Language and Tools for Cyber-Physical Systems: A Comparison between Acumen and CIF 3.0

B. Volmer

Technische Universiteit
Eindhoven
University of Technology

Externship (international)

Cyber-Physical Systems: A Comparison between Acumen and CIF 3.0

Subject

Cyber-Physical Systems (CPSs) are becoming omnipresent, as computers take control of the physical environment and constantly react to it. Early validation and verification of such systems is essential due to their criticality and to achieve that using rigorous modeling techniques is an essential pre-requisite. The Acumen language, `http://www.acumen-language.org/`, and the Compositional Interchange Format for Hybrid Systems (CIF), `http://se.wtb.tue.nl/sewiki/cif/start`, are two modeling languages for CPSs.

Assignment

The goal of this internship is to specify and verify a set of canonical examples (small systems capturing key features of typical CPSs) in both languages and thereby come up with a structured comparison of the commonalities, strengths and weaknesses of the two languages and their tools.
This exercise will comprise the following steps:

- Listing the typical phenomena appearing in CPSs. Typical references for finding such phenomena are overview papers, textbooks and tool tutorials for CPSs.
- Gathering a list of canonical examples of CPSs. Typical sources for such examples are textbooks and tool tutorials, particularly the two tools under study. This tasks also includes typical properties/scenario's that are to be verified/validated for each and every example.
- Modeling the examples in both tools and verifying the properties/validating the scenario's using tools.
- Comparing the two languages and tools based on the data and observations gathered during the modeling and verification/validation. Comparison criteria for modeling languages can include: learning time, expressiveness, modularity, and precision and clarity in the semantics. The tools can be compared with respect to: usability and verification and validation time.

Start	Sept 2014
Finish	Dec 2014
Student	B. Volmer
Supervisors	M.R. Mousavi (Halmstad University, Sweden), M.A. Reniers (TU/e)

Abstract

Comparing modelling and simulation tools can give more insight in the differences between tools and can give new insights for implementing interesting features of other tools. Based on this idea the tools CIF 3.0 and Acumen are compared with each other, these tools are developed with respect to hybrid systems containing continuous and discrete behaviour. During this research both tools are investigated and compared to each other with respect to Cyber Physical Systems, systems that contain properties of hybrid systems. Based on typical properties of Cyber Physical Systems canonical examples of small systems are defined after which they are modelled and simulated by using both tools, Acumen and CIF 3.0. The models and simulation results are compared to each other with respect to a number of criteria important for a modelling and simulation tool in general like structure, data types and working principle behind the semantics. Comparing Acumen and CIF 3.0 it turns out that in particular the structure of models and working principle of the semantics of both tools significantly differ from each other. Although no hard conclusion(s) can be drawn from the results because the modeller his or her modelling preferences also play a role, anyway it is interesting to observe the differences between tools and its cause. In the end, this research shows a way for comparing tools with each other. Next to this, the developers of Acumen and/or CIF could be encourage to implement feature of each others tool. While not all properties of Cyber Physical Systems could be taken into account during this research, so to complete the comparison between Acumen and CIF there is still work to be done.

Contents

Chapter 1

Introduction

1.1 Subject Matter

This report is about Cyber Physical Systems. These are systems containing dynamics of physical processes, software and networking, to provide abstractions and modelling, design, and analysis techniques for the integrated whole. Physical processes often feature continuous dynamics while the dynamics of software and networking are regularly discrete. These systems become more and more interesting in the field of Systems Engineering and Computer Science, due to the fact that computers become more intelligent and entrenched with the physical world. Modelling and simulation of these systems before they are implemented, can be really useful to recognize problems in an early state. Bugs can be fixed when the actual system is not even built and therefore a lot of time and costs can be saved. Tools based on hybrid systems, are available to model and simulate Cyber Physical Systems. Hybrid systems are models of systems where continuous dynamics interact with discrete dynamics. Therefore tools that handle these kind of systems can be very useful to model and analyse Cyber Physical Systems as well.

1.2 Context

This research was carried out at the Centre for Research on Embedded Systems (CERES), at Halmstad University in Sweden. At this university (and in collaboration with some other universities, such as Rice University in Texas, USA) "an experimental modelling and simulation environment for hybrid systems" [4] called Acumen is being developed. At University of Technology Eindhoven, the Compositional Interchange Format (CIF) for hybrid systems is commonly used for working on hybrid systems. This is an automata based modelling language for the specification of discrete event systems, timed systems, and hybrid systems [3]. The Acumen language is still being developed, whereas CIF has reached a relatively stable state. CIF is actually in the state where it becomes interesting to see in what way(s) it can be implemented in industry. This means for the research during this project, that the absence of properties and/or opportunities in Acumen can change in the nearby future due to new releases. In this project Acumen and CIF are compared with each other based on the suitability for modelling and simulation of Cyber Physical Systems. Therefore typical properties of this kind of systems are identified together with canonical examples of systems containing these properties. Subsequently, these systems are modelled and simulated by using Acumen and CIF, and compared with each other.

1.3 Problem Definition

The purpose of this project is to specify and verify a set of canonical examples in both languages and thereby come up with a structured comparison of the commonalities, strengths and weaknesses of the two languages and their tools. Typical properties of Cyber Physical Systems are defined, and connected to relatively simple systems that contain behaviour based on these properties. The systems are modelled by using both tools, and the models and simulation results are analysed with respect to each other to come up with differences and similarities. The analysis is based on several criteria such as structure, data, working principle and simulation results of the models. Defining a strict reference to compare both tools with is hard in this case, it is more about recognizing differences between both tools with respect to each other. In this report only the most important aspects of this analysis are discussed. Below the criteria are explained in more detail:

Structure is about the appearance of a model in Acumen and CIF. The way in which models need to be defined.

Data is about the different kinds of data that can be used in the program. The predefined categories and if possible the user defined types of data. In what way they can be used in the program.

Working Principle is about the steps taken by the simulator when it executes a model. What is the way the program evaluates the values of different variables and in what way affects this the model and its results.

Simulation Results is about the output of the simulation of a model. The ways in which results can be examined afterwards as well as the simulation and visualization possibilities when the program is running.

1.4 Results

The comparison between Acumen and CIF based on a number of typical properties of Cyber Physical systems showed that, next to the way a modeller models a system, the underlying techniques and principles of a languages can restrict the possibilities of modelling a system. This has to do with the working principle of the tool and the way in which the structure of models needs to be defined. Next to this, the way in which data can be used and simulation results are available to analyse the simulated model are the other two criteria taken into account comparing Acumen and CIF.

With respect to Cyber Physical Systems, Acumen and CIF are both able to be used for the modelling and simulation of these kind of systems. More explicitly, this project reveals that Acumen is a language that has less restrictions for the structure and definition of a model, compared to CIF. CIF models contain a certain structure and typical defined functions to change the model its behaviour. Where Acumen does not have such a defined structure and defined functions that need to be used for defining the behaviour of a (sub)system. Therefore, the modeller is more free to implement behaviour in the way he or she likes. This can be an advantage because for modelling behaviour of system there is often not explicitly one way of doing this. In a lot of situations equivalent behaviour can be modelled by different models.

So, in CIF the modeller needs to take into account the structure and functions of the tool that need to be used, where in Acumen that is not necessary because there is not such a strict structure. Structure and defined model definitions of a tool are important for the modeller to take into account when creating a model.

Next to that, Acumen is more straightforward to model the component of physical dynamics of a Cyber Physical System and CIF is more straightforward to model the software and networking dynamics. This does not mean in CIF it is hard to model physical dynamics. It just divides both language in the area of Cyber Physical Systems to the properties that fit them the best based on the results of this research.

1.5 Structure of the Report

The following two chapters (Chapter 2 and Chapter 3) contain a more detailed explanation of both languages and tools, Acumen and CIF, in therms of working principle and layout of the user interface. In Chapters 4 till 8 the typical properties of Cyber Physical Systems and their analysis are discussed based on canonical examples. Finally, in Chapter 9 a conclusion is drawn and recommendations are given for further research.

Chapter 2

Acumen

The Acumen language is designed for modelling and simulating hybrid systems. It is built around a small, textual modelling language. The purpose of the people working on Acumen is to keep the syntax of Acumen as close as possible to the mathematical description of systems. The program is explained by using a relatively simple system of a water tank. The basics and properties of Acumen relevant for this research are discussed with respect to this system. A more detailed explanation of Acumen can be found on: `http://www.acumen-language.org/`. The water tank considered in this section consist of a tank leaking at a constant rate. Water can be added to the volume at a constant rate through a hose that can be opened and closed. In Figure 2.1 a schematic representation of the system can be seen.

