Stratified Verification for MDP

```
 1: procedure STRATIFIED-DFS($M', \hat{p}$)
 2:     Entry[1] $\leftarrow \{s \in S | \eta_{\text{init}}(s) > 0\}$
 3:     $k \leftarrow 1$
 4:     while $\exists i \geq k$ s.t. Entry[$i$]$\neq \phi$ do
 5:         for all $s \in$ Entry[$k$] do
 6:             if $s \notin$ Layer[$i$], $\forall i \leq k$ then
 7:                 Insert $s$ into Layer[$k$]
 8:                 STRATIFIED-DFS-VISIT($M', \hat{p}, s, k$)
 9:             end if
10:         end for
11:         $k \leftarrow k + 1$
12:     end while
13: end procedure
14: procedure STRATIFIED-DFS-VISIT($M', \hat{p}, s, k$)
15:     if $s.q \in F$ then
16:         Insert $s$ into $Target$
17:     end if
18:     for all $(s, \alpha, t) \in trans(s)$ do
19:         if $P(s, \alpha, t) > \hat{p}$ then                          ▷ high prob. transition
20:             if $t \notin$ Layer[$i$], $\forall i \leq k$ and $t \notin I$ then
21:                 Insert $t$ into Layer[$k$]
22:                 STRATIFIED-DFS-VISIT($M', t, k$)
23:             end if
24:         else                                                       ▷ low prob. transition
25:             Insert $t$ into Entry[$k + \lfloor \log_{\hat{p}} P(s, \alpha, t) \rfloor$]
26:         end if
27:     end for
28: end procedure
```

If the procedure stops when the state space is completely traversed, *Target* contains all reachable error states. We decide whether $\langle A \rangle_{\geq p}$ holds by checking

$$Pr_M^{\min}(A) = 1 - Pr_M^{\max}(\Diamond \mathit{Target}) \geq p \tag{6.29}$$

Otherwise, suppose that the procedure stops at iteration k, we compute the minimum probability for the regular safety property A to hold over all adversaries. Let $U = \cup_{i \geq k}$Entry[i] be the set of frontier states. Since we do not know whether there are acceptance states reachable from states in U, $Pr_M^{\max}(\Diamond U)$ adds uncertainty to the result: $\langle A \rangle_{\geq p}$ holds if $1 - Pr_M^{\max}(\Diamond \mathit{Target} \vee \Diamond U)$, fails to hold if $1 - Pr_M^{\max}(\Diamond \mathit{Target}) < p$, or uncertain otherwise.

6.5 Conclusion

In this chapter, we presented stratified probabilistic verification technique. It is inherently different to the existing approaches to state explosion, which often aim at reducing the size of the state space. It adopts directed state traversal and prioritizes examining the more probable states. It is able to determine if a DMDP satisfies a given probabilistic safety property without completely traversal of state space. Furthermore, the stratified technique computes significantly tighter probability bounds than its predecessor algorithm. It is also worth noting that the stratified algorithm applies to not only the DMDP model introduced in this chapter but also MDP with standard definition.

The stratified verification technique is applicable to compositional verification. It is well-suited for the multiple stack architecture in that it verifies modules within the architecture individually, which reduce the complexity of verifying the complete system. As the modules in the architecture are often not perfect and fail with small probability or cannot be completely verified, stratified algorithm may assume that modules provide guarantee with at least some confidence and check properties on the system that depends on these modules.

Chapter 7

The Merge Protocol

This chapter evaluates the safety of the driver-assisted merging. We shall see that the techniques discussed in Chapter 3 through Chapter 6 are put to work. A merge protocol is designed at the top of multiple stack architecture. The result of stratified verification shows that the driver-assisted merge cannot fail with probability greater than $1 - 10^{-16}$. This degree of confidence cannot be provided by simulations and test tracks.

The merge protocol specified in this chapter is reasonably simple. It is specified as a collection of four FSMs. Within multiple stack architecture, the logic of the merge protocol can be simplified to an extent that it is able to focus on the the coordination of message exchange, commands to the components, and responses toward events on the roadway. Furthermore, the timing stack allows the timing to be extracted to a separate process that determines the time needed for a merge maneuver to complete.

In the verification, the components in the architecture are replaced by simple service models. They interact with the FSMs of the merge protocol using simple messages. We assume that these components deliver a set of probabilistic guarantees. For instance, the mapping protocol is expected not to have a sensor malfunction with a high probability. When the sensor malfunction does occur, it reports to the merge protocol so that merge protocol can notify the driver to abort merging. These set of probabilistic guarantees, along with the guarantees we expect the driver-assisted merging to deliver, are represented as a collection of DFAs. Furthermore, we separate the verification of the lock protocol and the verification of driver-assisted merge. The lock protocol is first verified to deliver its guarantee with required probability, after which the verification of driver-assisted merging is conducted given that the lock protocol along with the other components deliver the guarantees with promised probability. The stratified verification technique is able to bound the error probability of driver-assisted merging.

In this chapter, we start by discussing the role of the merge protocol in the multiple stack architecture and how driver-assisted merging is realized (section 7.1). We then present the specification of the merge protocol in FSMs (section 7.2). Next, we describe the service models of the components and the service guarantees in the form of probabilistic safety properties that are represented by DFAs. Finally, we apply the stratified verification to the composition of the FSM specification, the service models, and the DFAs (section 7.3).

7.1 Driver-assisted Merging

The driver-assisted merging is a collaborative effort that includes vehicle control, wireless communication, environment sensing and timing. We are mainly interested in the logic that coordinates joint actions among the three cooperating vehicle, sends commands to the peripheral components within a vehicle, and responds to the events generated by the components that interact with the environment. We call this logic the *merge protocol.* Fig.7.1 shows the relation between the merge protocol and the components that provide interfaces to interact indirectly with the physical world. The merge protocol is specified as a set of four FSMs, which will be explained further in the next section. In this section, we discuss the peripheral components and the functions they provide.

The merge protocol does not concern of the details of measuring distance, speed control, sending message wireless, etc. It interacts with the physical world through components including safe spacing system, lock protocol, mapping protocol, TRBP, and synchronized operation function in the timing stack, as shown in Fig.7.1. Based on currently feasible technologies, these components are assumed to provide the following services:

- The safe spacing system controls the distance between vehicles. It operates in a way similar to the CACC system in [Milanés *et al.*, 2014]. It uses ACC to adjust the speed and headway according to the output of the mapping protocol. During a merge, the

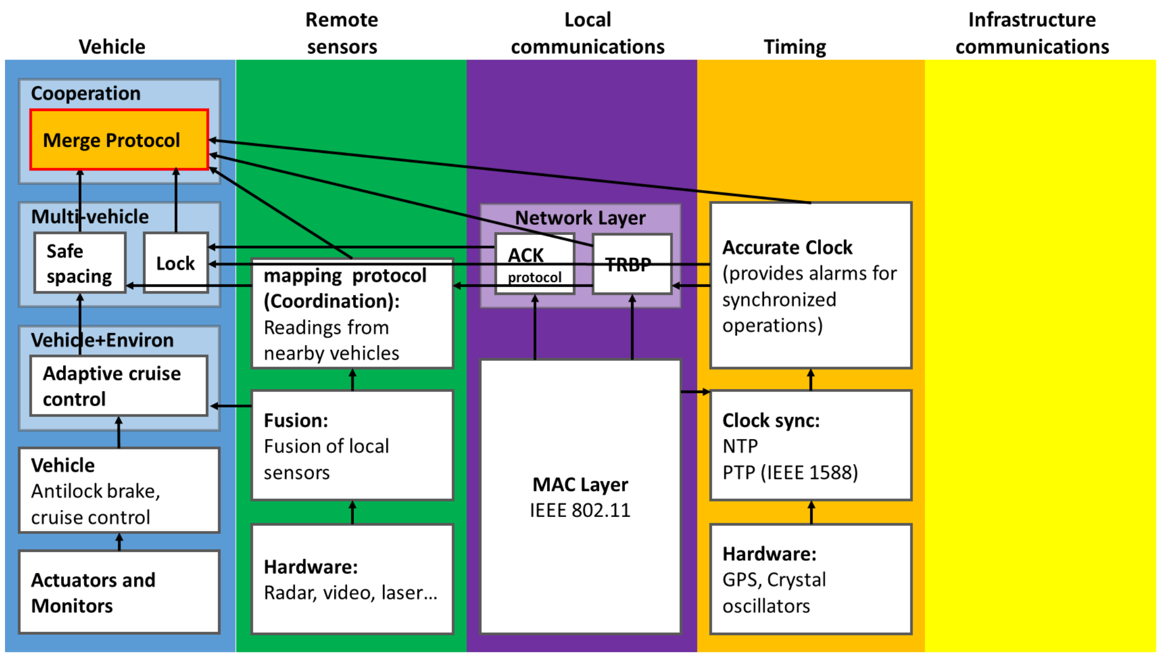


Figure 7.1: Driver-assisted Merge Protocol in the Architecture

safe system on the front car simply maintains constant headway to the preceding vehicle; the safe system on the back car creates a headway between itself and the front car so that the gap becomes large enough for the merging car to safely fit in; the safe system on the merging car aligns the merging vehicle with the gap between the front car and the back car by adjusting speed and headway settings accordingly.

- The lock protocol resolves conflicting merge requests issued by different vehicles. It creates a cooperating group among the three vehicle and ensures that each vehicle only participates in this group. The group only expires at a prespecified time. Its specification is presented in Chapter 5.

- TRBP detects communication failure and distributes messages when there are emergency incidents occurring in the vicinity of the cooperating group. It periodically circulates control messages, broadcasts and recovers messages among the three vehicles. The broadcasted emergency messages are guaranteed to be delivered to every vehicle in the cooperating group within a specified time, otherwise a communication failure is reported if any of the broadcasted messages is not recovered before the deadline.

- The mapping protocol maintains a list of objects in the surroundings of the three cooperating vehicles by fusing the object lists obtained from the fusion layer in each of the three vehicles. It uses TRBP to distribute lists of objects to every participant in the group. During normal operation, it monitors the gap size between the front car and the back car. It notifies the merge protocol when the gap is large enough to accomodate the merging vehicle and the mering vehicle is aligned with the gap. It also monitors non-participating vehicles around the cooperating group. When a vehicle brakes hard in front of the group or a interfering vehicle swoops into the gap, it dispatches warning to the merge protocol for it to take appropriate measures. TRBP guarantees that the object lists received by each partipant are up-to-date with a latency upper-bounded by token circulating period, which is in the order of few milliseconds, therefore we assume that the warning are received at each cooperating vehicle at the same time.

- The timing stack provides accurately synchronized clocks. It exposes an interface to the merge protocol as described in Chapter 5.

These components provide their service to the merge protocol through an interface comprising of simple messages. The messages are summarized in Table 7.1. The merge protocol interact with these components through handshaking, as they are connected by wire and are located reasonably close to the implementation of the merge protocol.

7.2 Specification of the Merge Protocol and its Safety Guarantees

The merge protocol is the main logic that dictates the actions of the three participating vehicle. In this section, we present the FSM specification of the merge protocol. The FSMs jointly describe how they work together with the other components in the system in order to assist the driver merging between the other two vehicles. The implementation details of the peripheral components are hidden from the FSMs by interfaces consisting of simple messages in Table 7.1. The FSMs send commands to them and receive notifications of

Component	Receiving	Sending
Safe spacing system	reset align make-gap maintain	
Mapping protocol		gap-taken gap-ready sensor-malfunction (malfunc)
TRBP		emergency communication-failure (comm-fail)
Lock protocol	attempt	success cooperate
Timing stack	set(d)	alarm(d)

Table 7.1: Messages on the Interface

incidents from them through these interfacing messages.

