BACHELOR

Real-time supervisory control synthesis program for one dimensional vehicle following system

Vlaar, Frank J.A.

Award date:
2022

Department of Mechanical Engineering

De Rondom 70, 5612 AP Eindhoven P.O.
Box 513, 5600 MB Eindhoven
The Netherlands
www.tue.nl

Author
Frank Vlaar (0960281)

Responsible Lecturer
dr.ir. M.A. Reniers
dr.ir. A. Rashidinejad

Date
January 31, 2022

Real-time supervisory control synthesis program for one dimensional vehicle following system

Frank Vlaar (0960281)
f.j.a.vlaar@student.tue.nl

Table of contents

Title
Real-time supervisory control synthesis program for one dimensional vehicle following system

Table of contents

Title
Real-time supervisory control synthesis program for one dimensional vehicle following system

List of Figures

1 Introduction

Imagine multiple vehicles following each other. If the first vehicle stops, everyone expects, and hopes, that the rest also stops. If the first vehicle accelerates, ideally would be if the other vehicles follow. If both are the case, the line of vehicles can be considered a platooning system. It would be nice if one of the vehicles is not in some state from where the breaks do not work, or that it is not possible to accelerate. This thesis attempts to solve that problem. Every vehicle is modeled as its own system. Each vehicle only knows the distance to the vehicle in front. The vehicles can also start accelerating or decelerating, depending on the distance. The distance is modeled as discrete events. Whenever the distance crosses a certain threshold, an event triggers, letting the vehicle know something happens. Depending on those triggered events, the vehicle can choose to trigger events in the form of a deceleration or acceleration.

The vehicles are modeled as timed automata. An automaton is a type of system that is only in one location at a time. The automaton can change locations when events occur. An event occurrence can be restricted in a location depending on variables. In a timed automaton the only type of variables is time. Time can only be reset and the value of a timer can be read. The states from where it is not possible to leave are called blocking states. An algorithm that changes a timed automaton such that it does not reach those blocking states is proposed by Rashidinejad et al., (2021). The goal of this thesis is to apply this algorithm to a model of one vehicle following another vehicle.

After the prerequisites in chapter two, in chapter three, the model is introduced. The model is separated into three layer: The plant layer where only the possibilities of the system is described, the observer layer that can only make observations of the plant layer, this is where time is introduced, and the requirement layer that puts restrictions on the plant layer. The system was modeled in CIF. Adjustments were made until the model worked.

The synthesis algorithm requires a single automaton. Therefore all the automata from chapter three had to be combined into one single automaton. This would be too much to do by hand so it is automated by a computer. Scripts that compute this single automaton are given in the appendix. In chapter 4.1 and 4.2 the scripts are layed out in pseudocode. In chapter 4.3 an attempt is made at automating the synthesis algorithm from Rashidinejad et al., (2021).

From the single automaton that is computed in chapter four, a small piece is taken and used in chapter five. Chapter five applies the synthesis algorithm by hand and shows how it works.

Finally, in the conclusion, recommendations are given on how the scripts can be improved and applied.

2 Prerequisites

An automaton is a system consisting of one or more locations (L), events (σ) and variables. Locations can also be called states, these two terms are used interchangeably in this thesis. The automaton always starts in an initial location (l_0). An event can for example be the push of a button or a sensor output. An event is contained in an edge. An edge (e) can make that, should en event occur, the automaton changes state to location l_t. An edge can also put restrictions on the event. This is called a guard (g). If the guard is not satisfied, the event cannot occur. Finally, the event can perform an update on variables. For example, if the event occurs, a variable should be multiplied by two or reset to zero.

While an automaton is in a location, requirements can be put on the variables. This means that the automaton is only allowed to stay in that location while the requirement evaluates to *True*. This requirement is called an invariant (I).

Supervisors are a type of automaton that keep the system from reaching an undesirable location and be stuck there. In order to do that, the supervisor has to know which locations are desirable, also called stable locations or marked locations (l_m). If there are no stable locations, the supervisor does not know in what location the automaton is allowed to be in. A location is blocking if there are no more ways to reach a stable location. The supervisor helps the automaton reach those locations by making the blocking locations unreachable (Rashidinejad et al., 2021).

The supervisor considers two types of events: controllable events and uncontrollable events. Controllable events are events that can be controlled by the supervisor. They can also be preempted if the supervisor so desires. Uncontrollable events are, as the name implies, uncontrollable. The supervisor is unable to change the guards of these events and cannot control when or whether they occur. Controllable events can for example be the turning on of a light bulb. An uncontrollable event is for example the light switch. If the uncontrollable event of switching the light switch occurs, the system should trigger the controllable event of turning on the light. Another type of event is a forcible event. A forcible event is an event that is able to preempt the passage of time in a location. They are not used in the system described in this thesis, but they can be used when synthesizing a supervisor. Bad locations are locations that lead to blocking locations through uncontrollable events or locations that are blocking themselves. They should therefore not be reachable (Rashidinejad et al., 2021).

This thesis deals with a specific type of automaton called a timed automaton (TA). The only type of variable in a TA are clock variable (C). All clock variables change at the same speed, at a rate of 60 seconds per minute. They can however be reset in the update of an edge. In that case, the clock resets to zero.

An automaton is said to be a seven-tuple. Meaning that it is made up out of seven elements. An edge is a five-tuple since it consists of five elements, an overview of the elements that make up an automaton is shown below:

- C: The set of clocks
- L: The set of locations. An individual location is indicated by l where $l \in L$
- Σ: The set of events. An individual location is indicated by σ where $\sigma \in \Sigma$
- E: The set of edges of the form $e \in E = \{l_s, \sigma, g, r, l_t\}$
    - l_s: The start location
    - $\sigma \in \Sigma$: The event causing the edge
    - g: The guard
    - r: The set of clocks to be reset
    - l_t: The target location
- L_m: The set of marked locations
- L_0: The set of initial locations
- I: The set of invariants. An individual invariant is indicated by i where $i \in I$

Multiple automata are used to describe the vehicle system that is the subject of this thesis. Supervisor synthesis requires those automata to be combined into one single automaton. That single combined automaton is called the synchronous product. The locations of the synchronous product is the product of all locations such that every possible combination is represented. In each of those combined locations, the new invariant is the conjunction of the two original invariants. The new edges are the sum of the original edges with, if they are shared, the new guard as the conjunction of the original guards. The target location of such a shared edge is the location that is the product of the the target locations of the original edges. This can intuitively be understood by imagining an edge that has a target location $l_t = l_1$, and another edge that has as its target location $l_t = l_2$, then the combined edge has as its target the location that is the product of both l_1 and l_2. The product is two automata G_1 and G_2 is denoted as follows (Rashidinejad et al., 2021):

$$G_1||G_2 = (C_1 \cup C_2, L_1 \times L_2, \Sigma_1 \cup \Sigma_2, E_P, L_{m,1} \times L_{m,2}, (L_{0,1}, L_{0,2}), I_P) \tag{2.1}$$

In words: The disjunction of the clocks, the product of all locations, the disjunction of all events, the set of the edges combined, the product of all marked locations, both sets of initial locations and the set of all invariants combined.

3 System Setup

The goal of the system is to make one vehicle follow another vehicle. In order to accomplish this goal, the vehicles are assumed to have a sensor that measures the distance to the vehicle in front. The speed of the vehicles is not used. Instead, the speed is measured indirectly by measuring the time it takes between events.

The input to the system is the distance sensor on the follower vehicle. This sensor causes events to happen when the distance crosses certain thresholds. The values for these thresholds are not discussed in this thesis, only the events are used. The distance is split into five discrete distances in the following events:

- $D_{\rightarrow none}$: This is the output of the sensor where the distance to the vehicle in front becomes equal to the ideal distance.
- $D_{\rightarrow front,small}$: This event happens when the distance to the vehicle in front is slightly smaller than ideal. This distance is still assumed to be save.
- $D_{\rightarrow front,large}$: When this event occurs, the distance to the vehicle in front is too small and the vehicle should decelerate to prevent accidents.
- $D_{\rightarrow back,small}$: This event happens when the vehicle gets slightly behind, but the deviation from the ideal distance is not yet considered to be a problem.
- $D_{\rightarrow back,large}$: This distance is considered to be too large and the vehicle should accelerate to catch up.

These distances are all uncontrollable and therefore cannot be influenced by the supervisor. In the graphical representations of the automata, shown in Figure 3.1, uncontrollable events are indicated by dashed lines. In the CIF representations, the uncontrollable event start with the letter *u*.

The events that the supervisor has control over are the acceleration and deceleration events. These can be imagined like a gas paddle and a break paddle. A list of these events is given below:

- $A_{\rightarrow none}$: When this event occurs, the acceleration is set to zero, the vehicle drives at a constant speed.
- $Decel_{\rightarrow small}$: This event sets the acceleration of the vehicle to be slightly negative. The vehicle is decelerating.
- $Decel_{\rightarrow large}$: The deceleration is large.
- $Accel_{\rightarrow small}$: The vehicle starts accelerating, but the acceleration is not large.
- $Accel_{\rightarrow large}$: This event sets the acceleration to be large.

These are controllable events and are indicated by a continues line for the edges in the graphical representations, as shown in Figure 3.2. In the CIF representation, these events start with the letter c. The ten events described above are the only events in the system.

The system is event-based. This means only the events are used to indicate something. It is not possible to say: 'When this automaton is in this location, do ...' since that requires the location. Only events are synchronized and the different automata do not know what location the other automata are in. In order to simplify this, all events are memory-less. This means only the destination location of the event matters, not where it came from. This makes the event based system easier because instead of saying: 'when in this location, do ...', there is only one event that results in the automaton being in that location. As an example, the event $A_{\rightarrow none}$ always points tot the locations where the acceleration is zero. There are not separate events that set the acceleration to zero if the acceleration was previously negative and one for the case where the acceleration was previously positive.

To make the purpose of the automata clear, they have been seperated into three layers: the plant layer, the observer layer and the requirement layer. The purpose of these layers is described below:

- **The plant layer**: This is where the plant is modeled. There are no real restrictions put onto the system yet. Everything is possible.
- **The observer layer**: This layer only observes the plant and adds something to it. In this case the observer adds timing to the plant layer.
- **The requirement layer**: This is where the requirements are modeled as automata. This layer restricts the behaviour of the plant.

The requirement automata are different from requirement automata in for example Rashidinejad et al., (2021). There, whenever a requirement is not met, the system goes to a blocking state. In the requirement automata described in this thesis, this is not the case. Here the requirement automata only limit the behaviour of the plant.

The goal of the system has been formalized into four requirements. These requirements are given below:

1. Whenever the events $D_{\rightarrow front,large}$ or $D_{\rightarrow back,large}$ happen, the vehicle should start to decelerate or accelerate respectively.
2. If the acceleration or deceleration does not result in the event $D_{\rightarrow none}$ within a certain time, the acceleration or deceleration should be faster.
3. If, depending on the speed at which the vehicle is getting closer or further away, a small deceleration or acceleration is not enough. The vehicle should immediately decelerate or accelerate heavily.
4. When the distance between the vehicle and the vehicle in front is equal to the ideal distance, the speed should be constant.

Designing the automata started with an draft. This draft was transferred to the CIF programming language and a simulation was made to check if everything worked. If something did not work, the automata were adjusted until it did work. Most of the problems that occurred resulted from one automaton blocking the events of another automaton. Supervisor synthesis is unable to add edges where there previously were none, so the automata were adjusted. A result of this is that the automata became more complicated since self loops and additional edges were added.

3.1 Plant automata

Plant automata do not restrict the system, they only model the possibilities of the system. The plant is split into two automata: one for the distance sensor input and one for the acceleration and deceleration output. The distance automaton is shown in Figure 3.1. The acceleration and deceleration automaton is shown in Figure 3.2. Neither automaton has a memory, meaning that when an event happens, only the destination is used. The distance automaton only consists of uncontrollable events because they are all sensor dependent. The acceleration and deceleration automaton only consists of controllable events.

The distance automaton starts in the initial location where the distance is equal to the ideal distance. When the sensor outputs that the distance is smaller or larger than the ideal distance, the events $D_{\rightarrow front,large}$ or $D_{\rightarrow back,large}$ happen and the system changes state to $D_{front,small}$ or $D_{back,small}$. If the distance is even closer or further away the states are $D_{front,large}$ or $D_{back,large}$ respectively. If the deviation is small, not action is taken. The vehicle only decelerates or accelerates when the deviation is large. A larger or smaller acceleration or deceleration depends on the time it takes for the automaton to go from a small to a large deviation.

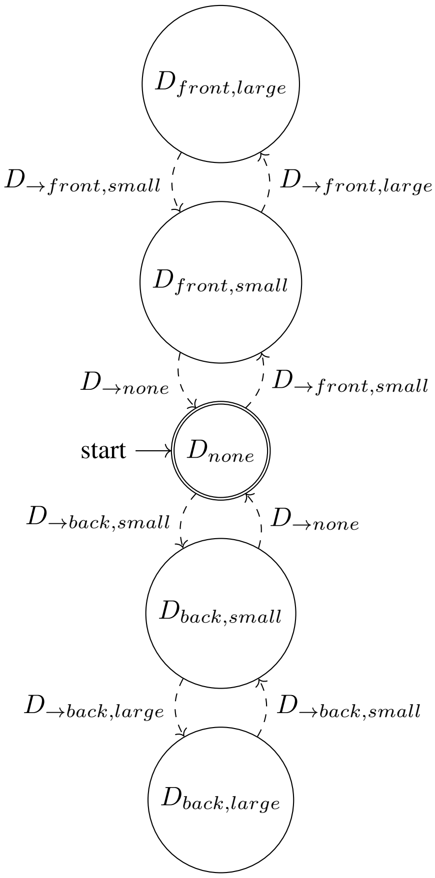

Figure 3.1: Distance automaton used to represent the distance to the truck in front. D_{none} represents the ideal distance. All events are uncontrollable because they represent sensor outputs. This is a visualization of the plant_distance automaton in Appendix 8.7

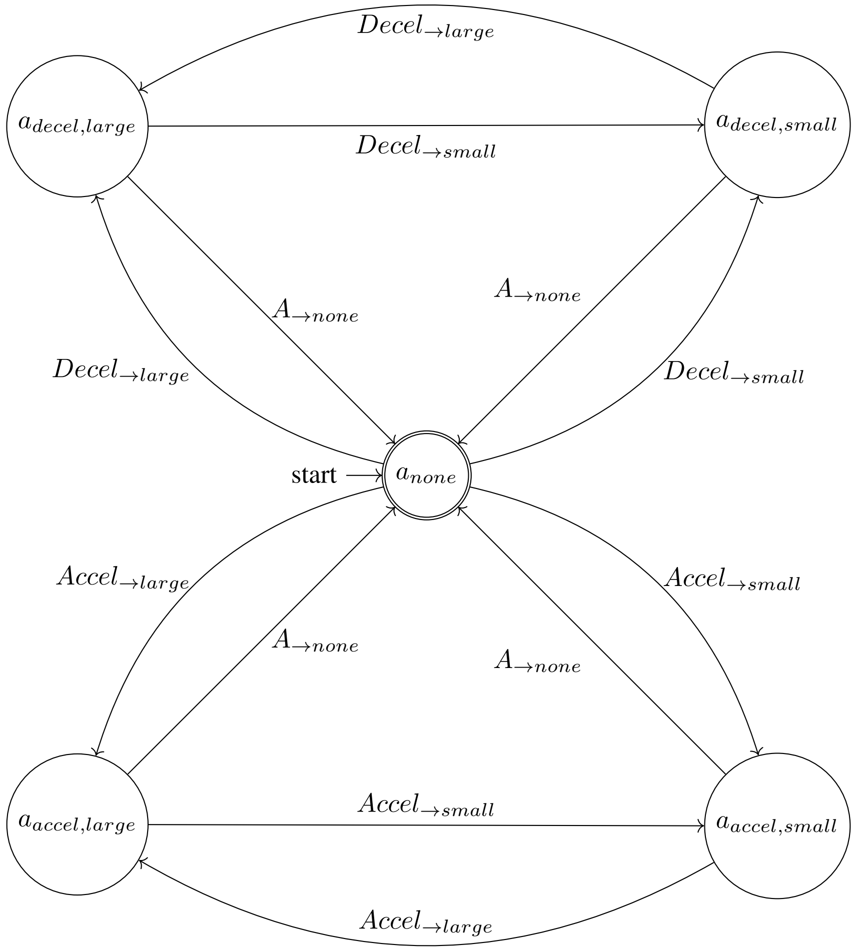

Figure 3.2: Acceleration and deceleration automaton. There are two levels of deceleration and two levels of acceleration, one small and the other larger. This automaton does not have a memory, meaning that the events only point to their destination and do not know where they come from. This is a visualization of the plant_acceleration automaton in Appendix 8.7

3.2 Observer automata

Observer automata only say something about the automata described in the plant layer, they do not influence anything. There are two observer automata in the second layer, one for the closer distance as shown in Figure 3.3 and one for the further distance as shown in Figure 3.4. These automata look similar to the plant automata, except that time has been added. If the time it takes to cross from the small deviation to the large deviation is known, the speed can be calculated. This is later used in the requirement automata to decide between a large or a small deceleration or acceleration. The requirement automata takes the value of t_1 and checks how long it takes before the $D_{\to front,large}$ event happens. If this takes long, it means that the approaching distance is slow. The speed that is considered slow can be set afterwards. In the system described here, the time is an arbitrary $3/8$ seconds. Quicker is considered fast. Slower is considered slow.