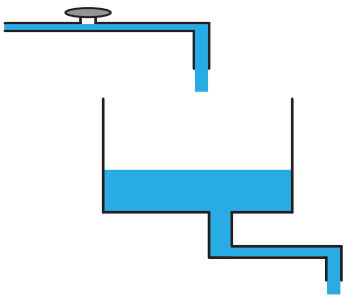


Figure 2.1: Schematic representation of the water tank system

2.1 The structure of an Acumen model

A system in Acumen is modelled by using objects of model declarations. Every model in Acumen needs to include at least a model declaration called "`Main`". Every model declaration can contain several parameter definitions. The model "Main" contains always just one parameter; `simulator`, this is defined by the developers. Executing a model always starts from the model "`Main`", so all objects of the system need to be defined in this model or created from this model, directly or indirectly. In Figure 2.2 the model of the water tank system in Acumen can be seen. It consist of the model "`Main`" and a model "`Tank`". A model consist of an "initially" section and an "always" section, respectively indicated by `initially` and `always`. In the initially section variables or vectors are declared and initiated. Continuous variables need to be initiated with a derivative as well, algebraic and discrete variables can be initiated by just an initial value. All variables

used in the current model need to be initiated in this section, otherwise they can not be used. As mentioned previously a model can also contain parameter definitions (`min_height` and `max_height` in this case), they do not need any initialization. The type of variable like real, integer or boolean for instance does not need to be declared. The program itself recognizes the type of the variable, and this can also change during simulations. When a variable has initial value 0.0 it is possible to assign the value "`true`" to it, which changes the variable type from real to boolean. This makes it possible to change variable types during the simulation of the model.

```
model Tank(min_height, max_height) =
initially
  t = 0 , t' = 1 ,
  V = 10.0 , V' = 0.0 ,
  Qi = 0.0 , Qo = 0.0 ,
  n = 0.0 ,
  c = red ,

  _3D  =  ()
always
  _3D = (Box      center=(0,0,V/2)      size=(10.0,5.0,V) color=blue  rotation=(0,0,0) ,
         Sphere   center=(-7.0,0,12.0)  size=0.4          color=c     rotation=(0,0,0) ,

         Cylinder center=(-4.5,0,12.0)  radius= 0.05  length=5.0  color=(0,0,0) rotation=(0,pi/2,pi/2) ,
         Cylinder center=(-8.5,0,12.0)  radius= 0.05  length=3.0  color=(0,0,0) rotation=(0,pi/2,pi/2) ,
         Cylinder center=(-2.0,0,11.25) radius= 0.05  length=1.5  color=(0,0,0) rotation=(pi/2,0,0) ,
         Cylinder center=(0,0,-1.5)     radius= 0.05  length=3.0  color=(0,0,0) rotation=(pi/2,0,0) ,
         Cylinder center=(2,0,-3)       radius= 0.05  length=4.0  color=(0,0,0) rotation=(0,0,pi/2) ,
         Cylinder center=(4,0,-3.5)     radius= 0.05  length=1    color=(0,0,0) rotation=(pi/2,0,0) ,

         Text center=( 1,0,11)      size=1 color=(0,0,0) rotation=(0,0,0) content="Qi=" ,
         Text center=( 3.0,0,11)    size=1 color=(0,0,0) rotation=(0,0,0) content=Qi ,
         Text center=( 6.5,0,5)     size=1 color=(0,0,0) rotation=(0,0,0) content="V=" ,
         Text center=( 8,0,5)       size=1 color=(0,0,0) rotation=(0,0,0) content=V ,
         Text center=( 1.5,0,-1.5)  size=1 color=(0,0,0) rotation=(0,0,0) content="Qo=" ,
         Text center=( 3.5,0,-1.5)  size=1 color=(0,0,0) rotation=(0,0,0) content=Qo) ,

  t' = 1.0 ,
  Qi = n*5.0 ,
  Qo = sqrt(V) ,
  V' = Qi - Qo ,

  if V <= min_height then
     n + = 1.0 ,
     c + = green

  else if V >= max_height then
          n + = 0.0 ,
          c + = red
  noelse

model Main (simulator) =
initially
  a  =  create Tank (2.0, 10.0) ,

  _3DView = ((-3,-80,20), (-3,0,0))
```

Figure 2.2: The Acumen model for the water tank

In the initially section of the model "`Main`" there is only one instantiation "`a`" which creates the model "`Tank`". The comment "`_3DView`" has to do with the visualisation and is discussed later in this chapter. By the "`create`" statement an object of the model "`Tank`" is created from the model "`Main`". This is the way the size of the model increases and more (sub)systems can be created in one model. In this example the objects are created statically.

There is no always section of the model "`Main`" in this case, but there can be one. Usually in that section the actual behaviour of the object is described. This can be done using several statements like; continuous assignments, if-statements, match-statements, for-statements and discrete

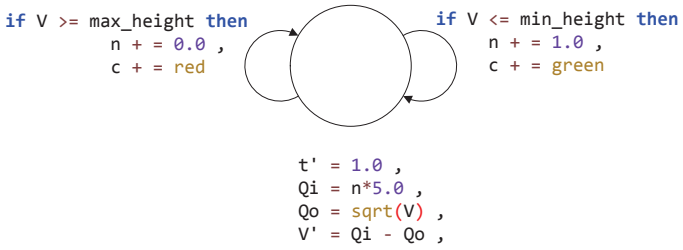


Figure 2.3: The model for the water tank in Acumen represented by a hybrid automaton

assignments. These are quite obvious from a mathematical point of view, but a more detailed explanation can be found in [4]. For the water tank example it can be seen that in the always section of the model "Tank" first four equations are defined globally (continuous assignments). These express the gradient in time of the time variable (t), the inflow (Qi), the outflow (Qo) and the volume of water in the tank (V) respectively. Subsequently, an if-statement is used to open and close the hose by changing the value of "`n`" using discrete assignments. In Figure 2.3 a graphical representation of the model is given by using hybrid automata. The variable "`c`" and the assignments denoted by "`_3D`" contain visualisation parameters and are discussed later on in this section.

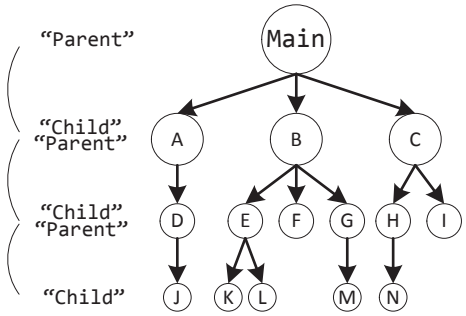


Figure 2.4: Schematic representation of the hierarchy in an Acumen model

An important aspect of Acumen with respect to sharing variables between different objects is the hierarchy in the system. The model "`Main`" is at the highest hierarchy level of the system, from there on other objects are created, see Figure 2.4. These other object are called "children" from model "`Main`", which is called the "parent". Hence, the position in the model where an object is created affects the relation to other objects in that system. This is important with respect to sharing data, because "parents" have only direct access to the data of their "children", lets explain this by Figure 2.4. For instance, if the value of variable "`share`" in model "E" need to be shared with model "G" this can be done by including the following expression in the always section of model "B":

```
G.share = E.share
```

When in the same system the variable "`share`" in model "E" need to be shared with model "C" it cannot be done directly. The following two expressions give the required result, but it would be better to maybe construct the model differently.

Always section of model B: `B.share = E.share`

Always section of model Main: `C.share = B.share`

2.2 Simulation of an Acumen model

The simulation of a model in Acumen consist of two different steps, a continuous step and a discrete step. In Figure 2.5 the principle of simulating is shown schematically. First, all variables and their initial values are saved in a store. Then it is checked if anywhere in the program discrete assignments are active. An assignment is active if the guard of, for instance, an if-loop is true. All active discrete assignments are collected by the program and executed simultaneously, which means the values of these variables are changed at the same simulation step and saved in a new store. Then the values of the new store are compared to the previous store. When both stores are equal the program continues to the continuous step, otherwise a next discrete step is taken. A continuous step is executed in a quite similar way as a discrete step; all active continuous assignments of the system are collected and executed simultaneously. So, all continuous variables are evaluated simultaneously over a fixed time step of 0.01 time units by default but this number can be adapted. Then the program checks if the simulation end time is reached. If so, the simulation ends and otherwise the program returns to the discrete step. This working principle of Acumen makes the program deterministic under the traditional semantics, used by default. Acumen has the possibility to change semantics to the enclosure semantics, where the program calculates with intervals to give a more confident solution for systems where the traditional semantics do not satisfy, the enclosure semantics are discovered in more detail in Chapter 7.

