FORMAL MODELING CAN IMPROVE SMART TRANSPORTATION ALGORITHM DEVELOPMENT

by

WATHUGALA GAMAGE DULAN MANUJINDA WATHUGALA

A THESIS

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

June 2017

THESIS APPROVAL PAGE

Student: Wathugala Gamage Dulan Manujinda Wathugala

Title: Formal Modeling Can Improve Smart Transportation Algorithm Development

This thesis has been accepted and approved in partial fulfillment of the requirements for the Master of Science degree in the Department of Computer and Information Science by:

Stephen Fickas Chair

and

Scott L. Pratt Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded June 2017

THESIS ABSTRACT

Wathugala Gamage Dulan Manujinda Wathugala

Master of Science

Department of Computer and Information Science

June 2017

Title: Formal Modeling Can Improve Smart Transportation Algorithm Development

Ensuring algorithms work accurately is crucial, especially when they drive safety critical systems like self-driving cars.

We formally model a published distributed algorithm for autonomous vehicles to collaborate and pass thorough an intersection. Models are built and validated using the "Labelled Transition System Analyser" (LTSA). Our models reveal situations leading to deadlocks and crashes in the algorithm.

We demonstrate two approaches to gain insight about a large and complex system without modeling the entire system: *Modeling a sub system* - If the sub system has issues, the super system too. *Modeling a fast-forwarded state* - Reveals problems that can arise later in a process.

Some productivity tools developed for distributed system development are also presented. *Manulator*, our distributed system simulator, enables quick prototyping and debugging on a single workstation. *LTSA-O*, extension to LTSA, listens to messages exchanged in an execution of a distributed system and validates it against a model.

CURRICULUM VITAE

NAME OF AUTHOR: Wathugala Gamage Dulan Manujinda Wathugala

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene
University of Colombo, Colombo

DEGREES AWARDED:

Master of Science, Computer and Information Science, 2017, University of Oregon
Bachelor of Science, Computer Science, 2002, University of Colombo

AREAS OF SPECIAL INTEREST:

Algorithm Modeling & Simulation
Distributed Systems
Autonomous Vehicles
Machine Learning

PROFESSIONAL EXPERIENCE:

Graduate Teaching Fellow, University of Oregon, 2015 – 2017.

Graduate Research Fellow, University of Oregon, 2016.

Lecturer, University of Colombo, 2002 – 2013.

GRANTS, AWARDS AND HONORS:

Felice Proctor Award (Non-theater major who has made a significant contribution to the theater production program in 2005), Virginia Polytechnic Institute and State University, 2005.

Fulbright Scholarship, US-SL Fulbright Commission & United States Government, 2004.

Mohan Munasinghe Award (Best Computer Science student), University of Colombo, 2002.

CINTEC Award (Best Computer Science senior thesis), University of Colombo, 2002.

Justin Samarasekara Award (Most outstanding science student), University of Colombo, 2002.

Joseph Nalliah Arumugam Memorial Award (Student having the highest average marks), University of Colombo, 2002.

Physical Science Award (Best undergraduate research project in Physics, Computer Science, Mathematics or Statistics), Sri Lanka Association for the Advancement of Science, 2002.

The Scholarship (Best results, B.Sc. 1^{st} year examination), University of Colombo, 1998.

Prof. J.E. Jayasuriya Prize for Mathematics, University of Colombo, 1998.

PUBLICATIONS:

Weliwitigoda, P. & Weerasinghe, A. R. & Wathugala, W. G. D. M. & Dharmaratne, A. T. (2004). Music Score Recognition with Waves, Purnima Weliwitigoda, Ruvan Weerasinghe. *International Information Technology Conference.*

Yogendirakumar, K. & Weerasinghe, A. R. & Wathugala, W. G. D. M. & Dharmaratne, A. T. (2004). Music Score Recognition with Waves, Purnima Weliwitigoda, Ruvan Weerasinghe. *International Information Technology Conference.*

Wathugala, W. G. D. M. & Kodikara, N. D. (2002). A Sinhala Finger Spelling Interpretation System Using Nearest Neighbor Classification. *International Information Technology Conference.*

ACKNOWLEDGEMENTS

Professional

We thank Weigang Wu, Jiebin Zhang, Aoxue Luo and Jiannong Cao, the authors of the paper we build our work on. We picked their work not because it is easy to pick on but because they have done excellent work and we thought that their work is worthy of carrying forward.

We extend our special thanks to Weigang Wu, for corresponding with us via email and helping us acquire a copy of the supplement that accompanies the main paper, which we were unable to locate on-line.

We thank Jeff Magee, Jeff Kramer, Robert Chatley, Sebastian Uchitel and Howard Foster, the developers of the "Labelled Transition System Analyser" (LTSA) tool for sharing their code-base with us. Without their generosity, our extension to LTSA, LTSA-O, would have been impossible.

Jeff Magee is put on a separate spotlight for a big thank you. Despite being a very busy person, he promptly responded in details for our clarifications regarding LTSA and our models.

I thank Stephen Fickas, my thesis advisor, for everything. Without the numerous discussions I had with him, ideas he contributed, time he spent on reading my thesis and providing constructive feedback, and caring he extended toward me as his student, this work would have been infeasible.

I thank C. W. W. Kannangara (13 October 1884 - 23 September 1969), the first Minister of Education and the *Father of Free Education* in Sri Lanka for establishing free education. I further thank all the governments and the general public of Sri Lanka for keeping free education alive for the generations to come

and funding my education till I graduated from college. Without this funding, the chances of I coming this far is mere.

Personal

I thank Deepa Wathugala, my aunt. During my first term, when I did not get a Graduate Employee position and considering going home without even beginning this journey, she lent me money to pay for my tuition.

My mother, Indrani Wathugala, gave birth to me and raised me to be a good person. She gave me all the psychological support from the other side of the globe. Thank you for everything.

I greatly appreciate my wife, Sharmila Iroshmi Thenuwara. She took all the responsibilities of managing our home on to her shoulder and allowed me to focus fully on my work. She even helped me in my work in various regards such as proof reading and finding articles on the web. She is the best wife a man could ever have. I love you more each day.

My father, Wimal Wathugla, is the "Higgs Boson" that glue all the dots together. He was the role model of my life and he taught me to be a good man. In 1957, when he was just 21 years old, he won the second price of a lottery worth Rs. 14,660. Instead of thinking of building a luxurious life for himself, he spent the money for the betterment of his family and to provide a better education to all his siblings including his youngest brother, Wije Wathugala. While my uncle is pursuing his Ph.D., he met my aunt Deepa Wathugala. This connects the dots and that made me survive my first term at the University of Oregon. Dear father, you are getting the returns of your investment in multiple orders and thank you for having a great vision.

Dots in the past are starting to connected well. I am glad about that.

I dedicate my thesis to the future generations to come.

I hope my work will aid in some way to make a better universe for them.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

LIST OF ALGORITHMS

Algorithm Page

LIST OF LISTINGS

Listing Page

Listing Page

CHAPTER I

INTRODUCTION

1.1 Autonomous Vehicles

When we prefix any one of the adjectives "*Autonomous*", "*Self-Driving*", "*Driver Less*" in front of the noun "*Vehicle*" (or Car), it means a vehicle that can navigate on its own without human assistance. The modern day dream of autonomous vehicles goes as far as the General Motor's Futurama exhibit at the 1939 World's Fair (Vanderbilt, 2012; Weber, 2014). In 1977, Japan's attempt of building an autonomous vehicle at Tsukuba Mechanical Engineering Laboratory has been recorded as the first breakthrough followed by VaMoRs project in 1987 and VaMP project in 1994 (Vanderbilt, 2012). Several successive U.S. Defense Advanced Research Projects Administration (DARPA) challenges held in 2004, 2005 and 2007 amplified the enthusiasm toward autonomous vehicle research and helped advance the field. Since 2010, Google has become the pioneer in autonomous vehicles with their Google Car project that aims at bringing an autonomous car at the consumer level.

1.2 Vehicle-to-Vehicle Communication

Vehicle-to-Vehicle (V2V) Communication (sometimes referred by Car-to-Car) is a new emerging technology that enables vehicles on road communicate with each other by sending and receiving information via a wireless channel. National Highway Traffic Safety Administration (NHTSA) of the U. S. Department of Transportation defines V2V communication as:

> A system designed to transmit basic safety information between vehicles to facilitate warnings to drivers concerning impending crashes. (Harding et al., 2014, p. xiii)

Along the V2V communication technology came the Vehicular Ad-hoc Network (VANET) (Al-Sultan, Al-Doori, Al-Bayatti, & Zedan, 2014). A VANET is a special kind of Mobile Ad-hoc Network (MANET), where nodes are vehicles and node movement is governed by the available road network and the vehicle movement dynamics.

Prior to the creation of V2V communication and VANETS, autonomous vehicles used various sensors like cameras, proximity sensors and radar to sense the surroundings of the vehicle, update a model of its environment, make individual decisions and act on them (sense-process-act loop). Each vehicle worked on its own and there was no collaboration among vehicles.

V2V communication opened the doors for vehicles to share information about each vehicle's state and intentions enabling a new range of distributed decision making tasks. Several vehicles arriving at a four-way road intersection communicating and deciding on their own, without any human assistance, on a schedule for them to pass through the intersection without causing an accident is one such application (Wu, Zhang, Luo, & Cao, 2015a).

1.3 Self-Driving Cars, Are they Safe?

A safety-critical system is defined as:

> A system whose failure could result in loss of life, significant property damage, or damage to the environment. (Knight, 2002)

With the advancement of autonomous vehicle research and technologies such as vehicle-to-vehicle communication and Global Positioning System (GPS), the day where self-driving cars start ubiquitously roaming our neighborhoods is not very far-fetched. When such vehicles transport humans and start sharing roadways with other humans, a malfunction in the underlying software systems that make driving

decisions could lead to accidents that cost human life. This fits the definition of a safety-critical system and hence we should employ methods and practices of designing and building such systems when it comes to designing and implementing software systems for self-driving cars.

We focused on a published distributed algorithm for self-driving cars to decide on an order to pass through a road intersection (Wu et al., 2015a). The authors of the algorithm claim it is deadlock free (live) and it prevents cars from crashing to each other (safe).

The paper passed the journal's review board, which implies that the board also found the algorithm to pass safety and liveness conditions. However, the paper uses a non-formal approach to demonstrate its claims. Our experience in working with complex distributed systems is that it is difficult to see all the nuances of an algorithm without using formal techniques to prove safety and liveness properties. This thesis sets out to re-examine the published algorithm using formal modeling tools. The results come from several areas:

1. We prove under what conditions the published algorithm is live and safe, and under what conditions it is not. This extends the results of the original paper.

2. We demonstrate how formal-modeling can be applied in the V2V area. Our review of the V2V literature points to a lack of formal modeling. We believe formal modeling can help with developing V2V algorithms that people can rely on, and hope to show V2V researchers how to use formal modeling techniques.

1.4 Labelled Transition System Analyser (LTSA)

The modeling tool of our choice is "Labelled Transition System Analyser" (LTSA) (Magee & Kramer, 2006; Magee, Kramer, Chatley, Uchitel, & Foster, 2013). LTSA is a verification and validation tool for concurrent systems. It enables modeling each concurrent process as a "Finite State Machine" using a program like notation called "Finite State Processes" (FSP). FSP builds upon the ideas from "Communicating Sequential Processes" (CSP) (Hoare, 1978, 2015) and "Calculus of Communicating Processes" (CCS) (Milner, 1982). Individual models of processes can then be composed to build a finite state machine for the complete concurrent system.

Along with the process descriptions, we can describe desirable properties the system must possess and undesirable properties that must be avoid. The composition can then be exhaustively analyzed against the desirable and undesirable properties to check whether the system reaches any undesirable state or it never reaches a desirable state.

We do not provide a survey of LTSA. Magee and Kramer (2006) is the official and only manual for LTSA, and an interested reader can get an excellent understanding of LTSA by referring it. However, if the reader is unable to acquire a copy of the book, the set of slides at the website accompanying the book (Magee & Kramer, 2015) and language specifications of Finite State Processes found online (Magee & Kramer, 2012; Magee, Kramer, Chatley, Uchitel, & Foster, 2007) are excellent resources to gain knowledge sufficient to make sense of the work described in this thesis. If anybody is interested in playing with the LTSA tool, it is freely available at Magee et al. (2013).

Our investigations revealed that the algorithm could deadlock under some circumstances and for some other circumstances it could end up causing a crash. Chapters IV, V and VI provide more detailed descriptions of how we built our model.

CHAPTER II

THE DISTRIBUTED ALGORITHM THAT WE MODEL

The paper, Wu et al. (2015a) presents two distributed algorithms to achieve mutual exclusion of vehicles with conflicting paths simultaneously passing through a road intersection. The first algorithm relies on a centralized control node while the second algorithm is purely distributed. In this thesis, we only focus on the second algorithm, which is more challenging and interesting.

A summary of the distributed mutual exclusion algorithm that we analyze is presented here. All the material in this chapter are either direct quotations from the original paper (Wu et al., 2015a), paraphrases of the ideas found in that paper or our commentaries regarding what is presented there.

2.1 The Intersection

An intersection where vehicles approach from four directions is considered (*Figure* 1). For each direction there are two incoming lanes. Lanes are numbered from 0 to 7 and denoted by l_0 to l_7. Odd numbered lanes are for turning left. As the paper mentions, even numbered lanes are for going forward. However, in a typical intersection, vehicles entering from lanes that are labeled with even numbers in the diagram are also allowed to make right turns. The paper mentions "*Obviously, the path of a vehicle is determined by the lane it is at*" (Wu et al., 2015a, p. 66). This allows us infer that disallowing vehicles from making right turns is an implicit assumption the authors of the paper are making for their algorithm to work. We believe that this assumption is too strong and it makes the intersection being modeled unrealistic.

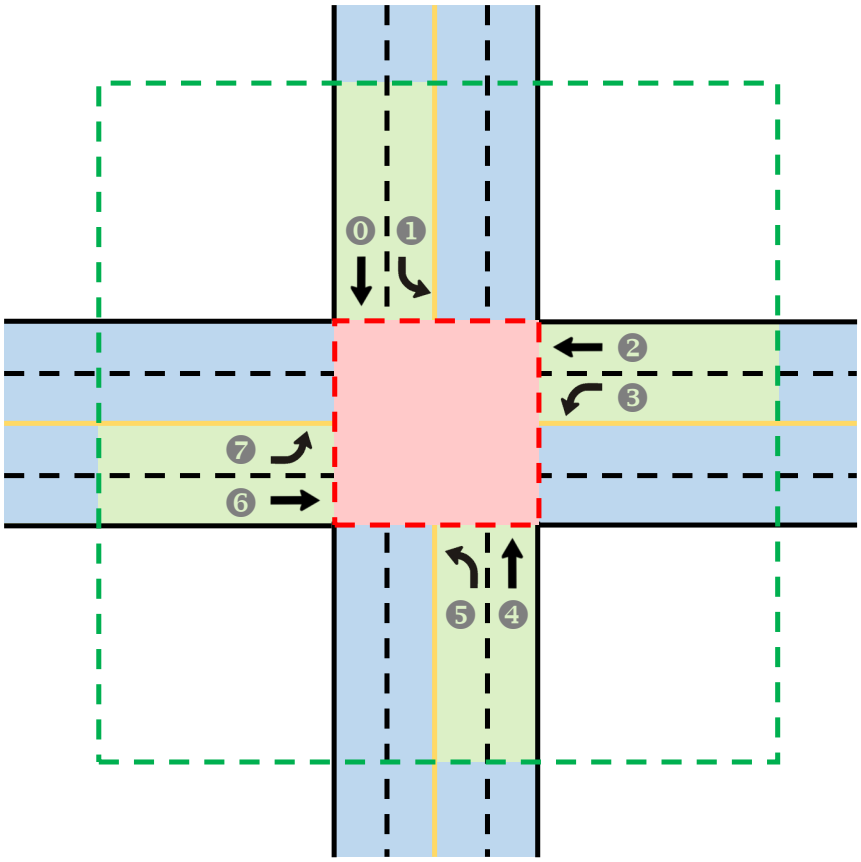

Figure 1. The intersection. The smaller red square in the middle of the intersection is the critical section, which is called the core area. The larger green square marks the queue area where vehicles wait to pass through the intersection. Vehicles within the larger green square collaborate with each other to pass through the intersection.

2.1.1 Relationships Between Lanes. The authors of the paper have elegantly summarized relationships between lanes in to a conflict graph, which ends up being a cube (*Figure* 2).

Conflicting ($\propto$): Two lanes are conflicting if their paths intersect within the core of the intersection (e.g. $l_0 \propto l_2$). Two vehicles coming from two conflicting lanes must not enter the intersection at the same time.

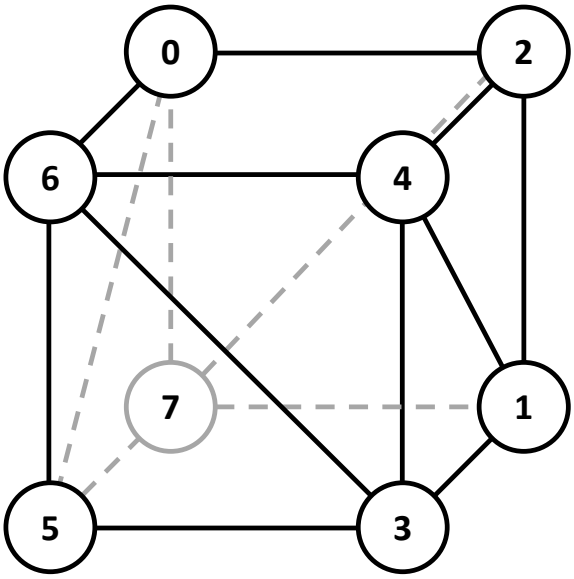

Figure 2. The conflict graph of lanes. Vertices represent lanes. Two lanes are *conflicting* ($\propto$) if and only if there is an edge between them. Vertices that are not directly connected are *concurrent* ($\approx$). Concurrent lanes that belong to the same face of the conflict graph cube are *strong concurrent* ($\cong$).

Concurrent ($\approx$): Two lanes are concurrent if their paths do not intersect within the core of the intersection (e.g. $l_0 \approx l_3$ and $l_0 \approx l_4$). Vehicles coming from two concurrent lanes can pass through the intersection simultaneously.

Strong concurrent ($\cong$): These are the pairs of concurrent lanes that belong to the same face of the conflict graph cube (*Figure* 2). In other words, two concurrent lanes that do not exit the intersection in the same direction are strong concurrent (e.g. $l_0 \cong l_4$).

2.2 Assumptions

This is a list of assumptions made by the authors of the algorithm we analyze (Wu et al., 2015a).

1. Assumptions about the vehicle

   (a) Each vehicle has a unique identification number.

   (b) Vehicles are capable of V2V communication.

   (c) As vehicles are approaching the intersection, they are aware of the respective lane numbers they are in.

   (d) Vehicles can detect the boundary of the queuing area and the core area (*Figure* 1). Further, they are aware when they cross those boundaries.

   (e) Vehicles are armed with sensors to self-navigate without colliding with other vehicles or infrastructure.

2. Assumptions about the communication channel

   (a) The transmission range of V2V communication devices is large enough so that any two vehicles within the "queue area" (*Figure* 1) can directly communicate with each other enabling all the vehicles in the "queue area" to form a single hop Vehicular Ad-hoc Network (VANET).

   (b) Wireless channel is first-in-first-out.

   (c) Wireless channel is reliable and it guarantees message delivery.

3. (Inferred) Assumptions about the intersection

   (a) Vehicles are not allowed to make right turns.

2.3 Vehicle Mutual Exclusion for Intersections (VMEI) Problem

The VMEI problem differs from a classical mutual exclusion problem in Computer Science due to two qualities (Table 1). Considering these differences the requirement of the VMEI problem is defined as "*Vehicles can pass the intersection*

Table 1. Classical Mutual Exclusion vs. Vehicle Mutual Exclusion for Intersections (VMEI) - A comparison

Classical ME	**VMEI**
A fixed group of processes compete to enter the critical section.	The group of processes (vehicles) that compete to enter the critical section (the intersection) is dynamic. After passing through the intersection, vehicles drive away (and might even never come back).
Only one process can be in the critical section at a time.	Vehicles entering from concurrent lanes can simultaneously pass through the intersection.
	Vehicles entering the intersection from the same lane can pass through the intersection as a group.

simultaneously if and only if they are in concurrent lanes or the same lane." (Wu et al., 2015a, p. 68)

2.3.1 Correctness Properties of VEMI Problem. For a solution to the VEMI problem to be correct, it should demonstrate three properties:

Safety (mutual exclusion): At any moment, if there is more than one vehicle in the core area, they must be concurrent with each other.

Liveness (deadlock free): If no vehicles are in the core area, some waiting vehicle must be able to enter the core area in a finite time.

Fairness (starvation free): Each vehicle must be able to pass the intersection after a finite number of vehicles do so. (Wu et al., 2015a, p. 68)

2.3.2 Priority Assignment. The paper does not explicitly elaborate a scheme for assigning priorities to vehicles competing to pass through the intersection. However, at several places it hints that arrival time is used to assign a priority to determine the passing order of vehicles:

> The vehicles compete by exchanging messages among themselves and the *order of passing is* ***determined*** *according to arrival time.* (Wu et al., 2015a, p. 66)

> The basic idea of our design is as follows. *Vehicles have different priorities to pass, which is* ***generally determined*** *by the arrival time.* A vehicle broadcasts request message and the receivers with higher priority will prevent the sender via response message. If no receivers prevent the sender, it can pass the intersection. (Wu et al., 2015a, p. 69)

> *The priority is* ***generally, but not always****, determined by the arrival time of vehicles.* (Wu et al., 2015a, p. 69)

> For example, when a vehicle u in l_0 is passing, vehicle v in l_1 may be blocked by w in l_7 if w is between u and v according to ***the priority (arrival time)***. (Wu et al., 2015a, p. 70)

Although in one occasion the paper mentions that arrival time is not always used to determine the priority of a vehicle to pass through the intersection, it never mentions what other attributes are considered. Further, when it comes to the pseudo-code of the algorithm (Algorithm 2), the concept of priority seems to be completely left out.

2.4 Distributed Algorithm for VMEI

2.4.1 High-Level View of the Algorithm. Algorithm 1 provides our high-level interpretation of the distributed algorithm without cluttering the idea with implementation oriented data-structures internal to each vehicle and the details of various types of messages passed among vehicles. Further, we omit various optimizations that make the algorithm more efficient.

2.4.2 Algorithm in Detail.

2.4.2.1 Vehicle Labeling Convention. For our discussions, we let V_{nl} be the n^{th} vehicle that approached the intersection and l is the lane it approached from. Thus, n signifies the order of arrival of vehicles. Since two vehicles in the same lane cannot arrive at the intersection at the same time, this labeling convention provides a total ordering of vehicles at a particular snapshot of an intersection.

2.4.2.2 States. The different states a vehicle goes through when passing through an intersection are summarized using a state transition diagram (*Figure.* 3).

2.4.2.3 Variables and Data-Structures. The pseudo-code of the algorithm (Algorithm 2) uses the following notation for different variables and data-structures each vehicle maintains during the execution of the algorithm:

st_i: Current state of vehicle i.

lid_i: The lane number of vehicle i.

$CntPmp$: The number of vehicles with priority lower than that of vehicle i that vehicle i did not object of passing through the intersection before vehicle

Algorithm 1 The gist of the algorithm

On entering the queuing area:

1: Ask "Is there anybody who objects to me passing through the intersection?"
2: ***Wait*** for a fixed duration for responses.

3: **if** nobody objects **then**
4: Pass through the intersection.

5: **if** I objected to others passing through the intersection **then**
6: Let them know that I passed through (and I don't object anymore).
7: **end if**

8: **else** ▷ others do object
9: ***Wait*** for everybody who objected to me passing through the intersection.
10: Now I can pass through the intersection and do so.

11: **if** I objected to others passing through the intersection **then**
12: Let them know that I passed through (and I don't object anymore).
13: **end if**

14: **end if**

On receiving a request from another vehicle to pass through the intersection:

15: **if** (I came ***earlier than*** the vehicle asking for permission) **and** (it is in my lane **or** its path conflicts with mine) **then**
16: Tell that vehicle that I object to it passing through the intersection.
17: **end if**

i. The authors call this a preemption and this is done to increase the concurrency and efficiency of the algorithm.

TH: Maximum number of preemptions given by a vehicle. This is a threshold to prevent a vehicle ending up preempting a large number of lower priority vehicles and having to wait for a longer time (starvation).

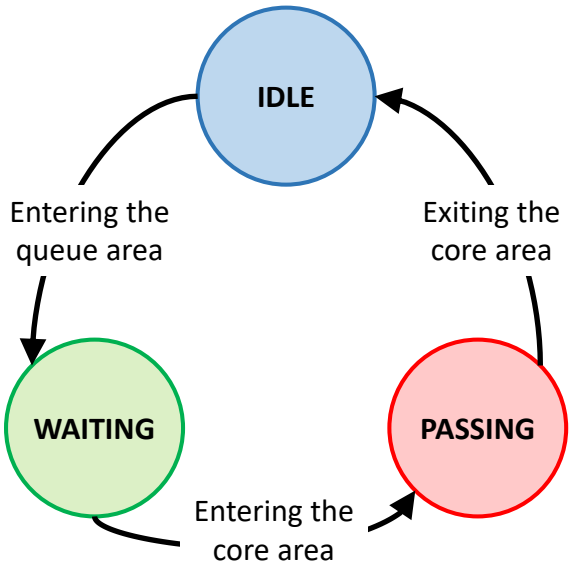

Figure 3. The state transition diagram of the distributed algorithm. *IDLE*: Vehicle is outside the queue area or has exited the intersection (Blue colored region of the intersection diagram *Figure.* 1). *WAITING*: Vehicle is waiting in the queue area to enter the core area (Green colored region of the intersection diagram). *PASSING*: Vehicle is moving through the core area (Red colored region of the intersection diagram).

flt: Follow list. A list of vehicles queued behind the leading vehicle in a lane that are allowed to pass through the intersection following the leading vehicle creating a convoy.

NP: Maximum number of vehicles that are allowed to pass through in a convoy behind the leading vehicle. This is to prevent starvation for other vehicles waiting to pass through.

HL_i: High List. Vehicle i is waiting until all the vehicles in this list pass through the intersection.

LL_i: Low List. Vehicles in this list are waiting till vehicle i passes through the intersection.

2.4.2.4 Messages Passed. Following is a list of messages exchanged among vehicles in the process of coordinating passage through the intersection:

REQUEST(i, lid): Vehicle with id i broadcasts this message once as it enters the queuing area informing the lane lid from which it is approaching the intersection to check whether anybody objects to vehicle i passing through the intersection.

REJECT(i, j): Vehicle i sends this message to make vehicle j wait till vehicle i passes through the intersection.

PERMIT(i): Vehicle i sends this message once after passing through and exiting the core area of the intersection to notify any vehicle that is waiting till vehicle i passes through the intersection.

FOLLOW(i, flt): Vehicle i, which is waiting in the front of a lane, sends this message once when it gets its chance to pass through the intersection. flt is a list of vehicles that are queued behind i and are allowed to pass through the intersection following i forming a convoy.

2.4.2.5 Optimizations. Besides the issues we have uncovered with our modeling and validation process, Algorithm 1 satisfies the correctness properties (Section 2.3.1) required. However, implementing this algorithm as it is would be less efficient and it would prevent from harnessing the maximum concurrency the problem offers. The authors have presented two types of optimizations:

1. Increasing concurrency: Authors have introduced two techniques to increase the amount of concurrency provided by the algorithm. In the bare-bones

algorithm, the order of arrival at the intersection is equal to the order of passing through the intersection. Both the optimizations meaningfully alter the order of passing through the intersection taking the concurrent nature of pairs of lanes vehicles come in.

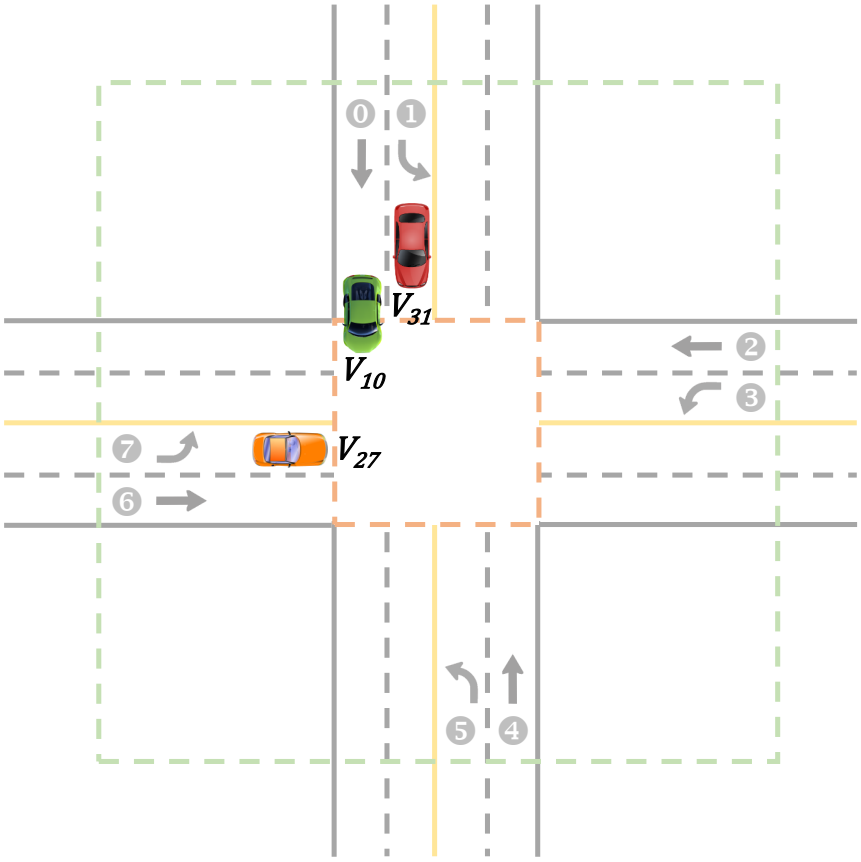

Figure 4. A situation for improving concurrency. Vehicles arrive at the intersection in the order V_{10}, V_{27}, V_{31}. Under normal operation they will pass through the intersection in the same order. However, since $V_{31} \cong V_{10}, V_{27}$ preempting its opportunity to pass through the intersection before V_{31} enables V_{31} to pass along V_{10} through the intersection. This improves the concurrency and hence the throughput of the algorithm.

1. Consider the situation depicted in *Figure.* 4. Since V_{10} (green car) arrives first, it is passing through the intersection and V_{27} (orange car) has to wait till V_{10} exits the intersection. Now, according to the basic algorithm, if we let

vehicles pass through the intersection strictly based on the order of arrival, when V_{31} (red car) arrives, it has to wait till V_{27} passes through. However, note that V_{31} is strong concurrent with V_{10} ($V_{31} \cong V_{10}$) and V_{31} can pass along V_{10} adding very little additional delay for V_{27} and improving the concurrency and the throughput of the intersection a lot. Authors have coined the term "***preemption***" to this type of giving way: A vehicle V giving way to a vehicle U that arrived at the intersection after V. Thus, a preemption is like we swap the order of arrival of V and U.

Implementing this optimization naively causes two negative repercussions. First, consider the sequence of vehicles $V_{10}, V_{27}, V_{31}, V_{(2i)0}, V_{(2i+1)1}, \quad i = 2, 3, 4, \ldots$, where each successive vehicle arrives at the intersection a very short time after its predecessor. Then, with a naive implementation of this optimization, V_{27} would end up alternatively giving way to each vehicle that comes after it making V_{27} wait for a longer period causing it to starve. To mitigate this effect, the authors have introduced a threshold TH, which is the maximum number of preemptions a vehicle is allowed to make. Lines 5 to 11 of Algorithm 2 implement this optimization.