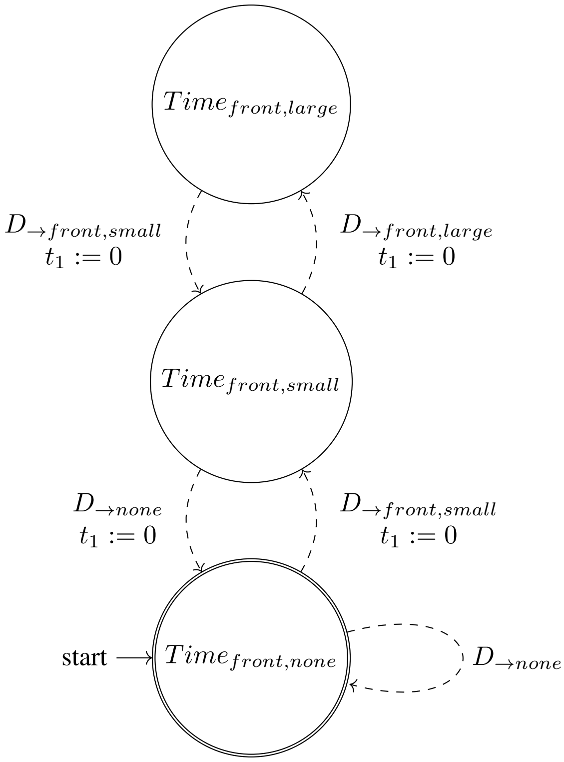

Figure 3.3: Automaton that adds timing to the "front" part of the distance automaton. This timing is later used in the requirement automata to say something about the speed of approach. This is a visualization of the time_close automaton in Appendix 8.8

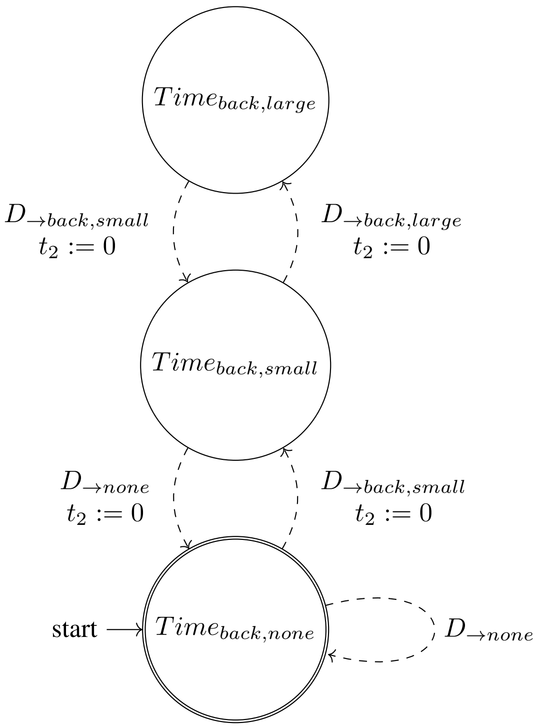

Figure 3.4: Automaton that adds timing to the "back" part of the distance automaton. This timing is later used in the requirement automata to say something about the speed of approach.This is a visualization of the time_far automaton in Appendix 8.8

3.3 Requirement automata

Requirement automata restrict the behaviour of the plant layer, depending on observations made in the observer layer. Below, the requirement automata are shown. The acceleration and deceleration automata for every speed work in a similar manner. The automata in Figure 3.5 and Figure 3.6 meet requirement 1: "*Whenever the events* $D_{\to front,large}$ *or* $D_{\to back,large}$ *happen, the vehicle should start to decelerate or accelerate respectively.*" For the slow deceleration automaton as shown in Figure 3.5, the steps are described below. The automaton from Figure 3.6 works in the same way:

1. The automaton starts in location $v : constant$. The event $A_{\to none}$ is a self loop so that it is blocked in all other locations. The uncontrollable events $D_{\to none}$ and $D_{front \to small}$ are there so that they are not blocked for the other automata.

2. When the event $D_{\to front,large}$ happens, the state changes to the $Deviation : large$ location and the timer $t_{3,slow}$ is reset to zero. In this location, the timer $t_{3,slow}$ is not allowed to grow greater than 0, so the automaton must immediately take the event $Decel_{\to small}$ in order to satisfy the invariant. The self loop $Decel_{\to large}$ is there because the other automaton from Figure 3.7 also goes to a location $Deviation : large$ when under the same condition, but this automaton takes the event $Decel_{\to large}$ and this event must not be blocked. This event is blocked in the $v : constant$ location because deceleration when the distance is good is undesirable.

3. When the event $Decel_{\to small}$ happens, timer $t_{3,slow}$ is reset and the automaton reaches the location $decel : small$. In this location the event $D_{\to front,small}$ is a self loop because the vehicle should only stop decelerating once the ideal distance is reached, so deceleration should only stop when the event $D_{\to none}$ happens. The event $Decel_{\to large}$ is a self loop in this location because of requirement 2: "If the acceleration or deceleration does not result in the event $D_{\to none}$ within a certain time, the acceleration or deceleration should be faster", which is satisfied in automaton Figure 3.7. The event $Decel_{\to large}$ is only allowed while this automaton is in the locations $Deviation : large$ or $Decel, small$ and not in the other locations because deceleration is only allowed when the distance is too close.

4. The automaton is also able to go directly from the location $v : constant$ to the location $Decel : small$. This is when the speed of approach is quick and the automaton should decelerate aggressively immediately. This event is so that the automaton of Figure 3.9 does not have its events blocked.

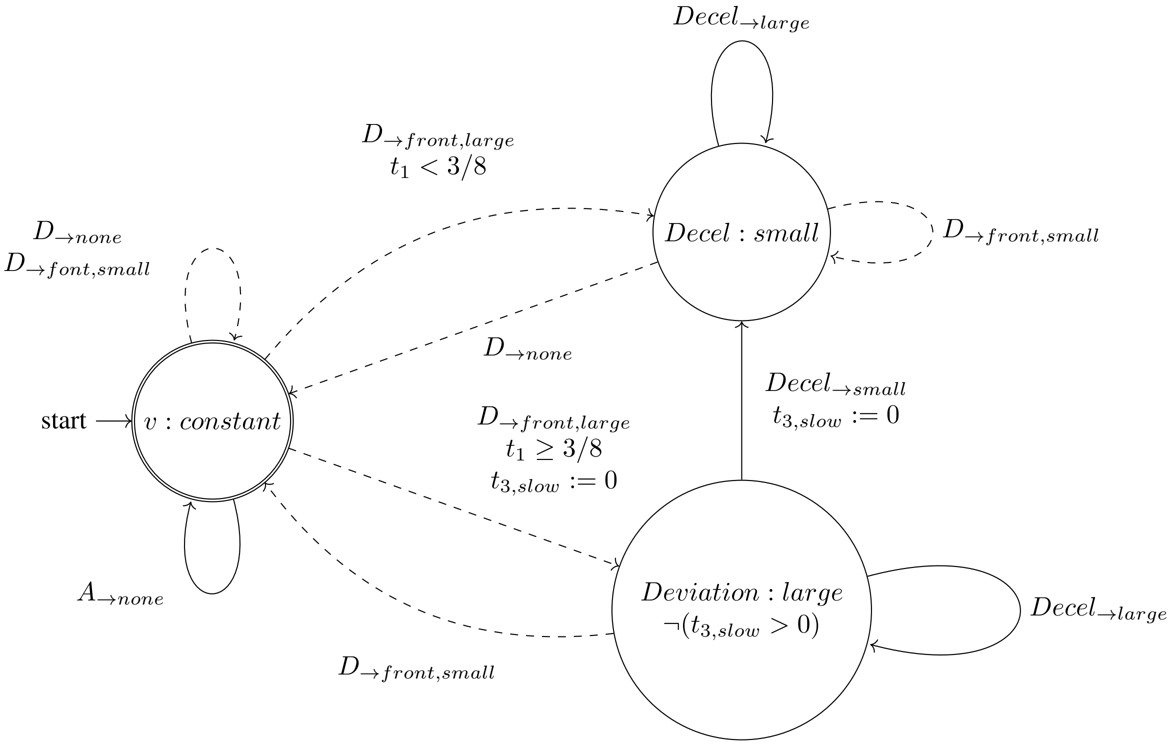

Figure 3.5: Automaton that requires the system to slow down when the approaching speed it slow. The automaton starts in the $v : constant$ location and when the event $D_{\rightarrow front,large}$ happens, depending on the timer t_1, the automaton goes to the $Deviation : large$ or the $Decel, small$ location. When in location $Deviation : large$, the event $Decel_{\rightarrow small}$ must happen to satisfy the invariant. This is a visualization of the decel_slow automaton in Appendix 8.9

.

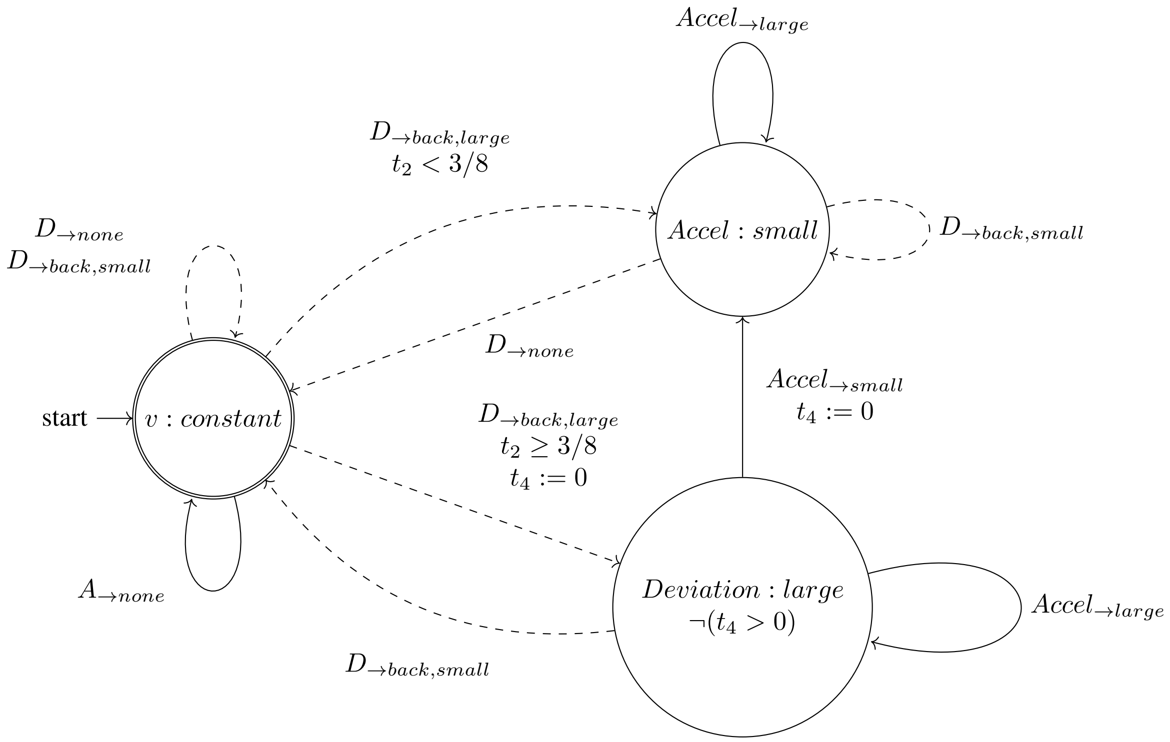

Figure 3.6: Automaton to meet the requirement that whenever the distance is very far and the speed with which the distance grows is slow, the vehicle should accelerate slowly. This is a visualization of the accel_slow automaton in Appendix 8.10

The automata from Figure 3.7 and Figure 3.8 are to meet requirement 2: "*If the acceleration or deceleration does not result in the event* $D_{\rightarrow none}$ *within a certain time, the acceleration or deceleration should be faster.*". The working of these automata is similar to the automata from Figure 3.5 and Figure 3.6, except that the automaton is allowed to stay in the $Deviation : large$ location for a while, as a result the small acceleration or deceleration happens earlier. The automaton will only take the event $Accel_{\rightarrow large}$ or $Decel_{\rightarrow large}$ after it has stayed in that location for a while. This gives the automaton time to go back to the $v : constant$ state if the distance has decreased enough. If the event where the distance is large is followed by the event where the distance is small, the automaton is going back to the ideal position.

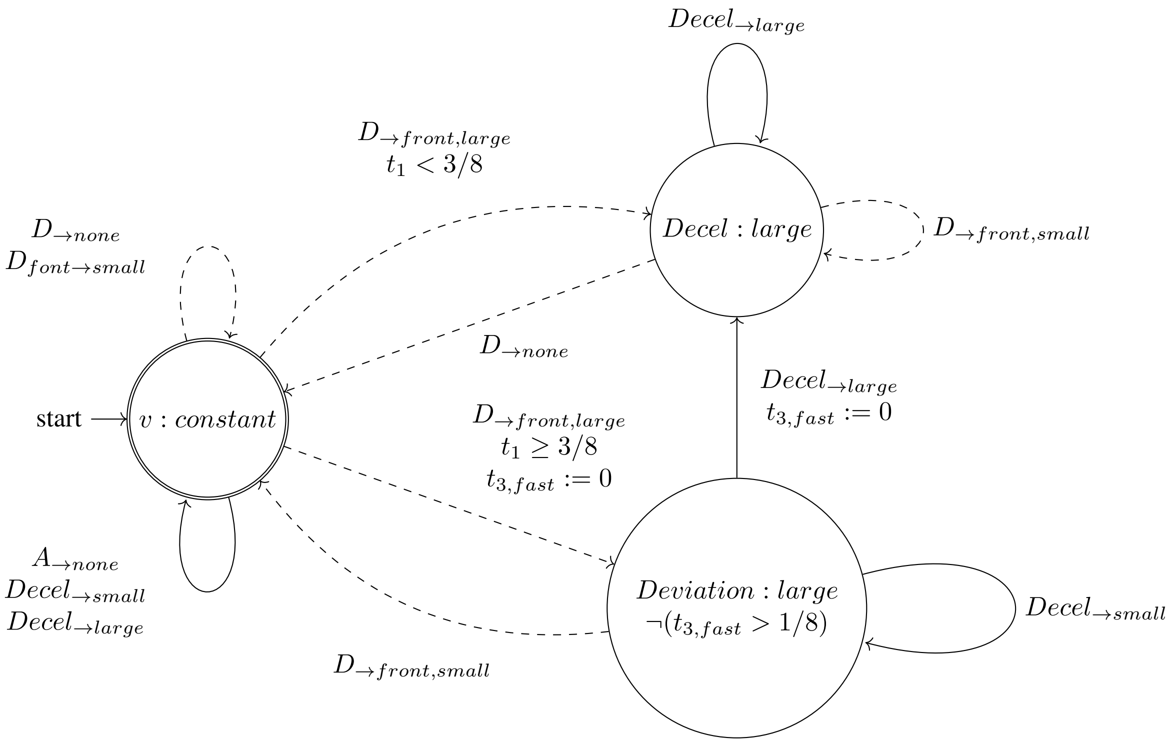

Figure 3.7: Automaton to meet the requirement that when the distance is very close but a small deceleration is not enough, then the deceleration should switch from a slow deceleration to a large deceleration. This is a visualization of the decel_panic automaton in Appendix 8.9