The merge protocol operates in three phases, with each phase accompanied by a request to the lock protocol. In phase 1 it uses the lock protocol to create a cooperating group consisting of the merging vehicle and the two in the target lane. Next, in phase 2 it instructs the safe spacing system to create a gap and to align the merging vehicle with the gap. Finally, in phase 3 the driver is prompted to steer into the gap. The merge protocol includes several escape sequences to handle undesirable events, such as an interfering driver or a sensor malfunction.

On each vehicle there are implementations of all four FSMs of the merge protocol, namely F_{merge} (Fig.7.2), F_{front}, F_{back} (Fig.7.4), and F_{hmi} (Fig.7.3). F_{merge} is activated whenever a driver switches on the turn signal so as to change lanes. F_{hmi} represents the human machine interface (HMI) between the implementation of the merge protocol and the driver. F_{front} and F_{back} are nearly identical in their FSM specification, except that they issue different commands to the safe spacing system. The labels of F_{back} are in the parentheses.

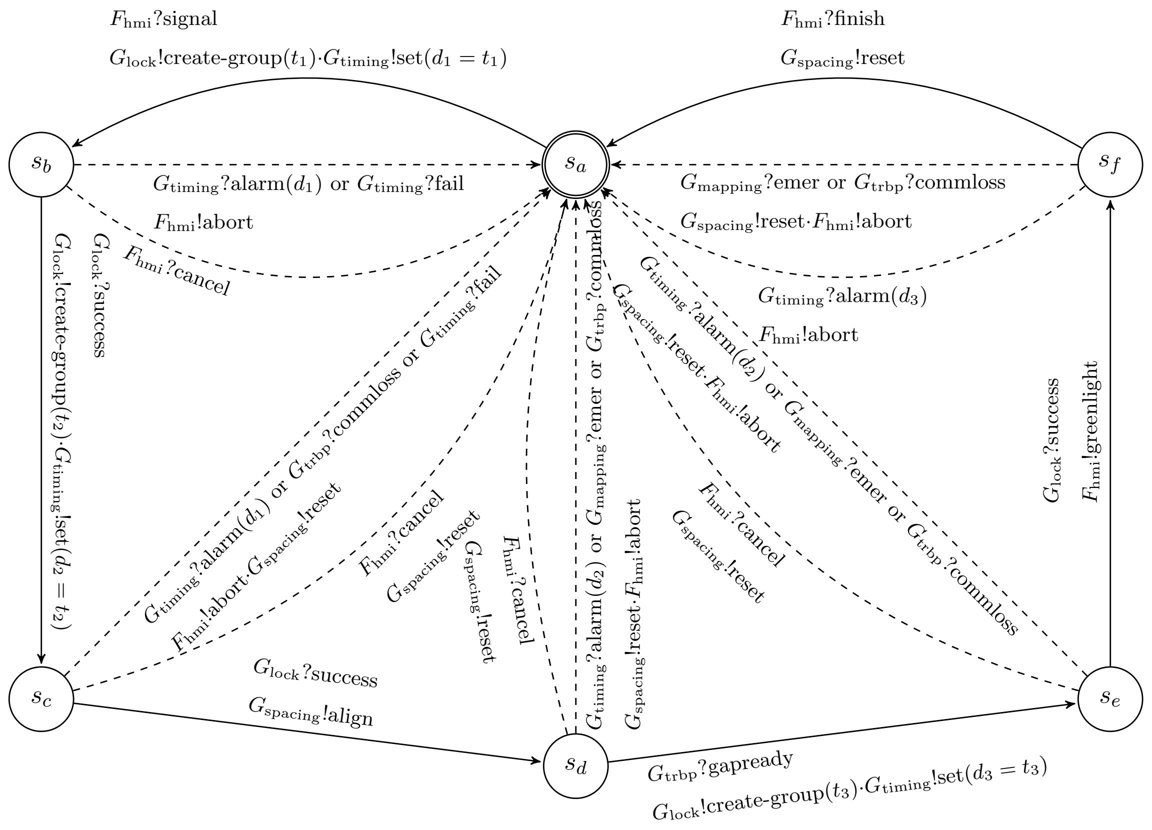

F_{hmi}?signal
G_{lock}!create-group$(t_1)\cdot G_{\text{timing}}$!set$(d_1 = t_1)$
F_{hmi}?finish
G_{spacing}!reset
s_b
s_a
s_f
G_{timing}?alarm(d_1) or G_{timing}?fail
F_{hmi}!abort
G_{mapping}?emer or G_{trbp}?commloss
G_{spacing}!reset$\cdot F_{\text{hmi}}$!abort
G_{lock}!create-group$(t_2)\cdot G_{\text{timing}}$!set$(d_2 = t_2)$
G_{lock}?success
F_{hmi}?cancel
G_{timing}?alarm(d_3)
F_{hmi}!abort
G_{timing}?alarm(d_1) or G_{trbp}?commloss or G_{timing}?fail
F_{hmi}!abort$\cdot G_{\text{spacing}}$!reset
F_{hmi}?cancel
G_{spacing}!reset
G_{spacing}!reset
F_{hmi}?cancel
G_{timing}?alarm(d_2) or G_{mapping}?emer or G_{trbp}?commloss
G_{spacing}!reset$\cdot F_{\text{hmi}}$!abort
G_{timing}?alarm(d_2) or G_{mapping}?emer or G_{trbp}?commloss
G_{spacing}!reset$\cdot F_{\text{hmi}}$!abort
F_{hmi}?cancel
G_{spacing}!reset
G_{lock}?success
F_{hmi}!greenlight
s_c
s_d
s_e
G_{lock}?success
G_{spacing}!align
G_{trbp}?gapready
G_{lock}!create-group$(t_3)\cdot G_{\text{timing}}$!set$(d_3 = t_3)$

Figure 7.2: Finite-state machine F_{merge}

F_{front} and F_{back} are activated when a vehicle that is not participating in another merge receives a request from the merging vehicle through the lock protocol; F_{front} is activated when the participating vehicle is in front of the target gap, while F_{back} is activated when the participating vehicle is behind the target gap.

To construct the model of the protocol system, we refer to the block diagram in Fig.7.5 that shows that how messages are passed between interfaces and the FSMs across different vehicle. Although the mapping protocol, TRBP, and the lock protocol appear as a single block across three vehicles, each of them consists of a three distinct implementations, one on each of the three vehicles. They appear as congruent blocks as the communications between the physically-separated implementations are hidden from the merge protocol. F_{merge}, F_{front} and F_{back} do not talk to each other directly. F_{merge} communicates with F_{front} and F_{back} through the lock protocol. F_{front} and F_{back} create gap by giving instructions to safe spacing

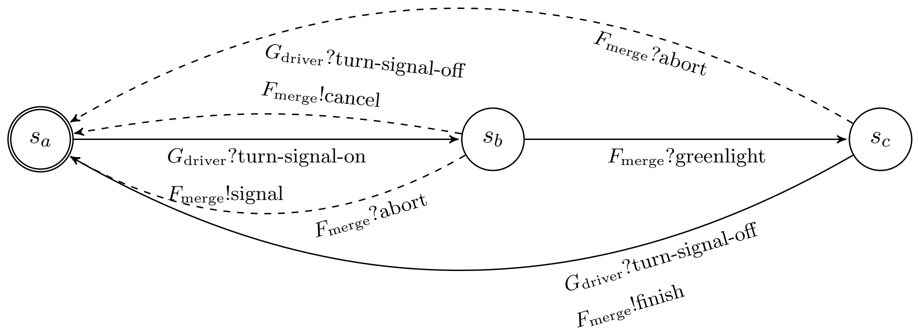

s_a
s_b
s_c
G_{driver}?turn-signal-off
F_{merge}!cancel
F_{merge}?abort
G_{driver}?turn-signal-on
F_{merge}?greenlight
F_{merge}!signal
F_{merge}?abort
G_{driver}?turn-signal-off
F_{merge}!finish

Figure 7.3: Finite-state machine F_{hmi}

systems on the respective vehicles in which they reside.

The merge protocol operates in three phases, each phase accompanied by a lock request which allocates time for the merge protocol to complete required actions during each phase. A merge maneuver is successful only if all three phases are completed. The successive completion of three phases is shown as the solid arrows in the specification diagrams.

1. Phase 1 starts when the driver switches on the turn signal ($F_{\text{hmi}} : s_a \to s_b$). As F_{merge} being activated, it uses the lock protocol to form a group until time t_1, and it also sets an alarm that will expire at time t_1 (F_{merge}: $s_a \to s_b$). If both car f and car b have the merge protocol available and not participating in any action, F_{front} and F_{back} are then activated and agree to cooperate ($F_{\text{front}}/F_{\text{back}}$: $s_a \to s_b$). The lock protocol returns success message and the merge protocol moves on to phase 2.

2. In phase 2 the merge protocol attempts to create the gap. It uses the lock protocol to extend the group to expire at $t_2 > t_1$, and it sets an alarm at time t_2 (F_{merge}: $s_b \to s_c$). On receiving cooperate request from the lock protocol, F_{front} instructs the safe spacing system to maintain speed while F_{back} instructs the safe spacing system to increase headway ($F_{\text{front}}/F_{\text{back}}$: $s_b \to s_c$). When notified of the success, F_{merge} instructs the safe spacing system to align itself with the gap by following car f or car x, whichever is closer, as shown in Fig.7.6a and Fig.7.6b (F_{merge}: $s_c \to s_d$). When the mapping protocol reports that the gap is large enough for merging, the protocol moves on to phase 3.

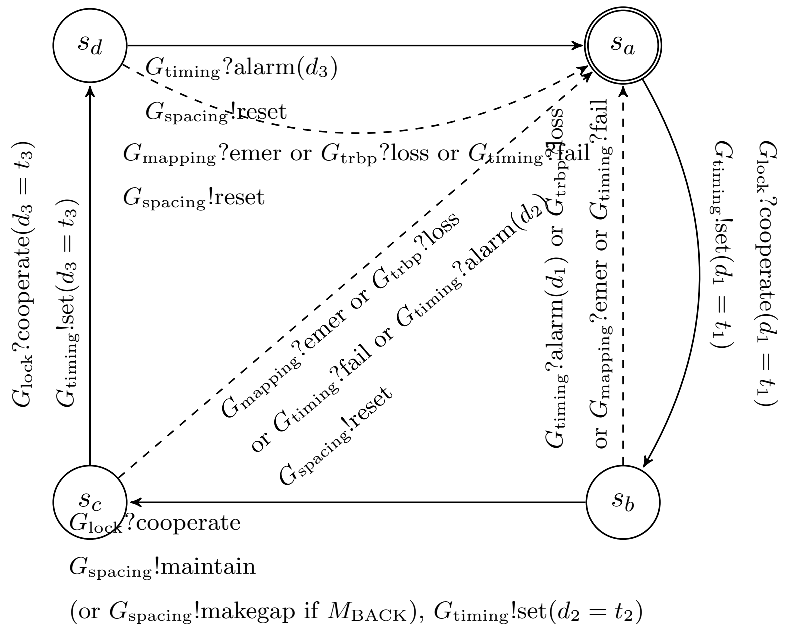

s_d
s_a
s_b
s_c
G_{timing}?alarm(d_3)
G_{spacing}!reset
G_{mapping}?emer or G_{trbp}?loss or G_{timing}?fail
G_{spacing}!reset
G_{lock}?cooperate($d_3 = t_3$)
G_{timing}!set($d_3 = t_3$)
G_{mapping}?emer or G_{trbp}?loss
or G_{timing}?fail or G_{timing}?alarm(d_2)
G_{spacing}!reset
G_{timing}?alarm(d_1) or G_{trbp}?loss
or G_{mapping}?emer or G_{timing}?fail
G_{timing}!set($d_1 = t_1$)
G_{lock}?cooperate($d_1 = t_1$)
G_{lock}?cooperate
G_{spacing}!maintain
(or G_{spacing}!makegap if M_{BACK}), G_{timing}!set($d_2 = t_2$)

Figure 7.4: Finite-state machine F_{front} (F_{back})

3. The driver is required to steer the vehicle into the gap in phase 3. It again uses the lock protocol to extend the expiration of the group to $t_3 > t_2$ and sets the alarm at t_3 (F_{merge}: $s_d \to s_e$). After both F_{front} and F_{back} agree to cooperate until t_3 ($F_{\text{front}}/F_{\text{back}}$: $s_c \to s_d$), the F_{merge} issue a greenlight message to the HMI that notifies the driver that it is clear to change lanes (F_{merge}: $s_e \to s_f$, $F_{\text{hmi}} : s_b \to s_c$). After steering into the target lane, the driver completes the action by switching off the turn signal. F_{merge} returns to the initial state (F_{merge}: $s_f \to s_a, F_{\text{front}}/F_{\text{back}} : s_d \to s_a, F_{\text{hmi}} : s_c \to s_a$).