The second repercussion is that this could lead to a deadlock situation. Consider the set of vehicles $V_{10}, V_{26}, V_{33}, V_{44}$ arrived in that order (*Figure* 5). Since V_{10} (green car) arrives before V_{26} (orange car), V_{26} waits till V_{10} passes through the intersection. According to the algorithm, when V_{33} (blue car) arrives, it waits till V_{26} (orange car) passes through the intersection adding the arc $V_{33} \rightarrow V_{26}$. Note that, at this point, if V_{26} preempts, V_{33} could pass through the intersection along with V_{10}. However, in this case, since V_{10} and V_{33} are not strong concurrent ($\ncong$), the algorithm is designed to prevent V_{26}

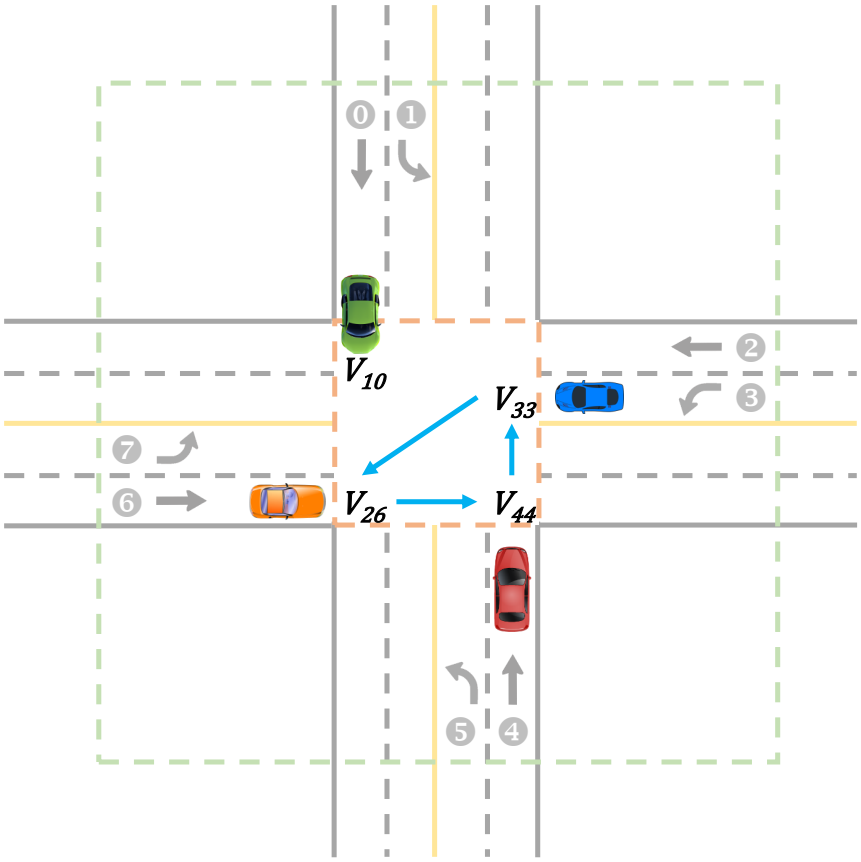

Figure 5. A wait for graph of a deadlock situation that can arise due to preemption. Suppose vehicles arrive in the order $V_{10}, V_{26}, V_{33}, V_{44}$. Since V_{10} (green car) arrives before V_{26} (orange car), V_{26} waits till V_{10} passes through the intersection. According to the algorithm, when V_{33} (blue car) arrives, it waits till V_{26} (orange car) passes through the intersection adding the arc $V_{33} \rightarrow V_{26}$. Note that, at this point, if V_{26} preempts, V_{33} could pass through the intersection along with V_{10}. However, in this case, since V_{10} and V_{33} are not strong concurrent ($\ncong$), the algorithm is designed to prevent V_{26} from preempting. When V_{44} (red car) arrives, it waits till V_{33} passes through, introducing the arc $V_{44} \rightarrow V_{33}$. With preemption in place, V_{26} gives way to V_{44} despite that V_{44} comes after V_{26}. So, in essence, V_{26} now waits till V_{44} passes through, adding the arc $V_{26} \rightarrow V_{44}$. This ends up creating a wait cycle $V_{33} \rightarrow V_{26} \rightarrow V_{44} \rightarrow V_{33}$.

from preempting. When V_{44} (red car) arrives, it waits till V_{33} passes through, introducing the arc $V_{44} \rightarrow V_{33}$. With preemption in place, V_{26} gives way to V_{44} despite that V_{44} comes after V_{26}. So, in essence, V_{26} now waits till V_{44}

passes through, adding the arc $V_{26} \rightarrow V_{44}$. This ends up creating a wait cycle $V_{33} \rightarrow V_{26} \rightarrow V_{44} \rightarrow V_{33}$.

This situation is avoided by making a vehicle V listen to objections destined to other vehicles. If a vehicle U, which V objected from passing through the intersection, objects to another vehicle P, which V preempted and allowed passing through (this creates a wait cycle: $P \rightarrow U \rightarrow V \rightarrow P$), then V revokes the preemption it extended toward P and objects to P from passing through (this breaks the wait cycle: $P \rightarrow U \rightarrow V$ and $P \rightarrow V$).

In addition to this, vehicle V performs the same action of revoking the preemption if U is strong concurrent with V ($U \cong V$). In this case, there is no threat of deadlock. But, if V preempts for P, since P has to wait till U passes, the order of passing through the intersection would be U, P, V. However, since $U \cong V$, instead of preempting and waiting till P passes, V can pass along with U improving the concurrency. This enhancement is implemented by lines 17 to 20 of Algorithm 2.

2. When the leading vehicle LV of a lane gets the chance to pass through the intersection, it looks for all the vehicles piled up behind it in the same lane, which it objected from passing through the intersection. Then it notifies those vehicles and all the other vehicles at the intersection about those vehicles. With this, the algorithm is enhanced in such a way those vehicles follow LV through the intersection forming a convoy. This makes the algorithm mimic the behavior of giving a green traffic light to a lane rather than just a single vehicle.

However, if the number of vehicles behind LV is very large, it would make all the vehicles in other conflicting lanes wait for a longer period, which makes the algorithm unfair. This issue is controlled by a threshold NP, which is the maximum number of vehicles that are allowed to be in a convoy. This mimics the duration of a green traffic light given to a lane. Lines 25, 26 and 29 to 31 of Algorithm 2 incorporate this optimization to the algorithm. This leads to the second type of optimization - reducing the number of messages exchanged.

2. Reducing the number of messages passed: This optimization is made possible due to the second optimization for improving concurrency. Under the basic algorithm, after each vehicle passes through the intersection, it should notify all the vehicles it objected from passing through the intersection. However, with the second concurrency optimization, when a group of vehicles in the same lane passes through the intersection forming a convoy they act as one long vehicle. So, any other vehicle that is in a lane that conflicts with the lane of the convoy has to wait till the last vehicle of the convoy clears the intersection. Thus, vehicles in the front and the middle of the convoy notifying that they are through only adds additional communication overhead to the vehicles. Therefore, the enhanced algorithm makes only the last vehicle in a convoy responsible for letting everybody else know that all the vehicles have cleared the intersection. Lines 32 to 39 of Algorithm 2 deal with this optimization.

However, our analysis of the algorithm uncovers one pitfall of this optimization and we discuss a way to overcome it (Chapter VI).

We further have a hunch that the preemption logic of the algorithm could still lead to a deadlock situation under specific message ordering. In the situation

portrayed in *Figure* 5, consider the message and action ordering (we use the colors of the cars to identify cars):

1) REQUEST(Blue)

2) Orange Receive REQUEST(Blue) → REJECT(Orange, Blue)

3) REQUEST(Red)

4) Blue Receive REQUEST(Red) → REJECT(Blue, Red)

5) Orange Receive REJECT(Blue, Red) → Orange ignore it

6) Orange Receive REQUEST(Red) → Orange preempt

— Deadlock —

The prevention of the deadlock situation that could arise due to preemptions relies on the *REQUEST(Red)* message broadcast by the Red car being received by the Orange car prior to Orange car receiving the *REJECT(Blue, Red)* message broadcast by the Blue car. Then only Orange car will have Red car on its high list for it to execute the logic for revoking the preemption upon hearing the *REJECT(Blue, Red)* message. When these two messages are received inverted by the Orange car, as in the message sequence above, when it receives the *REJECT(Blue, Red)*, it has not yet granted a preemption to revoke (lines 17 - 19 of Algorithm 2). Therefore, it just ignores that message (line 5). Later when it receives the *REQUEST(Red)* message, it carries out the usual preemption logic (lines 5 - 7 of Algorithm 2) leading to a deadlock (line 6).

During the lifespan of this thesis, we did not get to the point of modeling this situation and validating it formally.

2.4.2.6 Algorithm Pseudo-Code. With a clear idea of the gist of the algorithm (Algorithm 1), it would be easier for the reader to follow the details of the actual algorithm presented in the original paper (Wu et al., 2015a, p. 69-70)

(Algorithm 2). Please note that we have reproduced the algorithm exactly as it is presented in the original paper.

Algorithm 2 The distributed VMEI algorithm

CoBegin //for a vehicle i

On entering the monitoring area:

1: st_i = WAITING
2: broadcast REQUEST(i, lid_i)
3: **wait** for REJECT from others

On receiving REQUEST(j, lid_j) from j:

4: **if** ((st_i = WAITING | PASSING) **and** ($lid_i = lid_j \ \vee \ lid_i \propto lid_j$)) **then**
5: **if** (($\exists k, k \in HL_i \ \wedge \ j \cong k$) **and** $CntPmt < TH$) **then**
6: add j to HL_i
7: $CmtPmt++$
8: **else**
9: add j to LL_i
10: broadcast REJECT(i, j)
11: **end if**
12: **end if**

On receiving REJECT(j, k) from j:

13: **if** (st_i = WAITING) **then**
14: **if** ($i = k$) **then**
15: add j to HL_i
16: **end if**
17: **if** ($i \neq k$ **and** ($k \in HL_i$ with preemption) **and** (($j \propto i \ \wedge \ j \notin HL_i$) $\vee \ j \cong i$)) **then**
18: delete k from HL_i
19: broadcast REJECT(i, k)
20: **end if**
21: **end if**

▷ Continued on the following page

2.5 Concurrent Vs. Strong Concurrent

Although the authors of Wu et al. (2015a) make a distinction between pairs of *Concurrent* ($\approx$) lanes and pairs of *Strong Concurrent* ($\cong$) lanes, they do not

▷ Continued from the previous page

On receiving PERMIT(j) or timeout *tmt* occurs (no REJECT received):

22: delete j from HL_i
23: **if** (HL_i is empty) **then**
24: st_i = PASSING
25: construct the follow list,
$$flt = \{\ v \mid lid_v = lid_i\ \wedge\ v \in LL_i\ \wedge\ flt's\ length < NP\ \}$$
26: broadcast FOLLOW(i, flt)
27: **end if**
28: move and pass through the core area

On receiving FOLLOW(j, flt) from j:

29: **if** ($i \in flt$) **then**
30: st_i = PASSING
31: move and pass the core area
32: **else if** ($i \propto j$) **then**
33: delete j from HL_i
34: delete vehicles in flt from HL_i or LL_i
35: add the last one in flt to HL_i
36: **end if**

On exiting the intersection:

37: **if** (the passing is triggered by a FOLLOW(x, flt) **and** i is the last in flt) **then**
38: broadcast PERMIT(i)
39: **end if**

CoEnd

present a rational behind this distinction. We contacted the first author of the paper asking for a clarification but that went in vain.

In Algorithm 2 (lines 5 - 7 and lines 17 - 19), only strong concurrency is considered when implementing the preemption optimization (Section 2.4.2.5). In the situation depicted in *Figure* 6, it seems V_{26} (orange car) can preempt and allow V_{33} (blue car) to pass through the intersection along V_{10} (green car). However, the

algorithm prevents V_{26} from preempting since V_{10} and V_{33} are not strong concurrent with each other ($V_{10} \ncong V_{33}$).

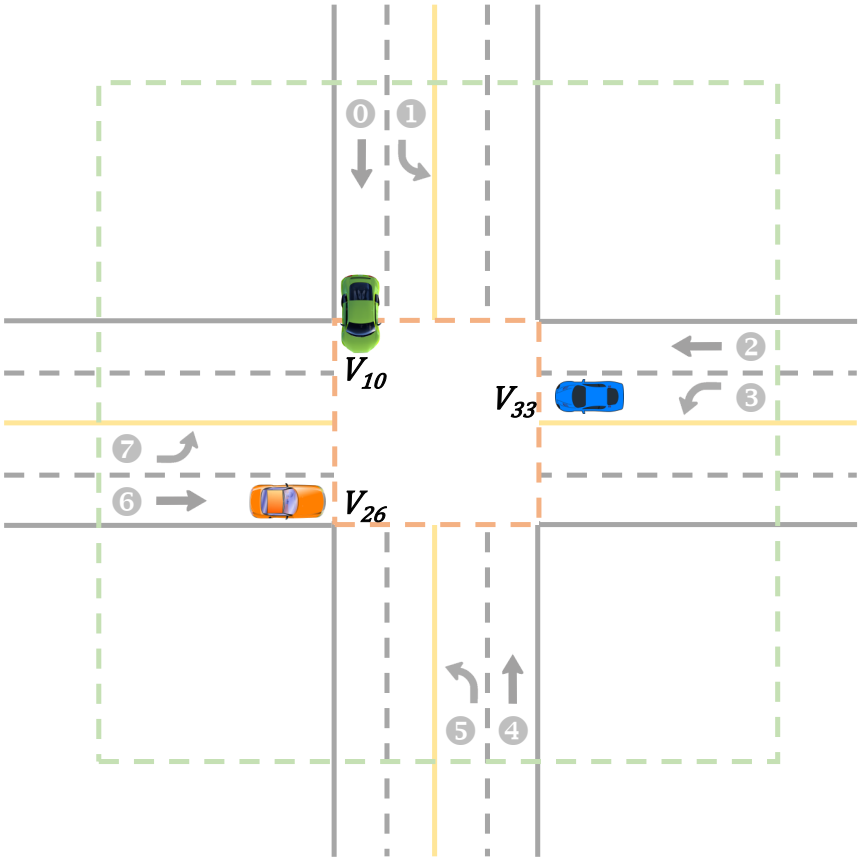

Figure 6. A situation where preemption is not applied due to lanes in question being not *Strong Concurrent* ($\cong$). Vehicles arrive at the intersection in the order V_{10}, V_{26}, V_{33}. Under normal operation they will pass through the intersection in the same order. However, since $V_{33} \approx V_{10}$, if V_{26} preempts its opportunity to pass through the intersection before V_{33}, that enables V_{33} to pass along V_{10} through the intersection. This would improve the concurrency and hence the throughput of the algorithm. However, since $V_{33} \ncong V_{10}$, Algorithm 2 does not allow preemption at this situation.

Looking deeper into a more complex scenario, we think we can infer the authors' intention behind this choice. Consider the situation depicted in *Figure* 5 with one difference. Instead of vehicles arriving in the order: green car, orange car, blue car and red car one after the other, suppose that the blue car and the red

car arrive at the same time. Then, according to our vehicle labeling scheme, the sequence of vehicles are labeled as: $V_{10}, V_{26}, \{V_{33}, V_{34}\}$. If the algorithm is designed to make V_{26} preempt any vehicle concurrent with V_{10}, it will preempt both V_{33} and V_{34}. Although, V_{33} and V_{34} are pairwise concurrent with V_{10} ($V_{33} \approx V_{10}$ and $V_{33} \approx V_{10}$), all three of them cannot pass through the intersection simultaneously since V_{33} and V_{34} conflict with each other ($V_{33} \propto V_{34}$). Thus, V_{26} preempting for both V_{33} and V_{34} simultaneously does not add any additional concurrency.

Therefore, we think that the authors of the algorithm have "hard-coded" a rule to automatically prioritize between two conflicting vehicles U and V ($U \propto V$), both pair-wise concurrent with a vehicle P that is already in the move through the core area of the intersection ($U \approx P$ and $V \approx P$), when deciding on preemptions.

Hard-coding such a rule to the algorithm reduces the concurrency in such situations as depicted in *Figure* 6. Going along with our example, we believe that even if V_{26} preempts both V_{33} and V_{34}, since $V_{33} \propto V_{34}$, one of them should *REJECT* the other for the algorithm to work. Therefore, V_{26} can listen to that *REJECT* message and revoke the preemption extended toward the vehicle that got rejected, making the algorithm work.

The process of handcrafting situations and trying to make sense of the algorithm is a daunting task. It is not only time consuming but also error-prone. There is a high chance that we are unable to manually enumerate and comprehend all the possibilities and nuances of the algorithm leading to inconclusive insights. This is where modeling tools like LTSA (Magee & Kramer, 2006; Magee et al., 2013), which can automatically and exhaustively search for problems and validate a system come into play.

Therefore, one sure way to figure out answers for these doubts is, modeling these situations in LTSA and letting LTSA do the hard work and tell us if something could go wrong. However, with the lifetime of this work we did not get to the point of modeling this specific situation.

CHAPTER III

MODELING - AN INTRODUCTION

The idea of modeling is to replicate an actual system without implementing or building it for real. The cost (time, effort, amount of money spent) of modeling should generally be less than the cost of building the actual system.

On one hand, if we try to model all (or almost all) the details of the actual system that is being modeled exactly as they operate in the actual system, the task of modeling would be as complex as building the actual system. This would defeat the sole purpose of modeling. Even if we can easily build such a complex model, it would complicate the analysis phase and make it difficult for us to focus on the specific aspects we are interested in analyzing.

On the other hand, if we abstract away too much from the actual system and make the model too simple, we run the danger of the insights we gain via the model not being applicable to the actual system.

So, we have to take the middle path where we omit the parts of the actual system that are irrelevant and do not interfere with the aspects we are interested in analyzing using the model, and simplify the parts and behaviors of the actual system that are of interest in the analysis. The model should be neither too complex nor too simple. It should capture the essence of the actual system to a level that is easy to analyze and validate the actual system.

For this thesis, we model using the "Labelled Transition System Analyser" (LTSA) (Magee & Kramer, 2006; Magee et al., 2013). In LTSA, individual components of a concurrent system are textually described using the language "Finite State Processes" (FSP). Then they are compiled and composed into bigger models that approximate how the actual system would behave as a whole. Both

individual models and compositions essentially are finite state machines and so we can validate these models in terms of reachability of undesired states and unreachability of desired states.

3.1 State Space

The state of a process is defined by the values and data stored in its variables, data structures and registers allocated to it (Magee & Kramer, 2006). LTSA performs an exhaustive search of the state space of the model to find anomalies. Thus, the higher the number of states, the costlier the model validation. If process P_i has S_{P_i} states and there are n processes in a concurrent system, a system composed of those processes has at most:

$$\prod_{i=1}^{n} S_{P_i}$$

states. The upper bound occurs when all the states of all the processes are independent of each other, which usually is not the common case.

3.2 Modeling

The task of modeling is to make sure that the model reaches all the desirable states at desirable moments avoiding any undesirable state. During the model validation phase, LTSA exhaustively searches the finite state machine of the composed model and reports three kinds of problems:

1. Any sequence of actions that leads to a deadlock in the system. A deadlock occurs when no further action is possible. Two cars at the intersection each waiting for other to pass through the intersection is an example for a deadlock.

2. Any sequence of actions that leads to a marked undesirable state in the system. An undesirable state could be two cars entering an intersection and crashing.

3. A marked desirable state not being reached. There is no path or set of actions that will get to a state that is desirable. One car never being able to pass through the intersection while other cars are passing through is an example for this case.

3.3 Making the State Space Manageable

The distributed mutual exclusion algorithm for intersection traffic control (Algorithm 2) that we model has a very large state space. Our LTSA model of a car, assuming that there are only two cars at the intersection, has 325 states. The intersection that we model (*Figure.* 1) contains eight lanes coming in and going out. So, building a full-scale model would require composing many cars coming in from different directions, which makes the final model prohibitively large. Thus, we take the approach of breaking down the problem into smaller sub problems and modeling and verifying those sub problems to shed more light into the bigger problem.

First, we model an intersection with only two approaching cars. If the two cars approach the intersection from two concurrent lanes, there would be no issue. Therefore, we only focus on the situation where the two cars are coming from two conflicting lanes. This enables us to strip away the lane related variables in the actual algorithm leading to further reduction of the logic required to check the relationship between the two lanes (*Figure.* 2). With two cars at the intersection, all the broadcast messages reduce to point to point communication, which is

another saving we make in modeling the V2V communication. All these reductions save a lot of state space. The rationale behind this choice is:

Since,

$$\text{Intersection with two cars} \subset \text{Intersection with more than two cars}$$

we have,

$$\left(\begin{array}{c}\text{There are issues with the}\\ \text{algorithm when two cars are at}\\ \text{the intersection}\end{array}\right) \implies \left(\begin{array}{c}\text{There are issues with the}\\ \text{algorithm when more than two}\\ \text{cars are at the intersection}\end{array}\right)$$

CHAPTER IV

MODELING - V2V COMMUNICATION

Since we are only considering an intersection with two cars (Chapter III), we model broadcasts as point to point messages between the two cars. To start the discussion of modeling communication, we define a simple sender process (Listing 1) and a simple receiver process (Listing 2).

4.1 Modeling Synchronous Communication

The action *before_send* of simple sender (Listing 1) resembles all the actions the simple sender preforms before sending a message (*send_message* action) and the action *after_send* resembles everything the sender does after sending the message. Similarly, the simple receiver (Listing 2) has two actions *before_receive* and *after_receive* to resemble all the actions the receiver performs before and after receiving the message.

Listing 3 models synchronized communication between the simple sender and the simple receiver. The line 7 (`send_message/receive_message`) synchronizes the *SENDER*'s *send_message* action with the *RECEIVER*'s *receive_message* action. In the final model, the only action visible is *send_message* because it is synonymous to *receive_message*. Due to the synchronization, the *send_message* action is not available to the *SENDER* until the *RECEIVER* is ready to receive (in this example model, before *RECEIVER* performs the *before_receive* action). In addition to being synchronous, this model does not take message propagation delay into account at all. *RECEIVER* receives the message instantly at the same moment the *SENDER* sends the message.

This level of synchronization is problematic when it comes to modeling the V2V communication involved in the distributed mutual exclusion algorithm for

Listing 1 Simple sender process

```
SENDER =
    (
          before_send
      -> send_message
      -> after_send        -> SENDER
    ).
```

Listing 2 Simple receiver process

```
RECEIVER =
    (
          before_receive
      -> receive_message
      -> after_receive     -> RECEIVER
    ).
```

Listing 3 Composition for synchronous communication

```
|| SYNC_COMMUNICATE =
    (
          SENDER
      || RECEIVER
    )
      / {
          send_message / receive_message
        }.
```

intersection traffic control. Assume that the message propagation delay within the queue area (*Figure* 1) is t_{prop} milliseconds. Consider the situation where vehicles V_{10} and V_{22} entering the queue area with V_{22} entering queue area less than t_{prop} milliseconds after V_{10}. Then, at the time V_{10} enters the queue area, there is nobody ready to receive the $REQUEST$ message of V_{10}. However, by the time V_{22} enters the queue area, the $REQUEST$ message V_{10} sent should be received by V_{22}. With a synchronous model of the message passing we are unable to model this behavior.

4.2 Making the Communication Asynchronous

To model an asynchronous message passing scheme, we have to introduce an intermediate process that facilitates the communication between the *SENDER* and the *RECEIVER*. We call this process the *ASYNC_NETWORK* (Listing 4). This setup is closer to how communication between vehicles is actually implemented in the real world. When the *SENDER* wants to send a message, it passes the message to the *ASYNC_NETWORK*. Then the message takes some time to route through the *ASYNC_NETWORK* and finally reaches the *RECEIVER*. Since we are not modeling a connection oriented communication protocol (E.g. TCP/IP), we do not have to model a handshaking phase, which establishes a connection. Thus, a *RECEIVER* that is ready to receive messages is not required at the time of a *SENDER* sending a message.

Composing the three processes *SENDER*, *RECEIVER* and *ASYNC_NETWORK* (Listings 1, 2 and 4), we can model an asynchronous message passing scheme as in Listing 5.

One limitation of this model is that if the *SENDER* sends the same message more than once before the first message is received by the *RECEIVER*, they will be lost in the *ASYNC_NETWORK* process and the *RECEIVER* will be unaware

Listing 4 Network process that enables asynchronous message passing

```
const False = 0
const True  = 1
range Bool  = False..True

ASYNC_NETWORK = BUFFER[ False ],

BUFFER[ in_transit : Bool ] =
    (
          accept_message  -> BUFFER[ True ]

      | when (in_transit)
          deliver_message -> BUFFER[ False ]
    ).
```

Listing 5 Composition for asynchronous communication

```
||  ASYNC_COMMUNICATE =
      (
           SENDER
        || RECEIVER
        || ASYNC_NETWORK
      )
        / {
            send_message    / accept_message,
            receive_message / deliver_message
          }.
```

of these multiple transmissions. However, since the algorithm we model assumes a reliable communication channel that provides guaranteed message delivery and vehicles do not retransmit messages, this limitation does not affect our models. We can easily rectify this limitation by making the *NETWORK* remember the number of times the sender transmits a particular message at the cost of increased state complexity. In our work, we favor less states and go with the more concise model.

4.3 Message Ordering

Since the algorithm being modeled assumes a first-in-first-out (FIFO) wireless channel, we need to enforce message ordering in the model. To start the discussion about message ordering, we need a sender who sends more than one message and a receiver who receives them (Listing 6).

A real-world packet switched network does not preserve the message ordering. The illusion of ordered message delivery is achieved by techniques such as adding sequence numbers to messages (or more accurately packets) sent by the sender and acknowledgments of messages received, sent by the receiver, handled by low level communication subsystems at the sender and the receiver.

We can model a channel that accepts messages from a sender in any order and then delivers them to a receiver in the same order they are accepted. This is done by making the network process, *MULTI_NETWORK_FIFO*, maintain a queue of messages accepted (Listing 7). Each new message accepted is appended to the tail of the queue. Only the message at the head of the queue is delivered when the receiver is ready to receive that message. Since the model is a finite state process (FSP), the queue should be bounded and in the current model we keep it to a size of 3. Any message sent while the network queue is full is dropped by the

Listing 6 Sender and receiver for multiple messages

```
const False = 0
const True  = 1
range Bool  = False..True

// Message types
const REQUEST = 100
const REJECT  = 200
const PERMIT  = 300

set MESSAGES = { [ REQUEST ], [ REJECT ], [ PERMIT ] }


MULTI_SENDER =
  (
        before_send                     -> SENDING
  ),

SENDING =
  (
        send_message[ msg : MESSAGES ] -> SENDING
    |   done_sending
      -> work_after_send                -> MULTI_SENDER
  ).


MULTI_RECEIVER =
  (
    before_receive                      -> RECEIVING
  ),

RECEIVING =
  (
        receive_message[ msg : MESSAGES ]-> RECEIVING
    |   done_receiving
      -> work_after_receiving          -> MULTI_RECEIVER
  ).
```

network. In fact, this is exactly how real world network components such as routers and switches operate when their internal buffers are full.

MULTI_NETWORK_FIFO process (Listing 7) can be tested by composing it with *MULTI_SENDER* and *MULTI_RECEIVER* (Listing 6) as shown in Listing 8.

4.4 Reducing the State Space of the Network Process

The general purpose model of the first-in-first-out network process (Listing 7) ends up having 121 states. To make the analysis more efficient, we are biased towards smaller models. In LTSA tool, smaller models bring the added value of being able to visualize the graph of the finite state model. In some cases, we can look at the graph of the model, make sense what is going on and debug the model if necessary. The current version of LTSA only visualizes models having 64 or less states.

So, we look closely at the specific algorithm (Algorithm 2) we are modeling and ask the question "Can we model a first-in-first-out communication channel that is problem specific instead of the general-purpose channel we modeled (Listing 7)?". To answer this question, we focus closely on the messaging pattern of a single vehicle.

The first message each vehicle sends is *REQUEST* and each vehicle sends only one such message as it enters the queue area of the intersection (*Figure.* 1). Further, each vehicle sends at most one *PERMIT* message and if it does, it is the last message a vehicle sends. In between those two messages, a vehicle may send multiple *REJECT* messages in response to *REQUEST* messages from other vehicles that are in conflicting lanes (*Figure.* 2). We can summarize this messaging

Listing 7 General purpose network that preserves the message order

```
MULTI_NETWORK_FIFO = QUEUE,

QUEUE =
  (
        accept_message[ _1st : MESSAGES ]
                    -> QUEUE[ _1st ]
  ),

QUEUE[ _1st : MESSAGES ] =
  (
        accept_message[ _2nd : MESSAGES ]
                    -> QUEUE[ _1st ][ _2nd ]

    |   deliver_message[ _1st ]
                    -> QUEUE
  ),

QUEUE[ _1st : MESSAGES ][ _2nd : MESSAGES ] =
  (
        accept_message[ _3rd : MESSAGES ]
                    -> QUEUE[ _1st ][ _2nd ][ _3rd ]

    |   deliver_message[ _1st ]
                    -> QUEUE[ _2nd ]
  ),

QUEUE[ _1st : MESSAGES ][ _2nd : MESSAGES ]
                                  [ _3rd : MESSAGES ] =
  (
        accept_message[ _4th : MESSAGES ]
      -> drop_message[ _4th ]
                    -> QUEUE[ _1st ][ _2nd ][ _3rd ]

    |   deliver_message[ _1st ]
                    -> QUEUE[ _2nd ][ _3rd ]
  ).
```

Listing 8 Composition for asynchronous FIFO communication

```
|| ASYNC_FIFO_COMMUNICATE =
    (
          MULTI_SENDER
      || MULTI_RECEIVER
      || MULTI_NETWORK_FIFO
    )
      / {
          send_message[ msg : MESSAGES ]
                                / accept_message[ msg ],
          receive_message[ msg : MESSAGES ]
                                / deliver_message[ msg ]
        }.
```

pattern by a regular expression:

$$(REQUEST)(REJECT)*(PERMIT)?$$

Since we are focusing on an intersection with only two vehicles, each vehicle sends one and only one *REQUEST* message and a *REJECT* message is sent only in response to a *REQUEST* message, in our model each vehicle sends at most one *REJECT* message. Thus, we have the opportunity to further reduce our algorithm specific network model to a context specific model that can only handle the messaging situation:

$$(REQUEST)(REJECT)?(PERMIT)?$$

We incorporate these observations into the network process to model a problem specific first-in-first-out channel (Listing 9), which we call *VMEI_NETWORK*. Since in the actual algorithm (Algorithm 2) each message passed includes the vehicle id, *VID*, (Chapter II - Section 2.4.2.4), we have modeled the *VMEI_NETWORK* process to handle that aspect as well. With these

modifications, we are able to reduce the number of states in the network process to 8.

To test our hypothesis regarding designing an algorithm plus context specific first-in-first-out channel, we design a sender process (Listing 10) that mimics only the message sending behavior of a vehicle that is running an implementation of the distributed mutual exclusion algorithm for intersection traffic control (Algorithm 2). The complete model for a vehicle is discussed in Chapter V. We further enhance the receiver process so that it can receive messages with the vehicle id (Listing 11). Composing *VMEI_NETWORK*, *VMEI_SENDER* and *VMEI_RECEIVER* (Listings 9, 10 and 11) as in the Listing 12, these new processes can be tested in action.

Listing 9 Network process that takes the messaging behavior of vehicles into account to provide a FIFO channel

```
VMEI_NETWORK_FIFO( VID = 0 ) = QUEUE,

QUEUE =
  (
      accept_message [ REQUEST ][ VID ]
                -> QUEUE[ REQUEST ]

    | accept_message [ REJECT  ][ VID ]
                -> QUEUE[ REJECT  ]

    | accept_message [ PERMIT  ][ VID ]
                -> QUEUE[ PERMIT  ]
  ),

QUEUE[ REQUEST ] =
  (
      accept_message [ REJECT  ][ VID ]
                -> QUEUE[ REQUEST ][ REJECT ]

    | accept_message [ PERMIT  ][ VID ]
                -> QUEUE[ REQUEST ][ PERMIT ]

    | deliver_message[ REQUEST ][ VID ] -> QUEUE
  ),

QUEUE[ REJECT ] =
  (
      accept_message [ PERMIT  ][ VID ]
                -> QUEUE[ REJECT  ][ PERMIT ]

    | deliver_message[ REJECT  ][ VID ] -> QUEUE
  ),

QUEUE[ PERMIT ] =
  (
      deliver_message[ PERMIT  ][ VID ] -> QUEUE
  ),
```

```
29 QUEUE[ REQUEST ][ REJECT ] =
30   (
31       accept_message [ PERMIT  ][ VID ]
32                 -> QUEUE[ REQUEST ][ REJECT ][ PERMIT ]

33     | deliver_message[ REQUEST ][ VID ]
34                 -> QUEUE[ REJECT  ]
35   ),

36 QUEUE[ REQUEST ][ PERMIT ] =
37   (
38       deliver_message[ REQUEST ][ VID ]
39                 -> QUEUE[ PERMIT  ]
40   ),

41 QUEUE[ REJECT ][ PERMIT ] =
42   (
43       deliver_message[ REJECT  ][ VID ]
44                 -> QUEUE[ PERMIT  ]
45   ),

46 QUEUE[ REQUEST ][ REJECT ][ PERMIT ] =
47   (
48       deliver_message[ REQUEST ][ VID ]
49                 -> QUEUE[ REJECT  ][ PERMIT ]
50   ).
```

Listing 10 Sender process that mimics the message sending behavior of a vehicle. Any sequence of messages sent by a complete model of a vehicle at an intersection with two vehicles can be generated by this sender.

```
1  // Message types
2  const REQUEST = 100
3  const REJECT  = 200
4  const PERMIT  = 300

5  set MESSAGES = { [ REQUEST ], [ REJECT ], [ PERMIT ] }


6  VMEI_SENDER( VID = 0 ) =
7    (
8            before_send
9      -> send_message[ REQUEST ][ VID ]   -> SEND_REJECT
10   ),

11 SEND_REJECT =
12   (
13           send_message[ REJECT  ][ VID ] -> SEND_PERMIT
14     |     skip_reject                     -> SEND_PERMIT
15   ),

16 SEND_PERMIT =
17   (
18           send_message[ PERMIT  ][ VID ]
19       -> work_after_send                 -> VMEI_SENDER

20     |     skip_permit                     -> VMEI_SENDER
21   ).
```

Listing 11 Receiver process that accepts messages with the vehicle id *VID*

```
VMEI_RECEIVER( VID = 1 ) =
  (
    before_receive                          -> RECEIVING
  ),

RECEIVING =
  (
        receive_message[ msg : MESSAGES ][ 1 - VID ]
                                            -> RECEIVING
    |   done_receiving
      -> work_after_receiving               -> VMEI_RECEIVER
  ).
```

Listing 12 Composition for algorithm plus context specific communication

```
|| VMEI_FIFO_COMMUNICATE =
    (
        VMEI_SENDER( 0 )
      || VMEI_RECEIVER( 1 )
      || VMEI_NETWORK_FIFO( 0 )
    )
      / {
          send_message[ msg : MESSAGES ][ 0 ]
                          / accept_message[ msg ][ 0 ],

          receive_message[ msg : MESSAGES ][ 0 ]
                          / deliver_message[ msg ][ 0 ]
        }.
```

CHAPTER V

MODELING - VEHICLE

This chapter discusses the modeling process of the vehicle. In our models, we use the term "Car" to refer to a vehicle just to keep things shorter. Our model of a car is based on the assumption that there are only two cars at the intersection (Chapter III - section 3.3). The only interesting scenario to model with two cars is when they approach the intersection from two conflicting lanes (*Figures* 1 and 2). These assumptions add several simplifications to the model of the car:

1. All broadcasts are reduced to point to point communication between the two cars.

2. No need to model the logic to check the relationship between the lanes where cars are coming from. Also, no need to include the lane a car is entering the intersection in the messages passed.