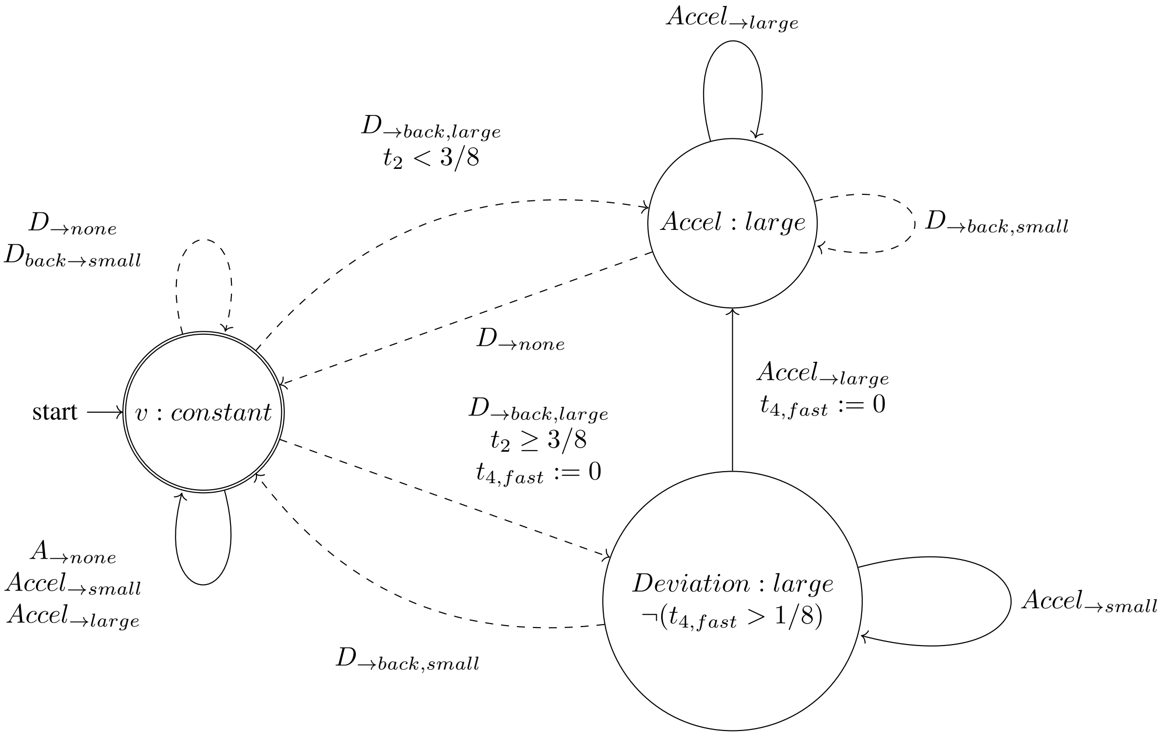

Figure 3.8: Automaton to meet the requirement that when the distance is very far but a small acceleration is not enough, then the acceleration should switch from a slow acceleration to a large acceleration. This is a visualization of the accel_fast automaton in Appendix 8.10

Requirement 3 is: "*If, depending on the speed at which the vehicle is getting closer or further away, a small deceleration or acceleration is not enough. The vehicle should immediately decelerate or accelerate heavily*". This requirement is met with the automata from Figure 3.9 and Figure 3.10. The idea behind these is similar to the previous automata, except that these automata take the $D_{\to front,large}$ or $D_{\to back,large}$ events to the $Deviation : large$ location if the timers t_1 and t_2 are small. Similar to the slow acceleration and slow deceleration automata, these automata are not allowed to stay in that temporary location for any period of time and therefore the events $Decel_{\to large}$ or $Accel_{\to large}$ must be taken immediately. These automata do not have an edge directly from the $v : constant$ state to the $Accel : large$ or $Decel : large$ state. These automata took a different approach to the same problem that the event $D_{\to back,small}$ should not be blocked. Here the event is added as a self loop to the location $v : constant$.

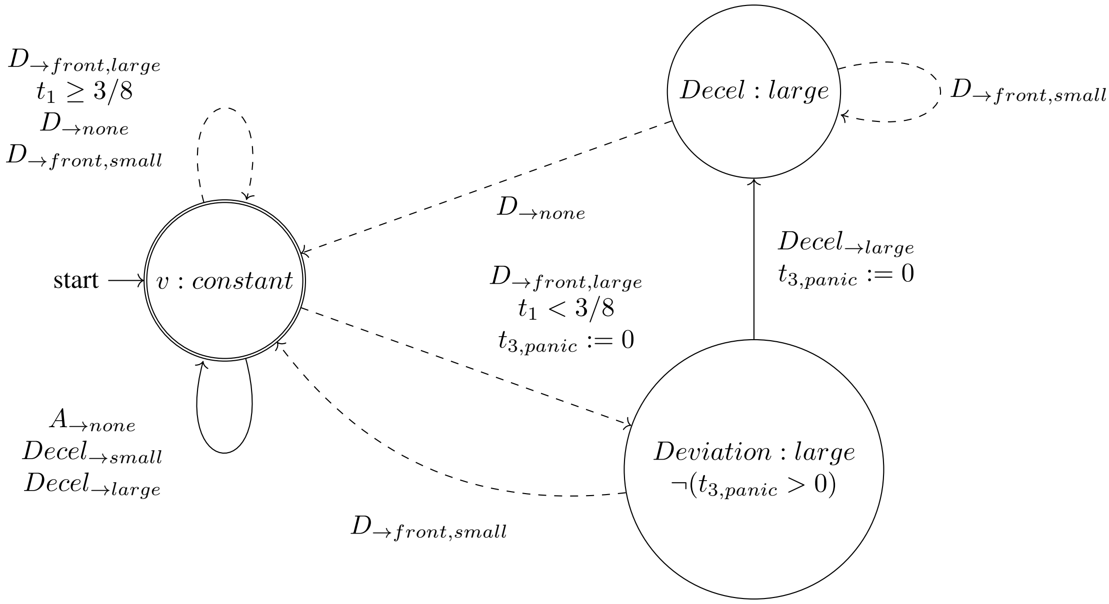

Figure 3.9: Automaton to meet the requirement that when the distance is very close and the speed of approach is fast, then the vehicle should immediately Decelerate fast. This is a visualization of the decel_panic automaton in Appendix 8.9

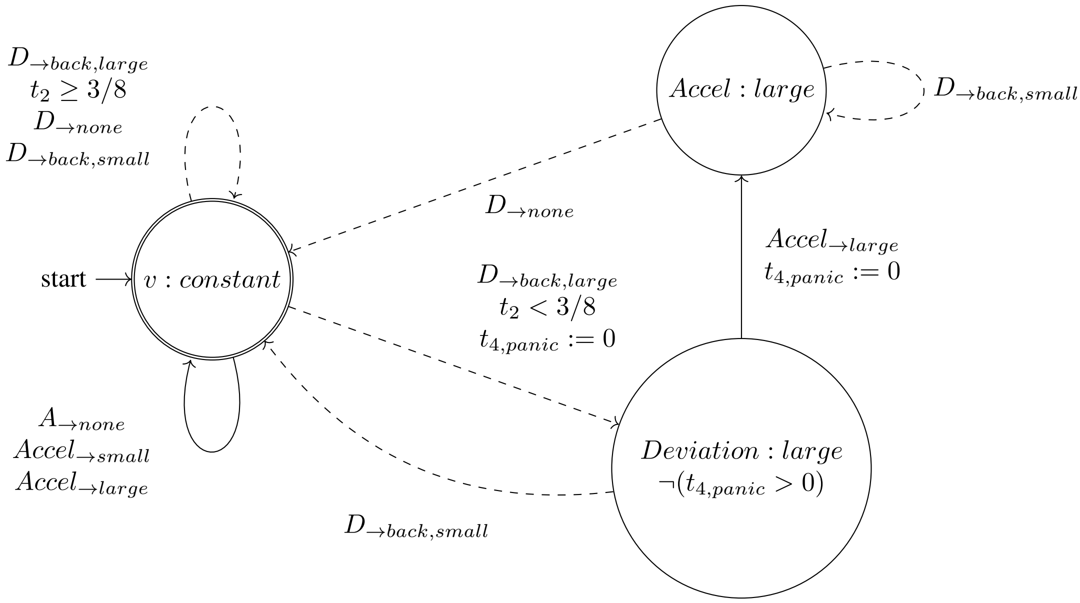

Figure 3.10: Automaton to meet the requirement that when the distance is very far and the leaving speed is fast, then the vehicle should immediately Accelerate fast. This is a visualization of the accel_panic automaton in Appendix 8.10

The last requirement is: "*When the distance between the vehicle and the vehicle in front is equal to the ideal distance, the speed should be constant*". This requirement is met by the automaton from Figure 3.11. The automaton transitions to location $D \neq 0, a \neq 0$ whenever the acceleration is not equal to zero. If the event $D_{\to none}$ happens, the automaton goes to a location $D = 0, a \neq 0$ where is is not allowed to stay for any time, so the event $a_{\to none}$ must happen immediately. Thus meeting the requirement that whenever the distance is equal to the ideal distance, the acceleration is zero, is met.

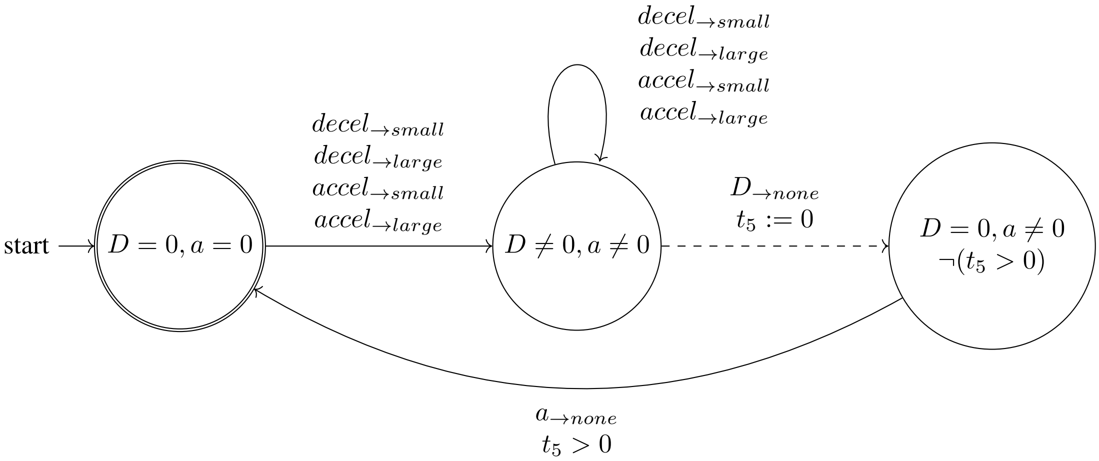

Figure 3.11: Automaton to meet the requirement that whenever the distance to the vehicles predecessor is equal to the ideal distance, the acceleration should be zero. This requirement is met because at the location $D = 0, a \neq 0$, the timer is not allowed to become larger then zero and thus the controllable event a_{none} must be taken. This is a visualization of a_none automaton in Appendix 8.11

4 Automation

There is a total of eleven automata that describe the system and the requirements. Computing the synchronous product of all these automata would be too much work to do by hand. The synchronous product computed by the CIF language was not in the correct form since it consisted of only one location. This makes supervisor synthesis more complicated since it relies on edges to marked or blocking locations. Therefore an attempt was made at creating a program that computes the synchronous product for the automata described in chapter three. The algorithms that were used for this are shown in Algorithm 1, 2 and 3. Of these three, Algorithm 1 is the main algorithm and both Algorithm 2 and 3 are called from the first one. It should be noted that these algorithms compute the product between two automata. The program is called many times, ten in total, to first compute the product between the first two, and add every consecutive automaton to the product. It can be visualized as follows for four automata:

$$A_1||A_2||A_3||A_4 = (((A_1||A_2)||A_3)||A_4) \tag{4.1}$$

Here the brackets indicate that every time, only the product between two automata needs to be computed. As is done in Appendix 8.1, line $102 - 105$.

4.1 Parsing

In order to compute a product, first the computer must know the automata. These automata initially only existed in text, so the text must be translated into something that the computer understands, this is called parsing. A parser reads the text and translates it into a structure that can be understood by the computer. The algorithm that is used by the parser is only described in words. The corresponding lines in the code from the appendix are given.

The scripts are written in Java, which is an object oriented language. These objects know things and they can do things. Take for example an object to perform operations on two numbers. First both numbers should be send to that object, then the object knows the numbers. Next there are a couple of possibilities. The main script could for example tell the object to compute the product between the numbers. The object would do that and then it also knows the product between the numbers. The main script could also tell the object to compute the sum. Then the object knows the sum as well. It is this idea that is used by the parser: an "*automaton*" object is created, this object is given a text string so that it knows the text string. The automaton is then told to "*organize*" the text. This means the automaton looks for the locations, its name and its alphabet. The steps that are taken by all objects are described below:

- The main script loads a text as a string and searches for the location of the "*automaton*" word, as well as the "*end*" word that is used to indicate the end of an automaton. The sub-string that runs from one occurrence of the word "*automaton*" till the next "*end*" occurrence is send to the automaton object. This is done in appendix 8.1, line $74 - 85$.
- The automaton object then takes that string and starts looking for the word that is between the "*automaton*" word and the first "*:*". This is saved as the name of the automaton. The automaton object then looks for the occurrences of the "*location*" word and sends the sub-string that runs from the occurrence of "*location*" till the next, to the location object. This is done in appendix 8.2, line $61 - 113$.
- The location object looks for the occurrence of the word "invariant" and saves that which runs from that word till the first "*;*" as the invariant. Next the location searches for the location of the word "*edge*" and sends that which runs from that word till the first "*;*" to the edge object. This in done in appendix 8.3, line $140 - 217$.
- The edge object does the same for the guards and the updates. The first word that comes after the "*edge*" word is saved as the event of the edge. The first word after the "*do*" word is saved as the update. The first word after the "*when*" word is saved as the guards and whichever comes after "*goto*" is saved as the target location. This is done in appendix 8.4, line $137 - 227$.

The parser does not check for the correctness of the CIF specification. This means that if something is wrong in the CIF model, the parser will either give an error or the output does not make sense.

The above method results in the following structure: The automaton object contains the alphabet, its name, the string it is composed of and the locations that make up the automaton. Locations contain the name of the locations, the string that originally made up the location, the owner automaton, whether it is an initial location, whether it is a marked location and the edges it contains. Edges contain the event, the string that makes up the edge, the update, the guard, the initial location, the target location, whether it is forcible and whether it is uncontrollable . The structure is also given below:

- Automaton (contains: $\downarrow$)
  - Alphabet
  - Name
  - Text string
  - Locations (contain: $\downarrow$)
    - Name
    - Text string
    - Owner automaton
    - Boolean initial
    - Boolean marked
    - Edges (contain: $\downarrow$)
      - Event
      - Text string
      - Update
      - Guard
      - Initial location
      - Controllable
      - Destination location
      - Owner automaton
      - Boolean uncontrollable
      - Boolean forcible

In the algorithms below, sub-elements are denoted by dots. Below, a couple of examples are given for how to read these:

- $e_1.\sigma$ denotes the event from an edge.
- $A.\Sigma$ denotes all events that are part of automaton A. This is also referred to as the alphabet.
- $l.\Sigma$ denotes all events that exist in the location l.
- l denotes a location $l \in L$. The notation $L \leftarrow$ **Combine locations**$(l_1, l_2, \Sigma_1, \Sigma_2)$ means the output of the function **Combine locations** is added to the set L.
- $l.E \leftarrow e_2$ means edge e_2 is added to the set of edges in l.
- $l \in A.L$ indicates a location in the automaton A.

The algorithms are meant to explain the scripts in the appendix. They do not explain how to perform the operations in the most efficient manner, only how it was done in the scripts. The scripts were not made to be efficiënt.

4.2 Computing the product

Algorithm 1 is the main algorithm that computes the product. It is the backbone of script 8.5 in the Appendix. The algorithm loops over every possible combination of locations in both automata and combines them. The output of the combination is stored in a new location. The final step in the algorithm is cleaning the locations. Here the locations that cannot be reached through events from the initial locations, are removed. The clean locations algorithm is explained in Algorithm 3. As a last step the new text file is generated. The first part of the CIF code is expected to contain the declaration of the variables. This part is copied and pasted back as the first part in the CIF file of the synchronous product. For all timers that exist in the product, a declaration of the form: *cont x der 1;* is added before the first locations.