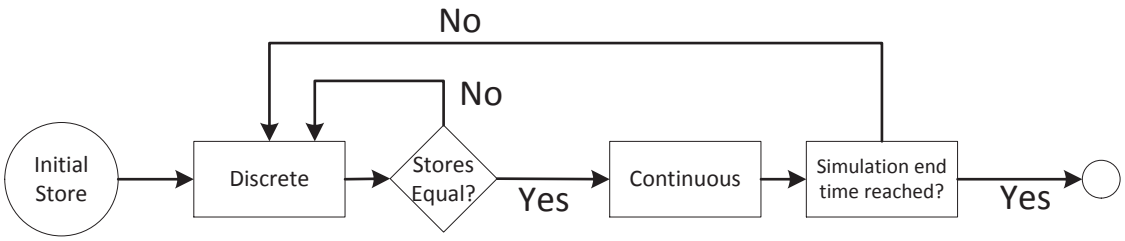


Figure 2.5: Schematic representation of the working principle of the simulator of Acumen

For the model of the water tank in Figure 2.2 this means the following. The volume "V" is initially 10.0 and thus the second condition in the if-statement of the body is enabled. This means the two discrete assignments declared in this section are executed, changing the value of "`n`" and "`c`", respectively representing the state of the hose (0 is closed, 1 is open) and the color of the hose to indicate its state. After this, these two discrete assignments are still the only two enabled and executing them again result in an equivalent store so the program will move to the continuous step. There, the four globally defined continuous assignments are evaluated for one time step. This will not result in any enabled discrete assignment, because the stores are equal, and thus a continuous step is executed again. This process repeats until the first guard of the if-loop is enabled, resulting in changing the values of "`n`" and "`c`" again. Then continuous steps are executed again until the second guard of the if-loop is enabled. This process continues till the simulation end time is reached.

2.3 Visualisation of models

During simulation a visualisation of the model can contribute to the analysis of the behaviour of the model. Potential bugs can be identified or better, correctness can be confirmed. Here, it must be taken into account that just a single trace of the system is visualised. So, when the model is non-deterministic it is possible that the model can also contain behaviour which is not represented by the visualisation.

When an Acumen model is executed the simulator creates a graph for each variable in the model by default. In these graphs the values of the variables are plotted over time. By moving the mouse over the graphs the exact values of a variable can be tracked for a more detailed analysis.

Next to this, it is possible to visualise the behaviour of a system in a 3D environment by using the "`_3D`" commands, mentioned earlier, in the model. Looking at Figure 2.2 it can be seen that the 3D environment needs to be initialized like every other variable in the initially section. Then in the always section the command is used to define the objects of the visualisation. In Acumen, next to text, there are a limited amount of shapes available, namely; boxes, cylinders, cones, spheres and lines. These shapes are defined by position, dimensions, color and orientation as can be seen in Figure 2.2 for the water tank. Dynamical shapes can be created by using variable values in their definition, like is done with the height of the box and the color of the sphere for the water tank. During this visualisation the view can be changed manually by using the mouse, or it can be set in the program by using the "`_3DView`" command like in Figure 2.2. Next to these two ways of visualising the behaviour of a model (by graphs and 3D environment) the program always produces the numerical results of each simulation step. In Figure 2.6 an example of the visualisation of Acumen can be seen in case of the water tank.

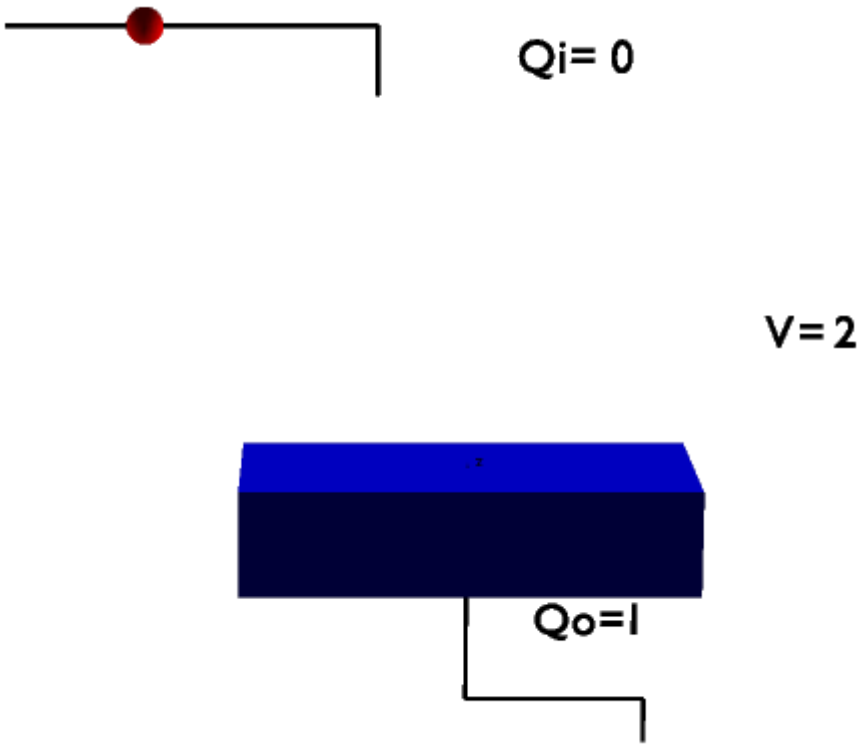


Figure 2.6: An example of the visualisation in Acumen for the watertank

Chapter 3

CIF

CIF 3.0 is the latest version of CIF. "It is a language designed for two purposes; as a specification language for hybrid systems and as an interchange format for allowing model transformations between other languages for hybrid systems" [1]. This project focuses on the purpose of being a specification language for hybrid systems. The program is explained again, by using the system of a water tank. The basics and properties of CIF interesting for this research are discussed with respect to this system. A more detailed explanation of CIF can be found on: `http://cif.se.wtb.tue.nl/index.html`. In Figure 3.1 a graphical representation of the system can be seen.

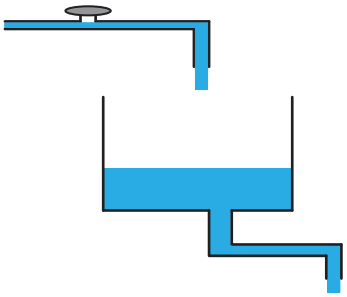


Figure 3.1: Schematic representation of the water tank system

3.1 The structure of a CIF model

A model in the CIF language consist of automata and groups (optional). The automata in CIF describe the behaviour of a (sub)system, where groups are named collections of automata and other declarations. In Figure 3.2 the CIF model of the water tank system can be seen. This is not the most efficient model for this system, but the model is adapted in such a way that it contains a lot of features of CIF, so they can be explained. The model consist of a group "`tank`" and an automaton "`controller`". In the group the volume of the tank (V), the inflow (Qi), the outflow (Qo) and the time derivative of the volume (V') are defined respectively along with several visualisation definitions explained later on in this chapter. The category of each variable defined in the group are declared with the corresponding variable such as; continuous (`cont`), algebraic (`alg`) or discrete (`disc`). Next to this, the data type of the algebraic, continuous and discrete variables need to be defined. For instance, this can be seen for the inflow and outflow variables of type real. The command "`equation`" can be used to define time derivatives, but also to define

algebraic variables at different locations in the system. This will be more clear if the locations of an automaton are discussed.

```
svgfile "tank.svg";

group tank:
  cont V = 10.0;
  alg real Qi = Controller.n * 5.0;
  alg real Qo = sqrt(V);
  equation V' = Qi - Qo;

  svgout id "water" attr "height" value 7.5 * V;
  svgout id "V"     text          value fmt("V = %.1f", V);
  svgout id "Qi"    text          value fmt("Qi = %.1f", Qi);
  svgout id "Qo"    text          value fmt("Qo = %.1f", Qo);
end

automaton def controller(alg int min_height, max_height):
  alg int n;

  location closed:
    initial;
    equation n = 0;
    edge when tank.V <= min_height  goto opened;

  location opened:
    equation n = 1;
    edge when tank.V >= max_height  goto closed;

  svgout id "n" attr "fill" value if n = 0: "red" else "lime" end;
end

Controller: controller(2, 10);
```

Figure 3.2: The CIF model for the water tank

As mentioned earlier the key parts of a CIF model are the automata, in this case just one automaton, describing the behaviour of a system, is used. When a system contains multiple nearly identical subsystems, automaton definitions and automaton instantiations can be used to create a template of an automaton and use this several times. By giving arguments to the automaton instantiations the small differences between two automata can be captured. Figure 3.2 shows the use of an automaton definitions to create the water tank, although in this example it is not necessary.