3. No need to deal with the logic for preemptions (Chapter II : Lines 5 - 7 of Algorithm 2).

4. No need to model the "follow list" related portions of the algorithm (Chapter II : Lines 25 - 26, 29 - 36 and 37 - 38 of Algorithm 2).

5.1 Extended State Diagram

The state diagram (*Figure* 3) that is presented in the paper where the distributed mutual exclusion algorithm for intersection traffic control is published (Wu et al., 2015a) is too high level to be modeled or implemented. After thoroughly analyzing the dynamics of the Algorithm 2, we extend it sufficiently so that we can directly map it into a LTSA model (*Figure* 7).

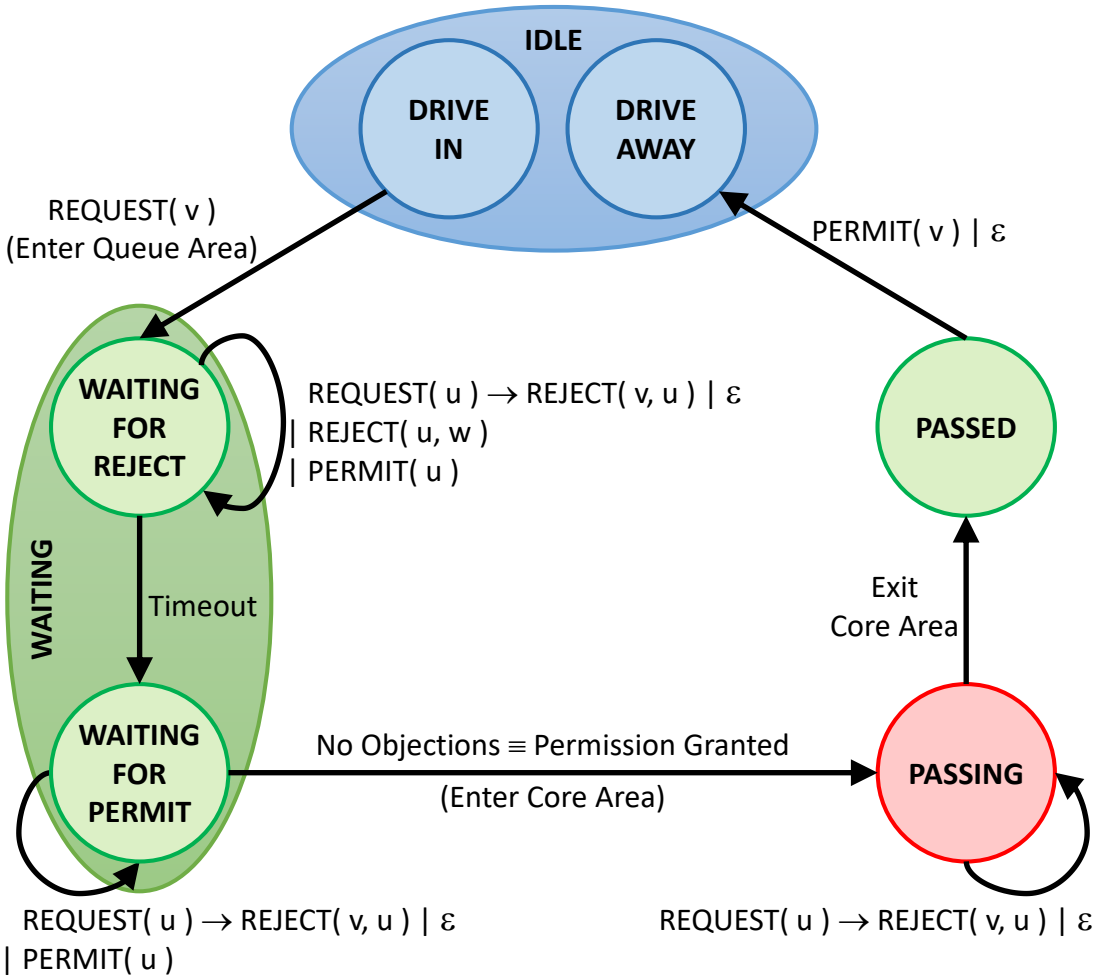

Figure 7. The extended state transition diagram of the distributed algorithm. Both *DRIVE_IN* and *DRIVE_AWAY* states are used as synonyms for *IDLE* state just to make LTSA treat them differently. *WAITING* state is divided into two states *WAITING_FOR_REJECT* (waiting before timeout) and *WAITING_FOR_PERMIT* (waiting after the timeout has expired). A *PASSED* state is introduced just after a vehicle exits the core area of the intersection for a vehicle to handle the housekeeping task of notifying other vehicles that are waiting till this vehicle clears the core area of the intersection that this vehicle is no longer in their way. We need two more states to process *REQUEST* messages each connected to the two *WAITING* states. We do not include them in this diagram for brevity and we use a shorthand notation $REQUEST(u) \rightarrow REJECT(v,u)|\varepsilon$ to represent that.

According to the state diagram in *Figure* 3, a vehicle is in state *IDLE* both before it enters the queue area (*Figure* 1) and after it exits the queue area of the

intersection, which means it is not actively executing the algorithm. However, in our model if we represent both these conditions using a single state, LTSA loops the same vehicle back into the same intersection, which is not a behavior we intend for. Thus, we have divided the *IDLE* state into two states *DRIVE_IN* and *DRIVE_AWAY*.

A vehicle starts in the *DRIVE_IN* state. As a vehicle enters the queue area, it broadcasts a *REQUEST* message and enters the *WAITING* state. According to how the algorithm works, a vehicle has two waiting stages. First, it waits for *REJECT* messages from other vehicles for a designated period that ends with the expiration of a timeout. Afterwards, a vehicle waits till all the vehicles that objected to it passing through give it *PERMISSION* to pass through. Thus, we divide the *WAITING* state into two states *WAITING_FOR_REJECT*, which is waiting before the expiration of timeout and *WAITING_FOR_PERMIT*, which is waiting after the expiration of timeout.

5.2 Handling *PERMIT* Messages

While a vehicle is in either of the *WAITING* states, it can receive *PERMIT* messages. Algorithm 2 uses the same function (Lines 22 - 28) to handle the three events: 1) receiving a *PERMIT* message before timeout, 2) receiving a *PERMIT* message after timeout and 3) the expiration of the timeout. We believe that this is not accurate for several reasons:

1. Line 22 of the function, $delete \quad j \quad from \quad HL_i$, is for deleting the vehicle j from the high list, HL_i, of vehicle i, upon vehicle i receiving *PERMIT(j)* from vehicle j. However, when this function is called upon the expiration of the timeout, no such vehicle j is defined, which causes a problem.

2. Lines 23 to 27, checks whether there is nobody objecting to vehicle i passing through and if that is the case, vehicle i starts passing through. Consider the situation depicted in *Figure* 8. Vehicles arrive in the order V_{10}, V_{25}, V_{36} and V_{36} arrives just before V_{10} exits the core area. Let the order of messages received at V_{36} be: *REJECT(V_{10}, V_{36})*, *PERMIT(V_{10})* and *REJECT(V_{25}, V_{36})*. Assume that V_{36} receives the first two messages before its timeout expires and *REJECT(V_{25}, V_{36})* gets delayed. With this setup, due to the use of the same function to handle *PERMIT* messages both before and after the timeout, after receiving the 2$^{\text{nd}}$ message, V_{36} sees that its high list is empty and starts to pass through the intersection while the *REJECT(V_{25}, V_{36})* is in flight. In the meantime, V_{25} too gets the *PERMIT(V_{10})* message and it too enters the core area leading to a crash.

To rectify these problems we propose breaking the "**On receiving PERMIT(j) or timeout *tmt* occurs**" function (Lines 22 - 28) in Algorithm 2 into two separate functions:

1. **On receiving PERMIT(j)** that just removes vehicle j from i's high list if j is present in i's high list (Line 22 of Algorithm 2). Vehicle does not change the state.

2. **Try to pass through** that checks whether the high list is empty and if so takes necessary steps and starts passing through (Lines 23 - 28 of Algorithm 2).

When a vehicle receives a *PERMIT(j)* message while it is in state *WAITING_FOR_REJECT*, it just executes **On receiving PERMIT(j)** function. When a *PERMIT(j)* message is received while a vehicle is in state

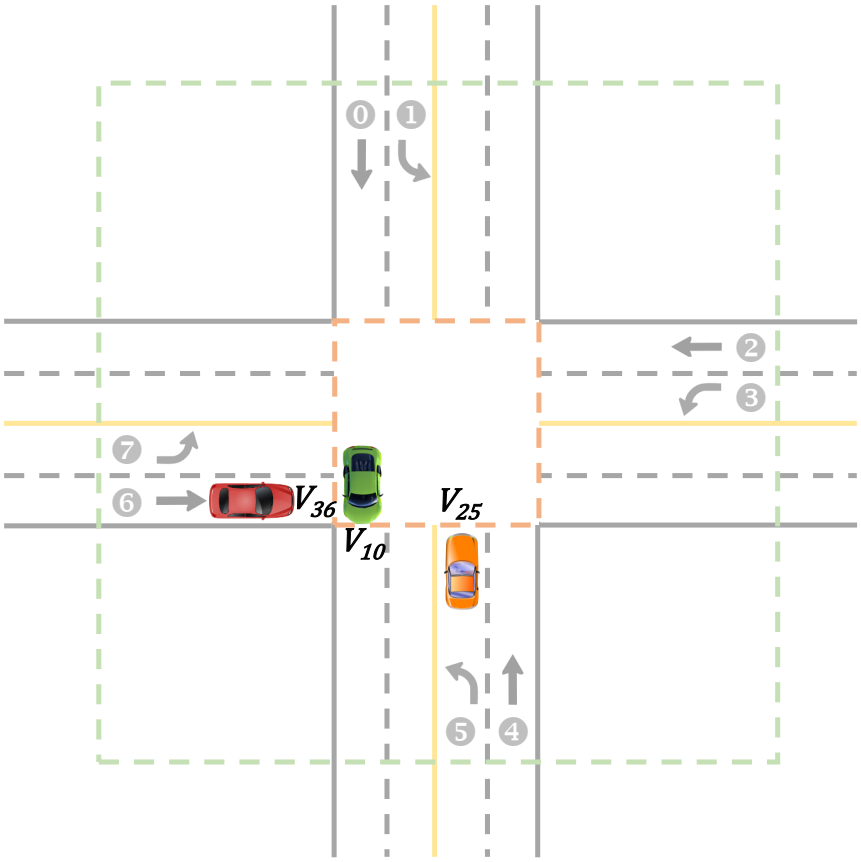

Figure 8. A scenario where using the same function to handle *PERMIT* messages both before and after the timeout can lead to a problem. Vehicles arrive in the order V_{10}, V_{25}, V_{36}. V_{36} (red car) arrives just before V_{10} (green car) exits the core area. Let the order of messages received at V_{36} be: *REJECT*(V_{10}, V_{36}), *PERMIT*(V_{10}) and *REJECT*(V_{25}, V_{36}). Assume that V_{36} receives the first two messages before its timeout expires and *REJECT*(V_{25}, V_{36}) gets delayed. With this setup, due to the use of the same function to handle *PERMIT* messages both before and after the timeout, after receiving the 2$^{\text{nd}}$ message, V_{36} sees that its high list is empty and starts to pass through the intersection while the *REJECT*(V_{25}, V_{36}) is in flight. In the meantime, V_{25} (orange car) too gets the *PERMIT*(V_{10}) message and it too enters the core area leading to a crash.

WAITING_FOR_PERMIT, it executes **On receiving PERMIT(j)** function and then executes **Try to pass through** function. In either of these cases the vehicle does not change its state. However, when the timeout expires, a vehicle

changes its state from *WAITING_FOR_REJECT* to *WAITING_FOR_PERMIT* and executes only **Try to pass through** function.

5.3 Exiting the Core Area

A vehicle performs the housekeeping tasks after exiting the core area of the intersection by executing the function "**On exiting the intersection**" (Lines 37 - 39 of Algorithm 2). However, that function prescribes what to do only if the passing is triggered by a *FOLLOW* message. But, there are situations where a vehicle starts passing through the intersection without being induced by a *FOLLOW* message. The scenario we model, an intersection with only two cars, is such a situation.

With only two cars approaching the intersection from two conflicting lanes, one vehicle (say V) proceeds to pass through the intersection because the other vehicle (say U) does not object (but not because it was included in a *FOLLOW* list). Vehicle U would wait till vehicle V passes through and clears the core area (*Figure* 1). In this case, vehicle V should broadcast a *PERMIT* message to notify vehicle U that it is safe for U to enter the core area.

We believe that we can use the low list LL_i (Section 2.4.2.3) for a vehicle to decide whether to broadcast a *PERMIT* message or not. Vehicles that end up in LL_i are the vehicles where vehicle i objected to passing through the intersection. Those vehicles are waiting for a *PERMIT* message from vehicle i. Therefore, upon exiting the core area, if vehicle i finds that its low list LL_i is not empty, it should broadcast a *PERMIT* message. We model this behavior for the car.

5.4 A Walk-through of the Extended State Transition Diagram

With the aforementioned clarifications, we are sufficiently armed to dive into the business of modeling the vehicle. To cement how a vehicle moves through the

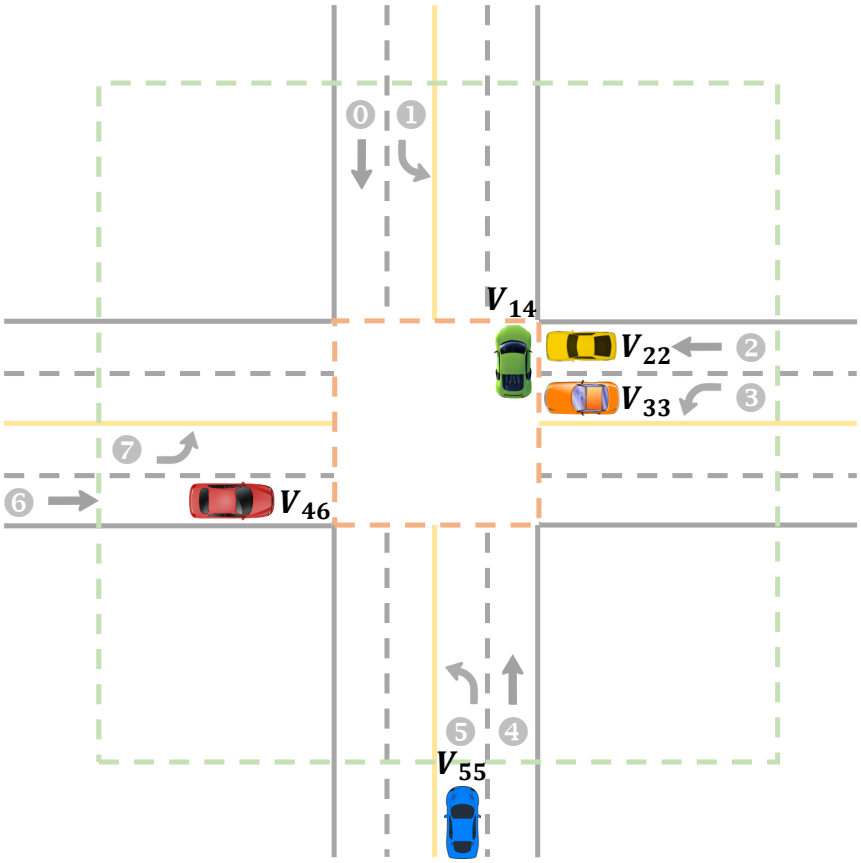

Figure 9. A scenario to walk-through the extended state transition diagram (*Figure* 7). Vehicles arrive in the order $V_{14}, V_{22}, V_{33}, V_{46}, V_{55}$. V_{14} (green car) is about to exit the core area of the intersection and V_{55} (blue car) is about to enter the queue area of the intersection. V_{22} and V_{33} (orange and yellow cars) are waiting for V_{14} to clear the core. V_{46} (red car) has just arrived in the queue area of the intersection. The sequence of states passed and actions taken by V_{46} from this point onward are presented in Table 2. Note that all the cars except V_{22} conflict with V_{46}. Therefore, V_{22} does not respond to the *REQUEST* of V_{46} with a *REJECT*. Further, V_{55} does not hear the *REQUEST* of V_{46} and hence it does not respond with a *REJECT*. However, V_{46} hears the *REQUEST* of V_{55} and it broadcasts a *REJECT*.

extended state transition diagram, we present an example scenario (*Figure* 9 and Table 2). *Figure* 9 sets up an example situation that could arise at an intersection. Table 2 summarizes the sequence of states passed in response to actions taking place in vehicle V_{46} (red car) in the figure. Note that all the cars except V_{22} (yellow car) conflict with V_{46}. Therefore, V_{22} does not respond to the *REQUEST* of V_{46}

Table 2. Extended state transition diagram - A walk-through: Sequence of states passed and actions taken by vehicle V_{14} (red car) depicted in *Figure* 9 starting from the situation depicted in that figure.

Start State	**Action**	**End State**	$HL_{V_{46}}$	$LL_{V_{46}}$
DRIVE IN	Broadcast REQUEST(V_{46})	WAITING FOR REJECT	{ }	{ }
WAITING FOR REJECT	Receive REJECT(V_{14}, V_{46})	WAITING FOR REJECT	{ V_{14} }	{ }
WAITING FOR REJECT	Receive REJECT(V_{33}, V_{46})	WAITING FOR REJECT	{ V_{14}, V_{33} }	{ }
WAITING FOR REJECT	Receive PERMIT(V_{14})	WAITING FOR REJECT	{ V_{33} }	{ }
WAITING FOR REJECT	timeout	WAITING FOR PERMIT	{ V_{33} }	{ }
WAITING FOR PERMIT	Receive REQUEST(V_{55})	WAITING FOR PERMIT	{ V_{33} }	{ }
WAITING FOR PERMIT	Broadcast REJECT(V_{46}, V_{55})	WAITING FOR PERMIT	{ V_{33} }	{ V_{55} }
WAITING FOR PERMIT	Receive PERMIT(V_{33})	WAITING FOR PERMIT	{ }	{ V_{55} }
WAITING FOR PERMIT	No Objections	PASSING	{ }	{ V_{55} }
PASSING	Exit Core Area	PASSED	{ }	{ V_{55} }
PASSED	Broadcast PERMIT(V_{55})	DRIVE AWAY	{ }	{ }

with a *REJECT*. Further, V_{55} (blue car) does not hear the *REQUEST* of V_{46} and

hence it does not respond with a *REJECT*. However, V_{46} hears the *REQUEST* of V_{55} and it broadcasts a *REJECT*.

5.5 Modeling the Car Begins

We start with the simplest model, which tries to mimic Algorithm 2 to the letter. After showing the pitfalls of the algorithm, we make the model more complex to rectify those issues. In this manner, we present a series of models for the vehicle each becoming a bit more complex and each addressing an issue in its predecessor.

To make models more readable and concise, we define some constants, sets and ranges (Listing 13). When composing these models in LTSA, this portion should be included for it to compile without errors.

5.6 Modeling the CAR Without a Sense of Priority

Although Wu et al. (2015a) mentions a notion of a priority in several places (Chapter II - section 2.3.2), Algorithm 2 does not include any notion of a priority. Our first model of the car (Listing 14) models this behavior following along the extended state diagram (*Figure* 7).

Please note that in our model we have the *timeout* action prefixed with a long tail of underscores (_____*timeout*). We make the choice to prefix a tail of underscores to time related actions in our models so that they stand out in the action traces leading to undesirable states produced by LTSA. There is no other significance than aesthetics in making this choice.

5.6.1 Testing the Model.

Listing 15 demonstrates how the model of the *CAR* in Listing 14 can be tested by composing it with the *VMEI_NETWORK_FIFO* process (Chapter IV - Listing 9).

Listing 13 Constants, sets and ranges that are used in LTSA models. When composing the models, include this section.

```
const False = 0
const True  = 1
range Bool  = False..True

// Vehicle IDs
const Car_0 = 0
const Car_1 = 1
set   CARS  = { [Car_0], [Car_1] }

// A literal to indicate an empty list
const EMPTY = -1

// Possible values that can be in
// High (hi) or Low (lo) lists
set LIST_VALS = { [EMPTY], [Car_0], [Car_1] }

// Message types
const REQUEST = 100
const REJECT  = 200
const PERMIT  = 300

set MESSAGES = { [REQUEST], [REJECT], [PERMIT] }
```

Listing 14 *CAR* without a sense of priority. This model exactly resembles the car in Algorithm 2.

```
CAR( VID = 0 ) = DRIVE_IN,

DRIVE_IN =
  (
        start
     -> broadcast[ REQUEST ][ VID ]
        -> WAITING_FOR_REJECT [ EMPTY  ][ EMPTY  ]
  ),


WAITING_FOR_REJECT[ lo:LIST_VALS ][ hi:LIST_VALS ] =
  (
        ____________________timeout
        -> WAITING_FOR_PERMIT [   lo   ][   hi   ]

    |   receive[ REQUEST ][ sender: {[1 - VID]} ]
        -> PROCESS_REJECT     [ sender ][   hi   ]

    |   receive[ REJECT  ][ sender: {[1 - VID]} ]
        -> WAITING_FOR_REJECT [   lo   ][ sender ]

    |   receive[ PERMIT  ][ sender: {[1 - VID]} ]
        -> WAITING_FOR_REJECT [   lo   ][ EMPTY  ]
  ),


PROCESS_REJECT[ lo:LIST_VALS ][ hi:LIST_VALS ] =
  (
        ____________________timeout
        -> WAITING_FOR_PERMIT [   lo   ][   hi   ]

    |   broadcast[ REJECT ][ VID ]
        -> WAITING_FOR_REJECT [   lo   ][   hi   ]
  ),
```

```
26 WAITING_FOR_PERMIT[ lo:LIST_VALS ][ hi:LIST_VALS ] =
27   (
28       when ( hi == EMPTY )
29          enter          -> PASSING [   lo   ][   hi   ]

30     |    receive[ REQUEST ][ sender: {[1 - VID]} ]
31       -> broadcast[ REJECT ][ VID ]
32             -> WAITING_FOR_PERMIT [ sender ][   hi   ]

33     |    receive[ PERMIT ][ sender: {[1 - VID]} ]
34             -> WAITING_FOR_PERMIT [   lo   ][ EMPTY  ]
35   ),

36 PASSING[ lo:LIST_VALS ][ hi:LIST_VALS ] =
37   (
38          exit           -> PASSED  [   lo   ]

39     |    receive[ REQUEST ][ sender: {[1 - VID]} ]
40       -> broadcast[ REJECT ][ VID ]
41                        -> PASSING [ sender ][   hi   ]
42   ),

43 PASSED[ lo:LIST_VALS ] =
44   (
45       when ( lo != EMPTY )
46          broadcast[ PERMIT ][ VID ]
47                        -> DRIVE_AWAY

48     | when ( lo == EMPTY)
49          drive          -> DRIVE_AWAY
50   ),


51 DRIVE_AWAY =
52   (
53          drive          -> DRIVE_AWAY
54   ).
```

Listing 15 Composition for testing *CAR* without a sense of priority

```
1  ||INTERSECION =
2      (
3           car[ i: 0..1 ]:CAR( i )
4        || VMEI_NETWORK_FIFO( 0 )
5        || VMEI_NETWORK_FIFO( 1 )
6      )
7        /   {
8              start       / car[ 0..1 ].start,
9              car[i: 0..1].broadcast[msg: MESSAGES][ i ]
10                         / accept_message [ msg ][  i  ],
11             car[i: 0..1].receive  [msg: MESSAGES][1-i]
12                         / deliver_message[ msg ][ 1-i ]
13           }.
```

When checked for safety of the model, LTSA points out a deadlock (*Figure* 10).

A deadlock is bad; it puts the whole system into a grinding halt. It greatly affects efficiency of the system. However, with respect to the specific problem we are modeling, two cars entering the core area of the intersection from two conflicting lanes (in Computer Science terms, a critical section violation) and crashing is catastrophic. To stress test the model for such flaws we need to use *fluent*s and *assert*s (Listing 16).

The *fluent* in line 1 of Listing 16 means that *CAR_0_PASSING* becomes true whenever car 0 enters the core area of the intersection and it becomes false whenever car 0 exits the core area. The *assert* in line 3 means that always ([ ]) it should not (!) be the case where both *CAR_0_PASSING* true and (&&)

Trace to DEADLOCK:

start

car.0.broadcast.100.0

car.1.broadcast.100.1

car.0.receive.100.1

car.0.broadcast.200.0

car.1.receive.100.0

car.1.broadcast.200.1

car.0.receive.200.1

car.0._____timeout

car.1.receive.200.0

car.1._____timeout

Figure 10. Deadlock situation in the composition of Listing 15 for a CAR without a sense of priority.

CAR_1_PASSING true. In essence, both car 0 and car 1 should never enter the core area of the intersection simultaneously.

Composing the model (Listing 15) in LTSA with the fluents and asserts (Listing 16) in place gives us the power to investigate the model for a potential crash by checking the Linear Temporal Logic (LTL) property "*CRASH*". It reveals an action trace that leads to a crash in the current model (*Figure* 11).

According to the action trace in *Figure* 11, that leads to a crash, each vehicle timeouts just after broadcasting the *REQUEST* (broadcast.100) message. This result provides us important insight regarding the duration of the timeout. As per the model of the car in Listing 14, the *timeout* action is available for a car immediately after entering the *WAITING_FOR_REJECT* state. This is just like

Listing 16 Fluents and asserts to test for crashes

```
1 fluent CAR_0_PASSING = <car[0].enter, car[0].exit>
2 fluent CAR_1_PASSING = <car[1].enter, car[1].exit>
3 assert CRASH = []!(CAR_0_PASSING && CAR_1_PASSING)
```

Trace to property violation in CRASH:

start	
car.0.broadcast.100.0	
car.0._____timeout	
car.0.enter	CAR_0_PASSING
car.1.broadcast.100.1	CAR_0_PASSING
car.1._____timeout	CAR_0_PASSING
car.1.enter	CAR_0_PASSING && CAR_1_PASSING

Figure 11. Action trace leading to a crash in the composition of Listing 15. The *CAR* does not have a sense of priority

a timeout duration of 0 seconds. In reality, it takes some time for a *REQUEST* message to propagate to any other car in the queue / core area of the intersection (*Figure* 1) and if such vehicle exits it takes some more time for a *REJECT* message (broadcast.200) to propagate back to the car requesting permission. The model of the car in Listing 14, allows the car to timeout immediately without waiting this round-trip time for a rejection to reach it and move into

Listing 17 Composition with low priority *timeout* actions for testing *CAR* without a sense of priority

```
||INTERSECION =
    (
          car[ i: 0..1 ]:CAR( i )
      || VMEI_NETWORK_FIFO( 0 )
      || VMEI_NETWORK_FIFO( 1 )
    )
      /   {
            start       / car[ 0..1 ].start,

            car[i: 0..1].broadcast[msg: MESSAGES][ i ]
                        / accept_message [ msg ][  i  ],

            car[i: 0..1].receive  [msg: MESSAGES][1-i]
                        / deliver_message[ msg ][ 1-i ]
          }
      >>  {
            car[0..1]._____________________timeout
          }.
```

WAITING_FOR_PERMIT state. This allows the car to prematurely check its high list and see that it is empty and enter the core area of the intersection leading to a crash.

Disabling the car from taking the *timeout* action if there are any other viable action (sending or receiving messages) is one way to avoid this timeout period of 0 seconds. In LTSA, this can be achieved by making the *timeout* a low priority action (Listing 17).

The composition of *INTERSECION* in Listing 17 prevents cars from crashing into each other. However, the danger of a deadlock still prevails. The fact that the *timeout* action is now low priority is evident in the new action trace LTSA provides for a deadlock (*Figure* 12).

At this stage of modeling we can gain three key insights:

1. If the timeout duration is 0, there is the danger of a crash.
2. If the timeout duration is as long as the time necessary to complete all the communication between the two cars, the danger of a crash goes away.
3. Cars could end up in a deadlock.

Trace to DEADLOCK:

start

car.0.broadcast.100.0

car.1.broadcast.100.1

car.0.receive.100.1

car.0.broadcast.200.0

car.1.receive.100.0

car.1.broadcast.200.1

car.0.receive.200.1

car.1.receive.200.0

car.0._____timeout

car.1._____timeout

Figure 12. Deadlock situation in the composition of Listing 17. Now the *timeout* action has low priority. The *CAR* does not have a sense of priority

Although the composition of *INTERSECTION* (Listing 17) solves the issue regarding a crash, the way we have defined the duration for the timeout is rather awkward. This model dictates a car to timeout whenever there is no other action it can take. This means that the timeout period is tightly coupled with all the

other actions a car can perform. Thus, the time has become a dependent variable of message passing actions of a car. This prevents us from quantifying the duration of timeout to different values and experimenting the behavior of the model.

In reality, time flows freely irrespective of whether a message is passed or not. Also, time passes concurrently while messages are in propagation. When it comes to digital computing, we cannot represent the analog, continuous nature of time as it is. Instead, time is represented as discrete clock ticks, which are conveyed by electric pulses. In an actual digital computer, all the actions take place in response to clock pulses. Thus, actions are dependent of clock pulses, the exact opposite of how the current model measures time leading to a timeout.

5.7 Modeling Time

We model time as a separate *CLOCK* process (Listing 18). The clock starts with 0 ticks recorded and for each *tick* action it records one more tick. Since we are modeling finite state processes, we must bound the maximum number of ticks the clock can record. We achieve this by the parameter *TICKS* passed into the *CLOCK* process. This does not limit the behavior of the model because cars only utilize a finite number of ticks to complete the communication. Setting the number of *TICKS* to a value greater than the number of ticks required to complete the communication and reach the timeout is sufficient to experiment the complete behavior of the system.

Now we have to enhance the model of the car to incorporate time (Listing 19). Vehicle ID (*VID*), maximum number of ticks the clock can record (*TICKS*) and the timeout duration (*TO*) are passed into the *CAR* process as parameters.

We augment the *CAR* process to read time from the clock (Lines 6, 11 and 28) to come up with the *CAR_WITH_CLOCK* process. Further,

Listing 18 *CLOCK* process that keeps track of time

```
1  CLOCK( TICKS = 6 ) =
2    (
3      start                                    -> CLOCK[  0  ]
4    ),

5  CLOCK[ t: 0..TICKS ] =
6    (
7        when ( t < TICKS )
8          ____________________tick             -> CLOCK[ t+1 ]

9      |   ____________________read_time[t]  -> CLOCK[  t  ]
10   ).
```

CAR_WITH_CLOCK process advances the global clock with the actions it performs. Each *tick* after a messaging action signals that that action consumes one clock tick. If we want to make an action consume more than one tick (say n ticks), we can have a sequence of n *tick* actions after such action. In any case, these ticks are dependent on the specific actions a *CAR* performs.