Algorithm 1 Computing the product

Input: Automaton $A_1 = (C_1, L_1, \Sigma_1, E_1, L_{m,1}, L_{0,1}, I_1)$;
Automaton $A_2 = (C_2, L_2, \Sigma_2, E_2, L_{m,2}, L_{0,2}, I_2)$;
Output: Automaton $A_P = A_1 || A_2 = (C_1 \cup C_2, L_1 \times L_2, \Sigma_1 \cup \Sigma_2, E_P, L_{m,1} \times L_{m,2}, (L_{0,1}, L_{0,2}), I_P)$;

$L_p = \emptyset$
for $l_1 \in A_1.L$ **do** ▷ Appendix 8.5, Line: $337 - 351$
  for $l_2 \in A_2.L$ **do**
    $L_P \leftarrow$ **Combine locations**$(l_1, l_2, \Sigma_1, \Sigma_2)$
  end for
end for

$A_P.L = L_P$
Clean locations$(A.L_P)$

4.2.1 Merging locations

The algorithm that combines two locations is given in Algorithm 2. This algorithm takes as input two locations and outputs a single location. This is done by merging all the edges in both automata. The actual Java script separates this into two functions, but in Algorithm 2 they are taken together. The algorithm takes the following steps:

1. The algorithm starts with an empty set $l_p.E$, which is filled up with all the combined edges. In the first loop (Line: $4 - 10$), the locations are checked for overlapping events. If two events are the same, they are not blocked in either location and thus are not blocked in the combined location. Therefore they are added to the set $l_p.E$. The "*Merge edges*" algorithm combines two edges. Merging edges means taking the conjunction of the guards, the conjunction of the clock resets and merging the target locations.
2. The next loop (line: $12 - 18$) goes over all the edges in the location L_1 and contains two booleans: x and y. The boolean x is *True* when the event causing the edge is in the alphabet of the automaton from the other location. The boolean y is *True* when the event causing the edge is in the other location. If y is *True*, this means that the event is in both locations and thus is is already added in the loop that checks for double events. If x is *True* and y is *False*, it means that is is in the alphabet, but not in the location and thus it is blocked by the other automaton. Only when it is not part of the other automaton and it is not already in added in the first loop, it can be added in this loop. The case where the event is not part of the alphabet of the other automaton, but it is part of the events in the other location, is impossible. Therefore $\neg x \wedge y = False$. In the script in Appendix 8.5, line 177, this is solved by using the logical "*and*". This means first the left part is evaluated and if that is *False*, the right part is skipped.
3. The last loop (line: $20 - 26$) does the exact same thing as the previous loop, except for the other edges. The event cannot be in the alphabet but not in the location because then it is blocked. It also cannot be in the other location either because then it was already added in the first loop.

In script 8.5 in the appendix, the function **merge_edges** from line 60 combines the guards and updates. The destination locations from the two original edges are added to a separate list in the new edge, called destination_sum. Later, the function **adjust_destination** from line 234 looks through all new locations and searches for the location that is the combination of the two target locations in the original edges. The name of the new target location of the edge is then adjusted according to the new name of the target locations. Any variables from other automata that are references are changed. The reference that occurs before the dot is removed. Also nothing is done to change the name of variables that have the same name. This means that if two automata share the same variable name, in the product they will be the same variable.

Algorithm 2 Merge locations

Input: Location l_1, Location l_2;
Alphabet Σ_1, Alphabet Σ_2;
Output: Location $l_P = l_1 \times l_2$;

```
lP.initial = l1.initial ∧ l2.initial
lP.marked = l1.marked ∧ l2.marked
lP.E := ∅
for e1 ∈ l1.E do                                ▷ Appendix 8.5, Line 151 − 160
    for e2 ∈ l2.E do
        if e1.σ = e2.σ then
            lP.e ← Merge edges(e1, e1)
        end if
    end for
end for

for e1 ∈ l1.E do                                ▷ Appendix 8.5, Line 162 − 172
    x := e1.σ ∈ Σ2
    y := e1.σ ∈ l2.Σ
    if ¬x ∧ ¬y then
        lp.E ← e1
    end if
end for

for e2 ∈ l2.E do                                ▷ Appendix 8.5, Line 174 − 184
    x := e2.σ ∈ Σ1
    y := e2.σ ∈ l1.Σ
    if ¬x ∧ ¬y then
        lp.E ← e2
    end if
end for
return lP
```

4.2.2 Cleaning unreachable locations

Algorithm 3 cleans all locations that cannot be reached from an initial location. It does this by adding the target locations for every edge in the reachable locations to the set of reachable location. This stops when for all locations no new locations are added to the set of reachable locations. The set of reachable locations consists of the names of the reachable locations, not of the locations themselves. This is the reason for the final loop on line $25-29$ over all locations. There the name of the location name is not part of the list of reachable locations, it is removed from the actual set of locations. This part is ineficiënt because of the constant loop over all locations. It would be faster to only check those locations that are reachable. In more detail, the algorithm works as follows:

1. The first step is looking for the initial locations. The names of these locations are added to the set of reachable locations (Line: $3-8$). Right now only one initial location is accepted. This is seen on line 7 in the algorithm: The *For*-loop ends once an initial location is found.
2. For the while loop, a boolean $Finished$ is used. The value is initially set to *False* and directly after the loop is set to *True*. If a new location is added, the value is set to *False* again. If no new location is found, the boolean is never set to *False* so the loop stops.

3. In the while loop, a sub loop runs over all location. If the location is a member of the $Reachable$ locations, for all edges in that location, the target location is added to the set of reachable locations if it was not already in there. If it was not already in there, the boolean $Finished$ its set to *False*.

4. The while loop finished when there is no new value added to the set of reachable locations.

5. The last loop goes over all locations and checks if it is in the set of reachable locations. If it is not in there, the location is removed.

Algorithm 3 Clean locations

Input: Locations L
Output: Locations L

```
Bool Finished = False
L_reachable = ∅

for l ∈ L do                                   ▷ Appendix 8.5, Line 267 – 274
    if l.initial then
        L_reachable ← l
        end for
    end if
end for

while ¬Finished do                             ▷ Appendix 8.5, Line 276 – 291
    Finished = True
    for l ∈ L do
        if l ∈ L_reachable then
            for l_t ∈ l do
                if ¬l_t ∈ L_reachable then
                    L_reachable ← l_t
                    Finished = False
                end if
            end for
        end if
    end for
end while

for l ∈ L do                                   ▷ Appendix 8.5, Line 294 – 303
    if ¬l ∈ L_reachable then
        Remove l
    end if
end for
Return L
```

Applying the algorithms on the system described in Chapter 3 results in 147 locations. When the guards of the edges are merged in the locations, they are not checked. This means some of the guards are *False* for all time. Such a guard may look as follows:

$$e.g = t_1 \geq 3/8 \wedge t_1 < 3/8 \tag{4.2}$$

These guards are a result of for example 3.9 and 3.5. Both locations start in location $v : constant$ and have $D_{\rightarrow front,large}$ in an outgoing edge. The guards for these edges are $t_1 \geq 3/8$ and $t_1 < 3/8$ respectively. Combining these edges and not simplifying the resulting guard results in a guards that is never satisfied. Later in this chapter a method that is able to simplify the guards is used, but this is not yet applied in the product-computation scripts.

4.3 Controller synthesis

An attempt was made at implementing Algorithm 5.3 from Rashidinejad et al., (2021) in Java. The first script from Algorithm 4 runs the outer loop and checks whether the program is finished. This is where the other algorithms are called. All algorithms and equations presented in this section are a direct implementation of algorithms and equations in chapter five of Rashidinejad et al., (2021).

4.3.1 Main file

In the scripts, simplification of logical expressions is done by Matlab in Script 4.1. The input to the function is a string and an optional argument. This string must contain the clocks that are reset. The string is first checked for the "*and*" and "*or*" words and they are changed for the Matlab equivalent of *&* and | respectively. The string that contains the logical expression is converted into an symbolic expression on line 4. The next *if*-statement checks if an optional argument was supplied. If this is the case, the clocks are substituted as zero in the symbolic expression. The assumption is made that all variables in the expression are real numbers and finally the expression is simplified. The simplify function is set to 100 iterations because some expressions can be very long. Finally the simplified expression is converted into a string so that it can be understood by Java. The "*and*" and "*or*" words are not put back because that is done at the end when the CIF text is generated.

Listing 4.1: Matlab simplifiction function. The input is a string that represents a logical expression, the ouput is a string with that expression simplified.

```
1  function simple = simpler(input, varargin)
2      input = strrep(input, ' and ', ' & ');
3      input = strrep(input, ' or ', ' | ');
4      sym = str2sym(input);
5      if (nargin >= 2)
6          resets = varargin{1};
7  
8          vars_old = symvar(sym);
9          vars_new = vars_old;
10         for i = 1:length(vars_new)
11             if ismember(vars_new(i), resets)
12                 vars_new(i) = 0;
13             end
14         end
15         sym = subs(sym, vars_old, vars_new);
16     end
17     assume(symvar(sym), 'real');
18     simple = simplify(sym, 100);
19     simple = string(simple);
20 end
```

For the actual synthesis, Algorithm 4 is used. This algorithm is directly related to Figure 5.4 in Rashidinejad et al., (2021). First the non-blocking conditions are computed for all locations. This is done in Algorithm 5. Next the bad-state conditions are computed in Algorithm 6 and the guards are adapted in Algorithm 7. Line 8 checks whether the guards have changed, if that is not the case and they have remained the same, the invariants are updated in Algorithm 8. If they have changed, the first small loop from line $4-5$ starts again. If the invariants have not changed, the program is finished.

Algorithm 4 Controller synthesis

Input: Automaton A;
Output: Automaton A;

1: **Boolean** $Loop_2 = False$
2: **Boolean** $Loop_1 = False$
3: **while** $\neg Loop_2$ **do** ▷ Appendix 8.6, Line $475-486$
4:     **while** $\neg Loop_1$ **do**
5:         **Non blocking**
6:         **Bad state**
7:         **Adapt guards**
8:         $Loop_1 =$ **Guards Equal**
9:     **end while**
10:     $Loop_1 = False$
11:     **Adapt invariants**
12:     $Loop_2 =$ **Invariants equal**
13: **end while**

4.3.2 Non-blocking predicate

The non-blocking predicate is computed in Algorithm 5. This program first sets the non-blocking conditions to the invariant for all the locations, otherwise the non-blocking predicate is set to *False*. This happens on line $3-9$. On line $12-16$, the first part of the non-blocking condition is calculated, corresponding to number 1 and 2 of Algorithm 5.1 in Rashidinejad et al., (2021). Here the new non-blocking conditions is calculated according to the following equation:

$$Predicate^{n+1} = Predicate^{n} \vee \bigvee_{l \xrightarrow{\sigma,g,r} l_t} (g \wedge l_t.i \wedge l_t.NB) \tag{4.3}$$

Number three from Algorithm 5.1 in Rashidinejad et al., (2021) could not be easily computed by the computer so the user is asked to solve that part, it is given below as well. This implies that if for a time delay Δ the non-blocking condition is *True*, then for all $\delta \leq \Delta$, the invariant should also be *True*.

$$\exists \Delta l.NB(t+\Delta) \wedge \forall \delta \leq \Delta l.i(t+\delta) \tag{4.4}$$

To check whether the non-blocking condition has changed from one iteration to another, the boolean $Finished$ is used. This variable is set to *False* if a condition has changed. Checking if the condition has changed is done by taking the conjunction of the old non-blocking condition with the negation of the new non-blocking condition. If neither of them is *False* in any case, this conjunction should evaluate to *False*. This can intuitively be understood by saying $x \wedge \neg y$, which is always *False* if $x = y$. The check for either of them being *False*, but not both, is done on line 21.

Algorithm 5 Non-blocking conditions

Input: Automaton A;
Output: Automaton A;

Boolean $Finished = False$
for $l \in A.L$ **do** ▷ Appendix 8.6, Line 204 − 219
  if $l.marked$ **then**
    $l.NB = l.i$
  else
    $l.NB = False$
  end if
end for

while $\neg Finished$ **do** ▷ Appendix 8.6, Line 222 − 294
  for $l \in A.L$ **do** ▷ Appendix 8.6, Line 225 − 269
    $Predicate = L.NB$
    for $e \in L.E$ **do**
      $Predicate = Predicate \vee (e.l_t.i \wedge e.g \wedge e.l_t.NB)$
    end for
    $Predicate = Predicate \vee$ **PartThree**$(l.i, l.NB)$
    $l.NB_{temp} = Predicate$
  end for

  for $l \in A.L$ **do** ▷ Appendix 8.6, Line 272 − 292
    if $(l.NB_{temp} = False) \oplus (l.NB = False)$ **then**
      $Finished = false$
      Break
    else if $\neg((l.NB_{temp} \wedge \neg l.NB) = False)$ **then**
      $Finished = false$
      Break
    else
      $Finished = true$
      Break
    end if
    $l.NB = l.NB_{temp}$
  end for
end while

To minimize the amount of times user input is needed, the trivial case where either or both $l.NB$ or $l.I$ are *False* is solved automatically in Matlab script 4.2. The script is explained below.

Listing 4.2: Function to check for the trivial case of part three from algorithm 5.1 from Rashidinejad et al., (2021). The trivial case is where either the non-blocking condition or the invariant is False or if one is True

```
function result = clock_regions(N, I)
    N_sym = str2sym(N);
    I_sym = str2sym(I);
    var_N = symvar(N_sym);
    var_I = symvar(I_sym);
    if (N_sym==symfalse)
        result = string(simplify(I_sym, 20));
        return;
    end
    if (I_sym==symfalse)
        result = string(simplify(I_sym, 20));
        return;
    end
    if (N_sym==symtrue || I_sym==symtrue)
        result = string(simplify(I_sym, 20));
        return;
    end
    result = "null";
    return;
end
```

This script takes as input the N and I. In this case, N can be either the non-blocking predicate or the bad-state predicate. The predicates are first converted to symbolic equations and then checked for either or both of them being *False*. The interpretation from Rashidinejad et al., (2021) of Equation 4.4 reads: "The condition to stay (for some time delay $\delta \leq \Delta$) in a non-blocking location as long as the invariant is satisfied". This means that, in the case that the invariant is *True* for all time, then the non-blocking predicate must also be *True* for all time. Else, if the invariant is *False* for all time, the non-blocking predicate must also be *False*. In the case that the non-blocking predicate is *False*, the requirement says that the automaton is allowed in the location for as long as the invariant is satisfied, so the invariant is returned.

4.3.3 Bad-state predicate

The bad-state predicate from Algorithm 6 is computed in a similar way as the non-blocking predicate and is based on Algorithm 5.2 from Rashidinejad et al., (2021). First the predicate is set to the negation of the non-blocking predicate in line $3-4$. Next, the first part of the new bad-state predicate is computed according to the following equation from Rashidinejad et al., (2021):

$$Predicate^{n+1} = Predicate^{n} \vee \bigvee_{l \xrightarrow{\sigma_{unc}, g, r} l_t} (g \wedge l_t.i \wedge l_t.BS) \tag{4.5}$$

In the above Equation 4.5 it should be noted that in this equation, only the uncontrollable edges are taken into account. The final part of the bad-state predicate could again not easily be automatically computed. It did however become simpler because there are no forcible events. Forcible events are events that can preempt the passage of time, but they are not used in the platooning system described in Chapter 3. Therefore, the final part of the bad-state predicate can be computed by the user according to the following equation:

$$\exists \Delta l.BS(t+\Delta) \wedge \forall \delta \leq \Delta l.i(t+\delta) \tag{4.6}$$

This Equation 4.6 can be interpreted similar to Equation 4.4. The automaton should be allowed to stay in a location as long as the invariant is satisfied. In Algorithm 6, the user is asked to solve this equation on line 15. The user is given the bad-state predicate along with the invariant.

Algorithm 6 Bad-state conditions

Input: Automaton A;
Output: Automaton A;

```
Boolean Finished = False

for l ∈ A.L do                                              ▷ Appendix 8.6, Line 314 − 326
    Predicate = ¬l.NB
end for

while ¬Finished do                                          ▷ Appendix 8.6, Line 328 − 405
    for l ∈ A.L do                                          ▷ Appendix 8.6, Line 330 − 381
        for e ∈ l.E do                                      ▷ Appendix 8.6, Line 335 − 358
            if e.σ.uncontrollable then
                Predicate = Predicate ∨ (e.l_t.i ∧ e.g ∧ e.l_t.BS)
                Apply resets(Predicate)
            end if
        end for
        Predicate = Predicate ∨ PartThree(l.i, l.NB)
        l.BS_temp = Predicate
    end for

    for l ∈ A.L do                                          ▷ Appendix 8.6, Line 383 − 403
        if (l.BS_temp = False) ⊕ (l.BS = False) then
            Finished = false
            Break
        else if ¬((l.BS_temp ∧ ¬l.BS) = False) then
            Finished = false
            Break
        else
            Finished = true
            Break
        end if
        l.BS = l.BS_temp
    end for
end while
```

4.3.4 Adapting guards and invariants

Adaptation of the guards and the invariants is done in Algorithm 7 and 8 below. The new guard is equal to, for all outgoing uncontrollable edges, the disjunction of the old guards and the negation of the bad-state predicate of the target location. Uncontrollable events are uncontrollable, therefore the supervisor cannot adapt them, otherwise they would be controlled. The invariant can only be adjusted if there is an outgoing forcible event in the location. If that is the case, the new invariant is equal to the disjunction of the old invariant and the negation of the bad-state predicate. Forcible events are not used therefore no invariants are adapted.