When any of the undesirable events occurs, such as communication loss, the appearance of an interfering vehicle that does not belong to the group, or driver ignoring or being unaware of the prompt to switch lanes, the protocol executes escaping transitions (dashed arrows in the specification), notifies the driver, reset the safe spacing system for a safe, autonomous driving mode in which the vehicle maintains its lane and safe spacing.

Note that determining the value of t_1, t_2, and t_3, which are essentially the respective duration of phase 1, phase 2, and phase 3, is beyond the scope of this paper. Phase 1 can be short, within which the merge protocol only has to establish a group. The duration of phase 2 must be long enough to account for the time needed for the ACC to create a gap,

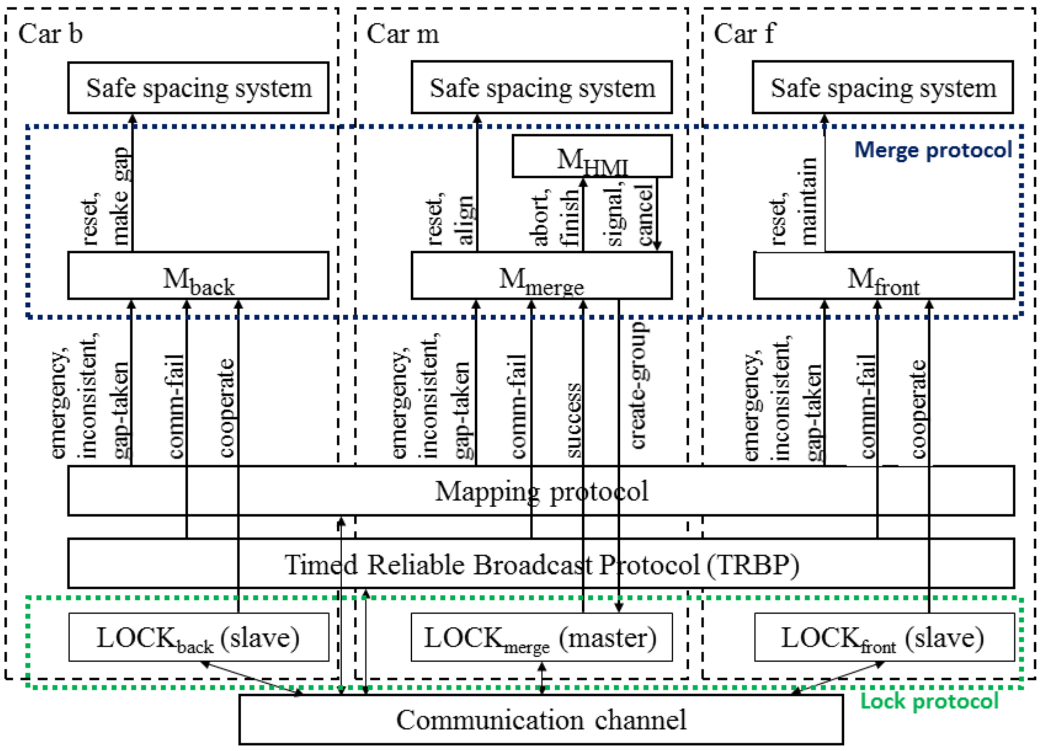


Figure 7.5: Interactions between FSMs and the interfaces

and it depends on the vehicles' speed, acceleration, traffic condition, etc. The duration of phase 3 must be long enough for the driver to react to the green light and steer into the gap without haste.

The merge protocol is expected to satisfy the following properties. We use stratified verification to establish the bound of the probability that these properties are violated.

1. Creates the gap and notifies the driver only if none of the undesirable events occurs
2. If any of the undesirable events occurs, the driver is notified and the safe spacing system is reset to conservative mode for emergency maneuver

Undesirable events include failures of any component, occurrence of incidents in the vicinity of the cooperating group, interfering vehicle occupying the gap, and driver ignoring the notification to change lanes.

The first property is checked by determinining if there is a path from the initial state to the state where the driver is notified to change lanes. The second property can be

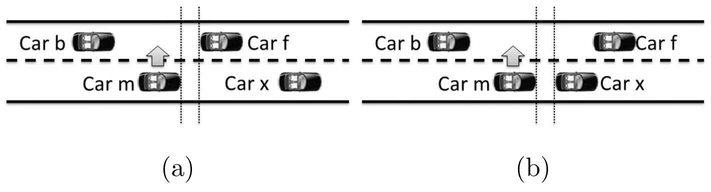


Figure 7.6: Car m attempts to merge between car f and car b

represented as a regular safety property A_{merge}, which is elaborated further in the following section.

7.3 Verification of the Driver-assisted Merging

In this section, the driver-assisted merging is put under scrutiny. We check if the merge protocol satisfies the properties presented at the end of last section. Recall that verification of a protocol systems consist of three stages: model construction, running the model checker, and result analysis. First the DMDP model M_{merge} of driver-assisted merging is constructed from the FSMs of the merge protocol and the service models that represent the component on which the merge protocol depends. Also, we construct the DFAs that represent various safety properties of the components and the merge protocol. Next, we use stratified verification to find the probability bound for the driver-assisted merging to violate the properties in last section, when all the components deliver their guarantees with required probabilities. Finally, we shows that the probability bound we find is sufficiently low.

The stratified verification is done under a compositional framework. The components other than the merge protocol are assumed to be verified to provide their guarantees before the verification of the whole system is conducted. The guarantees they provide are represented as DFAs and the restricted set of adversaries that reflect the failure probabilities of the guarantees. The following are the requirements of the mapping protocol, TRBP, and the lock protocol.

$$\begin{aligned}\langle true\rangle M_{\text{mapping}}\langle A_{\text{mapping},1}, A_{\text{mapping},2}\rangle_{\geq p_{\text{mapping},1}, p_{\text{mapping},2}} \\ \langle true\rangle M_{\text{trbp}}\langle A_{\text{trbp}}\rangle_{\geq p_{\text{TRBP}}} \\ \langle true\rangle M_{\text{lock}}\langle A_{\text{lock}}\rangle_{\geq p_{\text{lock}}}\end{aligned} \tag{7.1}$$

Safety property $A_{\text{mapping},1}$ means that the mapping protocol should not have a detectable sensor malfunction with probability greater than $p_{\text{mapping},1}$. Safety property $A_{\text{mapping},2}$ means that the mapping protocol should not have a failure goes undetected with probability greater than $p_{\text{mapping},2}$. Safety property A_{trbp} is about the communication failure experienced by TRBP. Such communication failure arises from unrecoverable token or messages and is assumed not to occur with probability greater than p_{TRBP}. In this thesis, we do not show that the sensor systems. Rather, we consider them assumptions that we place on the sensor system and TRBP.

Safety property A_{lock} is slightly more complex. It requires that the success in group creation is preceded by both the front vehicle and the back vehicle agreeing to cooperate, which are then preceded by the merging vehicle making an attempt to create a cooperating group. In this section, we use stratified verification to show that the lock protocol satisfies A_{lock} with probability greater than p_{lock}. That is, it exhibits an unexpected sequences with probability less than $1 - p_{\text{lock}}$.

We would like to show that driver-assisted merging satisfies A_{merge} given that the sensor system, TRBP, and the lock protocol deliver their guarantees with required probabilities. That is, we are to show that

$$\langle A_{\text{mapping},1}, A_{\text{mapping},2}, A_{\text{trbp}}, A_{\text{lock}}\rangle_{\geq p_{\text{mapping},1}, p_{\text{mapping},2}, p_{\text{trbp}}, p_{\text{lock}}} F_{\text{merge}}\langle A_{\text{merge}}\rangle_{\geq p_{\text{merge}}} \tag{7.2}$$

This can be shown by having stratified verification traverse that state space within the restricted set of adversaries. With (7.1) and (7.2), we have

$$\langle true\rangle M_{\text{mapping}} \parallel M_{\text{trbp}} \parallel M_{\text{lock}} \parallel F_{\text{merge}}\langle A_{\text{merge}}\rangle_{\geq p_{\text{merge}}} \tag{7.3}$$

Dividing the verification of driver-assisted merging into verification of (7.1) and (7.2) greatly reduces the complexity of verification. This can be seen from that the number of states traversed during verification of the merge protocol is a lot less than the number of states traversed when verifying the lock protocol.

Note that the design of the merge protocol cannot be satisfactory on the first try. The design process of the merge protocol involves repeating efforts of specifying the merge protocol, constructing the model, running the model checker, then analyzing the result to decide the causes of errors. The process is repeated until the error probability is low enough. The specification of the merge protocol presented in Section 7.2 is the final design of the merge protocol that is deemed robust enough.

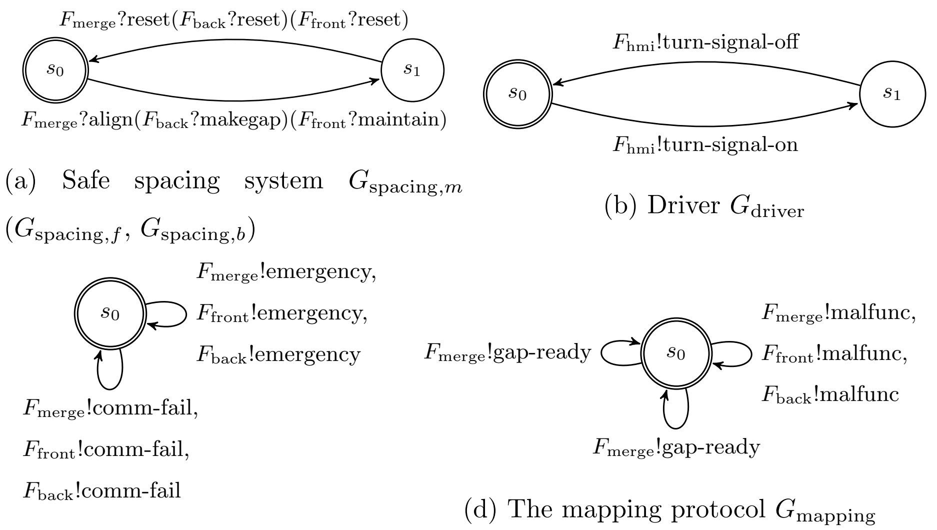
In this section, we first describe the service models of the components ($G_{\text{spacing},m}$, $G_{\text{spacing},f}$, $G_{\text{spacing},b}$, G_{trbp}, G_{mapping}, $G_{\text{lock},m}$, $G_{\text{lock},f}$, $G_{\text{lock},b}$, and G_{driver}) and the DFAs that represent the safety properties ($A^*_{\text{mapping},1}$, $A^*_{\text{mapping},2}$, A^*_{trbp}, and A^*_{lock}) as the requirements we place on the components and the guarantees we expect the driver-assisted merging to satisfy. We deviate a bit by using stratified verification to find the probability for the lock protocol to satisfy A_{lock}. Finally we run stratified verification on the restricted set of adversaries and show that the failure probability of driver-assisted merging is sufficiently small.