The standalone *tick* action in line 23 signifies that the time can flow independent of any action taken by the *CAR_WITH_CLOCK*.

Since we are only interested in investigating the behavior of the system with respect to different timeout durations, we have only augmented the parts of the model that influence the *timeout* action with *tick* actions. This is why we do not update the *CAR* with *tick* actions in states that follow after the expiration of the timeout.

In a real-world implementation, timeouts are handled by operating system interrupts that are captured by signal handlers of a process. In our model, `read_time[ t+TO..t+TICKS ]` actions in lines 11 and 28 act as signal handlers

Listing 19 *CAR_WITH_CLOCK* process that reads time from a global clock but without a sense of priority

```
1  CAR_WITH_CLOCK( VID = 0, TICKS = 6, TO = 5) = DRIVE_IN,

2  DRIVE_IN =
3    (
4            start
5         -> broadcast[ REQUEST ][ VID ]
6         -> ____________________read_time[ t: 0..TICKS ]
7         -> WAITING_FOR_REJECT [ EMPTY  ][ EMPTY  ][ t ]
8    ),

9  WAITING_FOR_REJECT[ lo: LIST_VALS ][ hi: LIST_VALS ]
10                                       [ t: 0..TICKS ] =
11   (
12           ____________________read_time[ t+TO..t+TICKS ]
13        -> ____________________timeout
14        -> WAITING_FOR_PERMIT [   lo   ][   hi   ]

15     |     receive[ REQUEST ][ sender: {[1 - VID]} ]
16        -> ____________________tick
17        -> PROCESS_REJECT     [ sender ][   hi   ][ t ]

18     |     receive[ REJECT  ][ sender: {[1 - VID]} ]
19        -> ____________________tick
20        -> WAITING_FOR_REJECT [   lo   ][ sender ][ t ]

21     |     receive[ PERMIT  ][ sender: {[1 - VID]} ]
22        -> ____________________tick
23        -> WAITING_FOR_REJECT [   lo   ][ EMPTY  ][ t ]

24     |     ____________________tick
25        -> WAITING_FOR_REJECT [   lo   ][ EMPTY  ][ t ]
26   ),
```

```
26 PROCESS_REJECT[lo:LIST_VALS][hi:LIST_VALS][t:0..TICKS]=
27   (
28           ____________________read_time[ t+TO..t+TICKS ]
29       -> ____________________timeout
30       -> WAITING_FOR_PERMIT [   lo   ][   hi   ]


31     |    broadcast[ REJECT ][ VID ]
32       -> ____________________tick
33       -> WAITING_FOR_REJECT [   lo   ][   hi   ][ t ]
34   ),

35 WAITING_FOR_PERMIT[ lo: LIST_VALS ][ hi: LIST_VALS ] =
36   (
37       when ( hi == EMPTY )
38          enter   -> PASSING [   lo   ][   hi   ]

39     |    receive[ REQUEST ][ sender: {[1 - VID]} ]
40       -> broadcast[ REJECT ][ VID ]
41       -> WAITING_FOR_PERMIT [ sender ][   hi   ]

42     |    receive[ PERMIT ][ sender: {[1 - VID]} ]
43       -> WAITING_FOR_PERMIT [   lo   ][ EMPTY  ]
44   ),

45 PASSING[ lo: LIST_VALS ][ hi: LIST_VALS ] =
46   (
47           exit    -> PASSED  [   lo   ]

48     |    receive[ REQUEST ][ sender: {[1 - VID]} ]
49       -> broadcast[ REJECT ][ VID ]
50                  -> PASSING [ sender ][   hi   ]
51   ),
```

```
54 PASSED[ lo: LIST_VALS ] =
55   (
56       when ( lo != EMPTY )
57          broadcast[ PERMIT ][ VID ]
58                   -> DRIVE_AWAY

59     | when ( lo == EMPTY)
60          drive    -> DRIVE_AWAY
61   ),


62 DRIVE_AWAY =
63   (
64          drive     -> DRIVE_AWAY
65   ).
```

for the timeout interrupt. When a car enters the *WAITING_FOR_REJECT* state, it reads the clock and remembers the time (t) it started waiting. The *read_time* actions in lines 11 and 28 are available to a car only after *TO* ticks have passed after the time it entered (t) the *WAITING_FOR_REJECT* state. Since the *timeout* action is available only after taking the `read_time[ t+TO..t+TICKS ]` action, we get the desired effect of a timeout in our model. We repeat the `read_time[ t+TO..t+TICKS ]` action and *timeout* in line 28 to be vigilant about the precise moment a *CAR* reaches its timeout.

5.7.1 Testing the Model. These time sensitive models can be composed for testing as in Listing 20.

When composing these new models, we must provide suitable values for the two parameters *TICKS* and *TO* constrained to the relationship: $TO \leq TICKS$. While we can choose any value for *TICKS*, on one hand choosing a value unnecessarily larger than required bloats the state space of the model slowing down

Listing 20 Composition for testing *CAR_WITH_CLOCK* without a sense of priority

```
1  ||INTERSECION( TICKS=10, TO=6 ) =
2      (
3           car[ i: 0..1 ]:CAR_WITH_CLOCK( i, TICKS, TO )
4        || VMEI_NETWORK_FIFO( 0 )
5        || VMEI_NETWORK_FIFO( 1 )
6        || car[i:0..1]::CLOCK( TICKS )
7      )
8        /   {
9             start       / car[ 0..1 ].start,
10            car[i: 0..1].broadcast[msg: MESSAGES][ i ]
11                        / accept_message [ msg ][  i  ],
12            car[i: 0..1].receive  [msg: MESSAGES][1-i]
13                        / deliver_message[ msg ][ 1-i ]
14           }.
```

the composition process. On the other hand, if we choose a very small value, it prevents from experimenting with all the interesting situations.

We use the time to complete all the communication between the two cars (*TTCC* - "*Time To Complete Communication*") as the guiding principle for choosing a good value for the parameter *TICKS*. Since we have associated a single *tick* action with each message passing actions in the *CAR_WITH_CLOCK* process (Listing 19), each car requires 3 ticks to complete all the communication. As we have discussed in Section 5.7, this is a choice we have made to keep the model size small. This does not alter the behavior of the model. With two cars at the intersection, we have $3 \leq TTCC \leq 6$. So, choosing *TICKS* to be a little larger than 6 enables us to keep the model small and pick different values for *TO* to experiment with all the interesting cases.

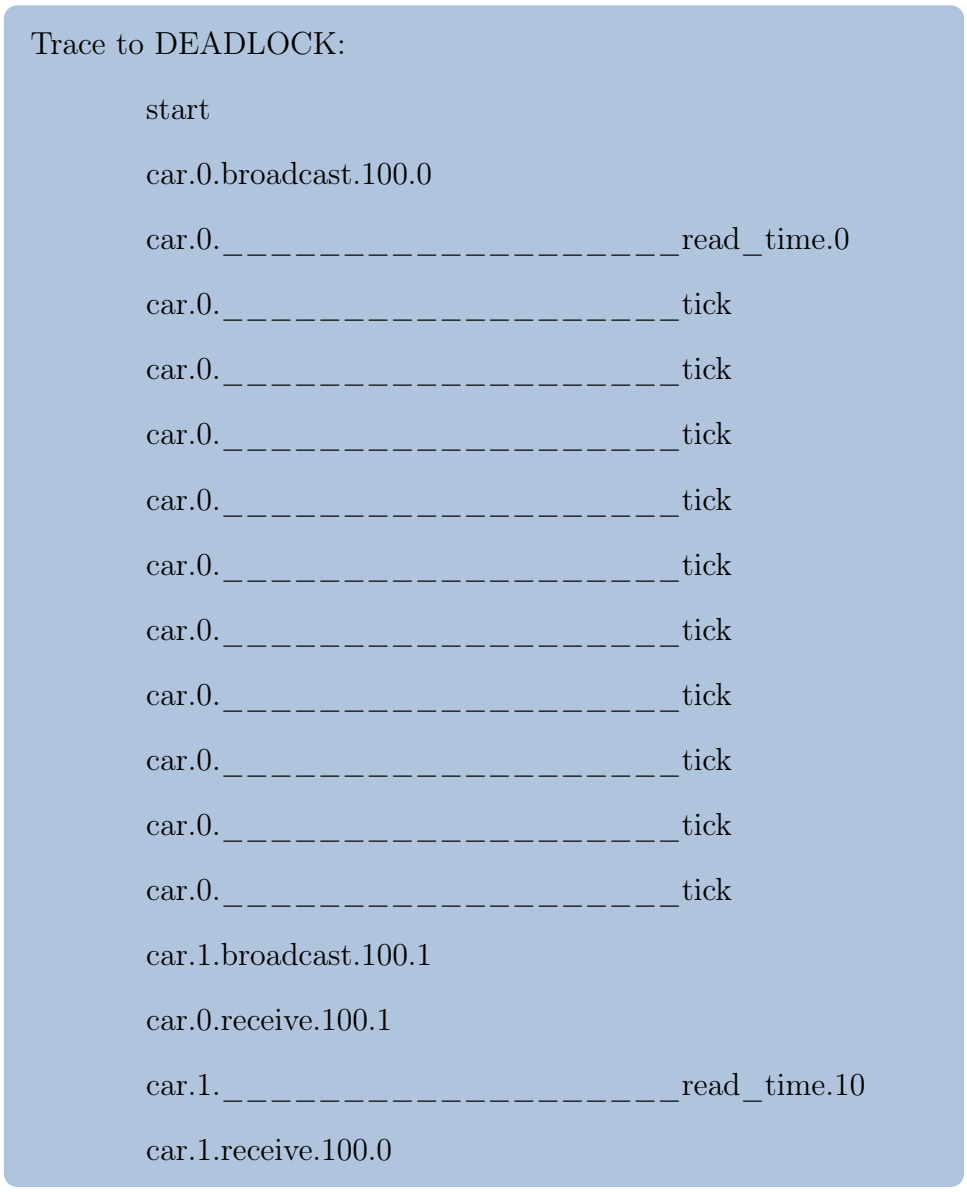

Figure 13. Deadlock situation in the composition of Listing 20. *CAR_WITH_CLOCK* reads and advances time in a globally shared *CLOCK* process with *read_time* and *tick* actions respectively.

When tested for safety, composition in Listing 20 provides the action trace in *Figure* 13 leading to a deadlock. According to this action trace, car 0 has been just passing time in the queue area of the intersection without doing any productive work. This is not the behavior we expect from a car that is trying to pass through

an intersection. The model reaches this deadlock state because the *tick* action is freely available to a *CAR_WITH_CLOCK* in the *WAITING_FOR_REJECT* state and the *CAR_WITH_CLOCK* repeatedly keeps on taking this action, disregarding other possible actions, until the *CLOCK* runs out of ticks.

Just like we make *timeout* action low priority in Listing 17, our first hunch is to make *tick* a low priority action and it will solve this problem. However, that introduces other anomalies to the model. In LTSA, when an action is made low priority, that action is removed wherever there are other actions available. This causes the *tick* action to be removed in places where we do not desire the *tick* action to be eliminated.

What we really want is an action in the *CAR_WITH_CLOCK* process to advance the clock independent of messaging actions and that action to have a low priority. Therefore, we replace the standalone *tick* action in *CAR_WITH_CLOCK* process (Line 23 of Listing 19) with a different action, *tick_idle*, which has the same effect as the *tick* action (Listing 21). We call this new process *CAR_TICK_IDLE*. Please note that in the interest of saving space, we only show the segment of the *CAR_WITH_CLOCK* process that is updated.

To enable the *CAR_TICK_IDLE* process, which has the *tick_idle* action, to update the *CLOCK* process (Listing 18), we should update the *CLOCK* process accordingly (Listing 22).

When composing the *CAR_TICK_IDLE* process (Listing 21) with *CLOCK_TICK_IDLE* process (Listing 22) and *VMEI_NETWORK_FIFO* process (Listing 9), in addition to making the *tick_idle* action low priority, we have to make some other actions high priority.

Listing 21 Portion of the *CAR_WITH_CLOCK* process of Listing 19 updated with the *tick_idle* action

```
9  CAR_TICK_IDLE( VID = 0, TICKS = 6, TO = 5 ) = DRIVE_IN,

10 DRIVE_IN =
11   (
12           start
13       -> broadcast[ REQUEST ][ VID ]
14       -> ____________________read_time[ t: 0..TICKS ]
15       -> WAITING_FOR_REJECT [ EMPTY  ][ EMPTY  ][ t ]
16   ),

17 WAITING_FOR_REJECT[ lo: LIST_VALS ][ hi: LIST_VALS ]
18                                       [ t: 0..TICKS ] =
19   (
20           ____________________read_time[ t+TO..t+TICKS ]
21       -> ____________________timeout
22       -> WAITING_FOR_PERMIT [   lo   ][   hi   ]

23     |     receive[ REQUEST ][ sender: {[1 - VID]} ]
24       -> ____________________tick
25       -> PROCESS_REJECT      [ sender ][   hi   ][ t ]

26     |     receive[ REJECT  ][ sender: {[1 - VID]} ]
27       -> ____________________tick
28       -> WAITING_FOR_REJECT [   lo   ][ sender ][ t ]

29     |     receive[ PERMIT  ][ sender: {[1 - VID]} ]
30       -> ____________________tick
31       -> WAITING_FOR_REJECT [   lo   ][ EMPTY  ][ t ]

32     |     ____________________tick_idle
33       -> WAITING_FOR_REJECT [   lo   ][   hi   ][ t ]
34   ),

35   // Rest of the process is the same as the CAR process
```

Listing 22 CLOCK process with the *tick_idle* action

```
1  CLOCK_TICK_IDLE( TICKS = 6 ) =
2    (
3      start                                  -> CLOCK[  0  ]
4    ),

5  CLOCK[ t: 0..TICKS ] =
6    (
7        when ( t < TICKS )
8          ____________________tick           -> CLOCK[ t+1 ]

9      | when ( t < TICKS )
10         ____________________tick_idle      -> CLOCK[ t+1 ]

11     |   ____________________read_time[t]  -> CLOCK[  t  ]
12   ).
```

With the *tick_idle* action in place, the sole purpose of the *tick* action is to take the time the *CAR_TICK_IDLE* process spends sending and receiving messages. So, it is better to try to force the model to take the *tick* action immediately after a messaging action. In other words, it is better to tightly couple each messaging action with its respective *tick* action that follows making it to be atomic. To achieve this we make the *tick* action high priority.

In a real-world implementation, operating system interrupts take precedence over the other actions a process is performing. In this light, we do not want the *CAR_TICK_IDLE* process to keep on performing other actions whenever the *timeout* action is available. We can make our model behave this way by making the *read_time* action of lines 11 and 28 of Listing 19, which is the signal handler in our model, high priority. As a byproduct, this makes the *read_time* action in line 6 high priority as well. This is where a car reads and notes the time when it enters

the queue area of the intersection. It becoming high priority forces a car to read and note the time as soon as it broadcasts the *REQUEST* message making these two actions atomic. This too is the logical thing for a car to do and hence it does not add any artificial behavior to our final model.

When *read_time* action in lines 11 and 28 of Listing 19 is available to a *CAR_TICK_IDLE* process, that means the signal handler for the timeout is called. In that case, we need the process to immediately take the *timeout* action. By making the *timeout* action high priority, we can achieve this behavior.

The prime motivation of each car is to pass through the intersection as quickly as possible. To achieve this, we make *enter* action high priority so that as soon as the conditions for a car to enter the core area of the intersection is met, *enter* action is taken.

With all the considerations in place, the final composition is presented in Listing 23. This composition is robust enough for us to experiment with different timeout durations.

5.7.2 Ideal Duration for the Timeout. In the final model we compose (Listing 23), the time to complete communication ($TTCC$) is 6 ticks. We test the model with varying values for the timeout (TO) duration. This covers three cases: $TO < TTCC$, $TO = TTCC$ and $TO > TTCC$.

When $TO \geq TTCC$, the system ends up in a deadlock. Since cars deadlock prior to entering the core area of the intersection, there is no danger of a crash. *Figure* 14 shows the action trace leading to a deadlock when $TO = TTCC$. We present the action trace leading to a deadlock when $TO = 9$, which represents the $TO > TTCC$ case, in *Figure* 15. The only difference between these two deadlock traces is that the trace for $TO = 9$ contains 3 *tick_idle* actions. During these

Listing 23 Composition for testing *CAR_ TICK_ IDLE* without a sense of priority

```
1  ||INTERSECION( TICKS=10, TO=6 ) =
2      (
3           car[ i: 0..1 ]:CAR_TICK_IDLE( i, TICKS, TO )
4        || VMEI_NETWORK_FIFO( 0 )
5        || VMEI_NETWORK_FIFO( 1 )
6        || car[ i: 0..1 ]::CLOCK_TICK_IDLE( TICKS )
7      )
8        /   {
9             start       / car[ 0..1 ].start,
10
11            car[i: 0..1].broadcast[msg: MESSAGES][ i ]
12                        / accept_message [ msg ][  i  ],
13
14            car[i: 0..1].receive  [msg: MESSAGES][1-i]
15                        / deliver_message[ msg ][ 1-i ]
16          }
17       <<  {
18            car[0..1].enter,
19            car[0..1]._____________________tick,
20            car[0..1]._____________________timeout,
21            car[0..1]._____________________read_time
22                                              [0..TICKS]
23          }.
24
25 ||INTERSECION_TICK_IDLE( TICKS=10, TO=6 ) =
26                                 INTERSECION( TICKS, TO )
27       >>  {
28            car[0..1]._____________________tick_idle
29          }.
```

Trace to DEADLOCK:

start

car.0.broadcast.100.0

car.0._____read_time.0

car.1.broadcast.100.1

car.1._____read_time.0

car.0.receive.100.1

car.0._____tick

car.0.broadcast.200.0

car.0._____tick

car.1.receive.100.0

car.1._____tick

car.1.broadcast.200.1

car.1._____tick

car.0.receive.200.1

car.0._____tick

car.1.receive.200.0

car.1._____tick

car.0._____read_time.6

car.1._____read_time.6

car.0._____timeout

car.1._____timeout

Figure 14. Deadlock situation in the composition of Listing 23 when timeout duration is equal to the time to complete communication ($TO = TOCC$).

Trace to DEADLOCK:

start

car.0.broadcast.100.0

car.0._____read_time.0

car.1.broadcast.100.1

car.1._____read_time.0

car.0.receive.100.1

car.0._____tick

car.0.broadcast.200.0

car.0._____tick

car.1.receive.100.0

car.1._____tick

car.1.broadcast.200.1

car.1._____tick

car.0.receive.200.1

car.0._____tick

car.1.receive.200.0

car.1._____tick

car.0._____tick_idle

car.0._____tick_idle

car.0._____tick_idle

car.0._____read_time.9

car.1._____read_time.9

car.0._____timeout

car.1._____timeout

Figure 15. Deadlock situation in the composition of Listing 23 when timeout duration is greater than the time to complete communication ($TO > TOCC$). Here we use $TO = 9$.

3 ticks, cars were waiting at the intersection just passing time till the timeout is reached. Therefore, setting timeout higher than necessary negatively affects the throughput at the intersection.

Setting $TO < TTCC$, leads to the danger of cars crashing into each other while passing through the core area of the intersection. Here, we present the action trace for a *CRASH* when $TO = 2$ (*Figure* 16). In this situation, each car timeouts prior to getting to know about the other car. Hence, each car thinks that it is the only car waiting at the intersection and that causes the crash.

We observed that when $TO = 5$, the model works without any deadlocks or crashes. However, this is just a case of being lucky. According to how we are timing the actions of a car, with 5 ticks there is not enough time for car 1 to receive the *REJECT* message sent by car 0 (or vice versa). This leads to the situation where car 0 is aware of car 1's rejection but car 1 is not. Thus, car 0 ends up waiting for car 1 to pass through first making the system work nicely just out of luck. Thus, $TO = 5$ is not the ideal timeout duration for the system to work smoothly.

With this modeling exercise, we gained a considerable amount of insight into the behavior of the distributed mutual exclusion algorithm for intersection traffic control algorithm (Wu et al., 2015a) with respect to the value for the duration of the timeout:

$TO < TTCC$: When the duration of the timeout is less than time to complete communication, there is the danger of a crash.

$TO \geq TTCC$: When the duration of the timeout is greater than or equal to the time to complete communication, the algorithm deadlocks.

Trace to property violation in CRASH:

```
start
car.0.broadcast.100.0
car.0._____________________read_time.0
car.1.broadcast.100.1
car.1._____________________read_time.0
car.0.receive.100.1
car.0._____________________tick
car.0.broadcast.200.0
car.0._____________________tick
car.0._____________________read_time.2
car.1._____________________read_time.2
car.0._____________________timeout
car.0.enter                                  CAR_0_PASSING
car.1.broadcast.100.1                        CAR_0_PASSING
car.1._____________________timeout           CAR_0_PASSING
car.1.enter                                  CAR_0_PASSING &&
                                                   CAR_1_PASSING
```

Figure 16. Action trace leading to a crash in the composition of Listing 23 when timeout duration is less than the time to complete communication ($TO < TOCC$). Here we use $TO = 2$.

$TO >> TTCC$: Setting the duration of the timeout greater than necessary reduces the throughput at the intersection.

In conclusion, setting and tuning the duration of the timeout is a serious business when it comes to implementing this algorithm for real. It is a trade-off between avoiding crashes vs. the throughput. A timeout duration less than the time to complete communication is not acceptable at any time, since it can lead to crashes.

Although our model is set to have a time to complete communication to be 6 ticks, in a real system it is dependent on many parameters such as, how reliable the network is, how congested the network is and the distance between vehicles that communicate. When it comes to a real-world implementation, we need good answers to questions regarding the time to complete communication like:

1. Is it the same for all the intersections?
2. Is it independent of the number of vehicles waiting at the intersection?
3. Is it the same for all the types of vehicles?
4. Is it the same for different weather conditions?

If the answer to any of these questions is "No", we need a way to adapt the timeout duration according to the situation. Finding a universal upper bound for the time to complete communication and setting the timeout duration higher than that could solve the problem of a crash. But it would result a very inefficient transportation system.

5.8 Global Clock vs. Local Clock

Line 6 of the composition in Listing 23 models the *CLOCK_TICK_IDLE* to be a global resource shared by both the cars:

```
6 car[ i: 0..1 ]::CLOCK_TICK_IDLE( TICKS )
```

In LTSA, " :: " signifies this. So, a *tick* or *tick_ idle* action advances the clock of both the cars despite which car executed that action. For example, in *Figure* 15, the 3 *tick_ idle* actions taken by car 0 mean both the cars were idle.

Due to this choice, in the model, none of the actions by the two cars happen concurrently within the same tick. In other words, within each tick only a single action by any car takes place.

Making the clock local to each car is quite straightforward and it is just the difference of a semicolon (:). We only need to change the " :: " in line 6 of the Listing 23 to " : ".

```
6 car[ i: 0..1 ]:CLOCK_TICK_IDLE( TICKS )
```

With a local clock, a tick taken by one car does not affect the clock of the other car. This reduces the time to complete communication ($TTCC$) to 3. Irrespective of this change, the insights we gain with a global clock remains unchanged with a local clock as well. We only include the action trace for the deadlock with $TTCC = 4$ in this case to show the behavior of the model (*Figure* 17). Note that in this action trace both car 0 and car 1 take independent *tick_ idle* actions.

As opposed to the version of the model with a global clock, when we make the clock local, the simulation is like all the actions are happening in parallel. For each clock tick both the cars perform some actions. Although the action trace in *Figure* 17 contains two *tick_ idle* actions, both these ticks resemble a single tick in a global clock. Another thing to note in this model is that despite the clocks are local to cars, they are perfectly synchronized.

Trace to DEADLOCK:

start

car.0.broadcast.100.0

car.0._____read_time.0

car.1.broadcast.100.1

car.1._____read_time.0

car.0.receive.100.1

car.0._____tick

car.0.broadcast.200.0

car.0._____tick

car.1.receive.100.0

car.1._____tick

car.1.broadcast.200.1

car.1._____tick

car.0.receive.200.1

car.0._____tick

car.1.receive.200.0

car.1._____tick

car.0._____tick_idle

car.0._____read_time.4

car.0._____timeout

car.1._____tick_idle

car.1._____read_time.4

car.1._____timeout

Figure 17. Deadlock situation in Listing 23 after making the CLOCK_TICK_IDLE local to each car. With a local clock, time to complete communication ($TTCC$) becomes 3. Here we use $TO = 4$.

With these we have covered the two extremes of all the actions being sequential and all the actions being concurrent. The algorithm has issues in either case. In reality, we get a mix of both. However, during this study we did not embark our efforts in modeling such a model. Another aspect we did not model is out of synchronized local clocks.

5.9 Using Arrival Time to Determine Priority

In several occasions, Wu et al. (2015a) discuss using arrival time to determine the priority of a vehicle to pass through the intersection (Chapter II - section 2.3.2). In this section, we bring arrival time into our models and experiment the caveats of this choice.

For the algorithm to use arrival time to decide which car should pass through the intersection first, each car should notify its arrival time to the other car. Therefore, we have to augment the *REQUEST* message with the arrival time of the car (Listing 24).

In addition to changing the *REQUEST* message, we have to change the way a car responses to a received *REQUEST* message. Instead of responding with a *REJECT* message to each *REQUEST* message received, now a car V sends a *REJECT* message only if the arrival time of the car U that sent the *REQUEST* message is greater than that of car V (lines 16 - 18 and 42 - 44 of Listing 24).

Since the current network process (Listing 9) is not designed to handle the updated *REQUEST* message format, we updated it accordingly (Listing 25).

CAR_P_TIME (Listing 24) and *VMEI_NETWORK_FIFO_TIME* (Listing 25) processes that can handle the new *REQUEST* message can be composed together with *CLOCK_TICK_IDLE* (Listing 22) to model the behavior at the intersection as in Listing 26.

Listing 24 *CAR_P_TIME* process that uses arrival time as priority to decide who should pass through the intersection first.

```
1 CAR_P_TIME( VID = 0, TICKS = 6, TO = 5 ) = DRIVE_IN,

2 DRIVE_IN =
3   (
4           start
5       -> ____________________read_time[ t_mine: 0..TICKS ]
6       -> broadcast[ REQUEST ][ VID ][ t_mine ]
7                             -> WAITING_FOR_REJECT[ EMPTY  ][ EMPTY  ][ t_mine ]
8   ),
```

```
9  WAITING_FOR_REJECT[ lo: LIST_VALS ][ hi: LIST_VALS ][ t_mine: 0..TICKS ] =
10   (
11           ____________________read_time[ t_mine + TO .. t_mine + TICKS ]
12       -> ____________________timeout
13                              -> WAITING_FOR_PERMIT[   lo   ][   hi   ][ t_mine ]

14     |     receive[ REQUEST ][ sender: {[1 - VID]} ][ t_other: 0..TICKS ]
15       -> ____________________tick
16       -> if ( t_other > t_mine )
17          then                  PROCESS_REJECT_1  [ sender ][   hi   ][ t_mine ]
18          else                  WAITING_FOR_REJECT[   lo   ][   hi   ][ t_mine ]

19     |     receive[ REJECT  ][ sender: {[1 - VID]} ]
20       -> ____________________tick
21                              -> WAITING_FOR_REJECT[   lo   ][ sender ][ t_mine ]

22     |     receive[ PERMIT  ][ sender: {[1 - VID]} ]
23       -> ____________________tick
24                              -> WAITING_FOR_REJECT[   lo   ][ EMPTY  ][ t_mine ]

25     |     ____________________tick_idle
26                              -> WAITING_FOR_REJECT[   lo   ][   hi   ][ t_mine ]
27   ),
```

```
28 PROCESS_REJECT_1[ lo: LIST_VALS ][ hi: LIST_VALS ][ t_mine: 0..TICKS ] =
29   (
30           ____________________read_time[ t_mine + TO .. t_mine + TICKS ]
31       -> ____________________timeout
32                               -> WAITING_FOR_PERMIT[   lo   ][   hi   ][ t_mine ]


33     |     broadcast[ REJECT ][ VID ]
34       -> ____________________tick
35                               -> WAITING_FOR_REJECT[   lo   ][   hi   ][ t_mine ]
36   ),

37 WAITING_FOR_PERMIT[ lo: LIST_VALS ][ hi: LIST_VALS ][ t_mine: 0..TICKS ] =
38   (
39       when ( hi == EMPTY )
40           enter                          -> PASSING [   lo   ][   hi   ]

41     |     receive[ REQUEST ][ sender: {[1 - VID]} ][t_other: 0..TICKS]
42       -> if ( t_other > t_mine )
43           then                   PROCESS_REJECT_2  [ sender ][   hi   ][ t_mine ]
44           else                   WAITING_FOR_PERMIT[   lo   ][   hi   ][ t_mine ]

45     |     receive[ PERMIT ][ sender: {[1 - VID]} ]
46                               -> WAITING_FOR_PERMIT[   lo   ][ EMPTY  ][ t_mine ]
47   ),
```

```
50 PROCESS_REJECT_2[ lo: LIST_VALS ][ hi: LIST_VALS ][ t_mine: 0..TICKS ] =
51   (
52         broadcast[ REJECT ][ VID ]
53                              -> WAITING_FOR_PERMIT[   lo   ][   hi   ][ t_mine ]
54   ),

55 PASSING[ lo: LIST_VALS ][ hi: LIST_VALS ] =
56   (
57         exit                             -> PASSED  [   lo   ]

58     |   receive[ REQUEST ][ sender: {[1 - VID]} ][ ot: 0..TICKS ]
59       -> broadcast[ REJECT ][ VID ]
60                                          -> PASSING [ sender ][   hi   ]
61   ),

62 PASSED[ lo: LIST_VALS ] =
63   (
64       when ( lo != EMPTY )
65         broadcast[ PERMIT ][ VID ]
66                                          -> DRIVE_AWAY

67     | when ( lo == EMPTY)
68         drive                            -> DRIVE_AWAY
69   ),

70 DRIVE_AWAY =
71   (
72         drive                            -> DRIVE_AWAY
73   ).
```

Listing 25 Updated network process that provides a FIFO channel that can handle updated *REQUEST* message that includes the arrival time.

```
VMEI_NETWORK_FIFO_TIME( VID = 0, TICKS = 6 ) = QUEUE,

QUEUE =
  (
      accept_message [ REQUEST ][ VID ][ t: 0..TICKS ]
          -> QUEUE[ REQUEST ][ t ]

    | accept_message [ REJECT  ][ VID ]
          -> QUEUE[ REJECT  ]

    | accept_message [ PERMIT  ][ VID ]
          -> QUEUE[ PERMIT  ]
  ),

QUEUE[ REQUEST ][ t: 0..TICKS ] =
  (
      accept_message [ REJECT  ][ VID ]
          -> QUEUE[ REQUEST ][ t ][ REJECT ]

    | accept_message [ PERMIT  ][ VID ]
          -> QUEUE[ REQUEST ][ t ][ PERMIT ]

    | deliver_message[ REQUEST ][ VID ][ t ]  -> QUEUE
  ),

QUEUE[ REJECT ] =
  (
      accept_message [ PERMIT  ][ VID ]
          -> QUEUE[ REJECT  ][ PERMIT ]

    | deliver_message[ REJECT  ][ VID ]       -> QUEUE
  ),

QUEUE[ PERMIT ] =
  (
      deliver_message[ PERMIT  ][ VID ]       -> QUEUE
  ),
```

```
29 QUEUE[ REQUEST ][ t: 0..TICKS ][ REJECT ] =
30   (
31       accept_message [ PERMIT  ][ VID ]
32           -> QUEUE[ REQUEST ][ t ][ REJECT ][ PERMIT ]

33     | deliver_message[ REQUEST ][ VID ][ t ]
34           -> QUEUE[ REJECT  ]
35   ),

36 QUEUE[ REQUEST ][ t: 0..TICKS ][ PERMIT ] =
37   (
38       deliver_message[ REQUEST ][ VID ][ t ]
39           -> QUEUE[ PERMIT  ]
40   ),

41 QUEUE[ REJECT ][ PERMIT ] =
42   (
43       deliver_message[ REJECT  ][ VID ]
44           -> QUEUE[ PERMIT  ]
45   ),

46 QUEUE[ REQUEST ][ t: 0..TICKS ][ REJECT ][ PERMIT ] =
47   (
48       deliver_message[ REQUEST ][ VID ][ t ]
49           -> QUEUE[ REJECT  ][ PERMIT ]
50   ).
```

Listing 26 Composition for testing *CAR_P_TIME* that uses arrival time to prioritize who passes through first

```
1  ||INTERSECION( TICKS=10, TO=6 ) =
2      (
3            car[ i: 0..1 ]:CAR_P_TIME( i, TICKS, TO )
4        || VMEI_NETWORK_FIFO_TIME( 0, TICKS )
5        || VMEI_NETWORK_FIFO_TIME( 1, TICKS )
6        || car[ i: 0..1 ]::CLOCK_TICK_IDLE( TICKS )
7      )
8        /   {
9              start       / car[ 0..1 ].start,
10             car[ i: 0..1 ].broadcast[ REQUEST ][ i ][ t: 0..TICKS ]
11                                       / accept_message[ REQUEST ] [i ] [t ],
12             car[ i: 0..1 ].broadcast[ msg: {[REJECT], [PERMIT]} ][ i ]
13                                       / accept_message[ msg ][ i ],
14             car[ i: 0..1 ].receive[ REQUEST ][ 1-i ][ t: 0..TICKS ]
15                                       / deliver_message[ REQUEST ][ 1-i ][ t ],
16             car[ i: 0..1 ].receive[ msg: {[REJECT], [PERMIT]} ][ 1-i ]
17                                       / deliver_message[ msg ][ 1-i ]
18           }
19       <<  {
20             car[0..1].enter,
21             car[0..1]._____________________tick,
22             car[0..1]._____________________timeout,
23             car[0..1]._____________________read_time[0..TICKS]
24           }.
```

```
25 ||INTERSECION_P_TIME( TICKS=10, TO=6 ) =
26                                   INTERSECION( TICKS, TO )
27       >>  {
28             car[0..1].____________________tick_idle
29           }.
```

We test the model that uses arrival time to prioritize cars while varying the timeout duration. It does not suffer from any deadlocks. However, when the arrival times of cars are the same there is the danger of a crash for every timeout duration (*Figure* 18).