An automaton contains one or more locations, representing a specific operational mode where the system may/will reside for a while. The graphical representation of the hybrid automaton of this system in Figure 3.3 gives a better understanding of the model. The automaton contains two locations, "`closed`" and "`opened`", representing the state of the hose. Each automaton in a model needs to include an initial state, defined by the expression "`initial`". The algebraic variable "`n`" of type integer is globally declared in the automaton "`controller`". But if the whole system (group `tank` and automaton `controller`) is taken into account, the same variable "`n`" is defined locally in automaton "`controller`". This difference can be identified by the way referring to this variable. In the locations of the automaton "`controller`" the expression "`n`" refers to this algebraic variable while in the group "`tank`" the expression "`controller.n`" is used. A similar situation can be applied to the continuous variable "`V`" used on the edges in the automaton "`controller`". These refer to the variable "`V`" in group "`tank`". In CIF it is possible to read a variable everywhere in the program, as long as the reference is correct. Next to variables events can also be defined globally or locally in a CIF model. As discussed previously in this model it is chosen to change the value of variable "`n`" by using an equation in each location. It is also possible

to change the value of variable "n" by updating it at the transitions, this is explained further on. In general, discrete variables need to be declared in the component of the system where their value is changed (locally) and can be read everywhere in the system (globally). On the other hand, the progress of continuous variables over time needs to be defined in every component of the system, globally in the automaton or locally in every location, by means of equations for the variable itself or its derivative.

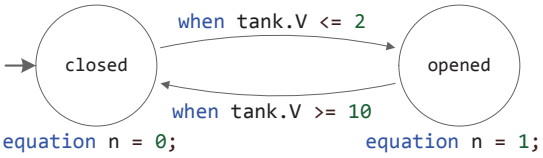


Figure 3.3: The model for automaton controller in CIF represented by a hybrid automaton

The transitions between both locations of the automaton are each defined by an event in the model, these are declared by the "edge" statement. Events are actions of the system in a certain location, called "transitions" when the location of the automaton is changed by executing an event. These events contain the location to which the system moves by executing the event, preceded by "goto". Next to this, edges can contain labels, updates and/or guards. These are all optional. Labels are used to identify certain edges, these become more important by using synchronizing events, explained later on in this chapter. Updates, preceded by "do", can be used to specify the effect of an event to variable values, and guards are used to enable events. Guards, preceded by "when", need to be an expression of type boolean. Whenever the expression is true the transition is enabled.

3.2 The CIF Simulator

Next to the structure and interpretation of a model it is important to have knowledge about the underlying principle of the tool when it simulates a model. This can be important to decide what modelling principles are used to model certain behaviour. In this section a simplified representation of the simulation loop used by the program during simulation is discussed.

When a simulation is performed in CIF the simulator creates an initial state of the system. This is the collection of the active states in each component (automaton), and the initial value of all variables defined. Subsequently, CIF works according to a standard procedure[1]:

- Check if the user-provided end time is reached. If so the simulation stops.
- Calculate all possible events for the current state.
- When there are no events possible, stop simulation.
- Choose one of the possible events.
- Take the chosen event, and set the state created by this event as the current state.

According to this procedure CIF collects all possible events in a list and executes one of them, chosen based on one of the following principles; first event in the list, last event in the list or a randomly chosen event from the list. Next to this, there is the possibility to simulate interactively; the user can influence the choice of the event to execute. These options makes the models in CIF

[1] http://cif.se.wtb.tue.nl/tools/cif3sim/traces.html

non-deterministic, because whenever multiple events are enabled an independent choice is made. Events in CIF are executed as long as there are enabled events in the current state of the system. The sequence of all events executed during a simulation is called the "trace" of a simulation. In CIF the simulator can only simulate one trace. Every simulation can therefore contain a different trace, depending on the choice of events made.

Although one event from the list of enabled events is chosen at a time by the simulator, it is possible multiple events are executed simultaneously, these are called synchronizing events. For instance, if two machines are modelled operating in series. This means the second machine gets a product when the first machines finished one, see Figure 3.4 for the graphical representation by hybrid automata. As can be seen each event is given a label. The event "`provide`" is present in both automata, this is the synchronizing event, which means this event can only be executed when it is enabled in both automata. This result in the situation multiple events are executed simultaneously, later on in this report this feature of CIF is considered in more detail.

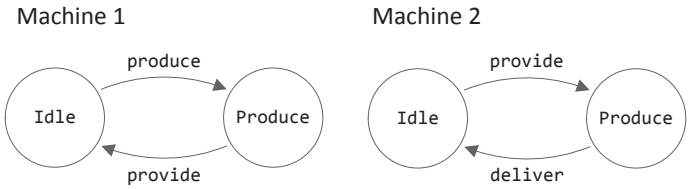


Figure 3.4: An example of two automate including synchronizing events

3.3 Visualisation by Scalable Vector Graphics

During simulation a visualisation of the model can contribute to the analysis of the behaviour of the model. Potential bugs can be identified or better, correctness can be confirmed. Although, just a single trace is visualised, so strong conclusions can not be drawn.

In CIF it is possible to set a lot of options when a model is executed. These are related to integration methods, input methods, simulation end time etc. One of the options is also to use the plot visualizer, this is an option in CIF that supports the possibility to plot the value of the variables of the model over time. It can be set if all variables need to be plotted or just a few of them. CIF does also provide a 2D visualization by using Scalable Vector Graphics files to visualise the behaviour of a model. These can be made using different programs, here Inkscape is used. In such a .svg file all kinds of conceivable shapes can be made in 2D. The .svg file contains all the objects needed for the visualization of a system, each object has an, adaptable, id. This is used in the CIF program to connect variables or states in the model to dimensions, position, color etc. of the objects in the SVG file. In CIF, functions are available to prescribe the relation between variables of the system and objects of the .svg file. This makes it possible to observe changes in system behaviour or system properties.

In Figure 3.2 the visualisation of the water tank is mainly defined in the group "`tank`", by the "`svgout`" statements. Here the volume of water is assigned to the height of the square representing the volume of the tank, and the values of the volume, water inflow and outflow are visualised respectively as can be seen in Figure 3.2. In the automaton "`controller`" it can be seen that the value of a visualisation object can also depend on a variable value using an if-statement. A more detailed explanation of the visualisation possibilities with CIF can be found in the language tutorial [3]. Next to the possibility of visualising the models behaviour by graphs or 2D environment it is also possible to obtain numerical results for every variable at each point in time. In Figure 3.5 an example of the visualisation of CIF can be seen in case of the water tank.