Algorithm 7 Guard adaptation

Input: Automaton A;
Output: Automaton A;

for $l \in A.L$ **do** ▷ Appendix 8.6, Line $147 - 180$
    for $e \in l.E$ **do**
        if $e.\sigma.controllable$ **then**
            $Predicate = e.g \wedge \neg e.l_t.BS$
            $e.g = Predicate$
            end for
        end if
    end for
end for

Algorithm 8 Invariant adaptation

Input: Automaton A;
Output: Automaton A;

for $l \in A.L$ **do** ▷ Appendix 8.6, Line $115 - 134$
    for $e \in l.E$ **do**
        if $e.\sigma.forcible$ **then**
            $Predicate = l.i \wedge \neg l.BS$
            end for
        end if
    end for
    $l.i = Predicate$
end for

5 Single test case

This chapter will synthesize a supervisor by hand for a small automaton according to the method proposed in Rashidinejad et al., (2021). In the automaton from Figure 5.1, a single modified string from the complete synchronous product is shown. The string has been modified because in the original string, the guards from location 2 to 3 and from location 3 to 4 are always *False*. The reason that this remained in the product, is because the simplifications with a matlab script were not yet used. An explanation as to why the guards are always *False* is given in equation 4.2.

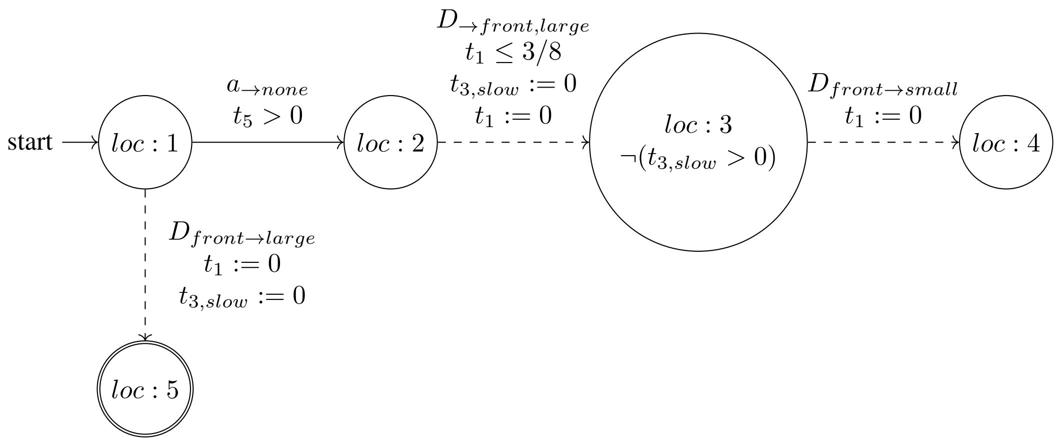

Figure 5.1: This automaton is a modified version of a single string from the synchronous product of the system. It is modified so that synthesis can be applied. In the actual test string, the edge from location 2 to 3 is not possible because the original guard is: $t_1 < 3/8 \wedge t_1 \geq 3/8$. Which is always False.

As a first step, the non-blocking conditions for all locations are computed. These are shown in Table 5.1. Initially, the non-blocking condition is set to the the invariant for all marked locations. Location 5 does not have an invariant so it is *True*. Next, for every location, Equation 4.3 is solved. In the case of location 1, this results in $False \vee True \vee (t_5 > 0 \wedge False) = True$. Part three from Equation 4.4 does not change anything about that since the the condition already is *True* for all time. Since there are no events going to location 1, iteration 3 does not change with respect to iteration 2.

Iteration	**loc:**1	**loc:**2	**loc:**3	**loc:**4	**loc:**5
1	*False*	*False*	*False*	*False*	*True*
2	*True*	*False*	*False*	*False*	*True*
3	*True*	*False*	*False*	*False*	*True*

Table 5.1: The non-blocking condition for the automaton from Figure 5.1

The bad-state conditions are shown in Table 5.2. Initially, all the bad-state conditions are set to the negation of the non-blocking conditions. This results in all conditions being *True* except for location 1 and 5. After one iteration, this does not change. For example, the bad-state predicate for location 2 resulted in: $True \vee (t_1 \leq 3/8 \wedge \neg(t_{3,slow} > 0) \wedge True) = True$.

Iteration	**loc:**1	**loc:**2	**loc:**3	**loc:**4	**loc:**5
1	*False*	*True*	*True*	*True*	*False*
2	*False*	*True*	*True*	*True*	*False*

Table 5.2: The bad-state condition for the automaton from Figure 5.1

The next step is updating the guards. Only the uncontrollable events can have their guards updated. The new guard is the conjunction between the old guards and the negation of the bad state of the target location. The only controllable event is the edge from location 1 to 2. The bad-state condition in location 2 is *True*, therefore the new guards is $t_5 > 0 \land \neg True = False$. The new automaton is shown in Figure 5.2. Invariants can not be adapted due to there being no forcible events in the automaton.

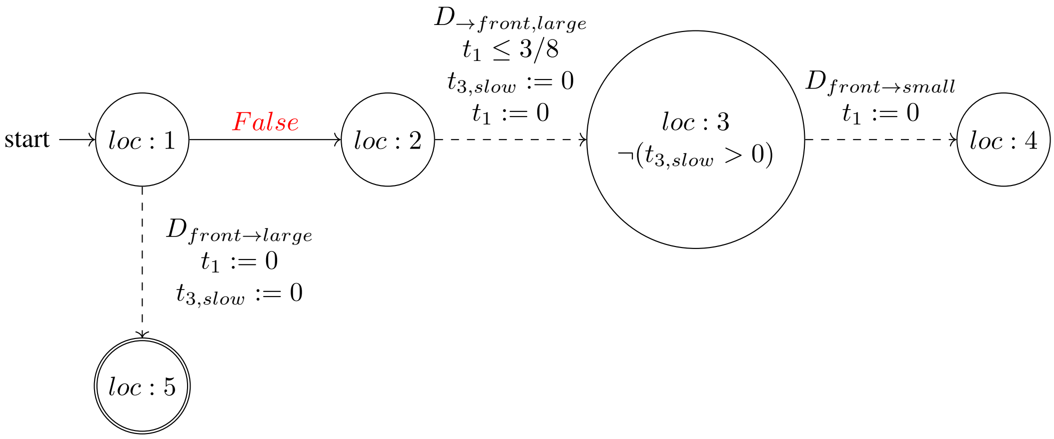

Figure 5.2: The supervisor automaton for Automaton 5.1. In red the new guard is shown. This guard is false so that the blocking location 4 can no longer be reached.

The supervisor for the automaton from Figure 5.1 is shown in Figure 5.2. In red the adapted guard is shown. The blocking location is location 4. Due to the guard from location 1 to location 2 being *False*, this location 4 can no longer be reached. The only possibility for this automaton is going from location 1 to the marked location 5.

6 Conclusion

First a model was made for one vehicle following another vehicle in a straight line, this resulted in eleven automata. In order to compute the synchronous product, a script was written in Java that takes as input the CIF code and outputs the synchronous product in CIF as well. The synchronous product had 147 states. Some of these states had guards that would never be satisfied. The reason for this is that the script that computes the product does not simplify the expression for the guards.

An attempt was made at automating the synthesis of a supervisor. The process relied heavily on the algorithm from Rashidinejad et al., (2021). Not all equations could be solved automatically so some are left to the user as input. Some trivial cases were solved automatically.

Since the a large amount of user inputs would be needed if synthesis would be applied to the product consisting of 147 states, a small string from the product is used. Synthesis is done by hand on this string. The result was that the blocking state could no longer be reached.

7 Notes

The scripts that compute the synchronous product and synthesize the supervisor require attention when used. Points to pay attention to and methods for improving the scripts are given below:

- As of right now, only automata can be parsed and not plants. The simple reason for this is that the parser looks for the word "automaton" and not for the "plant" word. This can easily be changed by changing the string in all cases.
- The parser does not check for the correctness of the CIF model. This means that if something is wrong with the CIF model, the product is also wrong.
- When timers from other automata are referenced in the CIF code with a dot, the program simply removes everything that precedes the dot. The means that if timers share the same name but different automata, in the synchronous product they will be equal.
- In the computation of the product, all information that precedes the word "*automaton*" is simply copied back in the code for the product. Information that is in between the word "*end*" and the word "*automaton*" is therefore lost in the product. This should be changed if the scripts is to be adapted.
- The scripts are inefficient. An example is cleaning the locations: In order to find the reachable locations, the script continuously runs over all locations. This can be improved by starting in the initial location and only run over the reachable locations.
- The simplification algorithm requires the Matab engine. Calling this engine from java is a slow process. Implementing a symbolic simplification algorithm would increase the speed.
- When using the algorithms to compute a product, the only variables can be timers. This has mostly to do with the way the variables are put back in the product. The product puts "*cont x der 1*" for all variables above the first location.
- For the computation of the synchronous product, the guards are not simplified. The script that is able to do this exists in Matab but is not yet implemented.
- The supervisor synthesis scripts asks the user to solve equations. If the scripts are to be used on large systems, those equations should be solved automatically.

Bibliography

Rashidinejad, A., Reniers, M., & Fabian, M. (2021). *Supervisory control synthesis of timed automata using forcible events*.

8 Appendix

8.1 Main file

```java
import java.io.FileWriter;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;

class main_file {
    private static String define_clocks(automaton [] automata) {
        ArrayList <String> clocks = new ArrayList<>();
        String text = "";
        for (automaton autom: automata) {
            for (edge edge: autom.edges) {
                for (String clock: edge.clock_resets) {
                    if (! clocks.contains(clock)) {
                        clocks.add(clock);
                    }
                }
            }
        }
        for (String clock: clocks) {
            text += "    cont " + clock + " der 1;\n";
        }
        return text;
    }

    private static int[] find (String main_text, String small_text) {           //Returns the occurances of a piece of small text in a larger text.
        int main_length = main_text.length();
        int small_length = small_text.length();
        String substring;
        int [] result_array;
        ArrayList<Integer> result = new ArrayList<Integer>();

        //Because arrays do not have a dynamic length, a list is used.
        for (int i=0; i<main_length-small_length+1; i++) {
            substring = main_text.substring(i, i+small_length);
            if (substring.equals(small_text)) {
                result.add(i);
            }
        }
        //The list is changed to an array because I find it easier to work with.
        result_array = new int[result.size()];
        for (int i=0; i<result.size(); i++) {
            result_array[i] = result.get(i);
        }
        return result_array;
    }

    private static automaton [] parse (String text) {
        automaton [] automata;
        String autom_name = "automaton ";
        String end_name = "end";
        String substring;
        int [] plant_occurances;
        int [] end_occurances;
        int automaton_amount;
        int end_length = end_name.length();
        int start;
        int end = 0;

        //Find the indices of teh 'plant' word.
        plant_occurances = find(text, autom_name);
        end_occurances = find(text, end_name);

        //The amound of occurances of the 'plant' word is an indication of the amount of plants in the system.
        automaton_amount = plant_occurances.length;
        automata = new automaton [automaton_amount];
```

```

        //The plant runs from the 'plant' word till the 'end' word.
        for (int i=0; i<automaton_amount; i++) {
            //The index of the start of the plant is the same as the index of the word 'plant'
            start = plant_occurances[i];
            //The plant ends the first time the 'end' word occurs.
            for (int j: end_occurances) {
                if (j > start) {
                    end = j;
                    end = end + end_length;
                    break;
                }
            }
            //Send the plant to the new automaton and organize the automaton.
            substring = text.substring(start, end);
            automata[i] = new automaton();
            automata[i].organize_text(substring);
        }

        return automata;
    }

    private static automaton compute_product (automaton [] automata, boolean print) {
        //Compute the product betweent the first two automata.
        if (automata.length == 1) {
            return automata[0];
        }
        int automaton_amount = automata.length;
        automaton product = new automaton();
        try {
            if (print) { System.out.println("Computing product between \"" + automata[0].name
    + "\" and \"" + automata[1].name + "\""); }
            product = ProductComputation.compute(automata[0], automata[1], print);
        }
        catch (ArrayIndexOutOfBoundsException e) {}

        //Compute the product between the subsequent automata.
        for (int i=2; i<automaton_amount; i++) {
            if (print) { System.out.println("\nAdding \"" + automata[i].name + "\" to the
    product. "); }
            product = ProductComputation.compute(product, automata[i], print);
        }

        return product;
    }
    public static void main(String args[]) throws Exception {
        automaton [] automata;
        automaton product;

        String clock_text;
        String pre_text;
        int [] plant_occurances;

        //Read the text file
        Path path = Path.of("text_BusPed.txt");
        String text = Files.readString(path);

        plant_occurances = find(text, "automaton");
        automata = parse(text);
        product = compute_product(automata, false);

        product = ControllerSynthesis.synthesize(product, true);

        //Generate the text as a large string.
        pre_text = text.substring(0, plant_occurances[0]);
        clock_text = define_clocks(automata);
        product.generate_text(clock_text, pre_text);
        String printing_text = product.text;


        //Write to a file.
        FileWriter myWriter = new FileWriter("Product_automaton.txt");
        myWriter.write(printing_text);
```

```
        myWriter.close();

    }
}
```

8.2 Automaton class

```
import java.util.ArrayList;

public class automaton {
    ArrayList<String> alphabet = new ArrayList<String>();
    String text = "";
    String name = "";
    location [] locations;
    edge [] edges;

    private int[] find (String main_text, String small_text) {
        int main_length = main_text.length();
        int small_length = small_text.length();
        String substring;
        int [] result_array;
        ArrayList<Integer> result = new ArrayList<Integer>();

        for (int i=0; i<main_length-small_length+1; i++) {
            substring = main_text.substring(i, i+small_length);
            if (substring.equals(small_text)) {
                result.add(i);
            }
        }
        result_array = new int[result.size()];
        for (int i=0; i<result.size(); i++) {
            result_array[i] = result.get(i);
        }
        return result_array;
    }

    private edge [] add_edge (edge [] edge_original, edge edge_new) {
        //Function that adds an edge to an array of edge methods.
        int length;
        try {
            length = edges.length + 1;
        }
        catch (Exception e){
            length = 1;
        }
        edge [] edges = new edge[length];
        for (int i=0; i<length-1; i++) {
            edges[i] = edge_original[i];
        }
        edges[length-1] = edge_new;
        return edges;
    }


    public void generate_text(String clocks, String pre_text) {
        for (location loc: locations) {
            loc.generate_text();
        }
        text += pre_text;
        text += "automaton " + name + ":\n";
        text += clocks;
        for (location loc: locations) {
            text += loc.text + "\n";
        }
        text += "end";
    }

    public void organize_text(String text) {
        this.text = text;

        String location_name = "location";
        String plant_name = "automaton";
        String end_name = "end";
        String substring;
        String DDot = ":";
        int [] location_indices;
        int [] end_index;
        int location_amount;
```

```
        int plant_length = plant_name.length();
        int [] index_1;
        int [] index_2;

        //Find the indices of the 'location' word and the 'end' word.
        location_indices = find(text, location_name);
        end_index = find(text, end_name);
        location_amount = location_indices.length;
        locations = new location[location_amount];

        // Find the indices for the plant name.
        index_1 = find(text, plant_name);
        index_1[0] += plant_length;
        index_2 = find(text, DDot);

        //Save the plant name
        this.name = text.substring(index_1[0]+1, index_2[0]);

        // Find the locations. The location ends when a new location start or the 'end' word
    occurs.
        for (int i=0; i<location_amount; i++) {
            //First try from one location till the next locaiton.
            try {
                substring = text.substring(location_indices[i], location_indices[i+1]);
                locations[i] = new location();
                locations[i].organize_text(substring, name);
            }
            //If there is no new location, the location gives an 'IndexOutOfBounds' error,
    when search for the 'end' word.
            catch (ArrayIndexOutOfBoundsException e){
                substring = text.substring(location_indices[i], end_index[0]);
                locations[i] = new location();
                locations[i].organize_text(substring, name);
            }
        }
        for (location i: locations) {
            for (edge j: i.edges) {
                this.edges = add_edge(edges, j);
                if (! alphabet.contains(j.event)) {
                    alphabet.add(j.event);
                }
            }
        }
    }
}
```