When cars are using arrival time as the priority, a car sends a *REJECT* message in response to a *REQUEST* message only if the car that is requesting permission arrived later than the car that is giving or denying permission. Consequently, when both cars arrive at the same time, neither of the cars reject the other car. This allows both the cars to enter the intersection creating a crash situation.

In Listing 24, if we change lines 16 and 42 from:

```
-> if ( t_other > t_mine )
```

to:

```
-> if (t_other >= t_mine )
```

, when cars arrive at the same time, each car would end up rejecting the other. A composition with a car (say V) modeled to prevent any other car (say U) that arrives at the same time or later than V, ends up in a deadlock when both cars arrive at the same time and the timeout duration is set to greater than or equal to

Trace to property violation in CRASH:

```
start
car.0._____________________read_time.0
car.1._____________________read_time.0
car.0.broadcast.100.0.0
car.1.broadcast.100.1.0
car.0.receive.100.1.0
car.0._____________________tick
car.1.receive.100.0.0
car.1._____________________tick
car.0._____________________tick_idle
car.0._____________________tick_idle
car.0._____________________tick_idle
car.0._____________________tick_idle
car.0._____________________read_time.6
car.1._____________________read_time.6
car.0._____________________timeout
car.0.enter                                  CAR_0_PASSING
car.1._____________________timeout           CAR_0_PASSING
car.1.enter                                  CAR_0_PASSING &&
                                                   CAR_1_PASSING
```

Figure 18. Action trace leading to a crash in the composition of Listing 26 when cars use arrival time to prioritize who passes through the intersection first. Here we use $TO = 6$.

the time to complete communication. In fact, when cars arrive at the same time, such a model behaves exactly the same as a model that does not use any notion of priority (Composition in Listing 23).

The lesson learned after these set of experiments is that the arrival time alone is not sufficient to prioritize which car should pass through the intersection first.

Since a car that arrives first passing through the intersection is the fair thing to do, we should use the arrival time in the formula to decide the priority. This way, we can maintain a first-come-first-served nature at an intersection. However, when both cars arrive at the same time, we face the dilemma of deciding who came first. Therefore, the algorithm fails when both the cars arrive at the same time. In this situation, we need some tie-breaker to give precedence to one car over the other.

5.10 Breaking the Tie

One quality of the attribute (or attributes) we choose to break the tie is that whenever the arrival times of the cars are the same, the value of the attribute used to break the tie should be unique for each car. Since arrival time gets precedence over the attribute used to break the tie, when arrival times are different (there is no tie), we do not need to worry about the value of the tie-breaker.

For example, in the situation where two human drivers arrive at a 4-way intersection at the same time from two conflicting lanes, the driver on the right gets the right of the way to pass though. In this case, the relative position of cars is being used as the tie-breaker.

The messages exchanged between cars in the current model already contain the vehicle id (VID). Further, it is unique to each car. Because of this, we can easily use VID as the tie-breaker and run experiments, without changing the models a lot.

The only minor change required is to the model of the *CAR_P_TIME* (Listing 24). Yet again, the change is to lines 16 and 42, which models the logic for processing an incoming *REQUEST* message and makes the decision - to reject or not to reject. We have to change lines 16 and 42 from:

```
-> if ( t_other > t_mine )
```

to:

```
-> if ( (t_other > t_mine ) ||
                ( t_other == t_mine && sender > VID ) )
```

Without loss of generality, we have given higher priority to the vehicle with the lower *VID*'s, whenever their arrival times are the same. In this experiment, our only intention is to demonstrate how a tie-breaker can aid remove the problems of the algorithm we have uncovered so far but not to decide what the best tie-breaker is.

With this change, when the duration for the timeout (TO) is set greater than or equal to the time to complete communication ($TTCC$), the algorithm works fine for two cars free of deadlocks and crashes. Whenever two cars arrive at the intersection at the same time, car 0, which has the lower *VID* gets to pass through before car 1. In this case, car 1 does not broadcast a *REJECT* message and hence car 0 does not receive a *REJECT* message. This takes two ticks off the $TTCC$, reducing it to 4 from the earlier value of 6 we had. In other cases, where cars arrive at different times, the car that comes late does not broadcast a *REJECT* message resulting in the car that came earlier not receiving such message keeping $TTCC$ to 4 ticks.

Consider the architecture of the intersection we are analyzing (*Figure* 1). Also, consider the situation where both car 0 and car 1 are arriving at the intersection at the same time.

In the case of human drivers and the driver on the right gets the right of the way rule, if car 0 arrives in lane 0 and car 1 arrives in lane 2, then car 0 gets to pass through the intersection first. Now if car 1 arrives in lane 0 and car 0 arrives in lane 2, car 1 gets to go first.

However, in the case of the algorithm with *VID* as the tie-breaker, car 0 always gets to pass through first irrespective of the lane it is coming from. This is a subtle fairness issue. Thus, the parameters we use to break the tie between cars that arrive at the same time dictate how fair the algorithm ends up. In this light, using something as *VID* which is tightly coupled with each vehicle makes the algorithm unfair. Looking for attributes that a car can dynamically inherit based on its context, such as the lane it is approaching the intersection or the direction it is traveling (e.g. north-south or east-west) would lead to a fairer algorithm.

5.10.1 A Better Tie-Breaker - A Suggestion. We suggest combining two attributes to come up with a better tie-breaker:

1. Out of the two cross-roads, designate one as the main road (say lanes 2, 3, 6, 7 of *Figure* 1) and the other as the sub road (lanes 0, 1, 4, 5). Vehicles on the main road get high priority.

2. Vehicles moving straight through the intersection get high priority.

Depending on which rule of the tie-breaker gets precedence, we get two distinct traffic patterns at the intersection. To implement this

mechanism, the *REQUEST(vid, lid)* message should be enhanced to: $REQUEST(vid, time, lid_{in}, lid_{out})$:

vid: Vehicle id.

time: Time when the vehicle enters the queue area of the intersection.

lid_{in}: Lane id where the vehicle is entering the intersection.

lid_{out}: Lane id where the vehicle will be exiting the intersection.

By inspecting lid_{in}, we can determine whether a vehicle is approaching the intersection from the main road or the sub road. The pair lid_{in} and lid_{out} serves as a turning signal. If $lid_{in} = lid_{out}$ then the vehicle is going straight. If $lid_{in} \neq lid_{out}$ then it is turning.

The function *iHiPriority* (Algorithm 3) describes how to use these attributes to decide who has the priority. It returns true if and only if the caller of the function (the vehicle that is trying to decide on the priority - referred to as I) has higher priority than the vehicle that is requesting permission (referred to as you). This function gives higher precedence to the rule "*vehicles on the main road (lanes 2, 3, 6, 7) have higher priority than vehicles on the sub road*" (lines 8 - 13) over the rule "*vehicles going straight have higher priority than vehicles that are turning*" (lines 14 - 17). The precedence of these two rules can be inverted in the Algorithm 3 by checking for turns before checking whether the vehicles are on the main road or not.

5.11 A Final Remark

With this final set of experiments, we have demonstrated that using a tie-breaker in addition to arrival time to decide who should pass first makes the

Algorithm 3 Suggested algorithm to decide priority.

▷ $iTime$ — Time I entered the queue area
▷ $uTime$ — Time you entered the queue area
▷ $iLid_{in}$ — Lane I am approaching from
▷ $uLid_{in}$ — Lane you are approaching from
▷ $iLid_{out}$ — Lane I am exiting to
▷ $uLid_{out}$ — Lane you are exiting to

▷ Returns ***True*** if and only if I (the caller of the function) have higher priority than you (the vehicle requesting permission)

1: **function** IHIPRIORITY($iTime$, $uTime$, $iLid_{in}$, $iLid_{out}$, $uLid_{in}$, $uLid_{out}$)

2: **if** $iTime < uTime$ **then**
3: **return** ***True*** ▷ I came earlier
4: **end if**

5: **if** $iTime > uTime$ **then**
6: **return** ***False*** ▷ You came earlier
7: **end if**
▷ We arrived at the same time

8: **if** $(iLid_{in} \in \{2, 3, 6, 7\})$ **and** $(uLid_{in} \notin \{2, 3, 6, 7\})$ **then**
9: **return** ***True*** ▷ I am in the main road, you are not
10: **end if**

11: **if** $(iLid_{in} \notin \{2, 3, 6, 7\})$ **and** $(uLid_{in} \in \{2, 3, 6, 7\})$ **then**
12: **return** ***False*** ▷ You are in the main road, I am not
13: **end if**
▷ Arrived at the same time on the same road

14: **if** $(iLid_{in} = iLid_{out})$ **and** $(uLid_{in} \neq uLid_{out})$ **then**
15: **return** ***True*** ▷ I am going straight, you are not
16: **end if**

17: **return** ***False***

18: **end function**

algorithm operate smoothly when there are two cars at the intersection. This does

not automatically certify that the algorithm works flawlessly with more than two cars at the intersection.

For example, in the 4-way intersection with human drivers, "*the driver on the right gets the right of the way*" rule works perfectly as long as less than 4 cars (say n and $n < 4$) simultaneously approach the intersection from n directions. However, if 4 cars arrive simultaneously from 4 directions, this rule would complete a wait cycle, which is a deadlock. Although such a situation is highly improbable, it is not impossible for sure. A group of civilized human drivers will be able to understand the situation, use their common sense, collaborate, improvise and resolve such a situation easily. Unfortunately, the drivers of self-driving cars, computers, not yet have invented common sense. Therefore, we need to stress test the algorithm with intersections with larger number of cars to give it the certificate of approval for real use. Making it work for two cars is just the first step.

CHAPTER VI

CONVOY CRASH

One goal of designing a distributed protocol is to keep the number of messages exchanged to a minimum. This leads to better network resource utilization.

In designing the distributed mutual exclusion algorithm for intersection traffic control (Algorithm 2), the authors try to reduce the number of messages by leveraging how vehicles behave at an intersection (Wu et al., 2015a). This optimization is discussed in detail in Chapter II - Section 2.4.2.5. We just present the gist of the idea behind the optimization here to give the reader some context.

When the vehicle at the head of a lane gets the chance to pass through the intersection, it allows vehicles queued behind it to form a convoy. Instead of each of the vehicles in the convoy broadcasting a *PERMIT* message to notify other vehicles waiting in conflicting lanes that it is no longer in their way, the algorithm designates the last vehicle of the convoy as the representative for the whole convoy and requires only that vehicle send a single *RERMIT* message.

For example, in the situation pictured in *Figure* 19, when the green car gets the turn to pass through the intersection, the yellow car and the orange car too get to follow along the green car. Only the orange car broadcasts a *PERMIT* message and when that is received the red car starts passing through the intersection.

The premise leading to this optimization is that the last vehicle in a convoy is the last to exit the core area (*Figure* 1) of the intersection. It seems that the authors of the paper are relying on the unspoken assumption of "*right turns are not allowed*" to achieve this reduction of messages exchanged (Chapter II - Sections

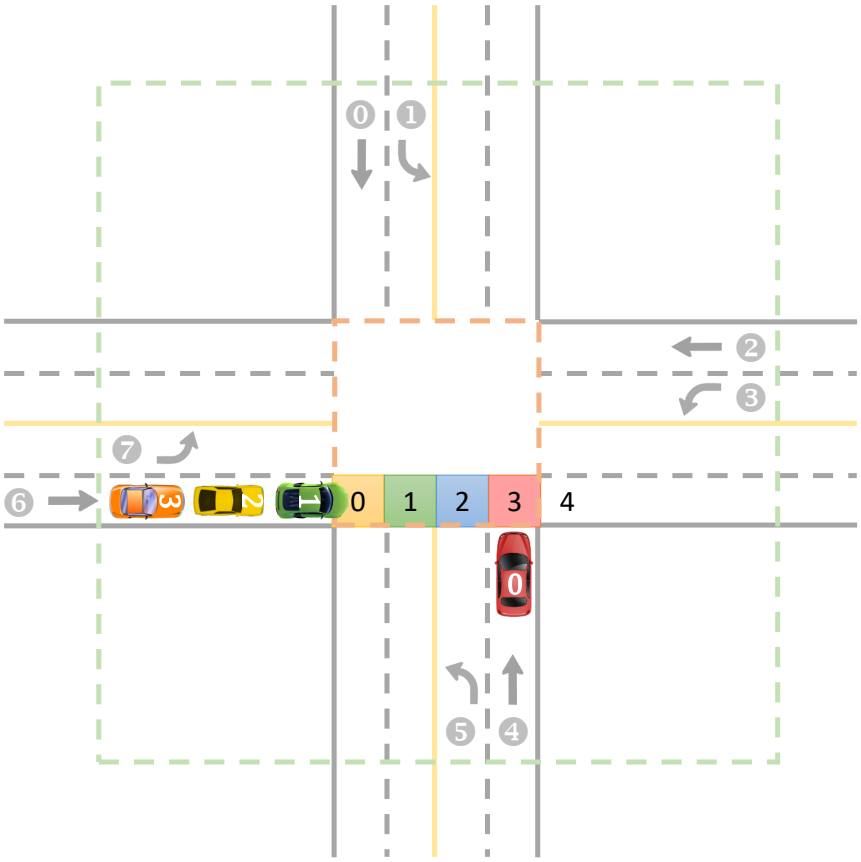

Figure 19. Example - A "*Follow List*" convoy, a situation induced by the follow list related logic of the algorithm. The green car (car 1) has broadcast a follow list. It contains the yellow car (car 2) and orange car (car 3). Therefore, cars 1, 2, and 3 form a convoy and pass through the intersection together. In the meantime, the red car (car 0) is waiting till the convoy passes through the intersection. According to follow list logic of the algorithm, orange car is responsible for notifying others (in this case the red car) that the convoy has cleared the core of the intersection. The convoy passes through segments 0, 1, 2, 3 of the core in sequence and exits the core area to the segment 4, which is the open road after the core. In the specific case shown in the figure, if car 0 enters the core while the convoy is still in transit through the core, a crash is possible in segment 3.

2.1 and 2.2). We believe that disallowing right turns at an intersection is a terrible price to pay in exchange for a reduction of a few messages.

In this chapter, we demonstrate how this optimization ends up creating a crash if right turns are allowed.

Building a complete model to highlight the problem in this situation requires a lot of effort. It requires at least 3 cars, 2 of which form the convoy while the other car enters the intersection from a lane that conflicts with the convoy. Further, we have to model all the logic behind the "*Follow List*" (Chapter II, Section 2.4.2.3) related portions of Algorithm 2: creation (line 25), broadcast (line 26) and processing after receiving (lines 29 - 37). This could end up with a quite large model.

Instead, we take a shortcut just to pinpoint the problem. We fast forward to an intersection where a follow list has been already broadcast and a convoy is passing through the intersection (e.g. car 1 - green, car 2 - yellow and car 3 - orange in lane 6) and there is another vehicle (e.g. car 0 - red) waiting in a lane that conflicts with the convoy (lane 4) till the convoy is through the intersection (*Figure* 19).

6.1 Fine-Grained Conceptual Model of the Intersection

To highlight the crash situation, we model the core area of the intersection with a finer granularity than we do in Chapter V. In Chapter V, we consider the core area of the intersection as a whole. Each car, after taking the *enter* action is considered to be in the core area of the intersection. Taking the *exit* action, a car exits the intersection. We do not model what happened to a car in between these two actions. In other words, we do not model the trajectory of a car within the core area of the intersection. Two cars simultaneously entering the core from conflicting lanes is considered a crash.

Since right turns are only applicable to lanes with even numbers (*Figure* 1), we only consider those lanes. We segment the portion of such a lane that runs through the core area of the intersection to 4 segments as shown in *Figure* 19.

Cars of the convoy enter the core area of the intersection into segment 0. Then they enter segments 1, 2, 3 in sequence and exit the core area when they exit the segment 3.

We label the road after segment 3 where cars exit the core area and drive away as segment 4. This choice makes our models a bit simpler and uniform. With this we can model exiting segment 3 as entering segment 4. This comes handy in several places such as building the convoy and defining fluents to test the model.

With these segments, we keep track which segment a car is in while passing through the intersection. A car can be in one segment for some time before moving into the next segment. Also, in the convoy, a car cannot enter a segment occupied by the car in front and cars cannot overtake while in transit as a convoy passes through the intersection. Further, a car in front can move faster than its followers if the segments in front of the fast-moving car are empty. This allows to have gaps of empty segments between cars of the convoy (e.g. The car at the head of the convoy moves fast up to segment 3 while the car behind it is still in segment 0. This leaves segments 1 and 2 empty.)

As per the *Figure* 19, if a car coming from any of the lanes 3, 4 or 5 enters the intersection while the convoy from lane 6 is still passing through the intersection, there is the possibility of a crash in segments 1, 3 and 2 respectively.

To make the models concise and more readable, we use the constants, ranges and sets defined in Listing 27. When compiling models these definitions should be included in the code.

6.2 Modeling a CAR in the Convoy

The new model of car breaks the passage of a car within the core area of the intersection into 4 states to resemble passage through the 4 segments we describe

Listing 27 Constants, sets and ranges that are used in LTSA models related to follow list. When compiling the models, include this section.

```
// Segment under investigation for a potential crash
// Acceptable values: 0, 1, 2, 3
const CRASH_SEG = 3

const False = 0
const True  = 1
range Bool  = False..True

// Number of vehicles in the convoy
const N = 3

// A literal to indicate an empty list
const EMPTY = -1

// Possible values that can be in
// High (hi) or Low (lo) lists
set LIST_VALS = { [EMPTY], [N] }

// Message types
const PERMIT  = 300
```

in Section 6.1 (*Figure* 19). With this the *PASSING* state of our extended state transition diagram (*Figure* 7) gets broken into to 4 smaller states (*Figure* 20).

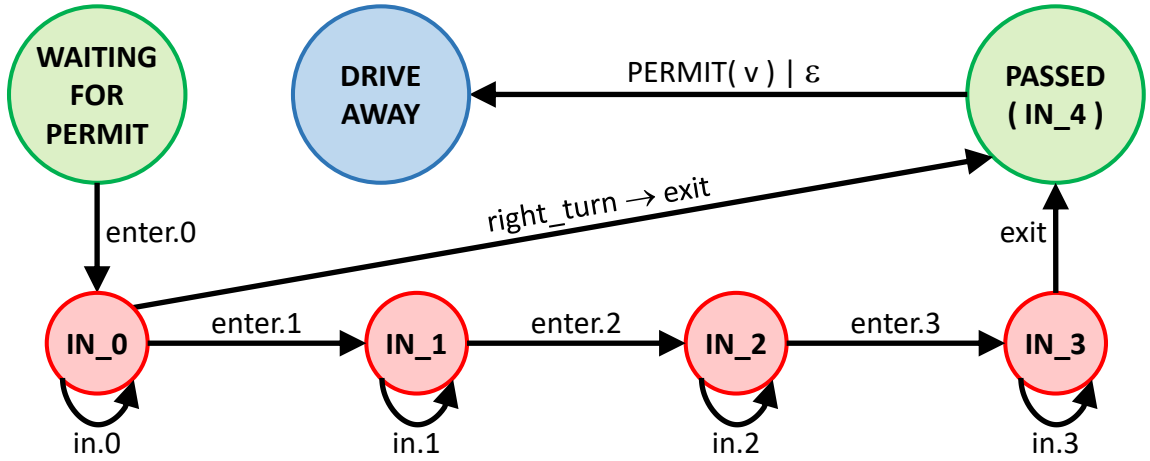

Figure 20. The finer-grained state transition diagram of a car. This breaks the *PASSING* state of our extended state transition diagram (*Figure* 7) into 4 smaller states to provide a more detailed description of a car passing through the core area of the intersection. Please note that we only show the part of the state diagram that is relevant for the fast-forwarded state of the algorithm we are modeling.

The model of a car in the convoy is presented in the Listing 28. Since we have fast forwarded to a state where follow list is created, broadcast and the convoy is about to enter the core area of the intersection, the *CAR_CONVOY* process starts straightaway in the *WAITING_FOR_PERMIT* state. Also, we do not bother to model the high and low lists as they are empty and do not matter at this stage. The process follows the state diagram shown in *Figure* 20.

We model the *CAR_CONVOY* in a way so that we can later model a convoy of N such cars. No matter how long the convoy is, we model car N to be the tail of the queue and it is the vehicle responsible for broadcasting the *PERMIT* message. Notice that we have hard-coded this fact into the model of *CAR_CONVOY* (lines 32 - 35 of Listing 28).

The *CAR_CONVOY* process accepts one special Boolean parameter *RT* to signal whether a car is to drive straight or to take a right turn. When *RT* is set to *True*, that car takes a right turn (lines 9 - 13 of Listing 28).

The lines 8, 17, 22 and 27 of Listing 28 ("`[ VID ].in[ s ]`", $s = 0, 1, 2, 3$), allow a car to spend some time in each segment before moving into the next segment.

6.3 Modeling the Convoy

Now, stitching several *CAR_CONVOY*'s (Listing 28) into a queue, we are building a *CONVOY*. In doing so, we have to make sure the final convoy demonstrates the behavior outlined in the Section 6.1.

For our experiments, we build a convoy of 3 cars ($N = 3$), where car 1 followed by car 2 followed by car 3. Car 3 is the last in the convoy (Listing 29).

The *ORDER* process (lines 10 - 18) enforces the order of entering into and exiting from segments 1, 2 and 3. When we compose the *CONVOY* we compose 3 *ORDER* processes, one per each segment (line 23). Line 12 (`[1].enter[SEG] -> [1].enter[SEG+1] ->`) enforces that car 1 must enter segments in sequence. Line 13 enforces the same constraint on car 2 and so forth. The line for car 2 (line 13) appearing after the line for car 1 (line 12) dictates that car 2 cannot enter a segment prior to car 1 vacating that segment by moving into the next segment. This logic follows for the other lines in the *ORDER* process and this prevents cars from overtaking while moving through the intersection as a convoy.

If cars are allowed to make right turns, a car performs this maneuver in segment 0. Therefore, a car has two options after entering segment 0 and hence we have to model the order of entrance and exit of segment

Listing 28 *CAR_CONVOY*, the model of a car in the convoy

```
CAR_CONVOY( VID = 1, RT = False ) = WAITING_FOR_PERMIT,

WAITING_FOR_PERMIT =
  (
            [ VID ].enter[ 0 ]              -> IN_0
  ),

IN_0 =
  (
            [ VID ].in[0]                   -> IN_0

    | when ( RT == False )
            [ VID ].enter[ 1 ]              -> IN_1

    | when ( RT == True )
            [ VID ].right_turn
        -> [ VID ].exit                     -> PASSED
  ),

IN_1 =
  (
            [ VID ].in[ 1 ]                 -> IN_1
    |       [ VID ].enter[ 2 ]              -> IN_2
  ),

IN_2 =
  (
            [ VID ].in[ 2 ]                 -> IN_2
    |       [ VID ].enter[ 3 ]              -> IN_3
  ),

IN_3 =
  (
            [ VID ].in[ 3 ]                 -> IN_3

    // This means the car is exiting the core area
    |       [ VID ].enter[ 4 ]              -> PASSED
  ),
```

```
30 PASSED =
31   (
32       when ( VID == N )
33            [ VID ].broadcast[PERMIT][VID] -> DRIVEAWAY

34     | when ( VID != N )
35            [ VID ].drive                  -> DRIVEAWAY
36   ),

37 DRIVEAWAY =
38   (
39            [ VID ].drive                  -> DRIVEAWAY
40   )
41     + {
42         [ VID ].enter[ 1 ],
43         [ VID ].right_turn
44       }.
```

0 separately. *ORDER_0* process takes care of this business. Line 3 (`[1].enter[0] -> { [1].enter[1], [1].right_turn } ->`) specifies that car 1 must enter segment 0 before either entering segment 1 or making a right turn. Line 4 specifies the same for car 2 and so forth. Similar to the logic in *ORDER* process, the order of lines prevents a car with a higher number entering segment 0 occupied by a car with a lower number.

Current *ORDER_0* and *ORDER* processes are set for a convoy of 3 cars. Uncommenting lines 6 and 15, and setting $N = 4$ in line 5 of Listing 27 give us a convoy of 4 cars. If we also uncomment lines 7 and 16, and set $N = 5$, it gives a convoy of 5 cars. Following this pattern, adding more lines and setting N accordingly will give longer convoys.

Line 21 (`forall [ i: 1.. N ] car:CAR_CONVOY(i, i == N)`) brings in N *CAR_CONVOY* processes to the composition. "`i == N`" becomes *true* only

Listing 29 A *CONVOY* of *CAR_CONVOY*'s

```
ORDER_0 =
  (
    [1].enter[0] -> { [1].enter[1], [1].right_turn } ->
    [2].enter[0] -> { [2].enter[1], [2].right_turn } ->
    [3].enter[0] -> { [3].enter[1], [3].right_turn } ->
  //[4].enter[0] -> { [4].enter[1], [4].right_turn } ->
  //[5].enter[0] -> { [5].enter[1], [4].right_turn } ->

    ORDER_0
  ).


ORDER( SEG = 1 ) =
  (
    [1].enter[SEG] -> [1].enter[SEG+1] ->
    [2].enter[SEG] -> [2].enter[SEG+1] ->
    [3].enter[SEG] -> [2].enter[SEG+1] ->
  //[4].enter[SEG] -> [4].enter[SEG+1] ->
  //[5].enter[SEG] -> [5].enter[SEG+1] ->

    ORDER
  ).

||CONVOY =
  (
        forall [   i: 1..N ] car:CAR_CONVOY(i, i == N)
    ||                       car:ORDER_0
    || forall [ seg: 1..3 ] car:ORDER( seg )
  ).
```

for car N, which is the car at the tail of the convoy. By changing N to any value between 1 and N, we can make any car in the convoy take a right turn. Further, in "`i == x`", by setting x to something other than 1, 2, ..., N (e.g. -1, 0, N+1 etc.), this clause becomes *false* for all the cars. By this we can have a convoy with all the cars that go straight.

6.4 The Trouble Maker

Just a convoy passing through the intersection does not give rise to any problems. We need another car, in a lane that conflicts with the convoy, waiting till the convoy exits the intersection to enter the intersection (Listing 30). We call this *CAR_0*.

Like the *CAR_CONVOY* (Listing 28), we boot-up the *CAR_0* process in the fast-forwarded state of *WAITING_FOR_PERMIT* where it has already received the follow list and its high list contains the last vehicle in the convoy (N) and its low list is empty. Only thing *CAR_0* needs to enter the intersection is a *PERMIT* message from car N.

The parameter *SEG* specifies which segment *CAR_0* is going to pass through. That is the segment where a crash is possible. If we fix a lane for the convoy, setting the *SEG* parameter determines which lane the *CAR_0* is coming in from. For example, if we follow the setting of the intersection in *Figure* 19 where the convoy is coming from lane 6, $SEG = 3$ means the *CAR_0* is coming from lane 4, which is the red car depicted in the figure. Similarly, other associations are: $SEG = 2 \implies$ lane 5, $SEG = 1 \implies$ lane 3, and $SEG = 0 \implies$ lane 0. If we change the lane of the convoy from 6 to another possible lane (lanes 0, 2, 4), we will get different lane associations for different *SEG* values.

Listing 30 *CAR_0* that conflicts with the convoy

```
1  CAR_0( SEG = 0 ) = WAITING_FOR_PERMIT[ EMPTY ][ N ],

2  WAITING_FOR_PERMIT[ lo: LIST_VALS ][ hi: LIST_VALS ] =
3    (
4        when (hi == EMPTY)
5           [ 0 ].enter[ SEG ]              -> IN_SEG

6      |    [ 0 ].receive[ PERMIT ][ N ]
7                 -> WAITING_FOR_PERMIT[ EMPTY ][ EMPTY ]
8    ),

9  IN_SEG =
10   (
11          [ 0 ].in[ SEG ]                 -> IN_SEG

12     |    [ 0 ].exit[ SEG ]
13       -> [ 0 ].pass_through
14       -> [ 0 ].exit                      -> DRIVEAWAY
15   ),

16 DRIVEAWAY =
17   (
18     [ 0 ].drive                          -> DRIVEAWAY
19   ).
```

We do not have any interest in the behavior of car 0 after it exits the segment *SEG* we are investigating for a potential crash (after car 0 takes the action `[ 0 ].exit[ SEG ]` in line 12). For completeness, we model the rest of the states of car 0 till it exits the core of the intersection. The "`[ 0 ].pass_through`" action in line 13 resembles any other additional distance car 0 has to travel within the core of the intersection before exiting the intersection (taking the action `[ 0 ].exit` in line 14).

Listing 31 *NEWTORK* process that carries the *PERMIT* message

```
1 NETWORK = BUFFER[ False ],

2 BUFFER[ in_transit : Bool ] =
3     (
4           accept_message [ PERMIT ][N]-> BUFFER[True]

5       | when (in_transit)
6           deliver_message[ PERMIT ][N]-> BUFFER[False]
7     ).
```

6.5 The Network

We do not need a network as elaborate as the networks we use in Chapter V. In this case, the only task of the network is to deliver the *PERMIT* message sent by car N to car 0. Although, we can even achieve this by directly synchronizing the "`[ N ].broadcast[ PERMIT ][ N ]`" action of car N, which broadcasts the *PERMIT* message (Listing 28) and the "`[ 0 ].receive[ PERMIT ][ N ]`" action of car 0, which receives the *PERMIT* message (Listing 30), we decided to use a simple *NETWORK* process (Listing 31) to keep things more realistic and uniform among chapters.

6.6 Modeling the Intersection

At this point, we have all the component processes to model the behavior at the intersection when a convoy is in transit after a follow list has been broadcast. It is just a matter of composing the processes and making network connections to enable communication (Listing 32).

Listing 32 The situation at the intersection after a follow list has been broadcast

```
||INTERSECTION =
  (
       CONVOY
    || car:CAR_0( CRASH_SEG )
    || NETWORK
  )
    / {
        car[ N ].broadcast[ PERMIT ][ N ]
                      / accept_message [ PERMIT ][ N ],

        car[ 0 ].receive  [ PERMIT ][ N ]
                      / deliver_message[ PERMIT ][ N ]
      }.


fluent IN_SEG_CAR_0 = <
                        car[ 0 ].enter[ CRASH_SEG ],
                        car[ 0 ].exit [ CRASH_SEG ]
                      >

fluent IN_SEG_CAR_[ c: 1..N ] =
  <
      car[ c ].enter[ CRASH_SEG ],
    { car[ c ].enter[ CRASH_SEG + 1 ], car[ c ].exit }
  >

assert CRASH = []!  (
                      exists[ c: 1..N ]
                          (
                              IN_SEG_CAR_0 &&
                              IN_SEG_CAR_[c]
                          )
                    )
```

The global constant *CRASH_SEG* (Listing 27) defines the segment we are focusing on to investigate the possibility of a crash. We can set it to 0, 1, 2 and 3 in turn to check the crash situation at each segment.

In addition to the composition, we include the necessary fluents and asserts to check for possible crashes. The *fluent* `IN_SEG_CAR_0` (lines 13 - 16) is true when car 0 is within the segment *CRASH_SEG*. The *fluent* `IN_SEG_CAR_[ c: 1..N ]` (lines 17 - 21) is defined for all the cars in the convoy. It is true for the car of the convoy that is in the segment *CRASH_SEG*.

Lines 23 - 27, which belong to the *assert*, check whether any car in the convoy and car 0 are in the segment *CRASH_SEG* at the same time. The *assert* dictates that at all times ([ ]) it should not (!) be the case that car 0 and any car of the convoy are together in the segment *CRASH_SEG*, which is a crash.