7.3.1 Service Models of Components

The environment and the interfaces of the components that the merge protocol depends on are represented as the service models shown in Fig.**??**. They are extremely simple, especially when being compared with their actual implementation.

The service models of the safe spacing systems ($G_{\text{spacing},m}, G_{\text{spacing},f}$, and $G_{\text{spacing},b}$, Fig.7.7a) on each vehicle are also highly simplified. There are only two states: s_0 represents normal driving and s_1 represents adjustment of headway and speed settings for gap creation. The safe spacing systems work differently when the vehicle takes different role in a merge. When on the mering car, safe spacing system takes command *align*; on the back car, safe spacing system takes command *make-gap*; on the front car, safe spacing takes command *maintain*.

The mapping protocol (G_{mapping}, Fig.7.7d) and TRBP (G_{trbp}, Fig.7.7c) have similar service models. These two components monitor the surroundings of the cooperating group


F_{merge}?reset(F_{back}?reset)(F_{front}?reset)
s_0
s_1
F_{merge}?align(F_{back}?makegap)(F_{front}?maintain)
F_{hmi}!turn-signal-off
s_0
s_1
F_{hmi}!turn-signal-on
s_0
F_{merge}!emergency,
F_{front}!emergency,
F_{back}!emergency
F_{merge}!comm-fail,
F_{front}!comm-fail,
F_{back}!comm-fail
F_{merge}!gap-ready
s_0
F_{merge}!malfunc,
F_{front}!malfunc,
F_{back}!malfunc
F_{merge}!gap-ready

(a) Safe spacing system $G_{\text{spacing},m}$ ($G_{\text{spacing},f}$, $G_{\text{spacing},b}$)

(b) Driver G_{driver}

(c) TRBP

(d) The mapping protocol G_{mapping}

Figure 7.7: Service models of the components, part 1

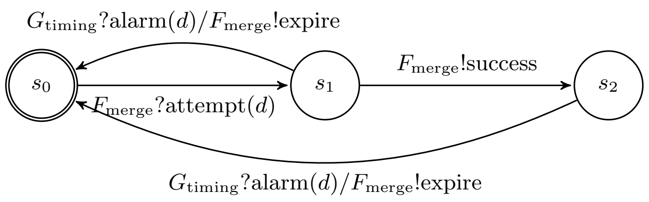
and the condition of communication. They are stateless, and they simply generate messages that indicate the events that might occur during a merge maneuver.

The service model of the driver (G_{driver}, Fig.7.7b) provides external stimuli that initiate the merge maneuver. Given the simple interface exposed to the driver, i.e., the FSM F_{HMI}, the driver can do but two things: switching on and off the turn signal.

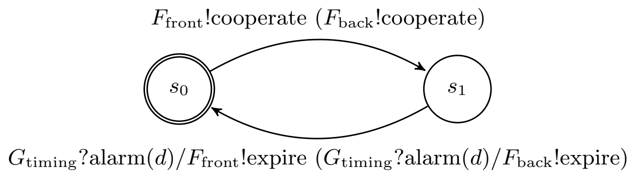
The service models of the lock protocol on the merging vehicle ($G_{\text{lock},m}, G_{\text{lock},f}$, and $G_{\text{lock},b}$, Fig.7.8a) allows F_{merge} to create a cooperating group (F_{merge}?attempt(d)). Normally it respond by saying a cooperating group is created successfully (F_{merge}!success), while sometimes it expires before a group can be created due to causes including message loss. Eventually the cooperating group is dismissed on reaching the timestamp d. The service models on the front car or the back car is simpler (Fig.7.8b). It prompts F_{front} or F_{back} to start cooperating with the merging car and to carry out required actions. It eventually expires on reaching timestamp d as well.

7.3.2 Service Guarantees as Regular Safety Properties

Next, we look at the probabilistic safety properties that we expect the merge protocol to satisfy and the probabilistic safety properties that are assumed to be satisfied by coordinated


G_{timing}?alarm(d)/F_{merge}!expire
s_0
s_1
F_{merge}!success
s_2
F_{merge}?attempt(d)
G_{timing}?alarm(d)/F_{merge}!expire

(a) Lock protocol on Merging Car $G_{\text{lock},m}$


F_{front}!cooperate (F_{back}!cooperate)
s_0
s_1
G_{timing}?alarm(d)/F_{front}!expire (G_{timing}?alarm(d)/F_{back}!expire)

(b) Lock protocol on Front Car $G_{\text{lock},f}$ (Back Car $G_{\text{lock},b}$)

Figure 7.8: Service models of the components, part 2

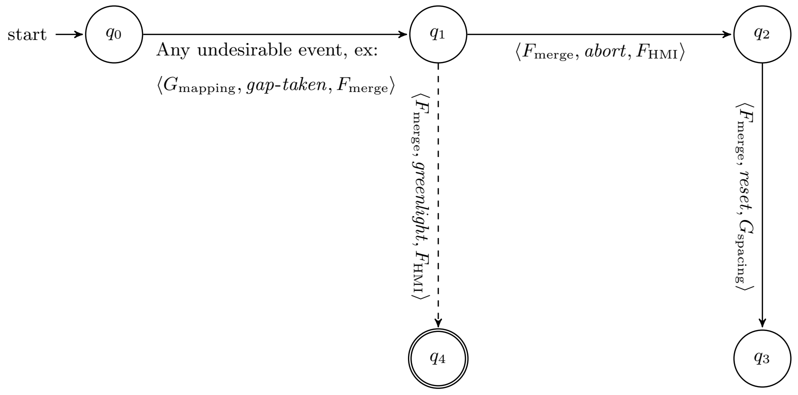
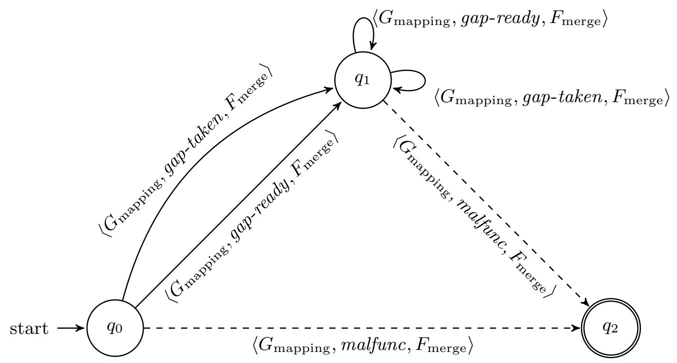

start
q_0
q_1
q_2
Any undesirable event, ex:
$\langle G_{\text{mapping}}, \textit{gap-taken}, F_{\text{merge}} \rangle$
$\langle F_{\text{merge}}, \textit{abort}, F_{\text{HMI}} \rangle$
$\langle F_{\text{merge}}, \textit{greenlight}, F_{\text{HMI}} \rangle$
$\langle F_{\text{merge}}, \textit{reset}, G_{\text{spacing}} \rangle$
q_4
q_3

Figure 7.9: DFA A^*_{merge}: the labels in the alphabet that are not shown result in self-loop

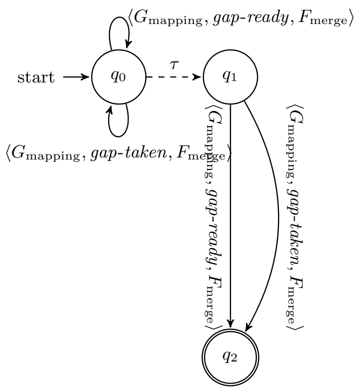
sensors, TRBP, and the lock protocol. They are $\langle A_{\text{mapping},1} \rangle_{\geq p_{\text{mapping},1}}$, $\langle A_{\text{mapping},2} \rangle_{\geq p_{\text{mapping},2}}$, $\langle A_{\text{trbp}} \rangle_{\geq p_{\text{trbp}}}$, and $\langle A_{\text{lock}} \rangle_{\geq p_{\text{lock}}}$. Their regular safety properties parts are represented as DFAs. For each probabilistic safety properties, we define the restricted set of adversaries that are required for proving (7.2) using stratified verification.

Since the mapping protocol uses multiple sensors to estimate a distance, it is resistant to single sensor failure. The fusion algorithm can reject an incorrect sensor readings by comparing it with readings of other sensors. Sensor malfunction is assumed to occur with probability less than $1 - p_{\text{mapping},1}$. However, it is possible on some rare occasions for all of the sensors to provide incorrect measurements, which can result in an incorrect mapping



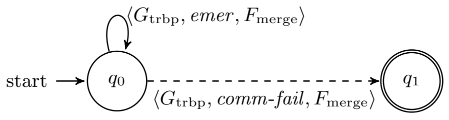
$\langle G_{\text{mapping}}, \textit{gap-ready}, F_{\text{merge}}\rangle$
q_1
$\langle G_{\text{mapping}}, \textit{gap-taken}, F_{\text{merge}}\rangle$
$\langle G_{\text{mapping}}, \textit{gap-taken}, F_{\text{merge}}\rangle$
$\langle G_{\text{mapping}}, \textit{gap-ready}, F_{\text{merge}}\rangle$
$\langle G_{\text{mapping}}, \textit{malfunc}, F_{\text{merge}}\rangle$
start
q_0
q_2
$\langle G_{\text{mapping}}, \textit{malfunc}, F_{\text{merge}}\rangle$

(a) DFA $A^*_{\text{mapping},1}$



$\langle G_{\text{mapping}}, \textit{gap-ready}, F_{\text{merge}}\rangle$
start
q_0
τ
q_1
$\langle G_{\text{mapping}}, \textit{gap-taken}, F_{\text{merge}}\rangle$
$\langle G_{\text{mapping}}, \textit{gap-ready}, F_{\text{merge}}\rangle$
$\langle G_{\text{mapping}}, \textit{gap-taken}, F_{\text{merge}}\rangle$
q_2

(b) DFA $A^*_{\text{mapping},2}$



$\langle G_{\text{trbp}}, \textit{emer}, F_{\text{merge}}\rangle$
start
q_0
q_1
$\langle G_{\text{trbp}}, \textit{comm-fail}, F_{\text{merge}}\rangle$

(c) DFA A^*_{trbp}

Figure 7.10: DFAs representing regular safety properties

of the surrounding objects. The mapping protocol still renders the mapping, and may incorrectly notify the merge protocol that the target gap is large enough while in fact otherwise. This failure should be extremely rare, and it is assumed to occur with probability less than $1 - p_{\text{mapping},2} \ll 1 - p_{\text{mapping},1}$. Even though the probability of such rare events can be reduced by either introducing more sensors or employing a more sophisticated fusion algorithm, the probability of failure $1 - p_{\text{mapping},1}$ cannot be reduced to zero.