8.3 Location class

```
import java.util.ArrayList;

public class location {
    ArrayList<String> next_stops = new ArrayList<String>();
    ArrayList<String> invariant_list = new ArrayList<>();
    String matlab_temp;
    String [] matlab_nonblocking_pred;
    String [] matlab_bad_state;
    String [] matlab_invariants;
    String [] name_sum = new String[2];
    String text;
    String name;
    String automaton;
    edge [] edges;
    boolean initial;
    boolean marked;

    public void create_matlab_invariants() {

        this.matlab_invariants = new String[1];
        for (String invariant: invariant_list) {
            if (this.matlab_invariants[0] == null) {
                this.matlab_invariants[0] = "( " + invariant + " )";
            }
            else {
                this.matlab_invariants[0] += " & ( " + invariant + " )";
            }
        }

    }

    private int[] find (String main_text, String small_text) {
        int main_length = main_text.length();
        int small_length = small_text.length();
        String substring;
        int [] result_array;
        ArrayList<Integer> result = new ArrayList<Integer>();

        for (int i=0; i<main_length-small_length+1; i++) {
            substring = main_text.substring(i, i+small_length);
            if (substring.equals(small_text)) {
                result.add(i);
            }
        }
        result_array = new int[result.size()];
        for (int i=0; i<result.size(); i++) {
            result_array[i] = result.get(i);
        }
        return result_array;
    }

    private String remove_whitespace (String word) {
        int length = word.length();
        String space = " ";
        String NewName = "";
        String substring;

        for (int i=0; i<length; i++) {
            substring = word.substring(i, i+1);
            if (!(substring.equals(space))) {
                NewName += word.substring(i, i+1);
            }
        }
        return NewName;
    }

    private String replace(String string, String initial, String result) {
        int [] index;
        int length_small = initial.length();
        int length_large;
        String substring1;
```

```
        String substring2;

        while (true ) {
            index = find(string, initial);
            try {
                length_large = string.length();
                substring1 = string.substring(0, index[0]);
                substring2 = string.substring(index[0]+length_small, length_large);
                string = substring1 + result + substring2;
            }
            catch (ArrayIndexOutOfBoundsException e) { return string;}
        }
    }

    private void create_invariant_text() {

        int length = matlab_invariants.length;
        String invariant = matlab_invariants[length-1];

        this.invariant_list = new ArrayList<String>();

        invariant = replace(invariant, "|", "or");
        invariant = replace(invariant, "&", "and");
        invariant = replace(invariant, "symtrue", "true");
        invariant = replace(invariant, "symfalse", "false");

        this.invariant_list.add(invariant);
    }

    public ArrayList<String> get_next_stops() {
        ArrayList<String> result = new ArrayList<String>();
        for (edge edges: edges) {
            result.add(edges.destination_location);
        }
        return result;
    }

    public void generate_text() {
        try {
            create_invariant_text();
        }
        catch (NullPointerException e) {}
        text = "";
        text = "    location " + name + ":\n";
        if (initial) {
            text += "        initial;\n";
        }
        if (marked) {
            text += "        marked;\n";
        }
        for (String invariant: invariant_list) {
            text += "        invariant " + invariant + ";\n";
        }
        for (edge edge: edges) {
            edge.generate_text();
            text += "        " + edge.text + "\n";
        }
        if (edges.length == 0) {
            text += "        " + "edge tau;\n";
        }
    }

    public void print_edges() {
        for (edge i: edges) {
            i.print_info();
        }
    }

    public void organize_text (String text, String automaton) {
        this.text = text;

        String substring;
        String initial = "initial";
```

```
        String marked = "marked";
        String location_name = "location";
        String DDot = ":";
        String DComma = ";";
        String edge = "edge";
        String invariant = "invariant";
        int name_length = location_name.length();
        int invariant_length = invariant.length();
        int edges_amount;
        int start = 0;
        int end = 0;
        int [] initial_index;
        int [] marked_index;
        int [] name_index;
        int [] DDot_index;
        int [] edge_index;
        int [] Dcomma_index;
        int [] invariant_index;

        initial_index = find(text, initial);
        marked_index = find(text, marked);
        name_index = find(text, location_name);
        name_index[0] += name_length;
        DDot_index = find(text, DDot);
        edge_index = find(text, edge);
        Dcomma_index = find(text, DComma);
        invariant_index = find(text, invariant);
        edges_amount = edge_index.length;


        // Determine weather the location is marked or initial;
        if (initial_index.length != 0) {
            this.initial = true;
        }
        else {
            this.initial = false;
        }
        if (marked_index.length != 0) {
            this.marked = true;
        }
        else {
            this.marked = false;
        }

        // Determine the name of the location;
        try {
            this.name = text.substring(name_index[0]+1, DDot_index[0]);
        }
        catch (StringIndexOutOfBoundsException e) {
            this.name = "NAMELESS";
        }
        // Find the invariants
        for (int i=0; i<invariant_index.length; i++) {
            start = invariant_index[i] + invariant_length;
            for (int j: Dcomma_index) { if (j > start) {end = j; break;}}
            substring = remove_whitespace(text.substring(start, end));
            invariant_list.add(substring);
        }

        // save the edges;
        edges = new edge[edge_index.length];
        for (int i=0; i<edges_amount; i++) {
            //The location starts when the 'edge' word occurs.
            start = edge_index[i];
            //The location ends at the first occurance of the ';' symbol.
            for (int j: Dcomma_index) { if (j > start) {end = j+1; break; } }
            substring = text.substring(start, end);
            edges[i] = new edge();
            edges[i].organize_text(substring, name, automaton);
        }

        //Convert to matlab-readable invariants
    }
```

```
}
```

8.4 Edge class

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class edge {
    String [] destination_sum = new String[2]; //index #0 is the original location, index #1
    is the location of the new automaton.
    ArrayList <String> clock_resets = new ArrayList<>();
    String text;
    String event;
    String update;
    String [] guard;
    String initial_location;
    String destination_location;
    boolean forcible = false;
    String automaton;
    boolean uncontrollable;

    private int[] find (String main_text, String small_text) {
        int main_length = main_text.length();
        int small_length = small_text.length();
        String substring;
        int [] result_array;
        ArrayList<Integer> result = new ArrayList<Integer>();

        for (int i=0; i<main_length-small_length+1; i++) {
            substring = main_text.substring(i, i+small_length);
            if (substring.equals(small_text)) {
                result.add(i);
            }
        }

        result_array = new int[result.size()];
        for (int i=0; i<result.size(); i++) {
            result_array[i] = result.get(i);
        }
        return result_array;
    }

    private String remove_whitespace (String word) {
        int length = word.length();
        String space = " ";
        String NewName = "";
        String substring;

        for (int i=0; i<length; i++) {
            substring = word.substring(i, i+1);
            if (!(substring.equals(space))) {
                NewName += word.substring(i, i+1);
            }
        }
        return NewName;
    }

    public void remove_update_reference () {
        if (guard != null) {
            String substring;
            int [] dot_loc;
            int length = guard[0].length();

            dot_loc = find(guard[0], ".");
            try {
                substring = guard[0].substring(dot_loc[0]+1, length);
                guard[0] = substring;
            }
            catch (ArrayIndexOutOfBoundsException e) {}
        }
    }

    private String replace(String string, String initial, String result) {
        int [] index;
```

```java
        int length_small = initial.length();
        int length_large;
        String substring1;
        String substring2;

        while (true ) {
            index = find(string, initial);
            try {
                length_large = string.length();
                substring1 = string.substring(0, index[0]);
                substring2 = string.substring(index[0]+length_small, length_large);
                string = substring1 + result + substring2;
            }
            catch (ArrayIndexOutOfBoundsException e) { return string;}
        }
    }

    public void generate_text() {
        String edge_text;
        String condition_text;
        String destination_location_text;
        String update_text;
        String temp_text = "";
        String temp_guard;
        try {
            temp_guard = this.guard[this.guard.length-1];
            temp_guard = replace(temp_guard, "|", "or");
            temp_guard = replace(temp_guard, "&", "and");
            temp_guard = replace(temp_guard, "symfalse", "false");
            temp_guard = replace(temp_guard, "symtrue", "true");
        }
        catch (NullPointerException e) {temp_guard = null;}

        edge_text = "edge " + event;
        temp_text += edge_text;

        if (destination_sum[0] != null && destination_sum[1] != null) {
            destination_location_text = destination_sum[0] + "__" + destination_sum[1];
        }

        if (temp_guard != null) {
            condition_text = " when " + temp_guard;
            temp_text += condition_text;
        }
        if (update != null) {
            update_text = " do " + update;
            temp_text += update_text;
        }
        if (destination_location != null) {
            destination_location_text = " goto " + destination_location;
            temp_text += destination_location_text;
        }
        temp_text += ";";
        this.text = temp_text;
    }

    public void print_info() {
        System.out.println("
    ------------------------------------------------------------------------------");
        System.out.println("Host automaton is: " + automaton);
        System.out.println("Initial location is: " + initial_location);
        System.out.println("Destination location is: " + destination_location);
        System.out.println("Event name is: " + event);
        System.out.println("Event condition  is: " + guard);
        System.out.println("Event update is: " + update);
        System.out.println("Text is: " + text);
    }

    public void organize_text(String text, String initial_location, String automaton) {
        this.text = text;
        this.initial_location = initial_location;
        this.automaton = automaton;

```

```
        String substring = "";
        String edge_text = "edge ";
        String Dcomma = ";";
        String goto_name = "goto ";
        String when_name = "when ";
        String do_name = "do ";
        String clock_reset = ":=0";
        List<Integer> index_list = new ArrayList<>();
        int when_length = when_name.length();
        int do_length = do_name.length();
        int goto_length = goto_name.length();
        int edge_length = edge_text.length();
        int [] clock_reset_index = null;
        int [] index_array;
        int [] do_index;
        int [] Dcomma_index;
        int [] goto_index;
        int [] edge_index;
        int [] when_index;
        int start = 0;
        int end = 0;

        do_index = find(text, do_name);
        edge_index = find(text, edge_text);
        Dcomma_index = find(text, Dcomma);
        goto_index = find(text, goto_name);
        when_index = find(text, when_name);

        for (int i: edge_index) { index_list.add(i); }
        for (int i: Dcomma_index) { index_list.add(i); }
        for (int i: goto_index) { index_list.add(i); }
        for (int i: do_index) { index_list.add(i); }
        for (int i: when_index) { index_list.add(i); }

        Collections.sort(index_list);
        index_array = new int[index_list.size()];
        for (int i=0; i<index_list.size(); i++) {
            index_array[i] = index_list.get(i);
        }

        //Find the event name:
        start = edge_index[0] + edge_length;
        for (int i: index_array) { if (i > start) { end = i; break;} }
        substring = text.substring(start, end);
        this.event = remove_whitespace(substring);

        //Find the event destination:
        if (goto_index.length != 0) {
            start = goto_index[0] + goto_length;
            for (int i: index_array) { if (i > start) { end = i; break;} }
            substring = text.substring(start, end);
            this.destination_location = remove_whitespace(substring);
        }

        //Find the event update:
        if (do_index.length != 0) {
            start = do_index[0] + do_length;
            for (int i: index_array) { if (i > start) { end = i; break;} }
            substring = text.substring(start, end);
            this.update = remove_whitespace(substring);
            clock_reset_index = find(this.update, clock_reset);
        }

        if (clock_reset_index != null) {
            for (int index: clock_reset_index) {
                this.clock_resets.add(this.update.substring(0, index));
            }
        }

        //Find the event condition:
        if (when_index.length != 0) {
            start = when_index[0] + when_length;
            for (int i: index_array) { if (i > start) { end = i; break;} }
```

```
            substring = text.substring(start, end);
            this.guard = new String[1];
            this.guard[0] = remove_whitespace(substring);
        }

        if (event.substring(0, 1).equals("u")) {
            this.uncontrollable = true;
        }
        else {
            this.uncontrollable = false;
        }
    }
}
```

8.5 Computing the product

```java
import java.util.ArrayList;

public class ProductComputation {
    static location [] locations;
    static edge [] edges;
    static ArrayList<String> alphabet = new ArrayList<String>();
    static String text = "";
    static String name = "";
    static boolean print;

    //Combine 2 locations
    private static edge [] insert_edge (edge edge_origin [], edge edge_new) {
        //Insert a new edge into a list.
        int length_result = edge_origin.length;
        length_result++;
        edge [] result = new edge[length_result];

        for (int i=0; i<length_result-1; i++) {
            result[i] = new edge();
            if (edge_origin[i].destination_sum[0] != null && edge_origin[i].destination_sum[1] != null) {
                result[i].destination_sum = new String[2];
                result[i].destination_sum[0] = new String(edge_origin[i].destination_sum[0]);
                result[i].destination_sum[1] = new String(edge_origin[i].destination_sum[1]);
            }
            result[i].text = edge_origin[i].text;
            result[i].event = edge_origin[i].event;
            result[i].update = edge_origin[i].update;
            result[i].guard = edge_origin[i].guard;
            result[i].initial_location = name;
            result[i].destination_location = edge_origin[i].destination_location;
            result[i].automaton = edge_origin[i].automaton;
            for (String clock_origin: edge_origin[i].clock_resets) {
                if (! result[i].clock_resets.contains(clock_origin)) {
                    result[i].clock_resets.add(clock_origin);
                }
            }
        }
        result[length_result-1] = new edge();
        if (edge_new.destination_sum[0] != null && edge_new.destination_sum[1] != null) {
            result[length_result-1].destination_sum = new String[2];
            result[length_result-1].destination_sum[1] = new String(edge_new.destination_sum[1]);
            result[length_result-1].destination_sum[0] = new String(edge_new.destination_sum[0]);
        }
        result[length_result-1].text = edge_new.text;
        result[length_result-1].event = edge_new.event;
        result[length_result-1].update = edge_new.update;
        result[length_result-1].guard = edge_new.guard;
        result[length_result-1].initial_location = name;
        result[length_result-1].destination_location = edge_new.destination_location;
        result[length_result-1].automaton = edge_new.automaton;
        for (String clock_new: edge_new.clock_resets) {
            if (! result[length_result-1].clock_resets.contains(clock_new)) {
                result[length_result-1].clock_resets.add(clock_new);
            }
        }

        return result;
    }

    private static edge merge_edges(edge edge_origin, edge edge_new) {
        //edge conditions and updates are merged.
        edge result = new edge();
        String condition;
        String update;
        int length_origin;
        int length_new;

        if ((edge_origin.guard != null) && (edge_new.guard != null)) {
```

```
            length_origin = edge_origin.guard.length;
            length_new = edge_origin.guard.length;
            condition = edge_origin.guard[length_origin-1] + " and " + edge_new.guard[length_new-1];
            result.guard = new String[1];
            result.guard[0] = condition;
        }
        else if ((edge_origin.guard != null) && (edge_new.guard == null)) {
            length_origin = edge_origin.guard.length;
            condition = edge_origin.guard[length_origin-1];
            result.guard = new String[1];
            result.guard[0] = condition;
        }
        else if ((edge_origin.guard == null) && (edge_new.guard != null)) {
            length_new = edge_new.guard.length;
            condition = edge_new.guard[length_new-1];
            result.guard = new String[1];
            result.guard[0] = condition;
        }

        if ((edge_origin.update != null) && (edge_new.update != null)) {
            update = edge_origin.update + ", " + edge_new.update;
            result.update = update;
        }
        else if ((edge_origin.update == null) && (edge_new.update != null)) {
            update = edge_new.update;
            result.update = update;
        }
        else if ((edge_origin.update != null) && (edge_new.update == null)) {
            update = edge_origin.update;
            result.update = update;
        }

        if ((edge_origin.destination_location != null) && edge_new.destination_location != null) {
            result.destination_sum[0] = edge_origin.destination_location;
            result.destination_sum[1] = edge_new.destination_location;
        }
        else if ((edge_origin.destination_location != null) && edge_new.destination_location == null) {
            result.destination_sum[0] = edge_origin.destination_location;
            result.destination_sum[1] = edge_new.initial_location;
        }
        else if ((edge_origin.destination_location == null) && edge_new.destination_location != null) {
            result.destination_sum[0] = edge_new.initial_location;
            result.destination_sum[1] = edge_new.destination_location;
        }

        for (String clock: edge_origin.clock_resets) {
            if (! result.clock_resets.contains(clock)) {
                result.clock_resets.add(clock);
            }
        }

        for (String clock: edge_new.clock_resets) {
            if (! result.clock_resets.contains(clock)) {
                result.clock_resets.add(clock);
            }
        }

        result.event = edge_origin.event;
        return result;
    }

    private static edge [] add_edge (String name, edge[] edges_origin, edge[] edges_new, ArrayList<String> alphabet_autom1, ArrayList<String> alphabet_autom2) {
        ArrayList<String> doubles = new ArrayList<String>();
        ArrayList<String> double_text = new ArrayList<String>();
        ArrayList<String> edges_loc_new = new ArrayList<String>();
        ArrayList<String> edges_loc_origin = new ArrayList<String>();
        String initial_location_origin = name;
        String initial_location_new = edges_new[0].initial_location;
```