6.7 Possible Crashes

We test the composition for possible crashes between the vehicles composing the convoy and vehicles coming in from all the lanes that conflict with the convoy by varying the global constant *CRASH_SEG*. The system does not suffer from deadlocks at all. Also, when the car at the tail of the convoy is made to go straight (by setting `i == 0` in line 21 of Listing 29 that describes the *CONVOY* process), there is no issue of crashes at all.

When $CRASH\_SEG = 0$, system does not suffer from the danger of a crash even when the last car of the convoy makes a right turn. This is intuitive because the last car broadcasts a *PERMIT* message only after taking the right turn and exiting segment 0, which results it to exit the intersection. Since the last car of the convoy has now cleared the segment 0, no other car of the convoy is or will be

in segment 0 to encounter with car 0, avoiding any possibility of a crash between the convoy and car 0.

In other 3 cases ($CRASH\_SEG = 1, 2, 3$), LTSA reveals action traces to crashes between cars of the convoy and car 0 when car 3 (the last car of the convoy) makes a right turn. While $CRASH\_SEG = 1$ gives rise to a crash between car 2 and car 0 (*Figure* 21), on the two other segments the crash is between car 1 and car 0 (*Figure* 22).

As an illustration, consider the situation in *Figure* 19. First, the green car enters segment 0 and as it moves to segment 1, the yellow car enters segment 0. When the green car and the yellow car advance to segments 2 and 1 respectively, the orange car enters segment 0. Now, irrespective of the locations of the green and yellow cars, the orange car takes a right turn, exits the core of the intersection and broadcasts a *PERMIT* message notifying everybody else that the convoy has passed through the core of the intersection. Relying on this message, the red car enters segment 3. At this moment, either of the green car or the yellow car could still be in segment 3 creating a crash.

The action traces LTSA reveals for crashes vary the speeds of vehicles through the intersection. For example, according to the action trace in *Figure* 22, car 1 quickly goes all the way till segment 3 and slows down. Until then, car 2 does not enter the segment 0. Although in our heads a convoy of cars is pictured as a queue of cars all moving at similar speeds, that should not be the case all the time. Thus, the danger of crashes LTSA reveals is not hypothetical.

When describing the problem and the models, as an example, we use a convoy coming from lane 6 (*Figure* 19) to aid the reader easily visualize and relate to the problem. However, none of our models are specific to lane 6. Therefore, our

Trace to property violation in CRASH:

car.1.enter.0	
car.1.enter.1	IN_SEG_CAR_.1
car.1.enter.2	
car.2.enter.0	
car.2.enter.1	IN_SEG_CAR_.2
car.3.enter.0	IN_SEG_CAR_.2
car.3.right_turn	IN_SEG_CAR_.2
car.3.exit	IN_SEG_CAR_.2
car.3.broadcast.300.3	IN_SEG_CAR_.2
car.0.receive.300.3	IN_SEG_CAR_.2
car.0.enter.1	IN_SEG_CAR_0 && IN_SEG_CAR_.2

Figure 21. Action trace leading to a crash in the composition of Listing 32 when $CRASH\_SEG = 1$.

models are generalized to any convoy coming in from any of the lanes 0, 2, 4, and 6, and any other car coming in from a lane that conflicts with the convoy. Hence, our results can be generalized to all these situations.

6.8 A Suggested Solution

The problems of the distributed mutual exclusion algorithm for intersection traffic control (Algorithm 2) discussed in this chapter can easily be resolved without prohibiting vehicles making right turns at the intersection. We need to modify the follow list related logic a bit.

Trace to property violation in CRASH:

car.1.enter.0	
car.1.enter.1	
car.1.enter.2	
car.1.enter.3	IN_SEG_CAR_.1
car.2.enter.0	IN_SEG_CAR_.1
car.2.enter.1	IN_SEG_CAR_.1
car.3.enter.0	IN_SEG_CAR_.1
car.3.right_turn	IN_SEG_CAR_.1
car.3.exit	IN_SEG_CAR_.1
car.3.broadcast.300.3	IN_SEG_CAR_.1
car.0.receive.300.3	IN_SEG_CAR_.1
car.0.enter.3	IN_SEG_CAR_0 && IN_SEG_CAR_.1

Figure 22. Action trace leading to a crash in the composition of Listing 32 when $CRASH\_SEG = 3$.

Instead of just making the last vehicle of the follow list broadcast a *PERMIT* message to notify others that the convoy has cleared the intersection, we should make the last vehicle of the convoy and the last vehicle of the convoy that is going straight both broadcast *PERMIT* messages. In the case of the last vehicle of the convoy is not making a right turn, both these vehicles refer to the same vehicle. In other cases, there will be two distinct vehicles. All the other vehicles that are waiting till the convoy is through the intersection must wait till they

receive both the *PERMIT* messages (and any other relevant *PERMIT* messages they are waiting for) before attempting to enter the core area of the intersection.

To implement this modification, we have to enrich the *REQUEST* message to carry the intention of a vehicle to go straight or turn right. This is just like blinking a turning signal on human operated vehicles. With this information available to all the vehicles waiting at the intersection, the vehicle that is compiling the follow list can include this right turn information into follow list. Vehicles receiving the follow list could then easily determine the two vehicles responsible for notifying that the convoy resulted due to the follow list has completely exited the core of the intersection.

CHAPTER VII

CORRECT MODEL — INCORRECT IMPLEMENTATION

In this chapter, we move out from our specific discussion on the algorithm and its pitfalls to the general realm of implementing distributed algorithms.

Designing a new algorithm or a protocol is a challenging task. We must make sure that it always produces the correct output within a reasonable amount of (short) time utilizing reasonable amount of (as little as) resources (memory, communication).

When it comes to designing a distributed algorithm, the task becomes even more challenging due to the necessity of proper coordination of multiple components. In such algorithms, we must make sure that the additional properties such as Liveliness, Fairness and Safety hold.

Manually checking whether a distributed algorithm adheres to the aforementioned qualities is only feasible if the algorithm at hand has few visible states. When the state space grows, comprehending all the possible states that the complete distributed system can reach and/or should avoid becomes a daunting task. That is where formal modeling tools such as *Labelled Transition System Analyser* (LTSA) (Magee et al., 2013) come to our rescue.

Using LTSA, we can model, test and debug a distributed algorithm and make sure that it does not suffer from the anomalies like deadlocks and critical section violations. A model is extremely useful in identifying issues and rectifying them early . However, a model, no matter how pristine, is unable to perform actual work the system being modeled is supposed to carry out.

After the modeling stage, the model is usually used as a communication tool that provides a formal specification of the system to be implemented. In the

implementation stage, a programmer interprets the model and translates it to a program written in some computer language. This translation stage is prone to human error. What if the programmer misinterprets some portion of the model? Or what if the programmer implements some details erroneously?

If such errors seep into to the implementation, all the time and effort invested in modeling, testing and validating an algorithm, and coming up with a clean model go in vain.

Since we already have a model that we have faith in, we ask the question: Can we employ the model as a higher authority to validate the implementation? This chapter presents our investigation to answer this self-imposed question.

7.1 Manulator - The Distributed System Simulator

Implementing, testing, and debugging a distributed system with multiple physical components (e.g. nodes) is a cumbersome and time consuming endeavor. Different parts of the system need to be coded and uploaded to respective nodes. Then all the nodes of the system should be executed and the execution of the nodes should be monitored. If necessary, execution logs should be downloaded from a dispersed set of nodes and analyzed. This cycle must be repeated until the implementation adheres to the model.

To simplify the implementation cycle of code–test–debug, we implement a distributed system simulator, *Manulator*[1], in python. Our goal is to provide a proof of concept but not to come up with a multi-purpose industry strength implementation of a simulator. Thus, Manulator can only handle implementations of a distributed system coded in python. It is very light weight and simple. The Manulator is comprised of three components:

[1] **Manu**jinda's Simu**lator** = **Manulator**

Listing 33 *Node* class that extends the *Communications* class. This is where the code for a node in a distributed system is implemented

```
1 import time
2 from Communications import Communications

3 class Node( Communications ):

4     def initialize_node( self, argc, argv ):
5         # argv[0] is the module name
6         # argv[1] is the node id
7         self.me = argv[1]

8     def main( self ):
9         print 'Execution of the node starts here'

10    def on_msg( self, sender, msg ):
11        print 'Process {} received {} from {}'
12                        .format( self.me, msg, sender )
```

Communications: The class that provides a communication interface to the nodes.

Node: A class that extends the *Communications* class (Listing 33), which provides a skeleton to implement a single node in the distributed system that is being simulated.

manulator: The python script that choreographs all the pieces of a distributed system and handles the simulation.

7.1.1 Communications Class. The *Communications* class provides a network interface to a node. The interface includes 3 network calls, two methods to initialize and start execution of a node and a helper method to write comments to a log file to help debugging:

send(msg, to):	Send the message *msg* to the node with id *to*. When extending the *Communications* class, this method should not be overridden.
broadcast(msg):	Broadcast the message *msg* to all the nodes. When extending the *Communications* class, this method should not be overridden.
on_msg(sender, msg):	Any message received from the node with id *sender* is available in this method. When extending the *Communications* class, a node can override this method to extract the message *msg* and its sender within this method and operate on it accordingly.
initialize_node(argc, argv):	The manulator passes the command line arguments for a node through this method. The node initialization (e.g. assigning a node id) should be done by overriding this method when extending the *Communications* class.
main( ):	This is where a node starts its execution. When extending *Communications* class, a node must override this method to start its process.
log(entry):	Used to add any log entry to a per node log file that is automatically created by the manulator, which logs the messaging behavior of a node. By using *log* method, additional messages can be added to

augment the default log file created. When extending the *Communications* class, this method should not be overridden.

7.1.2 Node Class. The *Node* class (Listing 33) extends the *Communications* class as described in Section 7.1.1 and is used to implement the logic of a single node. As per the necessity, additional methods can be added to the *Node* class to make the code of a node more modular.

To keep the Manulator simple, we have restricted the name of this class to be *Node*. If the system being simulated requires different nodes to have different implementations, they can be coded in separate python modules each having a class named *Node* that extends the *Communications* class.

A node receives command line arguments in the overridden method `initialize_node( argc, argv )`. The parameters, *argc* is the argument count and *argv* is a list of strings, where each string represents a command line argument. Again, to keep the Manulator simple, `argv[0]` always carries the module name and `argv[1]` always carries the node id. Any other arguments that are passed to a node follow these two.

A node starts its execution beginning at the first line of the overridden method `main( )`.

Whenever a node receives a message from some other node, Manulator makes it available to the node in the method `on_message( sender, msg )`. By overriding this method, a node can process incoming messages as required by the logic of the implementation.

The Manulator automatically logs all the messages sent and received by a node in a per node log file with the name *+_2_log_<node id>.txt*. A "+" sign is

prefixed to the name so that in a directory listing all the log files appear together close to the top of the listing. The "2" is to make all the log files appear below the Manulator global log file, which has the name *+_1_log_main.txt*.

A node can use the inherited `log( entry )` method to augment its personal log file with additional comments that are interleaved with message passing log entries to make it easier to decipher the automatically generated messaging log.

7.1.3 manulator. manulator is the script that brings the simulation to life. By default, manulator searches for a comma-separated configuration file named *nodes.csv*, which describes the nodes and the command-line arguments to be passed into each node. Each line of the configuration file describes one node and its command-line arguments:

<module name>, <node id>, <arg1>, <arg2>, ...

The module name is the module (name of the file) that holds the code for the *Node* class that extends the *Communications* class. Node id is the id for the node described by this line. It can be any valid python variable name or integer that suits the implementation of the node. arg1, arg2, ... are other command-line arguments that are passed into this node. Since we use comma (,) to separate files, module name, node id or any of the arguments cannot contain commas.

After the lines specifying nodes, the configuration file can contain an optional line to specify a timeout duration:

Timeout, <timeout duration in seconds>

The Manulator uses this to watch for periods longer than the timeout duration without any message being exchanged and notify the user. If this line is omitted, a default timeout period of 1 second is used. Further, lines starting with "#" signs are considered comments. Moreover, to enable keeping several

configurations commented in the same configuration file, we make manulator ignore all the lines from and below a line that starts with a hyphen ("-").

To make it possible to keep several configuration files to easily switch and test different compositions of a distributed system, manulator accepts an optional configuration file name as its first argument. If such a file name is provided, manulator searches the current directory for a configuration file with that name instead of the default (nodes.csv).

After reading and parsing the configuration file, the Manulator starts playing the role of a plumber. It creates a network of pipes (*16.6. multiprocessing - Process-based "threading" interface*, 2017), which is used to achieve inter-process-communication (ipc) to make the communication among different nodes and the Manulator possible (*Figure* 23).

Each node in the simulation is spawned as a separate process. Manulator maintains a per node monitoring thread. There is a sending pipe between each node process and the monitoring thread that is used by the node process to send messages. Further, each node process has receiving pipe shared among all the node monitoring threads that is used to receive messages from other nodes. The monitoring thread, upon receiving a message from the node it monitors, routes that message to the relevant receiver via the receiving pipe belonging to the designated receiver node.

In addition to the per node monitoring threads, the Manulator maintains another main monitor thread that talks to all the node monitoring threads and manages the whole simulation. This thread shares a pipe among all the

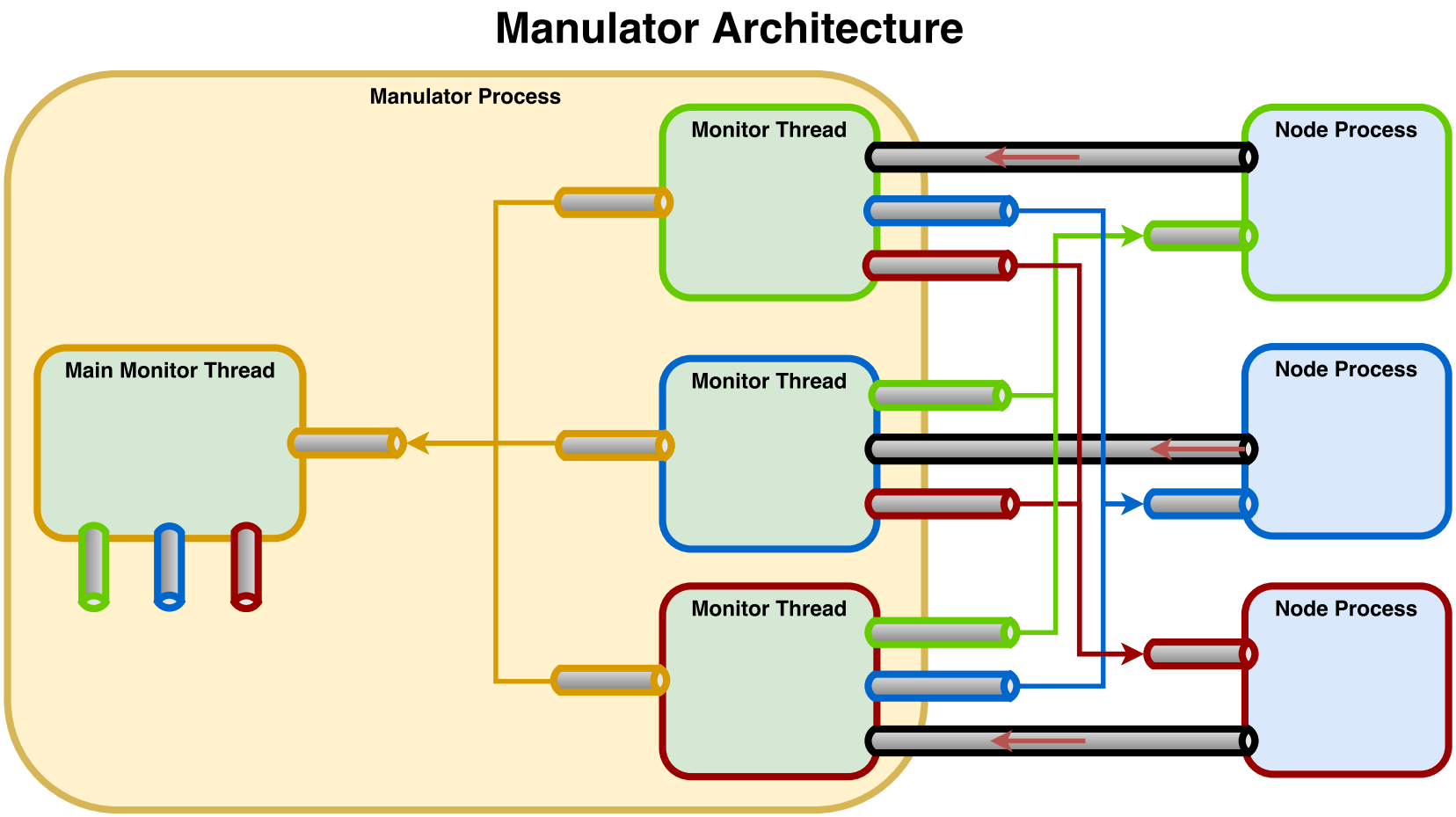

Figure 23. Manulator architecture. This diagram is of a simulation of a distributed system with 3 nodes. Pipes with black outline are the sending pipes, which each node process uses to send messages to its respective node monitoring thread. The pipe with the outline color that matches the outline color of the node process box is the receiving pipe of each process. The pipe with the brown outline is between the node monitoring threads and the main monitor thread.

node monitoring threads. Via this pipe, the main monitor thread receives all the messages passed in the simulation, which it logs to the main log file (+_1_log_main.txt). Further, the main monitor thread uses the receiving pipe of each node process to route control messages (e.g. termination message) to each node process.

After all the plumbing are in place, Manulator starts the main monitor thread. Then, it pairwise starts the node monitor thread and the respective node process that node monitor thread monitors. This kicks off the simulation. At any time, hitting return in the console where the simulation is running terminates the simulation.

7.2 Message Queue Telemetry Transport (MQTT)

We use python and java versions of the *paho* implementation (Craggs, Sutton, & Pagliughi, 2017) of the MQTT protocol (Stanford-Clark & Nipper, 2014) in our implementations to enable communication among distributed components. MQTT is a very lightweight publish/subscribe messaging protocol built on top of the TCP/IP protocol. With MQTT in place, we can quickly implement both point-to-point and broadcast message passing among nodes taking part in a distributed algorithm implementation without spending a lot of time on designing elaborate message formats or coding socket connections. This enables us easily prototype and experiment with our ideas and keep our attention more on the interesting problems we are going after for solutions.

In MQTT, the communication happens via a broker. Any process that wishes to send some message publishes it to the broker under some topic. Any process that is interested in receiving messages published on a topic can subscribe to that topic at the broker.

In our implementations, nodes share a common topic stem, say *manu*, and each node with id n subscribes to the topic *manu/n*. This serves as an ip address for that node. Any node wanting to send a message to node n, publishes that message to topic *manu/n*. By making all the nodes subscribe to another common topic, we can easily achieve broadcasts. Whatever published to this common topic is received by all the nodes. This approach uses topics just as ip addresses and everything that is needed to be communicated is encoded in the MQTT message.

Using a unique topic for each piece of information that needs to be communicated and a dummy MQTT message is the other extreme where everything is encoded in the topic. Another middle-ground approach is to encode what needs to be communicated in both the topic and the MQTT message. For example, each topic can signal a message type and actual message can be encoded in the MQTT message portion.

In addition to these, MQTT allows each node setup a last will. By nodes subscribing to a common will topic, whenever a node crashes, MQTT broker automatically delivers the will message of the crashed node to everybody else in the system. This comes handy when making an implementation respond or resilient to node failures.

7.3 The Model Meets the Implementation

The authors of the LTSA tool (Magee et al., 2013) were very generous to share their code-base (in java) with us, which enabled us to quickly test our ideas. Otherwise, the portion of work we present hereafter could have been nearly impossible or taken quite a long time.

We extend the LTSA tool to eavesdrop on the messages passed among nodes of an execution of an implementation and validate whether the sequence

of messages passed adheres to the language of messages generated by the LTSA model of the system, which is a finite state process. We call our extended version of LTSA the "*Labelled Transition System Analyser & Observer*" (LTSA-O). When the LTSA-O sees an out of sequence message, it notifies that to the user.

This work goes along the lines of the *BISIMULATOR* (Bergamini, Descoubes, Joubert, & Mateescu, 2005; Mateescu & Bergamini, 2017; Mateescu & Oudot, 2008), a tool that verifies whether two Linear Temporal Systems (LTSs) are equivalent. First LTS is built implicitly by following a system description while the second LTS explicitly describes a system. This way *BISIMULATOR* verifies whether the explicit description of the system matches the specification. In the case of a mismatch, it produces a counter example.

LTSA-O differs from *BISIMULATOR* since it verifies the execution of an implementation of a system against its finite state process model. BISIMULATOR checks the equivalence between two models.

Nodes of a MQTT based implementation of a distributed system publish messages to some broker to communicate with each other. LTSA-O subscribes to all the topics the nodes of the implementation use at the same broker used by the implementation. This way the LTSA-O can eavesdrop on each message exchanged in the implementation.

But, how does LTSA-O know which topics to subscribe to? LTSA-O gets to know this and some other information via a comma-separated configuration file. Once a model is composed, we can ask LTSA-O to spit out a skeleton for the configuration file. This is done by first selecting the menu items: Check > Run > Validate (*Figure* 24). This opens the "*Observer & Validator*" dialog (*Figure* 25).

On that dialog, clicking *Save Config* button gives the option to save the skeleton configuration file.

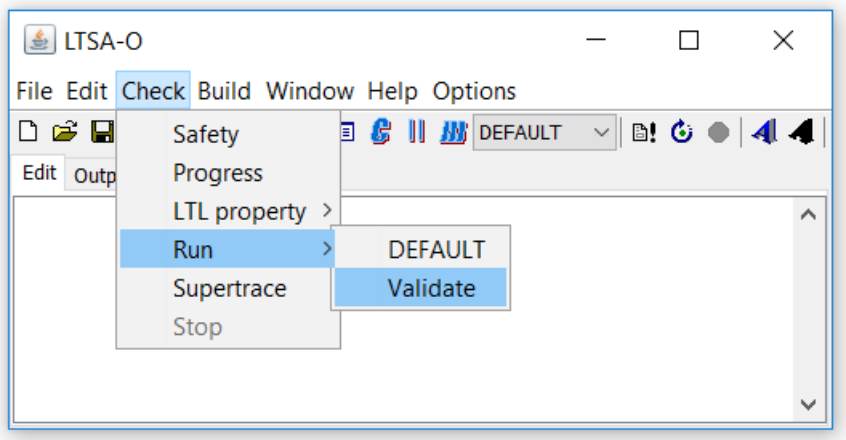

Figure 24. LTSA-O main window. The trail of menu selections to get the "*Observer & Validator*" dialog open.

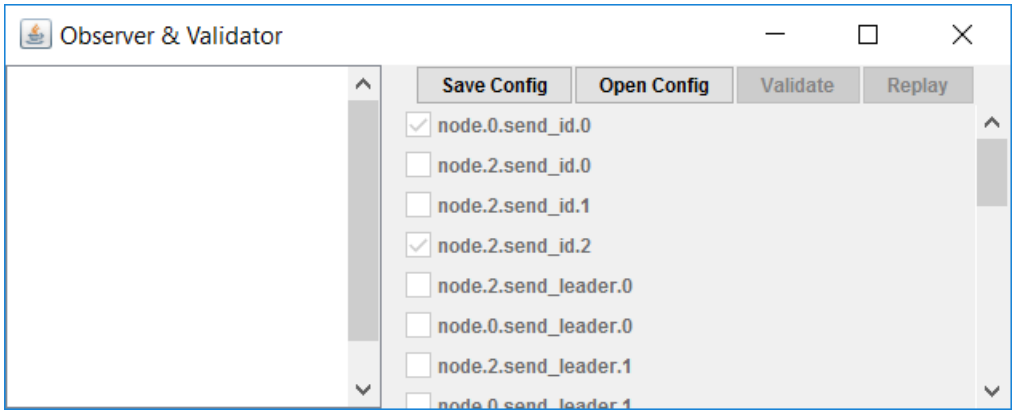

Figure 25. "*Observer & Validator*" dialog when a configuration file is not yet loaded. Use the *Save Config* button to get a skeleton of the appropriate configuration file. Use *Open Config* button to load a completed configuration file. When a configuration file is loaded, the *Validate* button enables. After clicking the *Validate* button, LTSA-O starts listening to topics specified in the configuration file published at the MQTT broker designated in the configuration file. After a validation run, the *Replay* button enables, where the last run of the implementation can be replayed.

As the first line of the configuration file, we must provide the MQTT broker and port where LTSA-O should subscribe for topics:

Broker, <MQTT Broker URL>:<Port>

The 2nd line specifies a MQTT topic used by LTSA-O to notify about misbehaving nodes (should not delete it). We discuss about this field later.

Bnode_ Topic, <Topic to be used to report bad nodes>

The 3rd line is just a row of column headings to guide the user (should not delete it):

Index, Action, Topic, Message

First two columns of the configuration file skeleton are filled by LTSA-O for the user. They must not be changed. 2nd column of the configuration file lists all the actions of the LTSA model. The user's responsibility is to fill the remaining 2 columns with the specific MQTT topics and messages used in the implementation being validated. Once filled, each such row provides a mapping between an action of the LTSA model with the corresponding message in the implementation and the topic the implementation uses to publish that message to the MQTT broker. Any of the actions, topics or messages should not contain commas (,) within them.

One caveat though is that the LTSA-O can eavesdrop only on the message passing layer of the implementation. Therefore, LTSA-O is unable to sense the internal node actions. However, during the modeling process, we do have to model them into the LTSA models and hence the skeleton configuration file lists such actions as well. We have to explicitly inform LTSA-O that such actions are local to a node and it will never hear nodes pass a message corresponding to such actions. For lines corresponding to internal node actions, mention "*local*" as the topic and some placeholder text (suggestion - a comment about the internal action) as the message. This adds one minor limitation to the selection of publish topics in the

implementation. Since LTSA-O has a special meaning attached to the topic *local*, none of the publish topics in the implementation should be *local*.

Figure A.27 in Appendix A provides a complete configuration file.

After populating the configuration file, it should be loaded into LTSA-O by clicking the *Open Config* button on the *Observer & Validator* dialog (*Figure* 25). At this point LTSA-O knows how to decrypt the communication among nodes of the implementation. Stating the validation process is just a matter of clicking the *Validate* button on the *Observer & Validator* dialog. Now, LTSA-O is busy listening to each message exchanged in the implementation.

After setting up the LTSA-O, it is time to execute the MQTT based implementation of the model. Once the implementation starts execution, for each message exchanged in the actual execution, the LTSA-O prints the corresponding action in the LTSA model in its *Observer & Validator* dialog window. If an out of order message is being passed in the actual execution, LTSA-O notifies that in the same window.

Moving on to the *Draw* tab of the LTSA-O main window and selecting a node of the model opens the graph of the finite state process driving that node. Selecting such a node while the validation process is in action lets you see the graph being animated by the messages passed in the actual implementation.

While all these actions are happening, LTSA-O remembers the most recent action trace resulted by the execution of the implementation. To aid debugging, that action trace can be replayed and relived as if it is driven by the execution of the implementation by clicking the *Replay* button on the *Observer & Validator* dialog (*Figure* 25).

7.4 Manulator Meets LTSA-O

The whole discussion about LTSA-O in Section 7.3 discusses at length how LTSA-O listens to communication taking place among nodes in an *actual implementation* of a model using MQTT as the underlying communication protocol. This requires someone to not only implement the system specified in the LTSA model but also have working knowledge in MQTT. This is somewhat counterintuitive, as in the prologue to this chapter, we expressed how daunting it is to actually implement a distributed system. Therefore, if somebody has to embark on a fully-fledged implementation of the system to get the assistance of LTSA-O, it is a rather long shot.

Luckily, that is not the case. We made Manulator capable of talking to LTSA-O via a MQTT broker. The main monitoring thread of the Manulator publishes all the messages exchanged in the simulation to a designated MQTT broker. By default, it uses the topic stem *manu* to publish these messages. Using this topic stem, a message destined to the node with id n is published to the topic *manu/n*. With this, LTSA-O can listen to the messages passed among nodes in the simulation the same way it does with an actual implementation and act as if it is listening to a real implementation (LTSA-O will not know the difference).

To enable all this, we just enhanced the Manulator configuration file to have two more fields.

Broker, <MQTT Broker URL>:<Port>

Topic, <Manulator publish topic root>

The first optional field, *Broker*, describes the MQTT broker address and its port where the Manulator should subscribe to and publish the messages exchanged in the simulation. This must be the same broker where LTSA-O is listening to,

so that it can eavesdrop on the communication and animate the respective model accordingly.

We add the second optional field, *Topic*, to make our system usable in a setting such as a class where multiple groups of students using the Manulator and the same MQTT broker simultaneously to debug their distributed implementations. If several instances of Manulator are in action simultaneously publishing with the same topic stem to the same broker and several LTSA-O instances are listening to communication happening within their respective simulations by subscribing to the same broker, there is high possibility that a particular LTSA-O instance ending up listing to communication happening within other simulations, which it is not interested of. To alleviate this, the optional field *Topic* can be set to define a simulation specific topic stem. Please note that topics are not allowed to contain commas (,) within them.

Figures A.26, A.29 and A.31 in Appendix A provide examples of complete Manulator configuration files.

7.5 Conductor & Launcher

Our choice of platform to implement a distributed system on an actual distributed hardware setting for testing is *Raspberry Pi* (*Raspberry Pi - Teach, Learn, and Make with Raspberry Pi*, n.d.). Being low cost, small in size and easy to attach sensors via GrovePi kit (*GrovePi Internet of Things Robot Kit*, 2017) are the main reasons behind our choice.

When testing our prototype programs, we had to move among different shells connected to each Raspberry Pi module and manually execute those nodes within a certain time bound, which is a bit bothersome when we have to do it multiple times during debugging.

Our *Conductor* and *Launcher* duo solves this problem for a distributed implementation done on the MQTT protocol. Once again, when making our design decisions, we were conscious of a class setup where multiple groups of students are developing distributed systems sharing the same MQTT broker instance.

7.5.1 Conductor. Conductor, as the name suggests, conducts the whole operation. It relies on a configuration file to start its operation. By default, it looks for the comma separated configuration file named *groups.csv*, which can be overridden with another file name provided as an optional command line argument at startup. The configuration file has the format:

Broker, <MQTT Broker URL>:<Port>

<group 1 name>, <number of nodes>

<group 2 name>, <number of nodes>

. . . , . . .

The format is very similar to other configuration files. First line specifies the MQTT broker to subscribe for. Then there can be any number of lines each specifying a group name and the number of nodes required to be up for the implementation in that group to kick off its operation. None of the group names can contain commas (,). Lines starting with "#" are treated as comments.

Prior to using the launcher, the conductor must be booted up with an appropriate configuration file.

7.5.2 Launcher. Launcher too reads a comma separated configuration file *launcher.csv* to collect the information about the MQTT broker to connect to and the name of the group. This file contains only two lines, and any other comments designated by a starting "#":

Broker, <MQTT Broker URL>:<Port>

<group name>, <number of nodes>

With the configuration file in place, to get the service of the conductor to start the execution of a program, instead of just typing:

$ *python <module name> [command line parameters]*

to start the program, it is executed through the launcher:

$ *python* ***launcher*** *<module name> [command line parameters]*

Then, the program does not start its execution immediately. It waits till the number of nodes submitted to conductor-launcher for execution for a group reaches the number of nodes specified in the conductor configuration file for that group. When the last node for a particular group is submitted, all the nodes automatically start execution in almost the same time.

CHAPTER VIII

CONCLUSION

The paper where the distributed mutual exclusion algorithm for intersection traffic control (Algorithm 2) is published claims that,

> The correctness proof is presented in Section S2 of the supplementary file, available online (Wu et al., 2015a, p. 69).

However, when we try to locate such a supplementary on-line at the address provided, there was no such document. With some effort, we managed to contact the first author of the paper and managed to get a copy of the supplement through him (Wu, Zhang, Luo, & Cao, 2015b).

The supplement proves that the algorithm satisfies safety, liveliness and fairness properties by mathematical plus verbal arguments. While such arguments are necessary to come up with a new algorithm ground up, they sometime fail to completely capture nuances that can go wrong due to some specific message ordering or timing.

With our work, we have demonstrated that relying only on a mathematical and/or verbal proof is not sufficient to provide a seal of correctness when an algorithm comes to life as a program. In Chapters V and VI, through formal modeling of the algorithm, we show several pitfalls of the algorithm that occur when certain conditions are met:

1. If the timeout duration (TO) is less than the time to complete communication ($TTCC$) among cars at an intersection, there is the danger of a crash.

2. When $TO \geq TTCC$,

   (a) if nothing is used to prioritize cars, as per Algorithm 2, cars could end up in a deadlock.

   (b) if only arrival time is used to prioritize cars,

      i. if a car that arrived later is considered having low priority, the algorithm bears the danger of a crash when cars arrive at the same time.

      ii. if a car that arrived at the same time or later is considered having low priority, there is the possibility of a deadlock when cars arrive at the same time.