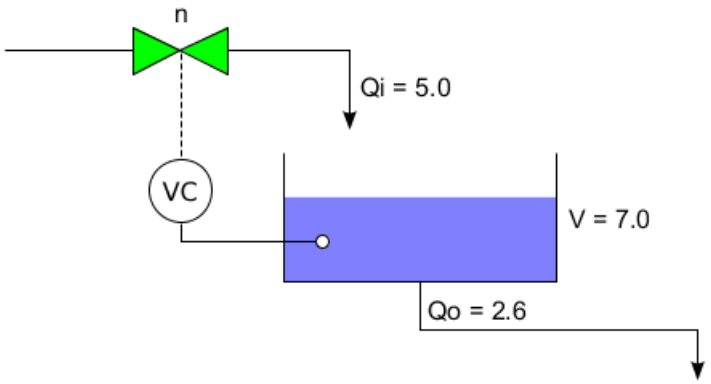


Figure 3.5: An example of the visualisation in CIF for the water tank

Chapter 4

Bouncing Ball

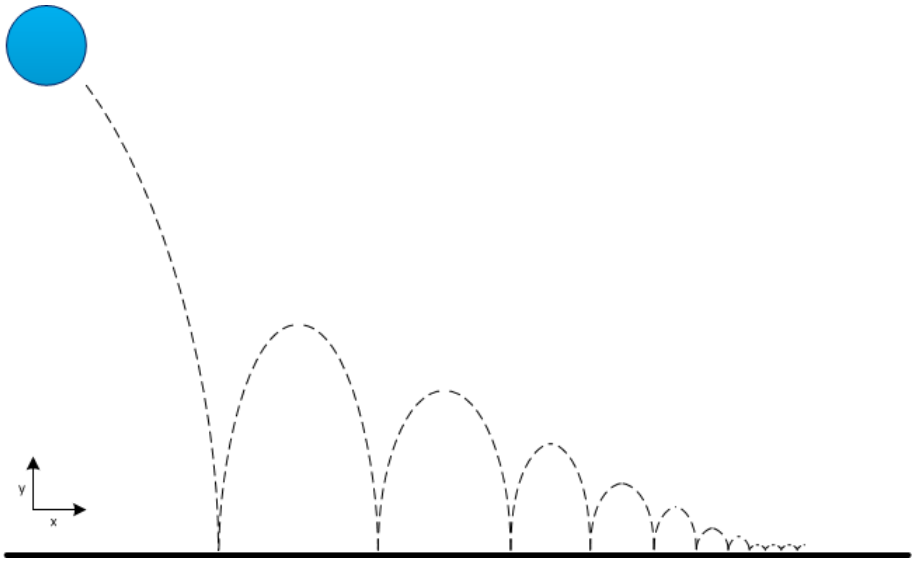


Figure 4.1: Schematic representation of the bouncing ball system

In this chapter a model of a ball bouncing on the ground is investigated, see Figure 4.1 for a graphical representation. The ball has a certain weight and is released from an initial position at a height y. During the process the acceleration in vertical direction is defined by equation 4.1. When the vertical position of the ball, y, becomes negative the vertical velocity, y', switches direction and is reduced by a factor, R, see equation 4.2. This expresses the loss of energy when touching the ground. Resulting in a ball that makes a number of jumps with decreasing maximal height.

The aspect of interest in this system is the possibility to describe a combination of continuous behaviour and discrete behaviour, and the way in which this can be done. This is essential for Cyber Physical Systems, because physical processes are mostly described by continuous equations while computers are commonly programmed discretely. The system is expected to behave like described, by jumps with decreasing height, till the end of the simulation. This can be a critical point, when the jumps become significantly small it is interesting in what way the program reacts.

$$y' = -m * g \quad (4.1) \qquad y' = -R * y' \quad (4.2)$$

The remainder of this chapter considers the comparison between the model of this system in both tools, Figure 4.2 and 4.3 give a graphical representation of both models by using hybrid automata. The models of the Acumen model and CIF model for the bouncing ball can be found in respectively Appendix A and Appendix B.

4.1 Acumen model compared to the CIF model

A model of a bouncing ball was present in both tools, thus by adapting several parameters two models for the equivalent system, described above, were created. Modelling hybrid systems require the possibility to define continuous variables whose values can be changed by using discrete assignments. This is possible in both, Acumen and CIF, see Figure 4.2 and 4.3. Where v represents the velocity of the ball and y represents the vertical position of the ball. In both models, when the ball hits the ground a discrete assignment is defined that changes the value of y' and v in respectively Acumen (`y' := -R*y'`) and CIF (`do v := -R*v`). When the corresponding condition, $y < 0$, is fulfilled the assignment changes the value of the continuous variable instantaneously. Then time elapses again and the continuous variable evolves according to the definition of its derivative. Interesting tool properties of CIF and Acumen with respect to continuous variables are:

- It is required to define the behaviour of a continuous variable in time in every state of the system, either globally or locally.
- In Acumen a second derivative can be defined directly while in CIF the second derivative is defined indirectly, by using the derivative of the first order derivative.
- When using derivatives in Acumen it is required to initiate the values of these derivatives. Although their definition with respect to time need to be defined in the always section of a model as well.

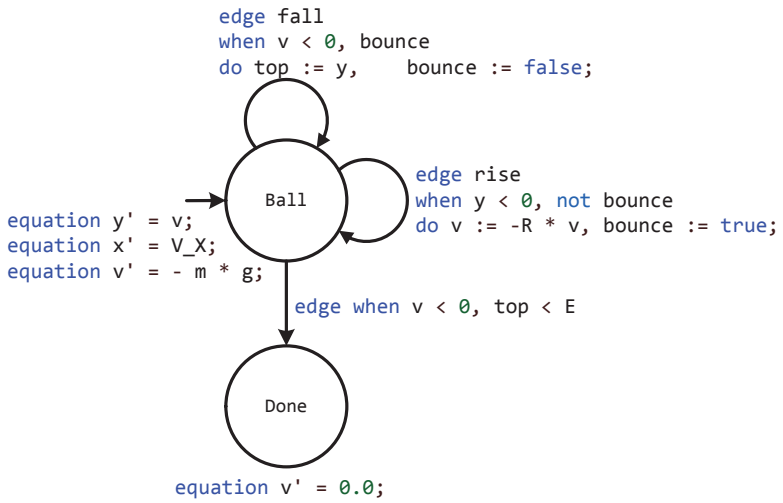


Figure 4.2: CIF model for the bouncing ball system represented by hybrid automata

So far, it can be determined that both tools can handle a combination of continuous behaviour and discrete behaviour for one variable in their models. In the Acumen model this is the only aspect modelled, while in CIF next to this behaviour a so-called "stopping criterion" is implemented. Observing the edge between states "`Ball`" and "`Done`", more accurately it can be seen that the constant "`E`" is used. This to decide if the system needs to move to state "`Done`" where the velocity becomes constant because the acceleration is set to 0.0 (`v' = 0.0`). This is an interesting difference compared to Acumen and it is related to the calculation method, used by the CIF simulator.

In general, several algorithms exists to solve ODE's. The working principle of the CIF simulator uses next to a particular ODE solving algorithm a root finding algorithm. This algorithm is used to approximate the exact point in time an event in CIF has become enabled. In short, for this system it means that; when the vertical position becomes negative in Acumen this is recognized and the value of the vertical velocity is changed, while in CIF the program first approximates at what point exactly the position became zero, then it changes the value of the vertical velocity and continues from that approximated point in time. This difference in dealing with enabling guards becomes critical when the ball makes significantly small bounces, known as Zeno-behaviour. This is the reason why a "stopping criterion" is introduced in CIF. When the top of a jump is smaller then the constant "`E`" the ball moves on with constant velocity in state "`Done`". The urgency of this stopping criterion and the difference in dealing with enabling guards is considered in more detail according to another system in Chapter 7.

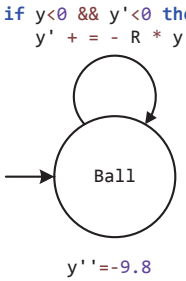


Figure 4.3: Acumen model for the bouncing ball system represented by hybrid automata

4.2 Conclusion

Acumen and CIF both has the possibilities to use their data for a combination of continuous- and discrete behaviour, which is not so surprising because both are developed on purpose of modelling hybrid systems. The interesting element that emerge while modelling this bouncing ball is the difference in managing the critical situations, when the bounces get significantly small. This is about the working principles of the simulators and needs to be taken into account when systems are simulated with one of both tools. At this point it is not clear what causes this difference. This is investigated more accurately in Chapter 7.

Chapter 5

Bifurcating Ball

In this chapter again a model of a bouncing ball is investigated, but when this ball hits the ground it bifurcates into two similar balls of smaller weight. These balls proceed, relative to each other, in the reverse direction and both bifurcate again when touching the ground, see Figure 5.1.

The physical relations are similar to the bouncing ball model of Chapter 4. However, when the first ball is released initially it drops straight down. Subsequently, when touching the ground the vertical velocity of the balls is reduced by a factor, R, and for simplicity the vertical acceleration is constant, $-9.8\ \mathrm{m\,s^{-2}}$, during the complete simulation. The weight of the balls is halved every time they hit the ground. Based on this there is a stopping criterion, namely: when the weight of the balls becomes smaller than 0.2 kg no new balls are created.