```
        edge [] checked = new edge[0];
        boolean alphabet_other;
        boolean location_other;
        boolean allowed;
        edge temp;

        for (edge edge_origin: edges_origin) {
            edges_loc_origin.add(edge_origin.event);
        }
        for (edge edge_new: edges_new) {
            edges_loc_new.add(edge_new.event);
        }

        //Check for all edges that occur in both locations, merge those.
        for (int j=0; j<edges_origin.length; j++) {
            for (int i=0; i<edges_new.length; i++) {
                if (edges_origin[j] != null && edges_new[i] != null && edges_origin[j].event.
    equals(edges_new[i].event)) {
                    temp = merge_edges(edges_origin[j], edges_new[i]);
                    checked = insert_edge(checked, temp);
                    doubles.add(edges_new[i].event);
                    double_text.add(edges_new[i].text);
                }
            }
        }
        //Check for all edges of the first automaton and see if they are allowed. They are
    allowed if the other automaton is not blocking that edge.
        for (edge edge_origin: edges_origin) {
            alphabet_other = alphabet_autom2.contains(edge_origin.event);
            location_other = edges_loc_new.contains(edge_origin.event);
            allowed = ! alphabet_other && ! location_other;
            if (allowed) {
                checked = insert_edge(checked, edge_origin);
                checked[checked.length-1].initial_location = name;
                checked[checked.length-1].destination_sum[0] = edge_origin.
    destination_location;
                checked[checked.length-1].destination_sum[1] = initial_location_new;
            }
        }
        //Do the same for the new edges.
        for (edge edge_new: edges_new) {
            alphabet_other = alphabet_autom1.contains(edge_new.event);
            location_other = edges_loc_origin.contains(edge_new.event);
            allowed = ! alphabet_other && ! location_other;
            if (allowed) {
                checked = insert_edge(checked, edge_new);
                checked[checked.length-1].initial_location = name;
                checked[checked.length-1].destination_sum[0] = initial_location_origin;
                checked[checked.length-1].destination_sum[1] = edge_new.destination_location;
            }
        }
        return checked;
    }

    private static location Merge_locations(location loc_origin, location loc_new, ArrayList<
    String> alphabet_origin, ArrayList<String> alphabet_new) {
        location location = new location();
        ArrayList<String> next_stops = new ArrayList<String>();
        ArrayList<String> invariant_list = new ArrayList<>();
        String text = "";
        edge [] edges;

        edge [] edges_origin = loc_origin.edges;
        edge [] edges_new = loc_new.edges;

        location.marked = loc_origin.marked && loc_new.marked;
        location.initial = loc_origin.initial && loc_new.initial;
        location.name = loc_origin.name;
        location.name_sum[0] = loc_origin.name;
        location.name_sum[1] = loc_new.name;
        location.automaton = "product";

        //Remove the reference from the original edges. time_slow.t_1 becomes just t_1.
```

```
        for (edge edges_loc_new: loc_new.edges) {
            edges_loc_new.remove_update_reference();
        }
        //Combine all edges.
        edges = add_edge(location.name, edges_origin, edges_new, alphabet_origin, alphabet_new
    );

        //Combine invariants
        for (String invariant_origin: loc_origin.invariant_list) {
            if (! invariant_list.contains(invariant_origin)) {
                invariant_list.add(invariant_origin);
            }
        }

        for (String invariant_new: loc_new.invariant_list) {
            if (! invariant_list.contains(invariant_new)) {
                invariant_list.add(invariant_new);
            }
        }

        location.name = loc_origin.name + "_" + loc_new.name;
        location.next_stops = next_stops;
        location.invariant_list = invariant_list;
        location.text = text;
        location.edges = edges;

        return location;
    }

    private static void adjust_destinations (location loc) {
        //Every combination of edges has a small list that has contains the destinations of
    the two original edges.
        //This function looks for the location that corresponds to the combination of
    destinations of edges.
        Boolean found1;
        boolean found2;

        for (edge edge: loc.edges) {
            for (location location: locations) {
                if (edge.destination_sum[0] != null && location.name_sum[0] != null) {
                    found1 = edge.destination_sum[0].equals(location.name_sum[0]);
                }
                else { found1 = false;}
                if (edge.destination_sum[1] != null && location.name_sum[1] != null) {
                    found2 = edge.destination_sum[1].equals(location.name_sum[1]);
                }
                else {found2 = false;}
                if (found1 && found2) {
                    edge.destination_location = location.name;
                }
            }
        }
    }

    //become automaton product
    private static void clean_locations() {
        ArrayList<String> nonBlocking = new ArrayList<String>();
        ArrayList<Integer> Accessible = new ArrayList<Integer>();
        ArrayList<String> Accessible_locations = new ArrayList<String>();
        Boolean finished = false;
        int removed_locations = 0;

        if (print) { System.out.println("Cleaning locations."); }

        for (int i=0; i< locations.length; i++) {
            if (locations[i].initial == true) {
                nonBlocking.add(locations[i].name);
                Accessible.add(i);
                Accessible_locations.add(locations[i].name);
                break;
            }
        }
        // Loop over all locations untill no new location is added to the list of accessible
```

```
    locations
        while (! finished) {
            finished = true;
            //Loop over all locations
            for (location loc: locations) {
                // If a location is part of the list of accessible locations, add it's edge
    destination to the list of accessible locations
                if (Accessible_locations.contains(loc.name)) {
                    for (edge edge: loc.edges) {
                        // Only add a new destination if it is not alreay part of the list of
    accessible locations
                        if (! Accessible_locations.contains(edge.destination_location)) {
                            Accessible_locations.add(edge.destination_location);
                            finished = false;
                        }
                    }
                }
            }
        }
        //If a location is not part of the list of accessible locations, remove it.
        finished = false;
        while (! finished) {
            finished = true;
            for (int i=0; i<locations.length; i++) {
                if (! Accessible_locations.contains(locations[i].name)) {
                    remove_location(i);
                    finished = false;
                    removed_locations++;
                }
           }
        }
        if (print) {System.out.println("Removed " + removed_locations + " locations.");}
    }

    private static void remove_location(int remove) {
        //Function that removes the location corresponding to the index 'remove'
        int new_length = locations.length-1;
        location [] temp = new location[new_length];
        for (int i=0; i<remove; i++) {
            temp[i] = locations[i];
        }
        for (int i=remove+1; i<new_length+1; i++) {
            temp[i-1] = locations[i];
        }
        locations = temp;
    }

    private static void become_product (automaton plant_origin, automaton plant_new) {
        //Generate locations equal to the total number of locations.
        int location_amount = plant_origin.locations.length * plant_new.locations.length;
        int i = 0;
        int index = 0;
        double percentage;
        double i_double;
        double location_amount_double = location_amount;
        if (print) {
            System.out.println("The number of new locations is: " + location_amount);
            System.out.print("Combining locations:    ");
        }

        locations = new location[location_amount];
        name = "product";

        //Loop over every combination of locations and merge all edges.
        for (location plant_origin_loc: plant_origin.locations) {
            for (location plant_new_loc: plant_new.locations) {
                locations[i] = new location();
                locations[i] = Merge_locations(plant_origin_loc, plant_new_loc, plant_origin.
    alphabet, plant_new.alphabet);
                i++;
                i_double = i;
                percentage = ((i_double / location_amount_double) * 100);
                if (percentage-index > 0 && print) {
```

```
                    for (int j=index ;j<percentage; j++) {
                        System.out.print("-");
                        index++;
                    }
                }
            }
        }
        i = 0;
        index = 0;
        if (print) {
            System.out.print("\n");
            System.out.print("Adjusting destinations: ");
        }
        //For all merged edges, find the new name of the destination.
        for (location loc: locations) {
            adjust_destinations(loc);
            i++;
            i_double = i;
            percentage = ((i_double / location_amount_double) * 100);
            if (percentage-index > 0 && print) {
                for (int j=index ;j<percentage; j++) {
                    System.out.print("-");
                    index++;
                }
            }
        }
        if (print) {System.out.print("\n");}
        //Remove the unaccessible destinations. This saves a lot of computing time.
        clean_locations();
        //Adjust the alphabet of the product.
        for (location loc: locations) {
            for (edge edge: loc.edges) {
                if (! alphabet.contains(edge.event)) {
                    alphabet.add(edge.event);
                }
            }
        }
    }

    //The main function to be called
    public static automaton compute(automaton automaton_1, automaton automaton_2, boolean
    printing) {
        automaton product = new automaton();
        print = printing;
        name = "product";
        locations = automaton_1.locations;
        edges = automaton_1.edges;

        become_product(automaton_1, automaton_2);

        product.name = name;
        product.locations = locations;
        product.edges = edges;
        product.alphabet = alphabet;

        return product;
    }
}
```

8.6 Controller synthesis

```
import java.util.ArrayList;
import java.util.Scanner;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.RejectedExecutionException;

import com.mathworks.engine.*;

public class ControllerSynthesis {
    static MatlabEngine matlab;
    static automaton automaton;
    static boolean print;

    private static void printing(String to_print) {
        if (print) {
            System.out.print(to_print);
        }
    }

    private static String last(String [] string) {
        int length;
        String result;
        try {
            length = string.length;
        }
        catch (NullPointerException e) {
            return null;
        }

        result = string[length-1];
        return result;
    }

    private static String [] add_string(String [] original_string, String new_string) {
        int length;
        try {
            length = original_string.length;
        }
        catch (ArrayIndexOutOfBoundsException e) {
            length = 0;
        }
        catch (NullPointerException e) {
            original_string = new String[0];
            length = 0;
        }
            String [] string = new String[length+1];

        for (int i=0; i<length; i++) {
            string[i] = original_string[i];
        }
        string[length] = new_string;

        return string;
    }

    private static String [] list_to_string(ArrayList<String> list) {
        String [] string;
        int length = list.size();

        if (list.size() == 0) {
            return null;
        }

        string = new String[length];
        for (int i=0; i<length; i++) {
            string[i] = list.get(i);
        }
        return string;
    }

    private static String target_invariant (String destination) {
        String predicate;
```

```
        for (location location: automaton.locations) {
            if (location.name.equals(destination)) {
                if (location.matlab_invariants != null) {
                    predicate = last(location.matlab_invariants);
                    return predicate;
                }
                else {
                    return "symtrue";
                }
            }
        }
        return null;
    }

    private static String target_nonblock (String destination) {
        String predicate;
        for (location location: automaton.locations) {
            if (location.name.equals(destination)) {
                predicate = last(location.matlab_nonblocking_pred);
                return predicate;
            }
        }
        return null;
    }

    private static String target_bad_state (String destination) {
        String predicate;
        for (location location: automaton.locations) {
            if (location.name.equals(destination)) {
                predicate = last(location.matlab_bad_state);
                return predicate;
            }
        }
        return null;
    }


    private static void adapt_invariants() throws RejectedExecutionException, EngineException,
     InterruptedException, ExecutionException {
        String predicate;

        // The new invariant for all locations is the conjuntion between the old invariant
        // and the negation of the bad-state predicate.
        printing("Adapting invariants\n");
        for (location location: automaton.locations) {
            for (edge edge: location.edges) {
                if (edge.forcible) {
                    printing("    New invariant for " + location.name + " is: ");
                    // If the location already has an invariant, the new bad-state predicate
    is added to that.
                    try {
                        predicate = "( " + last(location.matlab_invariants) + " )";
                        predicate += " & ~ ( " + last(location.matlab_bad_state) + " )";
                    }
                    // Else the invariant is set to true.
                    catch (NullPointerException e) {
                        predicate = "symtrue & ~ ( " + last(location.matlab_bad_state) + " )";
                    }
                    printing(predicate + "\n");
                    predicate = matlab.feval("simpler", predicate);
                    location.matlab_invariants = add_string(location.matlab_invariants,
    predicate);
                    break;
                }
            }
        }
        printing("\n----------------------------------------\n");
    }

    private static void adapt_guards() throws RejectedExecutionException, EngineException,
    InterruptedException, ExecutionException {
        String predicate;
        int guard_length;
```

```
        String target_bad_state;
        String [] resets;

        printing("Adapting guads\n");
        // Loop over all guards for all locations.  The new guard is the conjunction of the old guard
        // with the negation of the bad-state predicate of the target location.
        for (location location: automaton.locations) {
            printing("    New guards for location " + location.name + " are:\n");
            for (edge event: location.edges) {
                // See if there the event already has a guard, else the guard is set to true.
                try {
                    guard_length = event.guard.length;
                }
                catch (NullPointerException e) {
                    event.guard = new String[1];
                    event.guard[0] = "symtrue";
                    guard_length = 1;
                }
                // Only the guards of uncontrolleble events may be adjusted.
                if (! event.uncontrollable) {
                    target_bad_state = target_bad_state(event.destination_location);
                    predicate = "( " + event.guard[guard_length-1] + " )";
                    predicate += " & ~ ( "+ target_bad_state + " )";
                    resets = list_to_string(event.clock_resets);
                    if (resets != null) {
                    predicate = matlab.feval("simpler", predicate, resets);
                    }
                    else {
                        predicate = matlab.feval("simpler", predicate);
                    }
                    event.guard = add_string(event.guard, predicate);
                }
                //If the event was uncontrollable, the new guard is equal to the old guard.
                else {
                    event.guard = add_string(event.guard, event.guard[guard_length-1]);
                }
                printing("       " + event.event + ": " + event.guard[guard_length] + "\n");
            }

        }
        printing("\n-----------------------------------------\n");
    }

    private static void non_blocking() throws RejectedExecutionException, EngineException, InterruptedException, ExecutionException {
        String predicate;
        String edge_predicate;
        String edge_guard;
        String target_invariant;
        String target_nonblocking;
        String equality_test;
        String old_predicate;
        String new_predicate;
        String part_three;
        String invariant;
        String non_blocking;

        Scanner scan = new Scanner(System.in);
        boolean finished = false;
        printing("Computing the non-blocking conditions: \n");

        // Setting all the non-blocking conditions to the invariants for the marked locations
        // and the false for all other locations.
        printing("Initial non-blocking invariants are: \n\n");
        for (location location: automaton.locations) {
            location.matlab_nonblocking_pred = new String[1];
            if (location.marked) {
                if (location.matlab_invariants[0] == null) {
                    location.matlab_invariants[0] = "symtrue";
                    location.matlab_nonblocking_pred[0] = "symtrue";
                }
                else {
```

```
                location.matlab_nonblocking_pred = add_string(location.
matlab_nonblocking_pred, last(location.matlab_invariants));
            }
        }
        else {
            location.matlab_nonblocking_pred[0] = "symfalse";
        }
        printing(location.name + ": " + last(location.matlab_nonblocking_pred ) + "\n");
    }
    printing("\n");

    while (! finished) {
        finished = true;
        //Set the the predicate to the last non-blocking condition.
        for (location location: automaton.locations) {
            printing("    Computing non-blocking predicate for " + location.name + "\n");
            if (last(location.matlab_nonblocking_pred) != null) {
                predicate = last(location.matlab_nonblocking_pred);
            }
            else { predicate = "symtrue"; }
            location.matlab_temp = predicate;
            printing("    Initial predicate: " + predicate + "\n");
            // The new non-blocking condition is the last non-blocking condition and for
all edges,
            // the conjunction of the guard and the previous non-blocking condition.
            for (edge edge: location.edges) {
                printing("        add edge predicate for " + edge.event + ": ");
                edge_guard = last(edge.guard);
                if (edge_guard == null) {edge_guard = "symtrue";}

                target_nonblocking = target_nonblock(edge.destination_location);
                if (target_nonblocking == null) {target_nonblocking = "symtrue"; }

                target_invariant = target_invariant(edge.destination_location);
                if (target_invariant == null) {target_invariant = "symtrue"; }
                edge_predicate = " ( " + edge_guard + " & " + target_nonblocking + " & " +
 target_invariant + " )";
                printing(predicate + " | " + edge_predicate + " --> ");
                predicate += " |" + edge_predicate;
                predicate = matlab.feval("simpler", predicate);
                printing(predicate + "\n");
                location.matlab_temp = predicate;
            }
            invariant = last(location.matlab_invariants);
            if (invariant == null) {invariant = "symtrue";}
            non_blocking = last(location.matlab_nonblocking_pred);
            if (non_blocking == null) {non_blocking = "symtrue";}
            printing("        Part three inputs are: N: " + non_blocking + ", I: " +
invariant + ", ");
            part_three = matlab.feval("clock_regions", non_blocking, invariant);
            if (part_three.equals("null")) {
                printing("\n        Solve: " + non_blocking + " With some time delay Delta
 AND " + invariant + " with time delay delta for all time delta < Delta\n");
                part_three = scan.nextLine();
            }
            printing("Output is: " + part_three + "\n        Adding gives: ");
            predicate += " | ( " + part_three +  ")";
            printing(predicate + " --> ");
            predicate = matlab.feval("simpler", predicate);
            location.matlab_temp = predicate;
            printing(predicate + "\n\n");
            predicate = null;
        }
        // Here the finished condition is also checked. The condition is check by
computing (x & ~y), with x and y being logic equations. This is always false
        // if x is equal to y, or if y is false or if x is false.
        for (location location: automaton.locations) {
            printing("Final NBC are: " + location.name + ": " + location.matlab_temp + "\n
");
            old_predicate = last(location.matlab_nonblocking_pred);
            new_predicate = location.matlab_temp;
            equality_test = "test";
            if ((old_predicate.equals("symfalse")) ^ (new_predicate.equals("symfalse"))) {
```