   (c) if arrival time and a tie-breaker are used (e.g. vehicle id), the algorithm works fine when two cars or less are at the intersection. However, the choice of tie-breaker governs how fair the algorithm is.

   (d) if right turns are allowed at the intersection, the "*Follow List*" logic of the algorithm leads to crashes.

In addition to these, we further showed that setting timeout duration unnecessarily larger than time to complete communication ($TTCC$) reduces the throughput at an intersection.

We claim that in addition to mathematical and verbal argument based proofs, formal modeling can be used to formally verify an algorithm to gain better insight into the dynamics of an algorithm in action. Thus, formal modeling can lead to better algorithm development.

Although we focused on a single algorithm to pick on, our motivation is not just to talk about this one algorithm. We use our exercise of formally analyzing

this algorithm as a medium to start a much broader discussion and urge the smart transportation research community (and in general anybody who is in the business of developing new distributed algorithms), to incorporate formal modeling tools, like LTSA, to verify the new algorithms prior to those algorithms being used in the field. Prevention is always better than cure.

In formally modeling a distributed algorithm, we have to make a decision regarding how detailed the model is going to be. We should pick the middle path where the model is neither too complex (close to implementing the algorithm for real) nor too simple (does not sufficiently capture the nature of the algorithm). The discussion of the modeling process we go through highlights the design decisions we make and how we choose bite sized pieces of the problem to model that shed light into the pieces of the system in such a way we can generalize to the complex system.

With the distributed mutual exclusion algorithm for intersection traffic control (Algorithm 2), the total design space spans from an empty intersection to an intersection where both the queue and core areas of the intersection (*Figure* 1) are fully packed with cars to the maximum possible. Although this number is not infinite, it is quite large and we are unable to exhaustively model everything within a reasonable amount of time or resources. When different arrival times for cars are considered, the number of combinations become even larger.

If a sub system has issues, any super system too has issues. This is one guiding principle we used to make the problem simpler for analysis while allowing us to say something about the larger system. With the intersection mutual exclusion problem, if an intersection with two cars has problems, an intersection with more than two cars too has problems (Chapter V).

Another technique we use to simplify the problem is to model a fast-forwarded state of the system and analyze the algorithm from that point onwards. This way, we do not have to model the whole system and simulate it from start to the fast-forwarded state. However, with this approach, we are unable to comment anything regarding the system's behavior between the start and the fast-forwarded state. We can only comment something like given the system reaches the fast-forwarded state that has been modeled, such and such issues can arise (Chapter VI).

Besides providing formal modeling related insights, along the way, we develop some nifty tools to aid distributed algorithm implementation, validation and testing. Chapter VII presents an overview of all these tools.

8.1 Major Contributions

1. Uncovered specific issues with the distributed mutual exclusion algorithm for intersection traffic control (Algorithm 2) (Wu et al., 2015a). We further provided suggestions to rectify those issues (Chapters V and VI).

2. Demonstrated how to meaningfully break a larger modeling problem into smaller representative problems, which are feasible to model, and gain insight about the bigger problem (Chapters V and VI).

3. Developed a distributed system simulator — *Manulator* (Chapter VII).

4. Extended LTSA tool to come up with *LTSA-O*, which can listen to messages exchanged in an implementation (using MQTT as the messaging protocol) or a simulation of a model in Manulator, animate the model and flag when out of order messages are being passed in the implementation (Chapter VII).

5. Developed *Conductor-Launcher* duo that can be used to automatically start the execution of a distributed system once the desired number of nodes in a system are ready (Chapter VII).

8.2 Distributed VMEI Algorithm - Updated

We amalgamate all the ideas discussed in the thesis and all the insights gained through our models and verifications in to Algorithm 2 to come up with an updated version of the algorithm. To make the listing of the algorithm more modular, we define and use 3 routines in our updated algorithm.

Algorithm 3 is used to determine the priority of a vehicle to pass through the intersection. At the moment, we are using the method we suggested in Chapter V - Section 5.10.1 to determine the priority. Incorporating another priority scheme is just a matter of changing the implementation of this function.

Algorithm 4 determines the set of (1 or 2) vehicles that are responsible for notifying others that a convoy has cleared the core area of the intersection. This algorithm implements the solution we describe to avoid a crash situation created by the "*Follow List*" logic of the original algorithm (Chapter VI - Sections 6.7 and 6.8).

Algorithm 5 checks whether the "*High List*" of a vehicle is empty and if so, generates the "*Follow List*" and broadcasts it as necessary, and makes the vehicle start passing through the core area of the intersection.

We update the message formats to carry additional information required to perform the updated processing. The "*High List*" (HL_i) is made to be a list of triples: (x, $xLid_{in}$, *preempted*). x is the vehicle id and $xLid_{in}$ is the lane that vehicle is approaching the intersection. *preempted* is a Boolean that keeps track whether this vehicle got in the high list by preemption or not. This parameter is

Algorithm 4 Compute the representative vehicles for a convoy

▷ flt — A follow list

▷ Returns the set of vehicles responsible for notifying others
▷ that the convoy has passed through the core area of the intersection

1: **function** ConvoyAmbassadors(flt)

2: $ambas \leftarrow \{\}$
3: $ambas \leftarrow ambas \cup$ last entry of flt
▷ Add the last vehicle of the convoy that is going straight
4: $ambas \leftarrow ambas \cup \{\ (v, vLid_{in}, vLid_{out}) \mid (v, vLid_{in}, vLid_{out}) \in flt \wedge$ it is the last entry of flt such that $vLid_{in} = vLid_{out}\ \}$
5: **return** $ambas$

6: **end function**

Algorithm 5 Check whether the caller can start passing through

▷ HL — High List
▷ NP — Maximum number of vehicles that can join a convoy

▷ Checks whether nobody objects to the calling vehicle passing through
▷ the core area of the intersection and if so start passing through.

1: **function** TryToPassThrough(HL)

2: **if** $(HL = \{\})$ **then**
3: $st_i \leftarrow$ PASSING
▷ Construct the follow list
4: $flt \leftarrow \{\ (v, vLid_{in}, vLid_{out}) \mid (v, vLid_{in}, vLid_{out}) \in LL_i \ \wedge vLid_{in} = iLid_{in} \wedge flt's\ length < NP\ \}$
5: **if** ($flt \neq \{\}$) **then**
6: broadcast FOLLOW(i, flt)
7: **if** ((i, $iLid_{in}$, $iLid_{out}$) $\notin$ ConvoyAmbassadors(flt)) **then**
8: $LL_i \leftarrow \{\}$ ▷ I am not responsible for notifying others
9: **end if**
10: **end if**
11: **end if**
12: move and pass through the core area

13: **end function**

used when deciding to revoke a preemption (line 19 of Algorithm 6). At places where the algorithm ignores whether this parameter is true or false, we signal that by "**???**". We have further made the "*Low List*" (LL_i) a list of triples: (x, $xLid_{in}$, $xLid_{out}$), as required by the algorithm.

Finally, Algorithm 6 joins all these pieces and provides our updated distributed mutual exclusion algorithm for intersection traffic control. Although, we have an extensive discussion about preemptions in Chapter II - Sections 2.4.2.5 and 2.5, since we do not formally model and validate the preemption related logic of the original algorithm, we retain the same preemption logic in the updated algorithm as well.

Although, the updated algorithm addresses and patches some issues that were revealed through our study, as described in Chapter IX - Section 9.1, it needs much more modeling, validation and testing to be used in the field.

Algorithm 6 The distributed VMEI algorithm - updated

▷ xT — Arrival time of the vehicle x
▷ $xLid_{in}$ — Lane vehicle x is approaching from
▷ $xLid_{out}$ — Lane vehicle x is exiting to
▷ Hi_x — High list of the vehicle x
▷ Lo_x — Low list of the vehicle x
▷ flt — Follow list
▷ TH — Maximum number of preemptions allowed

CoBegin //for a vehicle i

On entering the monitoring area:

$st_i \leftarrow$ WAITING_FOR_REJECT
broadcast REQUEST(i, iT, $iLid_{in}$, $iLid_{out}$)
start timer and **wait** for REJECT from others

On receiving REQUEST(u, uT, $uLid_{in}$, $uLid_{out}$ **) from** u**:**

if (iHiPriority(iT, uT, $iLid_{in}$, $iLid_{out}$, $uLid_{in}$, $uLid_{out}$)) **then**
  if ((st_i = WAITING_FOR_REJECT | WAITING_FOR_PERMIT | PASSING) **and**
      ($iLid_{in} = uLid_{in} \vee iLid_{in} \propto uLid_{in}$)) **then**
    if (($\exists k$, (k, $kLid_{in}$, **???**) $\in HL_i \wedge uLid_{in} \cong kLid_{in}$) **and** $CntPmt < TH$) **then**
      $HL_i \leftarrow HL_i \cup$ { (u, $uLid_{in}$, **true**) } ▷ Preempt and let u go first
      $CmtPmt++$
    else
      $LL_i \leftarrow LL_i \cup$ { (u, $uLid_{in}$, $uLid_{out}$) }
      broadcast REJECT(i, $iLid_{in}$, u)
    end if
  end if
end if

▷ Continued on the following page

▷ Continued from the previous page

On receiving REJECT(u**,** $uLid_{in}$**,** k **) from** u**:**

15: **if** (st_i = WAITING_FOR_REJECT | WAITING_FOR_PERMIT) **then**
16: **if** ($i = k$) **then**
17: $HL_i \leftarrow HL_i \cup \{$ (u, $uLid_{in}$, **false**) $\}$
18: **end if**
19: **if** ($i \neq k$ **and** ((k, $kLid_{in}$, **true**) $\in HL_i$) **and**
(($uLid_{in} \propto iLid_{in} \wedge$ (u, $uLid_{in}$, **???**) $\notin HL_i$) $\vee$ $uLid_{in} \cong iLid_{in}$)) **then**
20: delete (k, $kLid_{in}$, **true**) from HL_i ▷ Revoke a preemption
21: broadcast REJECT(i, $iLid_{in}$, k)
22: **end if**
23: **end if**

On receiving PERMIT(u **) from** u**:**

24: delete (u, $uLid_{in}$, **???**) from HL_i
25: **if** (st_i = WAITING_FOR_PERMIT) **then**
26: TryToPassThrough(HL_i)
27: **end if**

On Timeout:

28: $st_i \leftarrow$ WAITING_FOR_PERMIT
29: TryToPassThrough(HL_i)

▷ Continued on the following page

▷ Continued from the previous page

On receiving FOLLOW(u, flt **) from** u**:**

if ((i, $iLid_{in}$, $iLid_{out}$) $\in$ flt) **then**
  $st_i \leftarrow$ PASSING
  if ((i, $iLid_{in}$, $iLid_{out}$) $\notin$ ConvoyAmbassadors(flt)) **then**
    $LL_i \leftarrow \{\}$ ▷ I am not responsible for notifying others that the convoy has cleared the core area
  end if
  move and pass the core area
else if ($iLid_{in} \propto uLid_{in}$) **then**
  delete (u, $uLid_{in}$, **???**) from HL_i
  delete vehicles in flt from HL_i or LL_i
  $HL_i = HL_i \cup$ ConvoyAmbassadors(flt) ▷ Remember the representatives of the convoy
end if

On exiting the intersection:

if ($LL_i \neq \{\}$) **then**
  broadcast PERMIT(i)
end if

CoEnd

CHAPTER IX

FUTURE WORK

9.1 Modeling and Validating the Algorithm

Making the distributed mutual exclusion algorithm for intersection traffic control (Algorithm 2) work without glitches for an intersection with two cars is just the first step. It does not guarantee that the algorithm works well when there are more than 2 cars.

9.1.1 Bigger Models. We need to research more on how to build bigger models of the system. This requires including more cars in the model. Consequently, it requires us to model broadcasts instead of point-to-point communication we have in the current models. Further, with more cars, we must model preemption and follow list related logic. We suggest growing the model slowly and trying a model with 3 cars at the intersection. This would give more insight to build even larger models.

9.1.2 Modeling and Validating Suggested Solutions. During our investigations of the algorithm, we suggested solutions to rectify two problems we discovered.

First, for the algorithm to work smoothly, there should be a prioritization scheme to decide the order of vehicles passing through the core area of the intersection. We suggested using arrival time, whether arriving in the main road or not and whether going straight or turning, in that order to decide the priority (Chapter V - Section 5.10.1).

Second, we demonstrated that the algorithm is prone to crashes if vehicles could take right turns at the intersection. Our suggestion to rectify this issue is to modify the "*Follow List*" related logic of the algorithm. We delegate the last vehicle

and the last vehicle that is going straight on the convoy to notify others when a convoy has exited the core area of the intersection (Chapter VI - Section 6.8).

We are confident that these solutions would solve the problems we found. However, for safety critical systems, relying solely on confidence is not wise. Those suggestions must be formally modeled and verified. By the time we devised those solutions, we ran out of time alloted for this thesis and hence we did not get a chance to formally model them and present our results.

To model the first solution, we have to incorporate the incoming lane and the outgoing lane for each vehicle in the *CAR_P_TIME* model (Chapter V - Listing 24). Since there are 8 lanes (*Figure* 1), if we add two state variables to the model, it would have a 64 fold increase in the state space, which is too much. Instead, since the incoming and outgoing lanes for a car are constants, we can augment the model by introducing two additional model parameters. Further, since our current models only deal with two cars, the upper bound for the number of lanes that could involve in one model is 4. This way, at the time of composing the system at the intersection, we can specify the lanes each car is traveling in. We hope, limiting the model by these means would make the size of the model more manageable.

In addition to introducing new model parameters, the *REQUEST* message should be enhanced to carry the incoming and outgoing lanes of a car. Further, we have to model how a car acts on a received *REQUEST* message by incorporating the priority scheme (Chapter V - Algorithm 3) into the model of the *CAR_P_TIME*.

With the enhanced *REQUEST* message, the network process *VMEI_NETWORK_FIFO* (Chapter IV - Listing 9) should be enriched to handle the new message format.

When modeling the second solution, we should make the *CAR_0* - trouble maker car - (Chapter VI - Listing 30) wait for two *PERMIT* messages instead of one and make both the last car and the last car that is turning right on the convoy broadcast *PERMIT* messages.

9.1.3 Experimenting with Preemptions. In the Chapter II, under Sections 2.4.2.5 and 2.5, we discuss some issues that could arise from the logic of the Algorithm 2 related to preemptions (lines 5 - 7 and lines 17 - 19). We manually build our arguments on handcrafted example scenarios. However, the better approach is to employ formal modeling in such investigations. Within the time frame of this thesis, we did not get the opportunity to build formal models of these situations in LTSA to verify and gain clear insights of the algorithm.

We propose to commence this investigation by modeling the example scenarios depicted in *Figures* 4, 5 and 6. These require bigger models involving at least 3 or 4 cars.

9.1.4 Reliable Communication Channel. The algorithm relies on the assumption that there is no packet loss and the communication channel is 100% reliable (Chapter II - Section 2.2) (Wu et al., 2015a, p. 67). However, this is far from reality.

Therefore, it is a worthwhile effort to gain insight about the behavior of the algorithm in the presence of message loss. One idea to proceed on this direction is to extend the model of the network (LTSA process *VMEI_NETWORK_FIFO* in Listing 9) to drop messages then and there. But with this, we should also extend

the *CAR_P_TIME* model (Listing 25) with a tie-breaker and mechanisms to work with an unreliable channel.

9.1.5 Clock Synchronization. Although the paper where the distributed mutual exclusion algorithm for intersection traffic control is published (Wu et al., 2015a) talks about using arrival time of the vehicles to assign priorities to vehicles to decide the order they pass through the intersection (Chapter II - Section 2.3.2), it never mentions how to measure or record arrival time. With a distributed setting, such as vehicles collaborating to determine the order of passing through the intersection, we lose the notion of a global clock. Instead, each vehicle has its own local clock.

Multiple local clocks tend to run out of synchronization. Using such clocks in a time sensitive system gives rise to the problem of clock synchronization. The paper never acknowledges this problem and how their algorithm (Algorithm 2) performs when clocks are out of synchrony. Although there are many clock synchronization algorithms (Cristian, 1989; Gusella & Zatti, 1989; Mills, 1991), none of them achieves 100% synchronized clocks. They only guarantee that the difference of times indicated by pairs of clocks would be under some upper bound.

Clocks not being synchronized can have negative effects on the operation of the algorithm. For example, algorithm relies on the fact that for two vehicles approaching the intersection on the same lane, the first car must have an earlier arrival time than the second car. This makes the first car have a higher priority, which prevents the second car from entering the intersection before the first car. However, if this order is swapped due to local clocks of these two cars being out of sync, it could lead to a deadlock: The first car waits till the second car passes through the intersection but the second car cannot proceed since the first car is

blocking its way (or if the second car advances without considering the first car in front of it, that could lead to a tail end collision).

Let t_{first} and t_{second} be the arrival times of the first and second cars respectively according to a global clock. Let t_{gap} seconds be the shortest time gap between two cars entering the queue area of the intersection on the same lane. Thus, if the clocks are perfectly synchronized, we have:

$$\begin{aligned} t_{second} &\geq t_{first} + t_{gap} \\ t_{first} &\leq t_{second} - t_{gap} \end{aligned} \tag{9.1}$$

Now suppose, due to a clock error, the clock of the second car runs t_{error} seconds slower. Therefore, when the second car enters the queue area of the intersection, it records $t_{second} - t_{error}$ as its arrival time instead of t_{second}. The aforementioned problems arise if:

$$t_{second} - t_{error} < t_{first} \tag{9.2}$$

Combining inequalities (9.1) and (9.2), we get,

$$\begin{aligned} t_{second} - t_{error} &< t_{second} - t_{gap} \\ t_{gap} &< t_{error} \end{aligned} \tag{9.3}$$

As per inequality (9.3), if the clocks go out of sync more than the shortest time gap between two cars entering the queue area of the intersection on the same lane, it would be problematic for the correct operation of the algorithm. Therefore, prior to implementing this algorithm for real world use, a careful investigation on such timing issues needs to be performed.

9.2 Manulator

Although, the simulation code written for the Manulator is closer to an implementation than the LTSA model, it still needs to be modified to run directly on hardware without the Manulator. This transformation too could introduce

errors in the implementation that were not present in the simulation. To eliminate such errors, we are in the process of enhancing and extending Manulator to enable the simulation code (possibly with minimal modifications) to be directly executed on actual hardware.

9.3 Enhancing the LTSA-O

LTSA tool animates the finite state graphs of component processes of a system. Although, it produces a finite state graph for the composition, it is not animated. Since the graph of the composition is the graph representative of the final system, we believe animating that graph is very important.

9.4 Game Tolerant Systems

Homo sapiens are a very clever group of species. They are so clever that they find loopholes in everything so that things can be turned into their advantage.

With respect to the distributed mutual exclusion algorithm for intersection traffic control, we asked the question, can a human user who takes the service of an autonomous vehicle that operates a "perfect" and "error free" implementation of the algorithm, game the system to take an unfair advantage? For example, even today, systems that boast unbreakable get hacked.

The algorithm relies on arrival time (and a tie-breaker) to prioritize and decide which car passes through an intersection first. In the fast-paced world, the owners and passengers of these autonomous vehicles are motivated to reach their destination as quickly as possible. Therefore, there is good reason to believe that at least some humans could try to customize their vehicles so that they could get an unfair higher priority to make their commute shorter. For example, what if somebody can just tamper with the local clock of the car to run a little slower and make their car broadcast a lesser time-stamp than others having the correct

time. This way a perfect algorithm (according to our current notion of being perfect) gives an advantage to such a person and he or she can game the system. Another example could be somebody wanting to make social havoc making a vehicle broadcast erroneous information to cause a crash at an intersection.

Even without any human intervention, electronic components could malfunction and start sending erroneous information. Further, a virus attack can lead to much more wide-spread and long lasting disruptions.

Thinking along these problems, we raised the question, even after deployment, can we utilize the model with our LTSA-O tool to keep an eye on the behavior of the system and alert anomalies in real-time to some monitoring authority that can take necessary action in a timely manner?

With the speed of today's computers and communication systems, reaction time of a human to such an alert could be very slow. By the time some resolution is in place, there could already be a lot of damage taken place. Therefore, we thought one more step ahead and asked the question, in addition to making the LTSA-O alert a human, can we make it directly alert the nodes of the distributed system and can we make the distributed system sufficiently game tolerant so that it can automatically take necessary actions to isolate the malfunctioning node and continue its regular operation?

We have already conducted some preliminary experiments in this line and gained some success. However, the results are not mature enough to present as an individual chapter of the thesis. Hence, we briefly mention our efforts here.

We experimented with the Chang-Roberts leader election algorithm for a ring topology (Chang & Roberts, 1979). According to the algorithm, the nodes run an election by sharing their node ids with others in the ring and the node with the

highest node id gets elected as the leader of the ring (Appendix A.1, A.2, A.3 and A.4).

The first question we asked is, can a node with a lesser id game the algorithm by tricking all the other nodes to believe that it is the leader? We came up with the answer as "*Yes*" by creating a bad node in the ring that can always become the leader of the ring irrespective of its id (Appendix A.6).

The second question was, can LTSA-O alert a human about the bad node? This was an easy question since LTSA-O already followed the execution of the implementation and matched it with the model. Thus, just the moment the bad node is about to game the system LTSA-O senses it immediately (Appendix A.7).

This leads to the subsequent question, can we make LTSA-O alert the system in execution about the bad node in real-time. At the time we were asking this question, LTSA-O was just a passive observer. It did not interfere with the conversation the nodes were having in the implementation.

We made the main thread of the Manulator in a simulation or each node in an implementation subscribe to a special topic *bnode_topic* at the MQTT broker. Then, whenever LTSA-O sees a bad node it publishes the identity of the bad node to the same topic at the same broker. We added the additional field:

bnode_topic, <Topic to reveal bad nodes>

to all the relevant configuration files so that all the parties share a common topic to discuss about bad nodes.

In a simulation, the main monitor thread gets the information about a bad node, which it then broadcasts to all the node processes in the simulation. In an implementation, each node gets the information directly from the broker.

Next we asked, can we enhance the algorithm to act on a bad node message, ostracize the bad node, recreate the ring without the bad node and elect the node that deserves the leadership. We are successful in this level as well (Appendix A.8).

This is where we have stopped at the moment. The moment the system in execution carves out the bad node and re-creates a new ring, the topology of the ring in the system in execution diverges from that of the static model LTSA-O has. With the current system, after catching the first bad node, the model with LTSA-O becomes outdated and hence is unable to properly synchronize with the new state of the system in action. If we can make LTSA-O more dynamic and re-compose a model eliminating the bad node, we might be able to make the model keep up with the changing topology of the system in execution. These are research questions not yet answered.

Moreover, showing that our ideas work with Chang-Roberts leader election does not generalize them to other algorithms. Can we make any distributed algorithm achieve this new sense of game tolerance? Can we build some theory or guideline so that anybody can follow and make any or many distributed algorithms robust at this level? These are more higher level questions that need more time and experiments to find solutions.

APPENDIX

MANULATOR AND LTSA-O - AN EXAMPLE

Here, we are providing a complete example of the *Manulator* and *LTSA-O* in operation. Along with the example, a commentary about what is happening is provided. We model and simulate the Chang-Roberts leader election algorithm for a ring topology (Chang & Roberts, 1979) in this example.

A.1 LTSA Model of a Chang-Roberts Node

We present a LTSA model of a node that executes the Chang-Roberts leader election algorithm. Listing 34 provides constants, ranges and sets used in the model. In this listing, changing the constant *NODES* creates rings with different number of nodes. Listing 35 for *CHANG_NODE* models the Chang-Roberts algorithm for a single node. In our models, we use the convention of prefixing internal node actions, which are not captured by the messaging layer, with a long tail of underscores (_____________). *LTSA-O* is unable to sense these actions. Composing 4 *CHANG_NODE*s into a ring topology is described in Listing 36. It creates a ring with 4 nodes with node ids $0, 1, 2, 3 : 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 0$. Finally, Listing 37 provides asserts to validate whether the model works per the definition of the algorithm. It further defines what progress means in our composition.

A.2 Implementing Chang-Roberts in Manulator

After having a validated model, it is time to implement it in the Manulator. To make it easy for the reader to map the connection between the model (Listing 35) and the implementation (Listing 38), we try to keep the implementation as close as possible to the model (same names for states, functions and variables,

Listing 34 Constants, sets and ranges used in Chang-Roberts LTSA models.

```
const False    = 0
const True     = 1
range Bool     = False..True

// Node IDs
const NODES    = 4   // Changes the number of nodes
range IDS      = 0..NODES - 1

// Message types
const ID       = 100
const LEADER   = 200
set MESSAGES   = { [ID], [LEADER] }
```

and a similar structure). Suppose that this code is stored in the python module *chang_node.py* (this is the file name where the code is stored).

A.3 Simulating Chang-Roberts in Manulator

The only thing that is missing to observe how our implementation behaves when executed is the Manulator configuration file (*Figure* A.26). The two command line parameters for each *chang_node* are: <*node id*> <*downstream node id*>. Please note that we have included the optional fields: *Timeout*, *Broker*, *Topic* and *Bnode_Topic* to provide a complete picture of the configuration file. *Timeout* defines the length of the maximum period without any message being passed. Manulator uses this to notify of such periods to the user. Other optional parameters are used to connect the simulation running in Manulator with LTSA-O. Also, assume that the configuration file is stored in a file named *nodes.csv*. If this is stored in a file with another name, the name of that file should be provided as the first command line argument to Manulator.

Listing 35 Model of a node that performs Chang-Roberts leader election

```
1  CHANG_NODE( NODE_ID = 1 ) = ACTIVE[ False ],

2  ACTIVE[ s: Bool ] =
3    (
4        when( s == False )
5             send   [   ID   ][ NODE_ID  ]
6                                  -> ACTIVE[   True   ]

7      |      receive[   ID   ][ nid: IDS ]
8                                  -> DECIDE[ nid ][ s ]

9      | when( s == True)
10            receive[ LEADER ][ nid:IDS  ]
11         -> send   [ LEADER ][   nid    ]
12                                 -> WORKING[   nid   ]
13     ),

14 DECIDE[ nid: IDS ][ s: Bool ] =
15   (
16       when( nid < NODE_ID )
17            ____________drop[ID][nid]
18                                 -> ACTIVE[    s     ]

19     | when( nid > NODE_ID )
20            send   [   ID   ][   nid    ]
21                                 -> PASSIVE

22     | when( nid == NODE_ID )
23            ____________i_am_leader
24                                 -> ANNOUNCE
25   ),
```

```
26 PASSIVE =
27   (
28             receive[   ID   ][ nid: IDS ]
29         -> send   [   ID   ][   nid    ]
30                                 -> PASSIVE
31
32       |     receive[ LEADER ][ nid: IDS ]
33         -> send   [ LEADER ][   nid    ]
34                                 -> WORKING[   nid   ]
35   ),
36
37 ANNOUNCE =
38   (
39             send   [ LEADER ][ NODE_ID  ]
40                                 -> WAITFORROUNDTRIP
41   ),
42
43 WAITFORROUNDTRIP =
44   (
45             receive[ LEADER ][ NODE_ID  ]
46                                 -> WORKING[ NODE_ID ]
47   ),
48
49 WORKING[ nid: IDS ] =
50   (
51     ____________working_for[ nid ]
52                                 -> WORKING[   nid   ]
53   ).
```

Listing 36 Composing the ring of *CHANG_NODE*s

```
1 || RING =
2     (
3       node[ n: IDS ]:CHANG_NODE( n )
4     )
5       /
6         {
7           node[n:IDS].send / node[(n+1)%NODES].receive
8         }.
```

Listing 37 Making sure the model of Chang-Roberts composition work properly

```
// Check whether the node with the largest node id
// eventually becomes the leader
assert Largest_is_leader =
        <>  ( node[ NODES-1 ]._____________i_am_leader )

// Check whether no other node becomes the leader
// along with the node with the largest id
assert Only_one_leader =
        []! (
              exists[ i: 0..NODES-2 ]
                ( node[ i ]._____________i_am_leader &&
                  node[NODES-1]._____________i_am_leader
                )
              )

// Check whether all nodes eventually work for the
// leader elect - the node with the highest id
assert Everyone_works =
  forall[ i: IDS ]
    <> (
          node[ i ]._____________working_for[ NODES-1 ]
        )

// Check whether the nodes always do not have to
// send their id for the algorithm to work
// This assert must be violated
assert Always_send_my_id =
                  forall[ i: IDS ]
                      <> ( node[ i ].send[ ID ][ i ] )

// In this system, progress means everybody ends up
// in the cycle of working for the leader.
progress ALL_WORKING =
    {
      node[ IDS ]._____________working_for [ NODES-1 ]
    }
```

Listing 38 Implementation of a *CHANG_NODE* (Listing 35) as a Node for Manulator. Suppose that this code is stored in the python module *chang_node.py*

```
1  import time
2  from Communications import Communications
3  from multiprocessing import Lock

4  class Node( Communications ):

5      def initialize_node( self, argc, argv ):
6          # argv[ 0 ] is the module name
7          # argv[ 1 ] is the node id
8          self.me = argv[ 1 ]
9          self.next = argv[ 2 ]
10         self.state = 'NOT ACTIVE'
11         self.lock = Lock()

12     def main( self ):

13         time.sleep( 1 )

14         with self.lock:

15             self.log( '[Node {}]\t{}'.
16                         format( self.me, self.state ))

17             if ( self.state == 'NOT ACTIVE' ):
18                 self.log( ('[Node {}]\tSending my id'
19                         'to {} for the 1st time').
20                         format( self.me, self.next ))

21                 self.send( 'ID {}'.
22                         format( self.me ), self.next )

23                 self.state = 'ACTIVE'
```

```
24      def on_msg( self, sender, msg ):
25          msg_parts = msg.split()
26          time.sleep( 1 )
27          with self.lock:
28              self.log( '[Node {}]\t{}'.
29                          format( self.me, self.state ))
30              if ( self.state == 'WAITFORROUNDTRIP' ):
31                  if ( msg_parts[0] == 'LEADER'
32                          and msg_parts[1] == self.me ):
33                      self.log( ('[Node {}]\tEverybody'
34                          'knows I am the boss').
35                          format( self.me ))
36                      self.state = 'WORKING'
37              elif ( self.state != 'WORKING' ):
38                  if ( self.state == 'PASSIVE' ):
39                      self.PASSIVE( msg )
40                  elif ( msg_parts[0] == 'ID' ):
41                      self.DECIDE( msg )
42                  # msg_parts[0] == 'LEADER'
43                  # and I am not the leader
44                  elif ( self.state == 'ACTIVE' ):
45                      self.send( msg, self.next )
46                      self.state = 'WORKING'
47                      self.log( '[Node {}]\t{}'.
48                          format( self.me, self.state ))
```

```
49      # Decide whether I am the leader or not
50      def DECIDE( self, msg ):
51          msg_parts = msg.split()
52          nid = msg_parts[ 1 ]

53          if ( int( nid ) > int( self.me )):
54              # I am not the leader
55              self.send( msg, self.next )
56              self.state = 'PASSIVE'

57          elif ( nid == self.me ):
58              # I am the leader
59              self.ANNOUNCE()

60      # I know that I am not the leader
61      # corporate with the election
62      def PASSIVE( self, msg ):
63          msg_parts = msg.split()
64          msg_type = msg_parts[ 0 ]

65          self.send( msg, self.next )

66          if ( msg_type ==  'LEADER' ):
67              # I have a new leader it is not me
68              self.state = 'WORKING'
69              self.log( '[Node {}]\t{}'.
70                          format( self.me, self.state ))

71      # Let everybody know that I am the boss
72      def ANNOUNCE( self ):
73          self.log( '[Node {}]\tI am the leader'.
74                          format( self.me ))

75          self.send( 'LEADER {}'.
76                          format( self.me ), self.next )

77          self.state = 'WAITFORROUNDTRIP'
```

```
# Nodes in the simulation
 chang_node,        0,                  1
 chang_node,        1,                  2
 chang_node,        2,                  3
 chang_node,        3,                  0
 Timeout,           3
# Broker to subscribe for
 Broker,            iot.eclipse.org:1883
# Topic Manulator publishes messages
# so that LTSA-O can eavesdrop
 Topic,             demo
# Topic LTSA-O notifies about nodes
# sending out of sequence messages
 Bnode_Topic,       bad_node
- - - Anything below this line is ignored
Manulator ignores anything below a line starting with a -
```

Figure A.26. An example Manulator configuration file to simulate a ring of 4 nodes that run the Chang-Robert leader election algorithm

With the Manulator configuration file in place, running the simulation is just a matter of executing the command:

$ *python manulator*

With this, Manulator reads the configuration file, creates the required amount of node processes and runs the simulation. While the simulation is running, anything logged using the `self.log()` command can be seen on the console in addition to being stored in the per node log file. At any time during the simulation, if the return key is pressed on the simulation console, the simulation terminates.