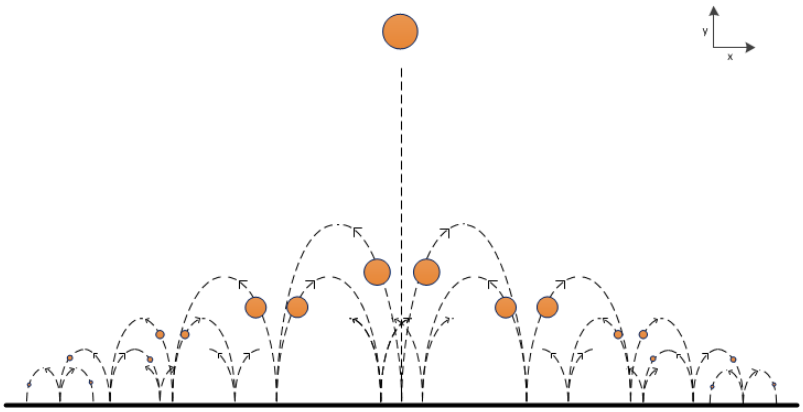


Figure 5.1: Representation of the system of bifurcating balls

An interesting property in modelling Cyber Physical Systems, discovered in this model, is the dynamical size of models. The system starts with just one ball, but in the end a certain amount of objects of an equivalent template, "`Ball`", are present in the model. The situation of a ball bifurcating into two smaller balls is not realistic, but it can be thought of, for example, Automated Highways Systems or Aerospace Systems. These are systems that need to react on changing environmental conditions, cars or planes arrive and leave in the environment of the examined situation. The behaviour of this particular model is not very interesting, because balls are bouncing similar to the bouncing ball discussed previously. What is of interest, is whether it is possible to capture the dynamical size of the system and in what way this influences the model, its performance and/or its complexity.

In Figure 5.2 and Figure 5.3 a graphical representation of the models by means of hybrid automata is given for both tools. The model of the system for Acumen and CIF can be found in respectively Appendix C and Appendix D.

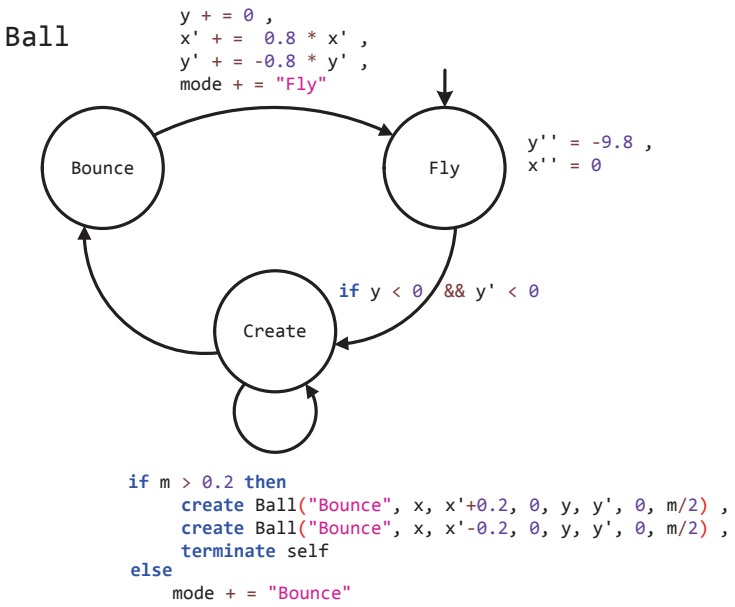


Figure 5.2: Acumen model for the system of bifurcating balls represented by hybrid automata

5.1 Acumen model compared to the CIF model

The aspect of dynamical system dimensions is managed differently in both tools based on the general requirements for a model. In Acumen a model consist of objects, objects can be created dynamically and statically. When an object is created dynamically it means that it is not present in the system initially, but created at a certain point in the simulation. Next to dynamically creating objects they can also be terminated during simulation with the "`terminate`" statement. This makes the size of the system, in terms of objects, dynamical during simulation. Because objects can be created and terminated while the model is executed. According to the defined structure of a model in CIF, it is required to define all automata, used during simulation, statically. These are active during the complete simulation. Based on these principles the models of the bifurcating balls described above are created. The models in Figure 5.2 and 5.3 present respectively, one object for the Acumen model and two automata for the CIF model.

5.1.1 Acumen

In Acumen the object of the model "`Ball`" is defined by values of the variables; position, velocity and acceleration in the x- and the y-direction. The first object of model "`Ball`" is created in the model "`Main`" and is initially in state "`Fly`". This results in the ball falling down until it bounces, then it creates two new objects of model "`Ball`" (`create ...`) initiated in state "`Bounce`", and terminates the current object of model "`Ball`" (`terminate self`). In the state "`Bounce`" of the two created balls, the velocity in the x- and the y-direction is defined ahead of reaching state "`Fly`", where this process is repeated. In this way every object of model "`Ball`" creates two objects of

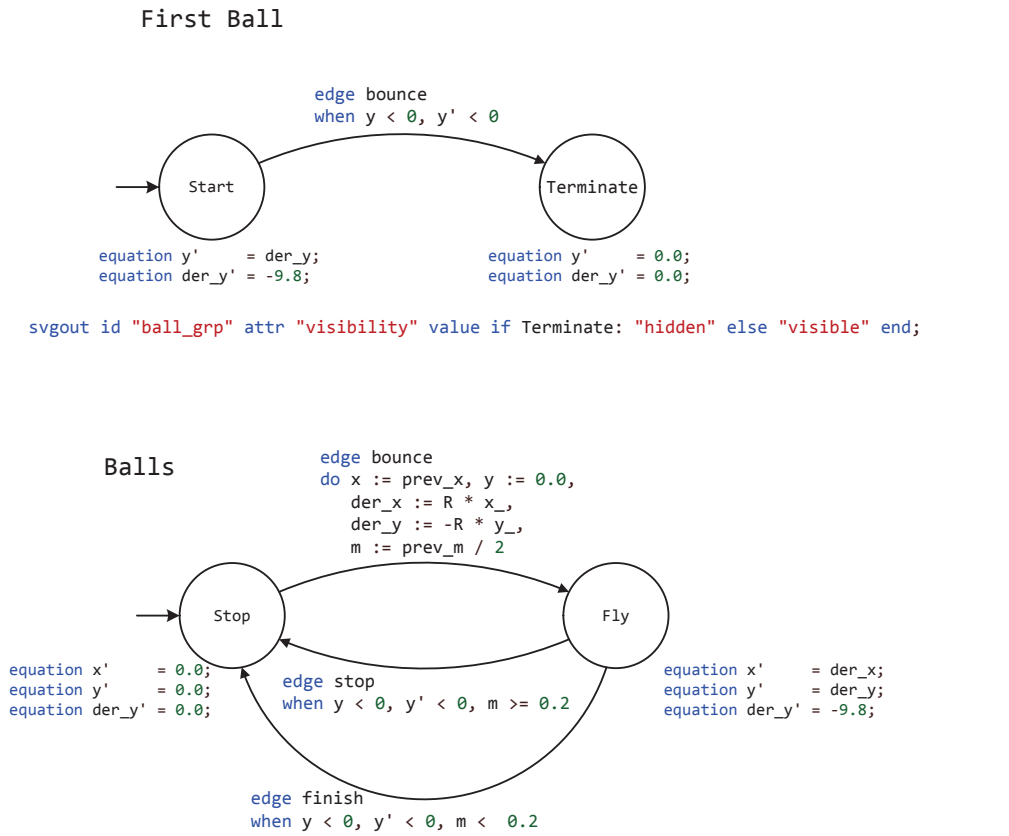


Figure 5.3: CIF model for the system of bifurcating balls represented by hybrid automata

this model before it terminates.

5.1.2 CIF

In CIF, the equivalent system is modelled, but in a different way. As known all automata need to be statically defined in the model, but reuse by using automaton definitions is possible and a useful property in this case. The CIF model consist of two automata, describing the first ball of the model, ("`First Ball`"), and all succeeding balls, ("`Balls`"). The automaton "`First Ball`" describes the behaviour of a ball falling down. When it hits the ground it moves to state "`Terminate`", by the synchronizing event "`bounce`". Here the velocity is zero, so this ball does not move any more. The synchronizing event is also declared in automaton "`Balls`" where it initializes a ball of type "`Balls`". The synchronizing event, `bounce`, is declared in two automaton of type "`Balls`", so two balls are initialized based on the first one. Their position and the reduction in velocity are defined based on the characteristics of the previous ball and they bounce one jump before they return to the state "`Stop`". The visualisation in CIF is adapted in such a way that the objects of the .svg file that only the "active" balls, the ones who move, are visible. This can be implemented using the state of each automaton, as can be seen in the lines in Figure 5.3 where the svg output is declared. It is stated in which case the object are "`visible`" or "`hidden`". The large difference with Acumen is that all automata required in the complete simulation and their relation

to each other needs to be defined in the CIF model. In the model of the system, see Appendix D, it can be seen that this comprises a large number of lines where faults can be introduced by common copy-paste mistakes. In addition, this influences the simulation in a way that it slows down occasionally, probably due to the large amount of calculations the processor executes for all the automata that are present in the model but not used in the simulation.