Whenever an execution includes a sensor malfunction event, it is an accepting run of property $A_{\text{mapping},1}$ (Fig.7.10a). The corresponding set of adversaries is

$$Adv^{\text{mapping},1} = \left\{ \sigma \in Adv \middle| \sigma(\langle s, q\rangle)(\alpha) \leq 1 - p_{\text{mapping},1} \text{if } \alpha \in Act(s) \wedge \delta_{A^*_{\text{mapping},1}}(q, \alpha) = q_2 \right\} \tag{7.4}$$

Mapping protocol also fails silently when all sensors malfunctioning so that there is no way to tell which sensor has failed (Fig.7.10b). The silent failure is represented as a τ-transition. The corresponding set of adversaries is

$$Adv^{\text{mapping},2} = \left\{ \sigma \in Adv \middle| \sigma(\langle s, q\rangle)(\alpha) \leq 1 - p_{\text{mapping},2} \text{if } \alpha \in Act(s) \wedge \delta_{A^*_{\text{mapping},2}}(q, \alpha) = q_1 \right\} \tag{7.5}$$

TRBP fails (Fig.7.10c) when a token or a broadcasted message is not recovered within a deadline. The causes could be consecutive message losses or vehicle moving out of range. The probability of vehicle moving out of range is difficult to characterize. The corresponding set of adversaries is

$$Adv^{\text{trbp}} = \left\{ \sigma \in Adv \,\middle|\, \sigma(\langle s, q \rangle)(\alpha) \le 1 - p_{\text{trbp}} \text{ if } \alpha \in Act(s) \wedge \delta_{A^*_{\text{trbp}}}(q, \alpha) = q_1 \right\} \tag{7.6}$$

Although the property of the lock protocol may seem complex (Fig.7.11), it simply requires that the whole process starts with an attempt of F_{merge}, which is then followed by both F_{back} and F_{front} agreeing to cooperate. The lock protocol then notifies F_{merge} that a cooperating group is formed after both F_{front} and F_{back} agrees to cooperate. In every state of DFA, it is possible for the group creation to fail and the attempt of group creation simply expires ($q_1 \to q_6$, $q_2 \to q_6$, $q_3 \to q_6$, $q_4 \to q_6$, and $q_5 \to q_6$). Any sequence other than the ones described above is an error sequence. The corresponding set of adversaries is

$$Adv^{\text{lock}} = \left\{ \sigma \in Adv \,\middle|\, \sigma(\langle s, q \rangle)(\alpha) \le 1 - p_{\text{lock}} \text{ if } \alpha \in Act(s) \wedge \delta_{A^*_{\text{lock}}}(q, \alpha) = q_7 \right\} \tag{7.7}$$

To see if $\langle true \rangle M_{\text{lock}} \langle A_{\text{lock}} \rangle_{\ge p_{\text{lock}}}$, we consider the 7-vehicle scenario shown in Fig.7.12. There are four users, namely user 2, user 3, user 5, and user 6, that attempt to create co-operating group. The system model of DMDP M_{lock} is contructed from the 7 FSMs of the lock protocol specification, full-duplex channel service models between each pair of FSMs, the service model of the timing stack, and the service models of the merge protocol, which make merge request. Specifically, we look at user 2, which attempt to create a group with user 6 and user 7 or a group with user 3 and user 5. The MDP-DFA product involves the alphabet set: $\{\langle G_{\text{merge}}, \mathit{attempt}, F_{\text{lock},m} \rangle, \langle F_{\text{lock},m}, \mathit{success}, G_{\text{merge}} \rangle, \langle F_{\text{lock},f}, \mathit{cooperate}, G_{\text{front}} \rangle, \langle G_{\text{timing}}, \mathit{alarm}(d), F_{\text{lock},m} \rangle\}$.

The stratified verification in Algorithm 4 is applied to $M^* = M_{\text{lock}} \otimes A^*_{\text{lock}}$ with all adversaries being considered. We require that the message loss rate of the communication channel is less than $\hat{p} = 10^{-5}$. This is also used as the discretization parameter. The algorithm is able to explore 1,002,242 reachable states, including all states up to equivalent class 4, using 2.2GB of memory. It concludes that $p_{\text{lock}} = 1 - 3.35 \times 10^{-20}$.

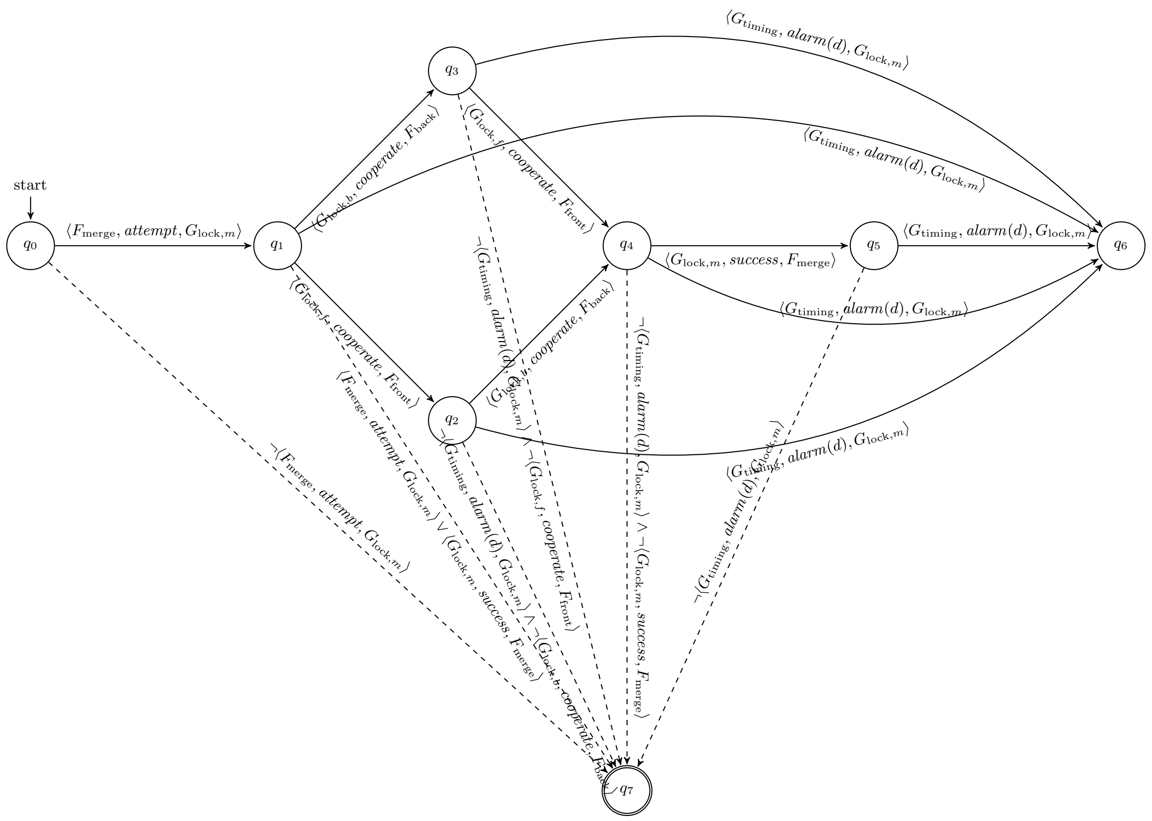

start
q_0
q_1
q_2
q_3
q_4
q_5
q_6
q_7
$\langle F_{\text{merge}}, \textit{attempt}, G_{\text{lock},m}\rangle$
$\langle G_{\text{lock},b}, \textit{cooperate}, F_{\text{back}}\rangle$
$\langle G_{\text{lock},f}, \textit{cooperate}, F_{\text{front}}\rangle$
$\langle G_{\text{timing}}, \textit{alarm}(d), G_{\text{lock},m}\rangle$
$\langle G_{\text{lock},m}, \textit{success}, F_{\text{merge}}\rangle$
$\neg\langle F_{\text{merge}}, \textit{attempt}, G_{\text{lock},m}\rangle$
$\langle F_{\text{merge}}, \textit{attempt}, G_{\text{lock},m}\rangle \vee \langle G_{\text{lock},m}, \textit{success}, F_{\text{merge}}\rangle$
$\neg\langle G_{\text{timing}}, \textit{alarm}(d), G_{\text{lock},m}\rangle \wedge \neg\langle G_{\text{lock},b}, \textit{cooperate}, F_{\text{back}}\rangle$
$\neg\langle G_{\text{timing}}, \textit{alarm}(d), G_{\text{lock},m}\rangle \wedge \neg\langle G_{\text{lock},f}, \textit{cooperate}, F_{\text{front}}\rangle$
$\neg\langle G_{\text{timing}}, \textit{alarm}(d), G_{\text{lock},m}\rangle \wedge \neg\langle G_{\text{lock},m}, \textit{success}, F_{\text{merge}}\rangle$
$\neg\langle G_{\text{timing}}, \textit{alarm}(d), G_{\text{lock},m}\rangle$

Figure 7.11: DFA A^*_{lock}

7.3.3 Verifying the Merge Protocol

We are now ready to apply stratified verification to driver-assisted mering. It is modeled as a DMDP

$$\begin{aligned} M_{\text{merge}} =& F_{\text{merge}} \otimes F_{\text{front}} \otimes F_{\text{back}} \otimes F_{\text{hmi}} \otimes \\ & G_{\text{spacing},m} \otimes G_{\text{spacing},f} \otimes G_{\text{spacing},b} \otimes \\ & G_{\text{trbp}} \otimes G_{\text{mapping}} \otimes G_{\text{lock},m} \otimes G_{\text{lock},f} \otimes G_{\text{lock},b} \otimes G_{\text{driver}} \end{aligned} \tag{7.8}$$

We verify if the merge protocol satisfies $\langle A_{\text{merge}}\rangle_{\geq p_{\text{merge}}}$ with a sufficiently high probability bound p_{merge}, assuming that all components provide their guarantees with required probability. Specifically, we use stratified verification to show that (7.2) holds by showing that

$$\forall \sigma \in Adv^{A_{\text{mapping},1}} \cap Adv^{A_{\text{mapping},2}} \cap Adv^{A_{\text{trbp}}} \cap Adv^{A_{\text{lock}}} \cdot P^{\sigma}_{M_{\text{merge}}}(A_{\text{merge}}) \geq p_{\text{merge}} \tag{7.9}$$

Then we show that the bound we find is sufficiently small.

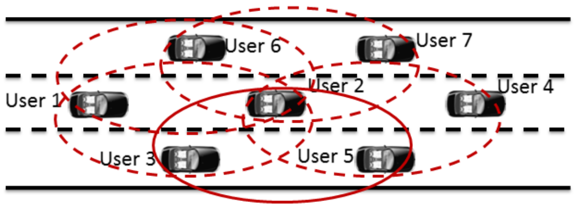


Figure 7.12: 7-party scenario of the lock protocol

In verification of the merge protocol, we assume that following: $p_{\text{mapping},1} = 1 - 10^{-4}$, $p_{\text{mapping},2} = 1 - 10^{-16}$, and $p_{\text{trbp}} = 0$. That is, a detectable sensor malfunction occurs with probability less than $1 - p_{\text{mapping},1} = 10^{-4}$, and a undetectable sensor failure occurs with extremely low probability $1 - p_{\text{mapping},2} = 10^{-16}$. TRBP may fail if the driver on the front car or the back car steering the car out of the cooperating group, so we conservatively assume that this probability is less than $1 - p_{\text{trbp}} = 1$ at best.