At the end of the simulation, per node log files and the Manulator log file can be inspected to see what happened in each node and in the simulation. These become vital when it comes to debugging.

A.4 Bringing LTSA-O to the Scene

Making LTSA-O eavesdrop on the messages exchanged in the simulation running in the Manulator is just a matter of completing the LTSA-O configuration file (*Figure* A.27) and commanding LTSA-O to validate the incoming stream of messages. As described in Chapter VII - Section 7.3, we can get the skeleton of the configuration file and then populate the missing information according to the specifics of the implementation. Note that this configuration file uses the same broker as in the Manulator configuration file (*Figure* A.26).

After populating the configuration file, we should load the configuration file to LTSA-O by using the "*Open Config*" button on the "*Observer & Validator*" dialog (*Figure* 25). Then by clicking the "*Validate*" button on the same dialog, LTSA-O starts listening to the message stream published by the Manulator to the broker. When messages start streaming in, LTSA-O animates the model accordingly. Whenever it sees an out of order message, it notifies that to the user and also publishes the message and topic where LTSA-O received the out of order message to the "*Bnode_ Topic*" specified in the configuration file. LTSA-O uses the message format: *<Topic where the out of order message has been published to>*, *<Payload of the out of order message>* to publish this information to the broker. Via this channel, Manulator gets to know about the bad node and it passes this information to all the node processes.

```
Broker,         iot.eclipse.org:1883
Bnode_Topic,    bad_node
Index,          Action,                 Topic,      Message
1,              node.0.send.100.0,      demo/1,     ID 0
2,              node.3.send.100.0,      demo/0,     ID 0
3,              node.3.send.100.1,      demo/0,     ID 1
4,              node.3.send.100.2,      demo/0,     ID 2
5,              node.3.send.100.3,      demo/0,     ID 3
6,              node.3.send.200.0,      demo/0,     LEADER 0
7,              node.0.send.200.0,      demo/1,     LEADER 0
8,              node.3.send.200.1,      demo/0,     LEADER 1
9,              node.0.send.200.1,      demo/1,     LEADER 1
10,             node.3.send.200.2,      demo/0,     LEADER 2
11,             node.0.send.200.2,      demo/1,     LEADER 2
12,             node.3.send.200.3,      demo/0,     LEADER 3
13,             node.0.send.200.3,      demo/1,     LEADER 3
```

Figure A.27. An example LTSA-O configuration file to make LTSA-O listen to communication taking place in the simulation of Chang-Roberts leader election algorithm with 4 nodes. The contents of the file span 4 pages.

14,	node.0._____________i_am_leader,	local,	Node 0 is the leader
15,	node.0.send.100.1,	demo/1,	ID 1
16,	node.0.send.100.2,	demo/1,	ID 2
17,	node.0.send.100.3,	demo/1,	ID 3
18,	node.0._____________working_for.0,	local,	Node 0 working for 0
19,	node.0._____________working_for.1,	local,	Node 0 working for 1
20,	node.0._____________working_for.2,	local,	Node 0 working for 2
21,	node.0._____________working_for.3,	local,	Node 0 working for 3
22,	node.1.send.100.1,	demo/2,	ID 1
23,	node.1.send.200.0,	demo/2,	LEADER 0
24,	node.1.send.200.1,	demo/2,	LEADER 1
25,	node.1.send.200.2,	demo/2,	LEADER 2
26,	node.1.send.200.3,	demo/2,	LEADER 3
27,	node.1._____________drop.100.0,	local,	Node 1 drop ID 0
28,	node.1._____________i_am_leader,	local,	Node 1 is the leader

```
29,     node.1.send.100.2,                          demo/2,     ID 2
30,     node.1.send.100.3,                          demo/2,     ID 3
31,     node.1.send.100.0,                          demo/2,     ID 0
32,     node.1._____________working_for.0,          local,      Node 1 working for 0
33,     node.1._____________working_for.1,          local,      Node 1 working for 1
34,     node.1._____________working_for.2,          local,      Node 1 working for 2
35,     node.1._____________working_for.3,          local,      Node 1 working for 3
36,     node.2.send.100.2,                          demo/3,     ID 2
37,     node.2.send.200.0,                          demo/3,     LEADER 0
38,     node.2.send.200.1,                          demo/3,     LEADER 1
39,     node.2.send.200.2,                          demo/3,     LEADER 2
40,     node.2.send.200.3,                          demo/3,     LEADER 3
41,     node.2._____________drop.100.0,             local,      Node 2 drop ID 0
42,     node.2._____________drop.100.1,             local,      Node 2 drop ID 1
43,     node.2._____________i_am_leader,            local,      Node 2 is the leader
```

```
44,  node.2.send.100.3,                          demo/3,  ID 3
45,  node.2.send.100.0,                          demo/3,  ID 0
46,  node.2.send.100.1,                          demo/3,  ID 1
47,  node.2._____________working_for.0,          local,   Node 2 working for 0
48,  node.2._____________working_for.1,          local,   Node 2 working for 1
49,  node.2._____________working_for.2,          local,   Node 2 working for 2
50,  node.2._____________working_for.3,          local,   Node 2 working for 3
51,  node.3._____________drop.100.0,             local,   Node 3 drop ID 0
52,  node.3._____________drop.100.1,             local,   Node 3 drop ID 1
53,  node.3._____________drop.100.2,             local,   Node 3 drop ID 2
54,  node.3._____________i_am_leader,            local,   Node 3 is the leader
55,  node.3._____________working_for.0,          local,   Node 3 working for 0
56,  node.3._____________working_for.1,          local,   Node 3 working for 1
57,  node.3._____________working_for.2,          local,   Node 3 working for 2
58,  node.3._____________working_for.3,          local,   Node 3 working for 3
```

Since nodes *CHANG_NODE* adhere to the Chang-Roberts protocol, LTSA-O shows a perfect action trace without any errors (*Figure* A.28). By clicking the *Replay* button on the *Observer & Validator* dialog, the last run recorded by LTSA-O can be replayed. This comes handy for debugging.

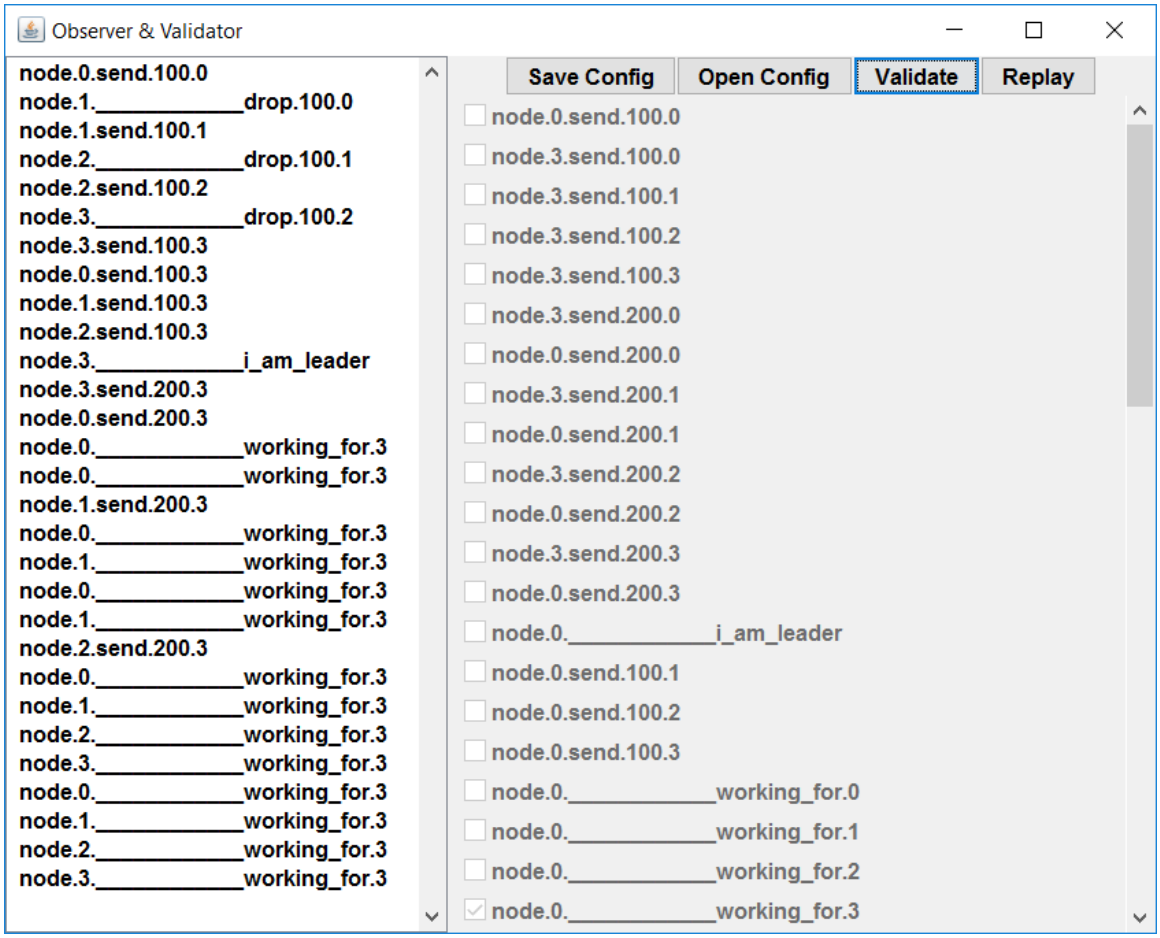

Figure A.28. LTSA-O action trace for a well behaved Chang-Roberts system

A.5 Summery of Steps to Run and Validate a Simulation

1. In LTSA-O, model the components of the system, compose the system and validate the model.

2. After the model is satisfactory, implement it in Manulator.

3. Create the Manulator configuration file *nodes.csv* (default name - or use any other name) to configure the simulation.

4. Compose the model in LTSA-O and generate the LTSA-O configuration file (*Observer & Validator - Save Config*).

5. Populate the LTSA-O configuration file according to the implementation.

6. Load the LTSA-O configuration file to LTSA-O (*Observer & Validator - Open Config*).

7. Make LTSA-O listen to the messages exchanged in the simulation (*Observer & Validator - Validate*).

8. Run the simulation in the Manulator (*$ python manulator*).

9. Observe the model in LTSA-O being animated according to the messages exchanged in the simulation in execution.

10. To debug the implementation

    (a) Replay the message sequence in LTSA-O and see what happened (*Observer & Validator - Replay*).

    (b) Examine the per node and manulator log files.

A.6 Meglomaniac Node

Since *CHANG_NODE* is well behaved, LTSA-O does not complain about the simulation governed by the Manulator configuration in *Figure* A.26. To demonstrate how LTSA-O behaves on an out of order message, we create a meglomaniac node that abuses the Chang-Roberts protocol and always becomes the leader of the ring irrespective of its node id.

In the LTSA model for the well behaved node, *CHANG_NODE*, (Listing 35), we should only change lines 14 - 18 from:

```
14 DECIDE[ nid: IDS ][ s: Bool ] =
15   (
16       when( nid < NODE_ID )
17            ____________drop[ID][nid]
18                                -> ACTIVE[    s     ]

19     | when( nid > NODE_ID )
20            send   [   ID   ][   nid    ]
21                                -> PASSIVE

22     | when( nid == NODE_ID )
23            ____________i_am_leader
24                                -> ANNOUNCE
25   ),
```

to:

```
  DECIDE[ nid: IDS ][ s: Bool ] =
    (
      ____________i_am_leader   -> ANNOUNCE
    ),
```

to come up with a power hungry node, which we call the "*BAD_NODE*".

By composing a *BAD_NODE* with 3 other *CHANG_NODE*s (Listing 35), we create a "*BAD_RING*" (Listing 39). In this composition, the node with id 0 abuses and games the protocol to always become the leader.

We can validate that node 0 always becomes the leader and no other node becomes the leader with the aid of asserts (Listing 40).

Implementing the bad node for Manulator is just a matter of changing the `def DECIDE(self, msg)` method (lines 47 - 56) of Listing 38 as:

Listing 39 Composing a ring with a *BAD_NODE* and 3 other *CHANG_NODE*s

```
|| BAD_RING =
    (
         node[    0    ]:BAD_NODE   ( 0 )
      || node[ n: 1..3 ]:CHANG_NODE( n )
    )
      /
        {
          node[n:IDS].send / node[(n+1)%NODES].receive
        }.
```

```
    def DECIDE( self, msg ):
        # I am the leader
        self.ANNOUNCE()
```

Assume that the code for the bad node is stored in the python module *bad_node.py* (this is the file name where the code is stored). Therefore, we can update the Manulator configuration file (*Figure* A.26) to simulate the *BAD_RING* composition (Listing 39) as in *Figure* A.29. Note that the only change is to the 2nd line of the earlier configuration file. This is how easy it is to simulate systems in Manulator.

A.7 Catching the *BAD_NODE* Red-handed

When LTSA-O with the model for the *ring with 4 CHANG_NODE*s (**model should be the correct ring without the bad node**) (Listing 36) listens to the communication taking place in the simulation of the *BAD_RING*, just the moment the bad node sends the message *i_am_leader*, LTSA-O notifies that to the user (*Figure* A.30). After the first out of order message, the sequence of messages exchanged in the simulation with the bad node does not align with the sequence of messages anticipated by the correct model (Listing 36). Therefore,

Listing 40 Making sure the *BAD_NODE* always becomes the leader

```
1  // Check whether the node with id 0
2  // eventually becomes the leader
3  assert Zero_is_leader =
4          <>  ( node[ 0 ]._____________i_am_leader )

5  // Check whether no other node becomes the leader
6  // along with the node with id 0
7  assert Zero_is_the_only_leader =
8          []! (
9                exists[ i: 1..NODES ]
10                 ( node[ 0 ]._____________i_am_leader &&
11                   node[i]._____________i_am_leader
12                 )
13               )

14 // Check whether all nodes eventually work for the
15 // leader elect - the node with id 0
16 assert All_work_for_Zero =
17   forall[ i: IDS ]
18     <> (
19           node[ i ]._____________working_for[ 0 ]
20         )

21 // Check that node 3 never becomes the leader
22 // This assert must be violated
23 assert Three_never_become_leader =
24     []! ( node[ 3 ].send[ LEADER ][ 0 ] )

25 // In this system, progress means everybody ends up
26 // in the cycle of working for the leader.
27 progress ALL_WORKING =
28     {
29       node[ IDS ]._____________working_for [ NODES-1 ]
30     }
```

```
# Nodes in the simulation
 bad_node,          0,                  1
 chang_node,        1,                  2
 chang_node,        2,                  3
 chang_node,        3,                  0
 Timeout,           3
# Broker to subscribe for
 Broker,            iot.eclipse.org:1883
# Topic Manulator publishes messages
# so that LTSA-O can eavesdrop
 Topic,             demo
# Topic LTSA-O notifies about nodes
# sending out of sequence messages
 Bnode_Topic,       bad_node
- - - Anything below this line is ignored
Manulator ignores anything below a line starting with a -
```

Figure A.29. Manulator configuration file to simulate the *BAD_RING* (Listing 39).

LTSA-O does not respond to any further messages it receives after the first out of order message.

A.8 Making the *CHANG_NODE* Game Tolerant

We did some initial experiments to try to make the node game tolerant by acting upon the bad node notification received from LTSA-O. As described in Chapter IX - Section 9.4, we were successful in acting on the first bad node message and repairing the ring by eliminating the bad node.

We updated the implementation of the `def initialize_node( self, argc, argv )` and `def on_msg( self, sender, msg )` methods (lines 5 - 11 and

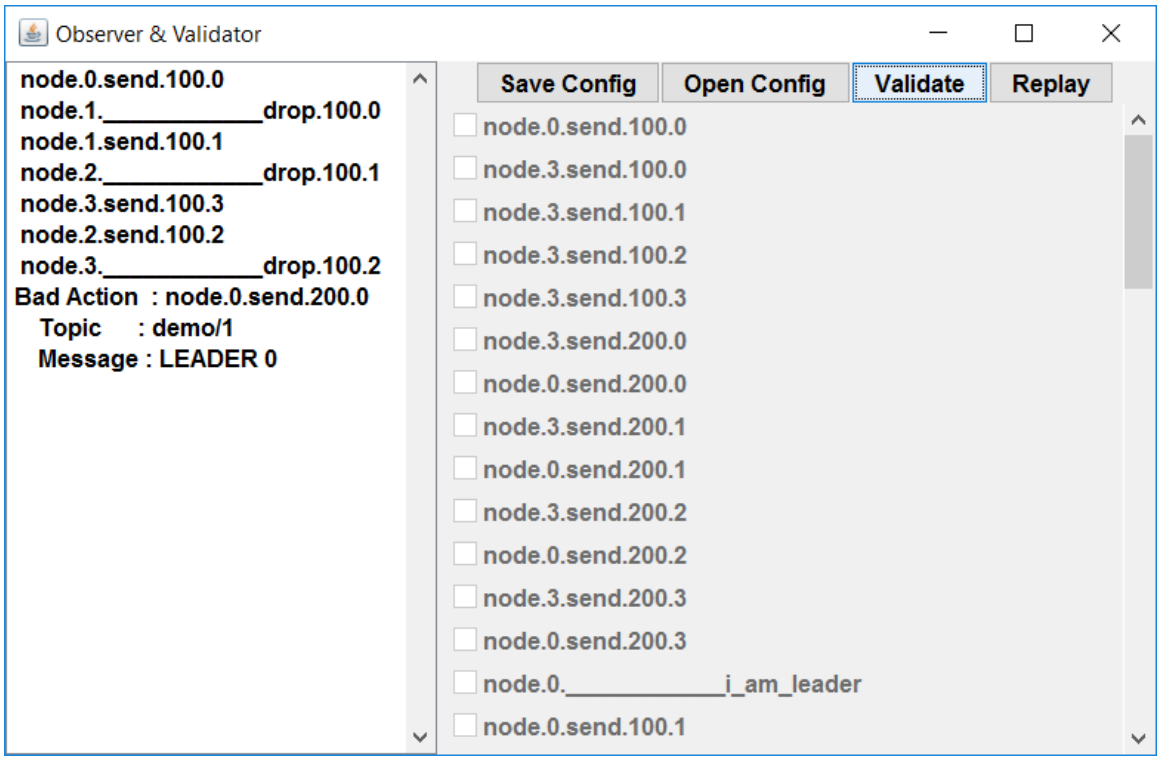

Figure A.30. LTSA-O notifying an out of order message (*Observer & Validator* dialog)

24 - 48) of the implementation of the *CHANG_NODE* (Listing 38) with the logic of the updated protocol that dynamically creates a new ring without the bad node (Listings 41 and 42). Suppose that the coding for the updated node is stored in python module *chang_node_gt.py*.

For our updated protocol to work, a node not only needs to know the id of its downstream node but also its upstream node. This is taken care of by the updated `def initialize_node( self, argc, argv )`. `def on_msg( self, sender, msg )` is updated to process the additional messages involved in reconfiguring the ring. Further, the updated node remembers the ids of all the reported bad nodes so that it can ignore further messages received from such nodes.

Listing 41 Updated *initialize_node* method of Listing 38 that makes the node game tolerant. Suppose that the coding for the updated node is stored in python module *chang_node_gt.py*

```
def initialize_node( self, argc, argv ):
    # argv[ 0 ] is the module name
    # argv[ 1 ] is the node id
    self.me   = argv[ 1 ]
    self.prev = argv[ 2 ]
    self.next = argv[ 3 ]
    self.state = 'NOT ACTIVE'
    self.lock = Lock()

    # List of known bad nodes
    self.bad_nodes = set()
```

The Manulator configuration file to simulate a ring of 3 game tolerant Chang-Roberts nodes along with a bad node is given in *Figure* A.31. The command line parameters to the *chang_node_gt* are: *<node id>*, *<upstream node id>* *<down stream node id>*.

Although our updated node is robust enough to handle any number of bad nodes by shrinking the ring appropriately all the way till there is only one node is left in the ring, LTSA-O is unable to dynamically update the model to reflect the eliminated bad nodes. Therefore, after reporting the first bad node, LTSA-O does not provide any meaningful responses.

Since we are still using the composition in Listing 36 that builds a ring of *CHANG_NODES*, it does not know the behavior of the updated game tolerant node. Therefore, LTSA-O considers all the messages related to ring update process and the elections afterwards as out of order messages and report them as bad nodes (*Figure* A.32). If all such messages are delivered to node processes, the ring ends

Listing 42 Updated *on_msg* method of Listing 38 that makes the node game tolerant. Suppose that the coding for the updated node is stored in python module *chang_node_gt.py*

```
1   msg_parts = msg.split()

2   if ( sender == '-1' ):
3     # This is a bad node message from the Manulator
4     with self.lock:
5         # Stop all other work
6         self.state = 'GAME_TOLERENCE'

7     # Decode the message
8     # 1. Bad node message := <receiver> <message>
9     # 2. <message> := <message type> <sender id>
10    # 1 = Format Manulator reports bad nodes
11    # 2 = Message format used by the Node
12    # Add this node to the set of bad nodes
13    self.bad_nodes.add( msg_parts[2] )

14    if ( msg_parts[ 2 ] == self.prev ):
15      # My upstream node is the bad node
16      if ( msg_parts[ 2 ] == self.next ):
17        # I am the only good node left
18        self.log( '{} > I am the only good node left'.
19                                        format( self.me ))
20      else:
21        # Notify the node who is sending messages to
22        # this bad node to re-route them to me
23        self.log( 'My upstream node {} is bad'.
24                                      format( self.prev ))
25        # Send a reconnect message
26        self.send( 'R {} {}'.
27                   format( self.me, self.prev ),
28                   self.next )
29    return
```

```
41    # Process only the messages from good nodes
42    if not ( msg_parts[1] in self.bad_nodes ):
43      time.sleep( 1 )
44      if ( msg_parts[ 0 ] == 'R' ):
45        if ( self.next == msg_parts[ 2 ] ):
46          # My downstream node is bad
47          # Update my downstream node
48          self.next = msg_parts[1]
49          # Notify all about the updated ring
50          self.send( 'NR {} {}'.
51                  format( self.me, self.next ),
52                  self.next )
53        else:
54          self.send( msg, self.next )
55      elif ( msg_parts[ 0 ] == 'NR' ):
56        if ( self.me == msg_parts[ 2 ] ):
57            # I have a new upstream node
58            self.prev = msg_parts[ 1 ]
59            # Forward the message
60            self.send( msg, self.next )
61        elif ( self.me == msg_parts[ 1 ] ):
62            # Everybody knows about the updated ring.
63            # Start another election
64            self.send( 'ID {}'.
65                    format( self.me ), self.next )
66        else:
67            self.send( msg, self.next )
68        with self.lock:
69            # Get ready for new election
70            self.state = 'ACTIVE'
71      else:
72        # Copy lines 27 - 28 of Listing 38 (chang_node.py)
```

```
# Nodes in the simulation
 bad_node,          0,                    1
 chang_node_gt,     1,                    0,    2
 chang_node_gt,     2,                    1,    3
 chang_node_gt,     3,                    2,    0
 Timeout,           3
# Broker to subscribe for
 Broker,            iot.eclipse.org:1883
# Topic Manulator publishes messages
# so that LTSA-O can eavesdrop
 Topic,             demo
# Topic LTSA-O notifies about nodes
# sending out of sequence messages
 Bnode_Topic,       bad_node
```

Figure A.31. An example Manulator configuration file to simulate a ring of 4 nodes that runs the Chang-Roberts leader election algorithm

up degenerating to just a single node. Hence, we instrumented Manulator to report only the first bad node message to the node processes. However, the action trace in *Figure* A.32 shows that the efforts node 0 took toward gaming the system and becoming the leader have become fruitless, and the rightful leader, node 3, has issued a *i_am_leader* message. The log of the Manulator clearly shows that node 3 becomes the leader of the new ring.

One idea to try is to model our game tolerant node, compose a ring out of them and see how that responds to the ring being updated. However, we have yet to get our hands dirty with this modeling process.

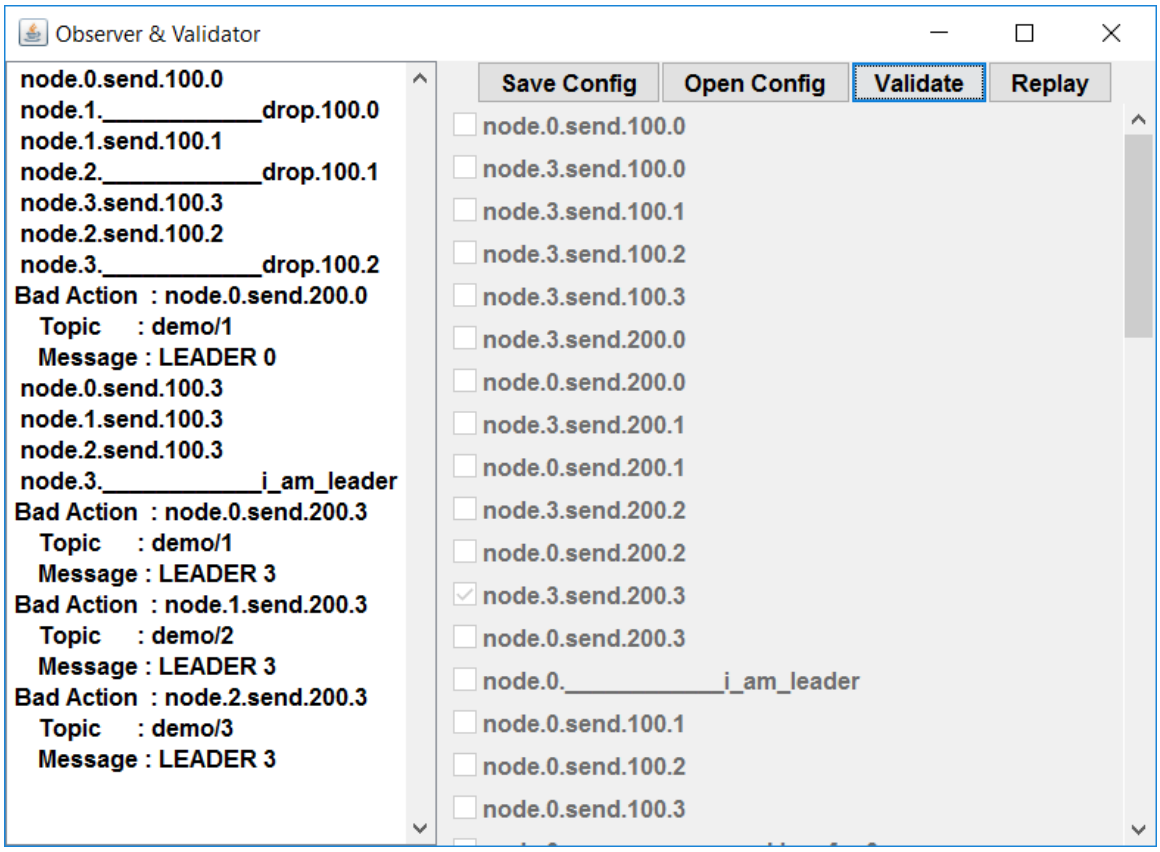

Figure A.32. LTSA-O notifying an out of order message and responding to ring update process (*Observer & Validator* dialog)

REFERENCES CITED

16.6. multiprocessing - process-based "threading" interface. (2017). https://docs.python.org/2/library/multiprocessing.html. The Python Software Foundation.

Al-Sultan, S., Al-Doori, M. M., Al-Bayatti, A. H., & Zedan, H. (2014). A comprehensive survey on vehicular ad hoc network. *Journal of Network and Computer Applications*, *37*, 380 - 392. Retrieved from http://www.sciencedirect.com/science/article/pii/S108480451300074X doi: http://doi.org/10.1016/j.jnca.2013.02.036

Bergamini, D., Descoubes, N., Joubert, C., & Mateescu, R. (2005). Bisimulator: A modular tool for on-the-fly equivalence checking. In N. Halbwachs & L. D. Zuck (Eds.), *Tools and algorithms for the construction and analysis of systems: 11th international conference, tacas 2005, held as part of the joint european conferences on theory and practice of software, etaps 2005, edinburgh, uk, april 4-8, 2005. proceedings* (pp. 581–585). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/978-3-540-31980-1_42 doi: 10.1007/978-3-540-31980-1_42

Chang, E., & Roberts, R. (1979, May). An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, *22*(5), 281–283. Retrieved from http://doi.acm.org/10.1145/359104.359108 doi: 10.1145/359104.359108

Craggs, I., Sutton, J., & Pagliughi, F. (2017). *paho.* https://eclipse.org/paho/.

Cristian, F. (1989). Probabilistic clock synchronization. *Distributed Computing*, *3*(3), 146–158. Retrieved from http://dx.doi.org/10.1007/BF01784024 doi: 10.1007/BF01784024

Grovepi internet of things robot kit. (2017). https://www.dexterindustries.com/grovepi/. Dexter Industries.

Gusella, R., & Zatti, S. (1989, Jul). The accuracy of the clock synchronization achieved by tempo in berkeley unix 4.3bsd. *IEEE Transactions on Software Engineering*, *15*(7), 847-853. doi: 10.1109/32.29484

Harding, J., Powell, G. R., Yoon, R., Fikentscher, J., Doyle, C., Sade, D., ... Wang, J. (2014, August). *Vehicle-to-vehicle communications: Readiness of V2V technology for application* (Report No. DOT HS 812 014). Washington, DC: U.S. Department of Transportation, National Highway Traffic Safety Administration. https://www.nhtsa.gov/staticfiles/rulemaking/pdf/V2V/Readiness-of-V2V-Technology-for-Application-812014.pdf.

Hoare, C. A. R. (1978, Aug). Communicating sequential processes. *Commun. ACM*, *21*(8), 666–677. Retrieved from http://doi.acm.org/10.1145/359576.359585 doi: 10.1145/359576.359585

Hoare, C. A. R. (2015). *Communicating sequential processes.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc. Retrieved 2017-04-26, from http://www.usingcsp.com/

Knight, J. C. (2002). Safety critical systems: Challenges and directions. In *Proceedings of the 24th international conference on software engineering* (pp. 547–550). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/581339.581406 doi: 10.1145/581339.581406

Magee, J., & Kramer, J. (2006). *Concurrency: State models & java programs* (2nd ed.). Wiley Publishing. Retrieved 2017-04-26, from https://www.doc.ic.ac.uk/~jnm/book/

Magee, J., & Kramer, J. (2012). *FSP notation.* https://www.doc.ic.ac.uk/~jnm/LTSdocumention/FSP-notation.html. Imperial College London.

Magee, J., & Kramer, J. (2015). *Concurrency: State models & java programs.* https://www.doc.ic.ac.uk/~jnm/book/.

Magee, J., Kramer, J., Chatley, R., Uchitel, S., & Foster, H. (2007). *FSP reference.* https://www.doc.ic.ac.uk/ltsa/eclipse/help/index.html?appendix_b___fsp_language_spec.htm. Imperial College London.

Magee, J., Kramer, J., Chatley, R., Uchitel, S., & Foster, H. (2013). *LTSA - labelled transition system analyser.* https://www.doc.ic.ac.uk/ltsa/.

Mateescu, R., & Bergamini, D. (2017). *BISIMULATOR manual page.* http://cadp.inria.fr/man/bisimulator.html. INRIA - Institut National de Recherche en Informatique et en Automatique (French Institute for Research in Computer Science and Automation).

Mateescu, R., & Oudot, E. (2008, June). Bisimulator 2.0: An on-the-fly equivalence checker based on boolean equation systems. In *2008 6th acm/ieee international conference on formal methods and models for co-design* (p. 73-74). doi: 10.1109/MEMCOD.2008.4547690

Mills, D. L. (1991, Oct). Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, *39*(10), 1482-1493. doi: 10.1109/26.103043

Milner, R. (1982). *A calculus of communicating systems.* Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Raspberry pi - teach, learn, and make with raspberry pi. (n.d.). `https://www.raspberrypi.org/`. Raspberry Pi Foundation.

Stanford-Clark, A., & Nipper, A. (2014). *MQTT.* `http://mqtt.org/`.

Vanderbilt, T. (2012). *Autonomous cars through the ages.* `https://www.wired.com/2012/02/autonomous-vehicle-history/`.

Weber, M. (2014). *Where to? a history of autonomous vehicles.* `http://www.computerhistory.org/atchm/where-to-a-history-of-autonomous-vehicles/`.

Wu, W., Zhang, J., Luo, A., & Cao, J. (2015a, January). Distributed mutual exclusion algorithms for intersection traffic control. *IEEE Transactions on Parallel and Distributed Systems*, *26*(1), 65-74. Retrieved from `http://ieeexplore.ieee.org/document/6747396/?tp=&arnumber=6747396` doi: 10.1109/TPDS.2013.2297097

Wu, W., Zhang, J., Luo, A., & Cao, J. (2015b, January). Distributed mutual exclusion algorithms for intersection traffic control (supplementary). *IEEE Transactions on Parallel and Distributed Systems*, *26*(1), 1-5. doi: 10.1109/TPDS.2013.2297097