5.2 Conclusion

Using dynamical objects is not equally easy for both tools, based on the difference in creating objects dynamically. In Acumen the available statements `create` and `terminate` can be used to model dynamic creation/termination of objects. However, when systems get more complex creating and terminating dynamical objects can cause losing the overview of the complete model. Considering the statically defined automata in CIF, these give a better overview of which automata are present in the model. But terminating automata by making them invisible is a modelling trick. This because in the visualisation they can not be seen any more, but the program still takes them into account for calculations. Moreover, the variables of the invisible automata are still evaluated. This costs unnecessary time and memory. Thus, in this case Acumen seems to be more straightforward to use, but it is possible to create an equal visualization for this kind of systems with CIF.

Chapter 6

Traffic Intersection

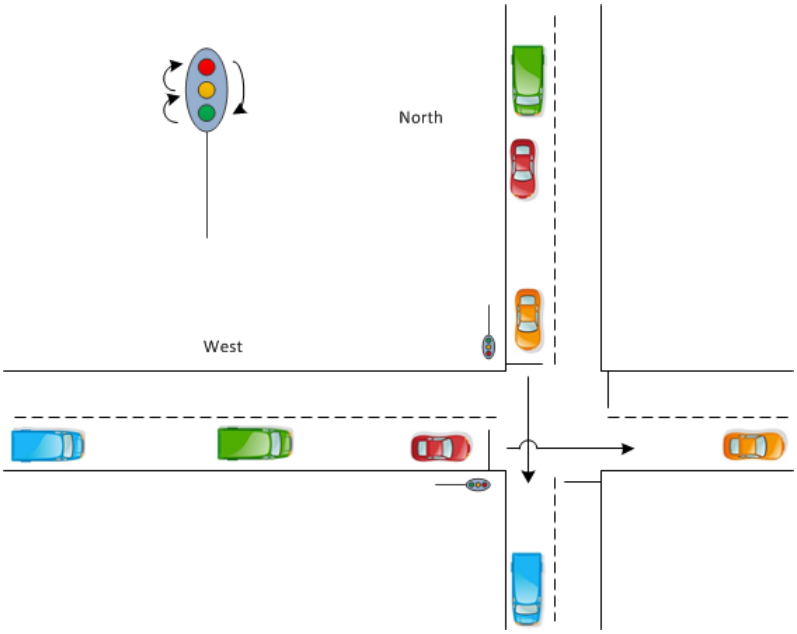


Figure 6.1: Schematic representation of a traffic intersection with two directions

6.1 Physical description

This chapter concerns control of traffic lights at a two road intersection. There are traffic lights for each road that coordinate the traffic in order to traverse the intersection securely, see Figure 6.1. The traffic lights switch color according to the following cyclic behaviour: red, green, yellow, red. Each traffic light is controlled by a controller and these two controllers communicate to ensure the secure behaviour. The working principle of the intersection contains the next set of properties:

- Cars arrive continuously at the queues, according to a predefined rate, when the traffic light is red, yellow or green.
- Cars leave the queue according to a predefined time interval when the traffic light is green.

- A traffic light turns green the moment a specified minimum number of cars is present in the queue.
- A traffic light turns yellow the moment the light has been green for a specified time interval or if the queue is empty.
- A traffic light turns red the moment the light has been yellow for a specified time interval.
- Cars are only allowed to drive straight.

These properties define the individual behaviour of different entities (cars and lights), while the software of the system makes sure the behaviour is correct according to a set of requirements:

- The traffic light in one of the directions is red at all situations. Hence, green lights in both directions or a green light and a yellow light is not allowed.
- In case both directions need to turn green, one of them gets priority. The next time this situation occurs the other direction gets priority. This implies a notion of fairness.

The main focus in this example is communication; the two systems communicate to ensure actions are allowed based on the situation at the other road. Two parallel communicating systems are chosen instead of one top-level controller, because in a more complicated system it is not useful to have a top-level controller. More components means that the top-level controller needs to take more data into account, where in the communicating situation controllers only communicate about data that is important for their own behaviour. Therefore, it is better to have communicating components, this saves a lot of complexity. This situation, communicating controllers instead of a top-level controller, is commonly used in Cyber Physical Systems containing networks. Components of systems cooperate by communication to create the required behaviour, but when two actions need to exclude each other the situation where both are enabled simultaneously is a critical one and therefore is considered more accurately.

6.2 Acumen model compared to the CIF model

The complete system of the traffic intersection is modelled by two parallel subsystems, each consisting of a traffic light, the queue of traffic and a controller. The controller activates the traffic light to turn green based on the observation of the the traffic in the queue. In addition, when a cycle at the traffic light has started, the traffic light turns yellow and red based on its own principles. Different principles are used to model the communication in Acumen and CIF. In CIF, the concept of synchronizing events is available and useful to model this system. The Acumen language does not contain a similar feature, so here a principle needs to be implemented by using the available functions. Considering the representations by hybrid automata in Figure 6.2 and Figure 6.3 it can be seen that both models contain locations/states with equivalent labels. Changing the state of a system happens when an enabled event is taken to another state. An event is enabled when the corresponding guard is true, the guard of each event can be found in the model in Appendix F and Appendix E, respectively for CIF and Acumen.

The basic principle is the following. A traffic light is initially “Red” and the controllers are initially in state “Observe” while traffic is arriving at the corresponding queue in automaton “Traffic”. Then, at the moment the guard corresponding to the event of turning green is true the controller can switch state to the state “Green” as well as the traffic light. Which means traffic can also leave the queue from that moment on. Next thing that could happen is that an intersecting direction

is ready to turn green as well. In that case, the corresponding controller will move to state "Request" and the controller of the direction that has a green light will move to state "Stop". Then, after a certain amount of time the traffic light automaton moves from state "Green" to "Orange" followed by "Red". At that point the corresponding controller moves to state "Observe" and the controller which is in state "Request" is able to turn green and thus move to state "Green". This shortly explains the basic idea behind the division of the system into different states. This idea is kept quite equivalent for both tools to make it possible to focus the comparison on the advantage or disadvantage of using synchronizing events. The functions of the extra state "Check" in the Acumen model is explained later on.

6.2.1 CIF

In Figure 6.2 the CIF model is shown schematically by using hybrid automata. Each controller communicates with the corresponding traffic automaton and traffic light automaton, because these are equal for both directions the figure shows only one of both. The events are designated by a label, events are defined as synchronizing events when multiple events have an equivalent label. Anyway, for simplicity and easy explanation, in the figure this is indicated by colors. The important feature of synchronizing events is that execution is allowed, only when at least one synchronizing event in each of the automaton where they are defined is enabled. This is a useful property to exclude the situation of both lights turning green simultaneously. That situation is considered step by step in more detail on the basis of Figure 6.2. For the queues of traffic, variable `x`, of the North-direction and West-direction equivalent arrival- and departure rates of cars are assumed.

- First, the lights and the controllers are in the initial state: "`Red`", and "`Observe`" respectively. Traffic arrives in both queues with equal rates.
- Based on the principle of synchronizing events, the events "`turn_green`" and "`turn_green`" are enabled at an equivalent point in time where both queues reach the minimum number of vehicles (`queue >= queue_length`).
- The CIF system can execute only one transition at a time, so based on the criteria it executes event "`turn_green`" or "`turn_green`". In this case, for instance, event "`turn_green`" is executed.
- The result of executing event "`turn_green`" is that Controller North is in state "`Green`", which means synchronizing event "`turn_green`" is not enabled in this automaton any more. This ensures that Controller West, and therefore the associated traffic light, can not turn green.
- Time passes by, while cars arrive in both directions and leave in the North direction.
- Now event "`do_Request`" becomes enabled, because Controller West is in state "`Observe`" and Controller North is in state "`Green`". Hence, Controller North moves to state "`Stop` and Controller West moves to state "`Request`.
- Subsequently, the defined time a traffic light is green elapses or there are no cars in the queue any more. Then the traffic light North turns yellow and after some time red. At that point Controller North moves back to state "`Observe`", by event "`turn_Red`".
- In this situation the only enabled transition is event "`answer_Request`", turning traffic light West green. In a similar way an equivalent procedure follows for the West direction, resulting in the situation where traffic light West turns red because traffic light North wants to turn green or both controllers end up in state "`Observe`" because there are not enough cars in both queues.