Setting the discretization parameter $\hat{p} = 10^{-4}$, the sets of adversaries that correspond to the assumptions can be discretized. Thus we have the following

$$Adv_D^{A_{\text{mapping},1}} = \left\{ \sigma \in Adv_D \,\middle|\, \sigma_D(\langle s, q_{s,1}\rangle)(\alpha) = \begin{array}{l} 1 \text{ if } \alpha \in Act(s) \wedge \delta_{A_{\text{mapping},1}^*}(q_{s,1}, \alpha) = q_2, \\ 0, \text{ otherwise} \end{array} \right\} \tag{7.10}$$

$$Adv_D^{A_{\text{mapping},2}} = \left\{ \sigma \in Adv_D \,\middle|\, \sigma_D(\langle s, q_{s,2}\rangle)(\alpha) = \begin{array}{l} 4 \text{ if } \alpha \in Act(s) \wedge \delta_{A_{\text{mapping},2}^*}(q_{s,2}, \alpha) = q_1, \\ 0, \text{ otherwise} \end{array} \right\} \tag{7.11}$$

$$Adv_D^{A_{\text{trbp}}} = \{\sigma \in Adv_D \,|\sigma_D(\langle s, q_t\rangle)(\alpha) = 0 \text{ if } \alpha \in Act(s)\} \tag{7.12}$$

$$Adv_D^{A_{\text{lock}}} = \left\{ \sigma \in Adv_D \,\middle|\, \sigma_D(\langle s, q_l\rangle)(\alpha) = \begin{array}{l} 4 \text{ if } \alpha \in Act(s) \wedge \delta_{A_{\text{lock}}^*}(q_l, \alpha) = q_7, \\ 0, \text{ otherwise} \end{array} \right\} \tag{7.13}$$

where $q_{s,1}$ is the state of $A^*_{\text{mapping},1}$, $q_{s,2}$ is the state of $A^*_{\text{mapping},2}$, q_t is the state of A^*_{trbp}, and q_l is the state of A^*_{lock}.

Let $\langle s, q_{s,1}, q_{s,2}, q_t, q_l\rangle$ be a state of $M_{\text{merge}} \otimes A^*_{\text{mapping},1} \otimes A^*_{\text{mapping},2} \otimes A^*_{\text{trbp}} \otimes A^*_{\text{lock}}$, where

s is the composite state of DMDP M_{merge}. The intersection of restricted adversary sets is

$$\begin{aligned} Adv_D^{\text{all}} &= Adv_D^{\text{mapping},1} \cup Adv_D^{\text{mapping},2} \cup Adv_D^{\text{trbp}} \cup Adv_D^{\text{lock}} \\ &= \left\{ \sigma_D \in Adv_D \;\middle|\; \begin{array}{l} \sigma_D(\langle s, q_{s,1}, q_{s,2}, q_t, q_l \rangle)(\alpha) = \\ \quad 1, \text{ if } \alpha \in Act(s) \wedge \delta_{A^*_{\text{mapping},1}}(q_{s,1}, \alpha) = q_2 \\ \quad 4, \text{ if } \alpha \in Act(s) \wedge (\delta_{A^*_{\text{mapping},2}}(q_{s,2}, \alpha) = q_1 \vee \delta_{A^*_{\text{lock}}}(q_l, \alpha) = q_7) \\ \quad 0, \text{ otherwise.} \end{array} \right\} \end{aligned} \tag{7.14}$$

Now we use Algorithm 3 to explore the state space of $M' = M_{\text{merge}} \otimes A^*_{\text{mapping},1} \otimes A^*_{\text{mapping},2} \otimes A^*_{\text{trbp}} \otimes A^*_{\text{lock}} \otimes A^*_{\text{merge}}$. Stratified verification traverses 402 distinct states without finding an error up to the equivalent class 3. While there are still reachable states that are yet to be explored, the procedure is able to compute the upper-bound of probability for the system to take those unexamined sequences. The linear program yields a bound of $1 - p_{\text{merge}} = \hat{p}^4 = 1 \times 10^{-16}$.

We may conclude that the protocol is safe if the bound is sufficiently small to provide the confidence required by the cooperative driving application. It means that an error occurs less than once every 1×10^{16} protocol invocations. Suppose there are 500 million vehicles [1] equipped with the merge protocol, and each of which travels 200,000 miles during its lifetime [Ford, 2012]. The driver of the vehicle should drive extremely aggressive and use the merge protocol to change lane 100 times every mile [2], on the average less than one unexplored state will occur during the lifetime of all these vehicles. Clearly, this level of confidence will never be achieved by driving vehicles on a test track.

If we continue probabilistic verification to consider those states reachable via sequences with bound as low as $\hat{p}^4$, we encounter level-4 low probability transitions. The level-4 low probability transitions are the 'silent' failures that cannot be detected by the merge protocol and therefore no handling of such failures can be done. These types of failure prevent the merge protocol from being perfectly safe, but stratified verification can upper-bound the error probability and ensure that the error probability is sufficiently small for the purpose

[1] 87 million vehicles are produced in the single year of 2013 [OIAC, 2015]

[2] US drivers average one lane change every 2.8 miles [NHTSA, 2015]

of cooperative driving application.

If, however, the bound we have still does not give the expected degree of confidence, probabilistic verification has identified the cause of failure being those level-4 low probability events that correspond to the silent failures. We may improve the mapping protocol by introducing redundancy to the set of sensor or using a more advanced fusion algorithm so as to drive the probability of silent failure smaller. We should also replace the acknowledgment-based communication protocol that the lock protocol depends on by a more sophisticated communication protocol so that the lock protocol exhibit unexpected behaviors with even smaller probability.

7.4 Conclusions

In this chapter, we designed driver-assisted merging and verified its safety. The multiple stack architecture divides driver-assisted merging into components with distinct functions. A reasonably simple top-level logic called the merge protocol coordinates joint actions among the three cooperating vehicles, sends commands to the lower-level components and responds to events detected by the components that interact with the physical world. We presented its specification, described as a collection of FSMs, and the safety guarantees which it should deliver.

Verification of driver-assisted merging was done in several steps. First, the components with which the merge protocol interacts are modeled as service models. The properties they are assumed to deliver are described as regular safety properties, which can be represented as DFAs. We placed probabilistic assumption on the failure probability of these components except for the lock protocol, to which we also applied stratified verification to prove that it satisfies the property with the assumed probability. Finally, as driver-assisted merging is the composition of the merge protocol and the components it depends on, we used stratified verification in the compositional framework to evaluate its safety. We were able to guarantee that driver-assisted merging can only fail with an extremely low probability that cannot be achieved by simulation or test tracks.

Chapter 8

Conclusions

In this thesis we consider the techniques that address the challenges we face when applying formal verification techniques to cooperative driving systems. They include a *multiple stack architecture*, a framework for leveraging *GPS clocks* in protocol design, and *stratified probabilistic verification.* Finally, we present our design of driver-assisted merging and use the discussed technique to prove that it is safe to a degree that cannot be guranteed by simulation or test tracks.

We faced several challenged when designing a driver-assisted merging application. It interacts with the physical world in multiple ways. The multiple stack architecture is designed to have each stack in the architecture address a particular way of interaction. The top-level logic of a cooperative driving application can be reasonably simplified with the layered structure. The top-level logic of driver-assisted merging application, the merge protocol, can be thus specified as FSMs. Having functions arranged in a layered fashion helps us divide the verification of the whole system into smaller subproblems of independent module verification. In driver-assisted merging application, both the merge protocol and the lock protocol that resolves conflicting merge requests for driver-assisted merging need to execute time-critical actions. We introduced the timing stack, which separates a process into two parts: the part modeled as an finite-state machine that controls state transitions and messages exchanges, and the part that determines the exact moment that a timed event should occur. This also reduces the complexity of protocol specification as timing constraints are controlled by a separate processes. Moreover, having accurate clocks at different locations allows processes to execute actions simultaneously, reducing interleaving that often arises in systems that use multiple timers to control timed events. To alleviate state explosion, we presented stratified probabilistic verification. It greatly improves the probability bound obtained in the original probabilistic verification algorithm. It prioritizes the states that are more likely to be encountered during system execution. When running out of memory, it constructs a linear program whose solution is the upper bound for the probability of reaching the unexplored states and the error states. Along with the multiple stack architecture, the stratified algorithm is particularly useful when verifying a protocol system that depends on several imperfect components that may fail with small but hard-to-quantify probabilities. We adopted a compositional approach to verify a collection of components, assuming that the components have inexact probability guarantees. We applied stratified verification to driver-assisted merging, which is already simplified by the architecture and the use of GPS clocks during its design stage. The verdict: the driver-assisted merging application fails less than once every 5×10^{13} merge attempts.

Bibliography

[Abubakari and Sastry, 2008] Hamza Abubakari and Shivakumar Sastry. Ieee 1588 style synchronization over wireless link. In *Precision Clock Synchronization for Measurement, Control and Communication, 2008. ISPCS 2008. IEEE International Symposium on*, pages 127–130. IEEE, 2008.

[Aeberhard *et al.*, 2012] M. Aeberhard, S. Schlichtharle, N. Kaempchen, and T. Bertram. Track-to-Track Fusion With Asynchronous Sensors Using Information Matrix Fusion for Surround Environment Perception. *IEEE Transactions on Intelligent Transportation Systems*, 13(4):1717–1726, 2012.

[AG, 2015] Daimler AG. Mercedes benz - intelligent drive, September 2015.

[Aho *et al.*, 1991] Alfred V Aho, Anton T Dahbura, David Lee, and M Umit Uyar. An optimization technique for protocol conformance test generation based on uio sequences and rural chinese postman tours. *Communications, IEEE Transactions on*, 39(11):1604–1615, 1991.

[Alur and Dill, 1994] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[Alur and Henzinger, 1999] Rajeev Alur and Thomas A Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[Alur *et al.*, 1990] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 414–425. IEEE, 1990.

[Alur, 1999] Rajeev Alur. Timed automata. In *Computer Aided Verification*, pages 8–22. Springer Berlin Heidelberg, 1999.

[Alvarez and Horowitz, 1999a] Luis Alvarez and Roberto Horowitz. Safe platooning in automated highway systems part i: Safety regions design. *Vehicle System Dynamics*, 32(1):23–55, 1999.

[Alvarez and Horowitz, 1999b] Luis Alvarez and Roberto Horowitz. Safe platooning in automated highway systems part ii: velocity tracking controller. *Vehicle System Dynamics*, 32(1):57–84, 1999.

[Amditis *et al.*, 2010] A. Amditis, E. Bertolazzi, M. Bimpas, F. Biral, P. Bosetti, M. Da Lio, L. Danielsson, A. Gallione, H. Lind, A. Saroldi, and A. Sjögren. A Holistic Approach to the Integration of Safety Applications: The INSAFES Subproject Within the European Framework Programme 6 Integrating Project PReVENT. *IEEE Transactions on Intelligent Transportation Systems*, 11(3):554–566, 2010.

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

[Banerjea *et al.*, 1996] Anindo Banerjea, Domenico Ferrari, Bruce A Mah, Mark Moran, Dinesh C Verma, and Hui Zhang. The tenet real-time protocol suite: Design, implementation, and experiences. *IEEE/ACM Transactions on Networking (TON)*, 4(1):1–10, 1996.

[Behere *et al.*, 2013] Sagar Behere, Martin Törngren, and De-Jiu Chen. A reference architecture for cooperative driving. *journal of Systems Architecture*, 59(10):1095–1112, 2013.

[Bento *et al.*, 2012] Luis Conde Bento, Ricardo Parafita, and Urbano Nunes. Inter-vehicle sensor fusion for accurate vehicle localization supported by v2v and v2i communications. In *Intelligent Transportation Systems (ITSC), 2012 15th International IEEE Conference on*, pages 907–914. IEEE, 2012.