```
                    equality_test = "dont";
                }
                else if (old_predicate.equals(new_predicate)){
                    equality_test = "symfalse";
                }
                else {
                    equality_test = "( " + old_predicate + " ) & ~ ( " + new_predicate + " )";
                    equality_test = matlab.feval("simpler", equality_test);
                }
                if (! equality_test.equals("symfalse")) {
                    finished = false;
                }
                location.matlab_nonblocking_pred = add_string(location.matlab_nonblocking_pred
    , location.matlab_temp);
                location.matlab_temp = null;
            }
            printing("------------------------------------------\n");
        }
    }

    private static void bad_state() throws RejectedExecutionException, EngineException,
    InterruptedException, ExecutionException {
        String predicate;
        String guard;
        String target_condition;
        String [] resets;
        String equality_test;
        String old_predicate;
        String new_predicate;
        String invariant;
        String bad_state;
        String part_six;
        boolean finished = false;

        Scanner scan = new Scanner(System.in);

        printing("Computing the bad-state conditions: \n");
        printing("Initial bad-state conditions are: \n");
        for (location location: automaton.locations) {
            if (location.matlab_nonblocking_pred != null) {
                predicate = last(location.matlab_nonblocking_pred);
            }
            else {
                predicate = "symtrue";
            }
            predicate = "~ ( " + predicate + " )";
            predicate = matlab.feval("simpler", predicate);
            location.matlab_bad_state = add_string(location.matlab_bad_state, predicate);
            location.matlab_temp = predicate;
            printing(location.name + ": " + predicate + "\n");
        }

        while (! finished) {
            finished = true;
            for (location location: automaton.locations) {
                printing("\n");
                printing("    Computing bad-state predicate for " + location.name + "\n");
                predicate = "( " + last(location.matlab_bad_state) + " )";
                printing("    Initial predicate is: " + predicate + "\n");
                for (edge event: location.edges) {
                    if (event.uncontrollable) {
                        printing("        add edge predicate for " + event.event);
                        if (event.guard != null) {guard = last(event.guard);}
                        else {guard = "symtrue";}

                        target_condition = target_bad_state(event.destination_location);
                        if (target_condition == null) {target_condition = "symtrue"; }
                        predicate += " | ( ( " + guard + " ) & ( " + target_condition + " ) )"
    ;
                        resets = list_to_string(event.clock_resets);
                        if (resets != null) {
                            printing(" with resets: ");
                            for (String reset: resets) {printing(reset + " ");}
```

```
                        printing(predicate + " --> ");
                        location.matlab_temp = matlab.feval("simpler", predicate, resets);
                        printing(location.matlab_temp + "\n");
                    }
                    else {
                        printing(predicate + " --> ");
                        location.matlab_temp = matlab.feval("simpler", predicate);
                        printing(location.matlab_temp + "\n");
                    }
                }
            }

            invariant = last(location.matlab_invariants);
            if (invariant == null) {invariant = "symtrue";}
            bad_state = last(location.matlab_bad_state);
            if (bad_state == null) {bad_state = "symtrue";}
            printing("        Part three inputs are: N: " + bad_state + ", I: " + invariant + ", ");
            part_six = matlab.feval("clock_regions", bad_state, invariant);
            if (part_six.equals("null")) {
                printing("\n        Solve: " + bad_state + " With some time delay Delta AND " + invariant + " with time delay delta for all time delta < Delta\n");
                part_six = scan.nextLine();
            }
            printing("Output is: " + part_six + "\n        Adding gives: ");
            predicate += " | ( " + part_six +  ")";
            printing(predicate + " --> ");
            predicate = matlab.feval("simpler", predicate);
            location.matlab_temp = predicate;
            printing(predicate + "\n\n");
            predicate = null;

            if (location.matlab_temp == null) {
                location.matlab_temp = predicate;
            }
        }
        printing("\n");
        for (location location: automaton.locations) {
            printing("Final BSP are: " + location.name + ": " + location.matlab_temp + "\n");
            old_predicate = last(location.matlab_bad_state);
            new_predicate = location.matlab_temp;
            equality_test = "test";
            if (old_predicate.equals("symfalse") ^ new_predicate.equals("symfalse")) {
                equality_test = "dont";
            }
            else if (old_predicate.equals(new_predicate)) {
                equality_test = "symfalse";
            }
            else {
                equality_test = "( " + old_predicate + " ) & ~ ( " + new_predicate + " )";
                equality_test = matlab.feval("simpler", equality_test);
            }
            if (! equality_test.equals("symfalse")) {
                finished = false;
            }
            location.matlab_bad_state = add_string(location.matlab_bad_state, location.matlab_temp);
            location.matlab_temp = null;
        }
        printing("------------------------------------------\n");
    }
}

private static boolean invariants_equal() throws RejectedExecutionException, EngineException, InterruptedException, ExecutionException {
    String equality_test;
    String old_invariant;
    String new_invariant;
    int length;

    for (location location: automaton.locations) {
        length = location.matlab_invariants.length;
```

```
            new_invariant = location.matlab_invariants[length-1];
            old_invariant = location.matlab_invariants[length-2];
            if (old_invariant.equals("symfalse") ^ new_invariant.equals("symfalse")) {
                equality_test = "dont";
            }
            else if (old_invariant.equals(new_invariant)) {
                equality_test = "symfalse";
            }
            else {
                equality_test = "( " + old_invariant + ") & ~ ( " + new_invariant + " )";
                equality_test = matlab.feval("simpler", equality_test);
            }
            if (! equality_test.equals("symfalse")) {
                return false;
            }
        }
        return true;
    }

    private static boolean guards_equal() throws RejectedExecutionException, EngineException,
    InterruptedException, ExecutionException {
        String equality_test;
        String old_guard;
        String new_guard;
        int length;

        for (location location: automaton.locations) {
            for (edge event: location.edges) {
                length = event.guard.length;
                old_guard = event.guard[length-2];
                new_guard = event.guard[length-1];
                if (old_guard.equals("symfalse") ^ new_guard.equals("symfalse")) {
                    equality_test = "dont";
                }
                else if (old_guard.equals(new_guard)) {
                    equality_test = "symfalse";
                }
                else {
                    equality_test = "( " + old_guard + ") & ~ ( " + new_guard + " )";
                    equality_test = matlab.feval("simpler", equality_test);
                }
                if (! equality_test.equals("symfalse")) {
                    return false;
                }
            }
        }
        return true;
    }

    public static automaton synthesize (automaton original_automaton, boolean print_controller
    ) throws IllegalArgumentException, IllegalStateException, InterruptedException,
    RejectedExecutionException, ExecutionException{
        boolean loop1 = false;
        boolean loop2 = false;

        matlab = MatlabEngine.startMatlab();
        automaton = original_automaton;
        print = print_controller;

        for (location location: automaton.locations) {
            location.create_matlab_invariants();
        }
        while (! loop2) {
            while (! loop1) {
                non_blocking();
                bad_state();

                adapt_guards();
                loop1 = guards_equal();
            }
            loop1 = false;
            adapt_invariants();
            loop2 = invariants_equal();
```

```
        }
        return automaton;
    }
}
```

8.7 CIF plant

```
import "requirement_far.cif";
import "requirement_close.cif";
import "requirement_a_none.cif";
import "observer.cif";
import "svg.cif";

uncontrollable  u_D_front_small, u_D_front_large;
uncontrollable  u_D_back_small, u_D_back_large;
uncontrollable  u_D_none;

controllable    c_decel_small, c_decel_large;
controllable    c_accel_small, c_accel_large;
controllable    c_a_none;

const real      time_constraint = 3/8;
const real      distance_small = 5;
const real      distance_large = 10;

plant plant_distance:
    alg real distance = x_leader - x_follower;

    location DNone:
        initial; marked;
        edge u_D_front_small    when distance<-distance_small    goto TimeFSmall;
        edge u_D_back_small     when distance>distance_small   goto TimeBsmall;

    location TimeFSmall:
        edge u_D_front_large    when distance<-distance_large    goto TimeFLarge;
        edge u_D_none           when distance>1                   goto DNone;

    location TimeFLarge:
        edge u_D_front_small    when distance>-distance_large    goto TimeFSmall;

    location TimeBsmall:
        edge u_D_back_large     when distance>distance_large   goto TimeBLarge;
        edge u_D_none           when distance<1                goto DNone;

    location TimeBLarge:
       edge u_D_back_small      when distance<distance_small   goto TimeBsmall;
end

plant plant_acceleration:
    location ANone:
        initial; marked;
        edge c_decel_small    goto ADecelSmall;
        edge c_decel_large    goto ADecelLarge;
        edge c_accel_small    goto AAccelSmall;
        edge c_accel_large    goto AAccelLarge;

    location ADecelSmall:
        edge c_decel_large      goto ADecelLarge;
        edge c_a_none           goto ANone;

    location ADecelLarge:
        edge c_decel_small      goto ADecelSmall;
        edge c_a_none           goto ANone;

    location AAccelSmall:
        edge c_accel_large      goto AAccelLarge;
        edge c_a_none           goto ANone;

    location AAccelLarge:
        edge c_accel_small      goto AAccelSmall;
        edge c_a_none           goto ANone;
end
```

8.8 CIF Observers

```
import "plant.cif";

plant time_close:
    cont t_1 der 1;
    location TimeFNone:
        initial; marked;
        edge u_D_front_small    do t_1 := 0 goto TimeFSmall;
        edge u_D_none;

    location TimeFSmall:
        edge u_D_none           do t_1 := 0 goto TimeFNone;
        edge u_D_front_large    do t_1 := 0 goto TimeFLarge;

    location TimeFLarge:
        edge u_D_front_small    do t_1 := 0 goto TimeFSmall;
end

plant time_far:
    cont t_2 der 1;
    location TimeBNone:
        initial; marked;
        edge u_D_back_small     do t_2 := 0 goto TimeBSmall;
        edge u_D_none;

    location TimeBSmall:
        edge u_D_none           do t_2 := 0 goto TimeBNone;
        edge u_D_back_large     do t_2 := 0 goto TimeBLarge;

    location TimeBLarge:
        edge u_D_back_small     do t_2 := 0 goto TimeBSmall;
end
```

8.9 CIF Requirement Close

```
import "plant.cif";

plant decel_slow:
    cont t_3_slow der 1;
    location DecelSlowVConst:
        initial; marked;
        edge u_D_front_large        when time_close.t_1 >= time_constraint
                                    do t_3_slow := 0
                                    goto DecelSlowDLarge;
        edge u_D_front_large        when time_close.t_1 < time_constraint
                                    goto DecelSlowASmall;
        edge u_D_front_small;
        edge u_D_none;
        edge c_a_none;

    location DecelSlowDLarge:
        invariant t_3_slow <= 0;
        edge c_decel_small          do t_3_slow := 0
                                    goto DecelSlowASmall;
        edge u_D_front_small        goto DecelSlowVConst;
        edge c_decel_large;

    location DecelSlowASmall:
        edge u_D_none               goto DecelSlowVConst;
        edge u_D_front_small;
        edge c_decel_large;
end

plant decel_large:
    cont t_3_large der 1;
    location DecelLargeVConst:
        initial; marked;
        edge u_D_front_large        when time_close.t_1 >= time_constraint
                                    do t_3_large := 0
                                    goto DecelLargeDLarge;
        edge u_D_front_large        when time_close.t_1 < time_constraint
                                    goto DecelLargeALarge;
        edge u_D_front_small;
        edge u_D_none;
        edge c_decel_small;
        edge c_decel_large;
        edge c_a_none;

    location DecelLargeDLarge:
        invariant t_3_large <= 1/8;
        edge c_decel_large          do t_3_large := 0
                                    goto DecelLargeALarge;
        edge u_D_front_small        goto DecelLargeVConst;
        edge c_decel_small;

    location DecelLargeALarge:
        edge u_D_none               goto DecelLargeVConst;
        edge u_D_front_small;
        edge c_decel_large;
end

plant decel_panic:
    cont t_3_panic der 1;
    location DecelPanicVConst:
        initial; marked;
        edge u_D_front_large        when time_close.t_1 < time_constraint
                                    do t_3_panic := 0
                                    goto DecelPanicDLarge;
        edge u_D_front_large        when time_close.t_1 >= time_constraint;
        edge u_D_front_small;
        edge u_D_none;
        edge c_decel_small;
        edge c_decel_large;
        edge c_a_none;

    location DecelPanicDLarge:
```

```
        invariant t_3_panic <= 0;
        edge c_decel_large          do t_3_panic := 0
                                    goto DecelPanicALarge;
        edge u_D_front_small        goto DecelPanicVConst;

    location DecelPanicALarge:
        edge u_D_none               goto DecelPanicVConst;
        edge u_D_front_small;
end
```

8.10 CIF Requirement Far

```
import "plant.cif";

plant accel_slow:
    cont t_4_slow der 1;
    location AccelSlowVConst:
        initial; marked;
        edge u_D_back_large          when time_far.t_2 >= time_constraint
                                     do t_4_slow := 0
                                     goto AccelSlowDLarge;
        edge u_D_back_large          when time_far.t_2 < time_constraint
                                     goto AccelSlowASmall;
        edge u_D_back_small;
        edge u_D_none;
        edge c_a_none;

    location AccelSlowDLarge:
    invariant t_4_slow <= 0;
        edge c_accel_small           do t_4_slow := 0
                                     goto AccelSlowASmall;
        edge u_D_back_small          goto AccelSlowVConst;
        edge c_accel_large;

    location AccelSlowASmall:
        edge u_D_none                goto AccelSlowVConst;
        edge u_D_back_small;
        edge c_accel_large;
end

plant accel_large:
    cont t_4_large der 1;
    location AccelLargeVConst:
        initial; marked;
        edge u_D_back_large          when time_far.t_2 >= time_constraint
                                     do t_4_large := 0
                                     goto AccelLargeDLarge;
        edge u_D_back_large          when time_far.t_2 < time_constraint
                                     goto AccelLargeALarge;
        edge u_D_back_small;
        edge u_D_none;
        edge c_accel_small;
        edge c_accel_large;
        edge c_a_none;

    location AccelLargeDLarge:
        invariant t_4_large <= 1/8;
        edge c_accel_large           do t_4_large := 0
                                     goto AccelLargeALarge;
        edge u_D_back_small          goto AccelLargeVConst;
        edge c_accel_small;

    location AccelLargeALarge:
        edge u_D_none                goto AccelLargeVConst;
        edge u_D_back_small;
        edge c_accel_large;
end

plant accel_panic:
    cont t_4_panic der 1;
    location AccelPanicVConst:
        initial; marked;
        edge u_D_back_large          when time_far.t_2 < time_constraint
                                     do t_4_panic := 0
                                     goto AccelPanicDLarge;
        edge u_D_back_large          when time_far.t_2 >= time_constraint;
        edge u_D_back_small;
        edge u_D_none;
        edge c_accel_small;
        edge c_accel_large;
        edge c_a_none;

    location AccelPanicDLarge:
```

```
        invariant t_4_panic <= 0;
        edge c_accel_large          when t_4_panic >= 0
                                    do t_4_panic := 0
                                    goto AccelPanicALarge;
        edge u_D_back_small         goto AccelPanicVConst;

    location AccelPanicALarge:
        edge u_D_none               goto AccelPanicVConst;
        edge u_D_back_small;
end
```

8.11 CIF Requirement Acceleration None

```
import "plant.cif";

plant a_none:
    cont t_5 der 1;
    location D0A0:
        initial;
        marked;
        edge c_decel_small          goto D1A1;
        edge c_decel_large          goto D1A1;
        edge c_accel_small          goto D1A1;
        edge c_accel_large          goto D1A1;

    location D1A1:
        edge u_D_none do t_5:=0     goto D0A1;
        edge c_decel_small;
        edge c_decel_large;
        edge c_accel_small;
        edge c_accel_large;

    location D0A1:
        edge c_a_none when t_5>0    goto D0A0;
end
```