[Bertolazzi *et al.*, 2010] E. Bertolazzi, F. Biral, M. Da Lio, A. Saroldi, and F. Tango. Supporting Drivers in Keeping Safe Speed and Safe Distance: The SASPENCE Subproject Within the European Framework Programme 6 Integrating Project PReVENT. *IEEE Transactions on Intelligent Transportation Systems*, 11(3):525–538, 2010.

[Broggi *et al.*, 2013] A. Broggi, M. Buzzoni, S. Debattisti, P. Grisleri, M.C. Laghi, P. Medici, and P. Versari. Extensive Tests of Autonomous Driving Technologies. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1403–1415, 2013.

[Brun *et al.*, 2003] R Brun, R Rausch, and Claude Henri Sicard. Using worldfip for synchronization and time stamping in the lhc accelerator. Technical report, 2003.

[Caveney, 2010] D. Caveney. Cooperative Vehicular Safety Applications. *Control Systems, IEEE*, 30(4):38 –53, August 2010.

[Chu and Liu, 1988] Peil-Ying M Chu and Ming T Liu. Synthesizing protocol specifications from service specifications in fsm model. In *Computer Networking Symposium, 1988., Proceedings of the*, pages 173–182. IEEE, 1988.

[Clarke and Emerson, 1982] Edmund M Clarke and E Allen Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*. Springer, 1982.

[Clarke and Wang, 2014] Edmund M Clarke and Qinsi Wang. 2ˆ 5 years of model checking. In *Perspectives of System Informatics*, pages 26–40. Springer, 2014.

[Clarke *et al.*, 1989] Edmund M Clarke, David E Long, and Kenneth L McMillan. Compositional model checking. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*, pages 353–362. IEEE, 1989.

[Clarke *et al.*, 2001] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal methods in system design*, 19(1):7–34, 2001.

[Cooklev *et al.*, 2007] Todor Cooklev, John C Eidson, and Afshaneh Pakdaman. An implementation of ieee 1588 over ieee 802.11 b for synchronization of wireless local area network nodes. *Instrumentation and Measurement, IEEE Transactions on*, 56(5):1632–1639, 2007.

[Corbett *et al.*, 2013] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[Dang *et al.*, 2012] Thao Dang, Jens Desens, Uwe Franke, Dariu Gavrila, Lorenz Schäfers, and Walter Ziegler. Steering and evasion assist. In *Handbook of intelligent vehicles*, pages 759–782. Springer, 2012.

[D'Argenio *et al.*, 1997] Pedro R D'Argenio, J-P Katoen, Theo C Ruys, and Jan Tretmans. *The bounded retransmission protocol must be on time!* Springer, 1997.

[Dye, 2014] Jessica Dye. Deaths linked to GM ignition-switch defect rise to 27. *Reuters*, October 2014.

[Edelkamp *et al.*, 2004] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International journal on software tools for technology transfer*, 5(2-3):247–267, 2004.

[Eidson and Lee, 2003] John C Eidson and Kang Lee. Sharing a common sense of time. *Instrumentation & Measurement Magazine, IEEE*, 6(1):26–32, 2003.

[Etessami *et al.*, 2007] Kousha Etessami, Marta Kwiatkowska, Moshe Y Vardi, and Mihalis Yannakakis. Multi-objective model checking of markov decision processes. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 50–65, 2007.

[Fecko *et al.*, 2003] Mariusz Fecko, M Ümit Uyar, Ali Y Duale, Paul D Amer, et al. A technique to generate feasible tests for communications systems with multiple timers. *Networking, IEEE/ACM Transactions on*, 11(5):796–809, 2003.

[Ford, 2012] Dexter Ford. As Cars Are Kept Longer, 200,000 Is New 100,000. *The New York Times*, March 2012.

[Geiger *et al.*, 2012] Andreas Geiger, Martin Lauer, Frank Moosmann, Benjamin Ranft, Holger Rapp, Christoph Stiller, and Julius Ziegler. Team annieway's entry to the 2011 grand cooperative driving challenge. *Intelligent Transportation Systems, IEEE Transactions on*, 13(3):1008–1017, 2012.

[Gerth *et al.*, 1995] Rob Gerth, Doron Peled, Moshe Y Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *International Symposium on Protocol Specification, Testing and Verification*. IFIP, 1995.

[Godbole *et al.*, 1998] Datta N Godbole, Raja Sengupta, and Veit Hagenmeyer. Distributed hybrid controls for automated vehicle lane changes. In *Decision and Control, 1998. Proceedings of the 37th IEEE Conference on*, volume 3, pages 2639–2644. IEEE, 1998.

[Godefroid *et al.*, 1996] Patrice Godefroid, J van Leeuwen, J Hartmanis, G Goos, and Pierre Wolper. Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem. 1996.

[Grosu and Smolka, 2005] Radu Grosu and Scott A Smolka. Monte carlo model checking. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 271–286, 2005.

[Gu *et al.*, 2015] Yitian Gu, Shou-pon Lin, and Nicholas F Maxemchuk. A fail safe broadcast protocol for collaborative intelligent vehicles. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2015 IEEE 16th International Symposium on a*, pages 1–6. IEEE, 2015.

[Guo *et al.*, 2012] Chunzhao Guo, Nianfeng Wan, Seiichi Mita, and Ming Yang. Self-defensive coordinated maneuvering of an intelligent vehicle platoon in mixed traffic. In *Intelligent Transportation Systems (ITSC), 2012 15th International IEEE Conference on*, pages 1726–1733. IEEE, 2012.

[Guvenc *et al.*, 2012] Levent Guvenc, Ismail Meriç Can Uygan, Kerim Kahraman, Raif Karaahmetoglu, Ilker Altay, Mutlu Senturk, Mümin Tolga Emirler, Ahu Ece Hartavi Karci, Bilin Aksun Guvenc, Erdinç Altug, et al. Cooperative adaptive cruise control implementation of team mekar at the grand cooperative driving challenge. *Intelligent Transportation Systems, IEEE Transactions on*, 13(3):1062–1074, 2012.

[Hallé *et al.*, 2004] Simon Hallé, Julien Laumonier, and Brahim Chaib-draa. A decentralized approach to collaborative driving coordination. In *Intelligent Transportation Systems, 2004. Proceedings. The 7th International IEEE Conference on*, pages 453–458. IEEE, 2004.

[Hansson and Jonsson, 1994] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5):512–535, 1994.

[Hardin *et al.*, 1996] Ronald H Hardin, Zvi Har'El, and Robert P Kurshan. Cospan. In *Computer Aided Verification*, pages 423–427. Springer Berlin Heidelberg, 1996.

[Hartenstein and Laberteaux, 2008] Hannes Hartenstein and Kenneth P Laberteaux. A tutorial survey on vehicular ad hoc networks. *Communications Magazine, IEEE*, 46(6):164–171, 2008.

[HAVEit,] HAVEit. HAVEit project.

[Hérault *et al.*, 2004] Thomas Hérault, Richard Lassaigne, Frédéric Magniette, and Sylvain Peyronnet. Approximate probabilistic model checking. In *Verification, Model Checking, and Abstract Interpretation*, pages 73–84. Springer, 2004.

[Hinton *et al.*, 2006] Andrew Hinton, Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: A tool for automatic verification of probabilistic systems. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 441–444, 2006.

[Holzmann, 1997] Gerard J Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[Holzmann, 1998] Gerard J Holzmann. An analysis of bitstate hashing. *Formal methods in system design*, 13(3):289–307, 1998.

[Holzmann, 2007] Gerard J Holzmann. Design and validation of computer protocols. *Computer Protocols*, 2007.

[Horowitz and Varaiya, 2000] Roberto Horowitz and Pravin Varaiya. Control design of an automated highway system. *Proceedings of the IEEE*, 88(7):913–925, 2000.

[Hsu *et al.*, 1991] Ann Hsu, Farokh Eskafi, Sonia Sachs, and Pravin Varaiya. Design of platoon maneuver protocols for ivhs. *California Partners for Advanced Transit and Highways (PATH)*, 1991.

[Huang *et al.*, 1996] Chung-Ming Huang, Jenq-Muh Hsu, and Shiun-Wei Lee. Probabilistic fuzzy timed protocol verification. *Computer Communications*, 19(5):407–425, 1996.

[Ioannou and Chien, 1993] P.A. Ioannou and C.C. Chien. Autonomous intelligent cruise control. *IEEE Transactions on Vehicular Technology*, 42(4):657 –672, November 1993.

[Ioannou *et al.*, 1993] Petros Ioannou, Cheng-Chih Chien, et al. Autonomous intelligent cruise control. *Vehicular Technology, IEEE Transactions on*, 42(4):657–672, 1993.

[Ishida and Gayko, 2004] Shinnosuke Ishida and Jens E Gayko. Development, evaluation and introduction of a lane keeping assistance system. In *Intelligent Vehicles Symposium, 2004 IEEE*, pages 943–944. IEEE, 2004.

[Jo *et al.*, 2012] Kichun Jo, Keounyup Chu, and Myoungho Sunwoo. Interacting multiple model filter-based sensor fusion of gps with in-vehicle sensors for real-time vehicle positioning. *Intelligent Transportation Systems, IEEE Transactions on*, 13(1):329–343, 2012.

[Jones, 1997] Mike B Jones. What really happened on mars, 1997.

[Katz and Peled, 1992] Shmuel Katz and Doron Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6(2):107–120, 1992.

[Kianfar *et al.*, 2012] Roozbeh Kianfar, Bruno Augusto, Alireza Ebadighajari, Usman Hakeem, Josef Nilsson, Ali Raza, Reza S Tabar, NV Irukulapati, Christer Englund, Paolo Falcone, et al. Design and experimental validation of a cooperative driving system in the grand cooperative driving challenge. *Intelligent Transportation Systems, IEEE Transactions on*, 13(3):994–1007, 2012.

[Kim and Maxemchuk, 2012] Bo Hyun J Kim and Nicholas F Maxemchuk. A safe driver assisted merge protocol. In *Systems Conference (SysCon), 2012 IEEE International*, pages 1–4. IEEE, 2012.

[Kodaka *et al.*, 2003] Kenji Kodaka, Makoto Otabe, Yoshihiro Urai, and Hiroyuki Koike. Rear-end collision velocity reduction system. Technical report, SAE Technical Paper, 2003.

[Kwiatkowska *et al.*, 2010] Marta Kwiatkowska, Gethin Norman, David Parker, and Hongyang Qu. Assume-guarantee verification for probabilistic systems. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 23–37, 2010.

[Lam and Katupitiya, 2013] Stanley Lam and Jayantha Katupitiya. Cooperative autonomous platoon maneuvers on highways. In *Advanced Intelligent Mechatronics (AIM), 2013 IEEE/ASME International Conference on*, pages 1152–1157. IEEE, 2013.

[Larsen *et al.*, 1997] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[Li and Nashashibi, 2013] Hao Li and Fawzi Nashashibi. Cooperative multi-vehicle localization using split covariance intersection filter. *Intelligent Transportation Systems Magazine, IEEE*, 5(2):33–44, 2013.

[Li *et al.*, 2014] Qingquan Li, Long Chen, Ming Li, Shih-Lung Shaw, and Andreas Nuchter. A sensor-fusion drivable-region and lane-detection system for autonomous vehicle navigation in challenging road scenarios. *Vehicular Technology, IEEE Transactions on*, 63(2):540–555, 2014.

[Lidström *et al.*, 2012] Kristoffer Lidström, Katrin Sjoberg, Ulf Holmberg, Johan Andersson, Fredrik Bergh, Mattias Bjade, and Spencer Mak. A modular cacc system integration and design. *Intelligent Transportation Systems, IEEE Transactions on*, 13(3):1050–1061, 2012.

[Lin and Maxemchuk, a] Shou-pon Lin and N.F. Maxemchuk. Improved probabilistic verification of systems with a large state space. [submitted to FMCAD 2015].

[Lin and Maxemchuk, b] Shou-pon Lin and N.F. Maxemchuk. Probabilistic model checking of systems with a large state space: A stratified approach (extended abstract). [accepted for poster presentation in FMCAD 2015].

[Lin and Maxemchuk, 2012] Shou-pon Lin and Nicholas F Maxemchuk. An architecture for collaborative driving systems. In *Network Protocols (ICNP), 2012 20th IEEE International Conference on*, pages 1–2. IEEE, 2012.

[Lin and Maxemchuk, 2014] Shou-pon Lin and Nicholas F Maxemchuk. The fail-safe operation of collaborative driving systems. *Journal of Intelligent Transportation Systems*, (ahead-of-print):1–14, 2014.

[Lin and Maxemchuk, 2015] Shou-pon Lin and N.F. Maxemchuk. A case study on using probabilistic verification to find failures in a cooperative driving application. In *Vehicular Technology Conference (VTC Fall), 2015 IEEE*. IEEE, 2015.

[Lin *et al.*, 1989] Fuchun Joseph Lin, Ming T Liu, and Charles J Graff. On the verification of time-dependent protocols using timed reachability analysis. In *System Sciences, 1989. Vol. II: Software Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*, volume 2, pages 285–294. IEEE, 1989.

[Lin *et al.*, 2014] Shou-pon Lin, Yitian Gu, and Nicholas F Maxemchuk. A multiple stack architecture for intelligent vehicles. In *Intelligent Vehicles Symposium Proceedings, 2014 IEEE*, pages 268–273. IEEE, 2014.

[Linn Jr and Uyar, 2013] RJ Linn Jr and MU Uyar. Protocol development success stories: Part i. In *Protocol Specification, Testing and Verification, XII: Proceedings of the IFIP TC6/WG6. 1. Twelfth International Symposium on Protocol Specification, Testing and Verification, Lake Buena Vista, Florida, USA, 22-25 June, 1992*, page 149. Elsevier, 2013.

[Mahmood and Gaderer, 2009] Aneeq Mahmood and Georg Gaderer. Timestamping for ieee 1588 based clock synchronization in wireless lan. In *Precision Clock Synchronization for Measurement, Control and Communication, 2009. ISPCS 2009. International Symposium on*, pages 1–6. IEEE, 2009.

[Mårtensson *et al.*, 2012] Jonas Mårtensson, Assad Alam, Sagar Behere, Muhammad Altamash Ahmed Khan, Joakim Kjellberg, Kuo-Yun Liang, Henrik Pettersson, and Dennis Sundman. The development of a cooperative heavy-duty vehicle for the gcdc 2011: Team scoop. *Intelligent Transportation Systems, IEEE Transactions on*, 13(3):1033–1049, 2012.

[Maurer, 2012] Markus Maurer. Forward collision warning and avoidance. In *Handbook of Intelligent Vehicles*, pages 657–687. Springer, 2012.

[Maxemchuk and Sabnani, 1989] Nicholas F Maxemchuk and Krishan Sabnani. Probabilistic verification of communication protocols. *Distributed Computing*, 3(3):118–129, 1989.

[Maxemchuk and Shur, 2001] Nicholas F Maxemchuk and David H Shur. An internet multicast system for the stock market. *ACM transactions on computer systems*, 19(3):384–412, 2001.

[Maxemchuk *et al.*, 2007] Nicholas F Maxemchuk, Patcharinee Tientrakool, and Theodore L Willke. Reliable neighborcast. *Vehicular Technology, IEEE Transactions on*, 56(6):3278–3288, 2007.

[Maxemchuk *et al.*, 2015] NF Maxemchuk, Shou-pon Lin, and Yitian Gu. Architectures for intelligent vehicles 13. *Vehicular Communications and Networks: Architectures, Protocols, Operation and Deployment*, 2:275, 2015.

[Michaud *et al.*, 2006] François Michaud, Pierre Lepage, Patrick Frenette, Dominic Letourneau, and Nicolas Gaubert. Coordinated maneuvering of automated vehicles in platoons. *Intelligent Transportation Systems, IEEE Transactions on*, 7(4):437–447, 2006.

[Milanés *et al.*, 2014] Vicente Milanés, Steven E Shladover, John Spring, Christopher Nowakowski, Hiroshi Kawazoe, and Mitsutoshi Nakamura. Cooperative adaptive cruise control in real traffic situations. *Intelligent Transportation Systems, IEEE Transactions on*, 15(1):296–305, 2014.

[Misra and Enge, 2006] Pratap Misra and Per Enge. *Global Positioning System: Signals, Measurements and Performance Second Edition.* Lincoln, MA: Ganga-Jamuna Press, 2006.

[Musuvathi *et al.*, 2004] Madanlal Musuvathi, Dawson R Engler, et al. Model checking large network protocol implementations. In *NSDI*, volume 4, pages 12–12. Citeseer, 2004.

[Nagano *et al.*, 1996] Shin'ichi Nagano, Yoshinori Hatakeyama, Yoshiaki Kakuda, and Tohru Kikuno. Timed reachability analysis method for efsm-based communication protocols and its experimental evaluation. In *Network Protocols, 1996. Proceedings., 1996 International Conference on*, pages 92–99. IEEE, 1996.

[NHTSA, 2015] NHTSA. A comprehensive examination of naturalistic lane-changes, February 2015.

[Nieuwenhuijze *et al.*, 2012] Maarten RI Nieuwenhuijze, Thijs van Keulen, Sinan Öncü, Bram Bonsen, and Henk Nijmeijer. Cooperative driving with a heavy-duty truck in mixed traffic: Experimental results. *Intelligent Transportation Systems, IEEE Transactions on*, 13(3):1026–1032, 2012.

[of the Secretary of Defense, 2008] Office of the Secretary of Defense. Global positioning system standard positioning service signal specification, September 2008.

[OIAC, 2015] OIAC. 2013 production statistics, February 2015.

[Ölveczky and Meseguer, 2002] Peter Csaba Ölveczky and José Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285(2):359–405, 2002.

[Papadimitratos *et al.*, 2009] Panos Papadimitratos, A La Fortelle, Knut Evenssen, Roberto Brignolo, and Stefano Cosenza. Vehicular communication systems: Enabling technologies, applications, and future outlook on intelligent transportation. *Communications Magazine, IEEE*, 47(11):84–95, 2009.

[PATH,] PATH. California path.

[Pnueli, 1977] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[Reynisson *et al.*, 2014] Arni Hermann Reynisson, Marjan Sirjani, Luca Aceto, Matteo Cimini, Ali Jafari, Anna Ingolfsdottir, and Steinar Hugi Sigurdarson. Modelling and simulation of asynchronous real-time systems using timed rebeca. *Science of Computer Programming*, 89:41–68, 2014.

[Rudin, 1992] Harry Rudin. Protocol development success stories: Part 1. In *Proceedings of the IFIP TC6/WG6. 1 Twelth International Symposium on Protocol Specification, Testing and Verification XII*, pages 149–160. North-Holland Publishing Co., 1992.

[Sabnani and Dahbura, 1988] Krishan Sabnani and Anton Dahbura. A protocol test generation procedure. *Computer Networks and ISDN systems*, 15(4):285–297, 1988.

[Sankaranarayanan *et al.*, 2013] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. *ACM SIGPLAN Notices*, 48(6):447–458, 2013.

[SARTRE,] SARTRE. SARTRE project.

[Scopigno and Cozzetti, 2009] Riccardo Scopigno and Hector Agustin Cozzetti. Gnss synchronization in vanets. In *New Technologies, Mobility and Security (NTMS), 2009 3rd International Conference on*, pages 1–5. IEEE, 2009.

[Segata *et al.*, 2014] Michele Segata, Bastian Bloessl, Stefan Joerer, Falko Dressler, and Renato Lo Cigno. Supporting platooning maneuvers through ivc: An initial protocol analysis for the join maneuver. In *Wireless On-demand Network Systems and Services (WONS), 2014 11th Annual Conference on*, pages 130–137. IEEE, 2014.

[Sichitiu and Kihl, 2008] Mihail L Sichitiu and Maria Kihl. Inter-vehicle communication systems: a survey. *Communications Surveys & Tutorials, IEEE*, 10(2):88–105, 2008.

[Sinha and Suri, 1999] Purnendu Sinha and Neeraj Suri. On the use of formal techniques for analyzing dependable real-time protocols. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 126–135. IEEE, 1999.

[Sistla and Godefroid, 2004] A Prasad Sistla and Patrice Godefroid. Symmetry and reduced symmetry in model checking. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(4):702–734, 2004.

[Swaroop *et al.*, 1994] DVAHG Swaroop, JK Hedrick, CC Chien, and P Ioannou. A comparision of spacing and headway control laws for automatically controlled vehicles1. *Vehicle System Dynamics*, 23(1):597–625, 1994.

[Tientrakool *et al.*, 2011] Patcharinee Tientrakool, Ya-Chi Ho, and Nicholas F Maxemchuk. Highway capacity benefits from using vehicle-to-vehicle communication and sensors for collision avoidance. In *Vehicular Technology Conference (VTC Fall), 2011 IEEE*, pages 1–5. IEEE, 2011.

[Tsugawa *et al.*, 2000] Sasayuki Tsugawa, Shin Kato, Takeshi Matsui, Hiroshi Naganawa, and H Fujii. An architecture for cooperative driving of automated vehicles. In *Intelligent Transportation Systems, 2000. Proceedings. 2000 IEEE*, pages 422–427. IEEE, 2000.

[Uzcategui and Acosta-Marum, 2009] R Uzcategui and Guillermo Acosta-Marum. Wave: a tutorial. *Communications Magazine, IEEE*, 47(5):126–133, 2009.

[van Nunen *et al.*, 2012] Ellen van Nunen, RJAE Kwakkernaat, Jeroen Ploeg, and Bart D Netten. Cooperative competition for future mobility. *Intelligent Transportation Systems, IEEE Transactions on*, 13(3):1018–1025, 2012.

[Varaiya, 1993] Pravin Varaiya. Smart cars on smart roads: problems of control. *Automatic Control, IEEE Transactions on*, 38(2):195–207, 1993.

[West, 1989] Colin H West. *Protocol validation in complex systems*, volume 19. ACM, 1989.

[Wu *et al.*, 1997] Sue-Hwey Wu, Scott A Smolka, and Eugene W Stark. Composition and behaviors of probabilistic i/o automata. *Theoretical Computer Science*, 176(1):1–38, 1997.

[Yang *et al.*, 2006] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems (TOCS)*, 24(4):393–423, 2006.

[Yannakakis and Lee, 1993] Mihalis Yannakakis and David Lee. An efficient algorithm for minimizing real-time transition systems. In *Computer Aided Verification*, pages 210–224. Springer, 1993.

[Yi, 1991] Wang Yi. Ccs+ time= an interleaving model for real time systems. In *Automata, Languages and Programming*, pages 217–228. Springer, 1991.

[Yovine, 1997] Sergio Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):123–133, 1997.

[Yuan *et al.*, 2015] Ting Yuan, Tobias Roth, Qi Chen, Jakob Breu, Miro Bogdanovic, and Christian A Weiss. Track-to-track association for object matching in an inter-vehicle communication system. *SPIE Optical Engineering+ Applications*, pages 959609–959609, 2015.