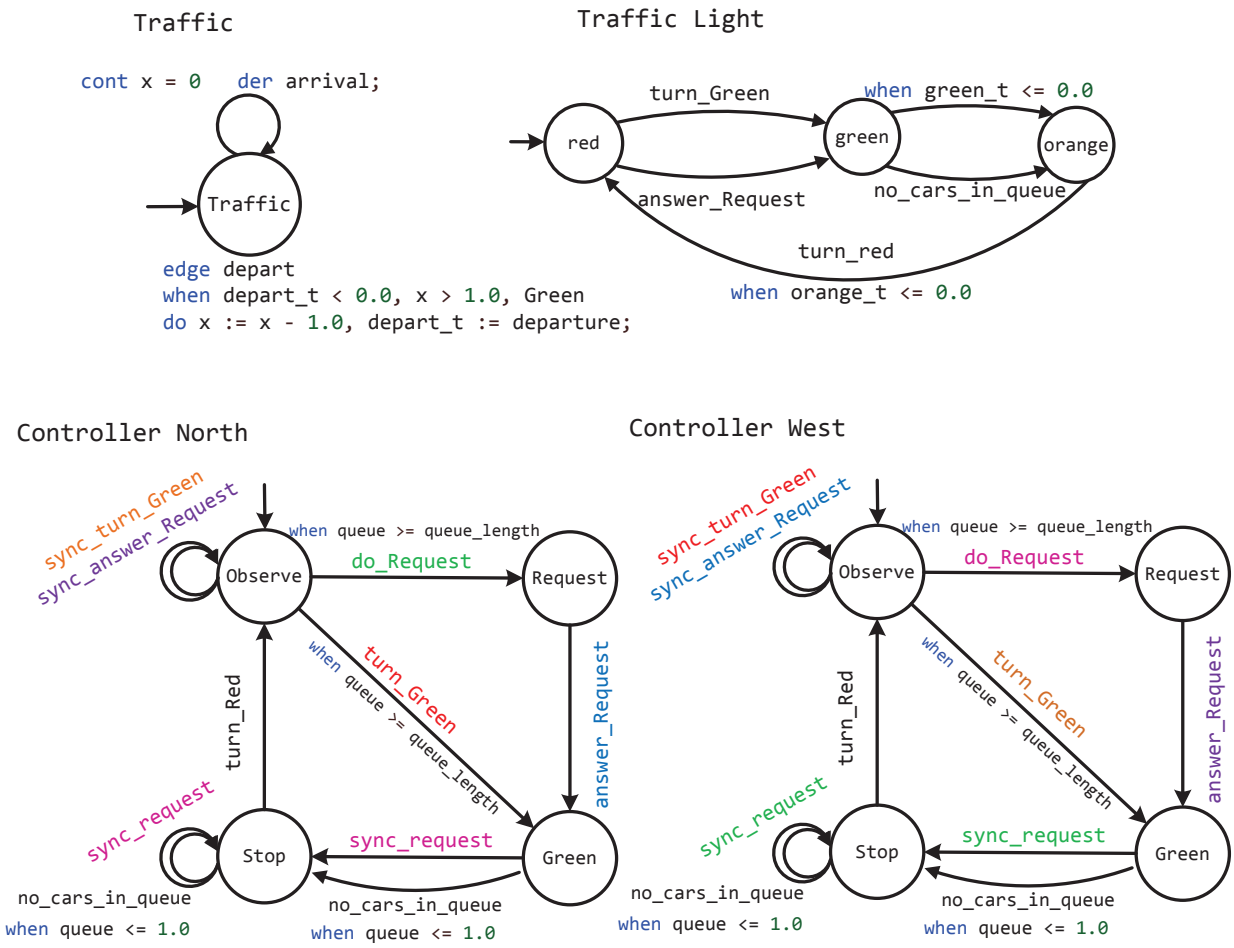


Figure 6.2: Representation by hybrid automata of the CIF model for the traffic, traffic light and controllers

The important part of this synchronization is the fact that both events "turn_green" and "turn_green", turning the light directly green, are defined as a self-loop in the state "Observe" of the other controller. This means to turn a traffic light green the other controller needs to be in the state "Observe", and thus the corresponding traffic light is red.

The second requirement for the system is to ensure a certain notion of fairness, so not every time the same direction gets priority in the critical situation. By including the state "Request" in both controllers, this behaviour is ensured. For instance, consider a similar situation as before, with equal arrival rates, but cars arrive faster at the queue than they depart. Assume the state "Request" is deleted, so when one of the lights returns red and the associated controller returns to state "Observe" the system chooses again between executing event "turn_green" and "turn_green", based on the defined criteria. This can cause the system to give priority to the same traffic light every time, for example traffic light North. Then traffic light West never turns green. To prevent this, controller West can move to state "Request" at the moment controller North is in state "Green" or "Stop". In that situation when controller North returns to state "Observe" it is not an option any more to turn green immediately, if there are still enough vehicles in the queue. This because the event "turn_green" is not enabled in controller West, so event "answer_Request" is enabled and traffic light West turns green. Using the state "Request" makes

sure the system alternates between both directions when both queues fulfil the requirements to turn green all the time. In case a queue becomes empty while the corresponding light is green the event "no_cars_in_queue" makes sure a traffic light turns red. The current queue is then empty, so no synchronizing event is needed, the current controller does not want to turn green when it returns in state "Observe" anyway.

6.2.2 Acumen

The Acumen model has an equivalent structure as the model in CIF, the large difference is that shared variables between controller North and controller West are used instead of synchronizing events in CIF. In Figure 6.3 one object of type Traffic, Traffic Light and Controller can be seen where in the actual model there are two of each (one for each direction).

In the top level of the model the relation between the variables of both objects is declared. This means both controllers have access to read the state of the other controller by variable "other_Cont". In this model it is ensured whenever a queue satisfies the minimal length and the other controller is not in the state "Observe" it is not allowed to turn green. In that case the current controller moves to state "Request". This to create fairness similar to the CIF model. When this is modelled it needs to be considered that assignments for sharing variables are continuous behaviour in Acumen. Therefore, when the critical case is taken into account, where both queues fulfil the minimal length simultaneously and controller North and West are in the state "Observe". Then, both lights turn green simultaneously because the two discrete assignments are collected and executed. Afterwards the states are shared, but this is too late.

This is undesired behaviour and needs to be prevented. This is done by assigning the epoch at which both lights want to turn green to a variable of type real, check_t, in both controllers, and they move to state "Check" instead of directly to state "Green" (arrow 2 in Figure 6.3). In this state it is checked if the variable check_t in the simulation exceeds the epoch saved in the variable of type real. When this is the case the system has executed at least one continuous step, so the states of both controllers are shared. Now the controllers can make a decision safely based on the value of variable "other_Cont". Hence, if the other controller is in the equivalent state the direction with priority turns green (arrow 4 in Figure 6.3). The other controller returns to state "Observe" (arrow 3 in Figure 6.3) and later on to state "Request". Otherwise, when one controller moves to state "Check" and the other controller is still in state "Observe" the current controller can turn green without any risks (arrow 1 in Figure 6.3).

This principle where a variable is used to check whether permission is given to access a certain part of the system is based on Fischer's protocol [6]. By implementing this protocol correctly the complete system operates according to the requirements without using a top level controller or synchronizing events.

6.3 Conclusion

Comparing both models it turns out that the structure of a CIF model becomes more abstract by using synchronizing events. Hence, when events at different automata need to exclude each other, like in this case. This can be modelled by using the synchronizing events, using self-loops, such that executing one of them disables the other one, because the state of the current automaton changes.

In Acumen, this is harder because only the change in state of each object can be used. And because variables are shared during the continuous step of the simulation an extra state is introduced based on Fischer's protocol, to exclude undesired behaviour.

This represents a significant difference between Acumen and CIF: CIF models are based on a strict structure and multiple defined functions are available, while in Acumen very little functions are available and the structure of a model is more free. Therefore, where in CIF it is possible to use functions, in Acumen these need to be defined and implemented by the user. This increases the likelihood of bugs, where in CIF these modelling mistakes are prevented because functions can be used. Based on their description it is interesting to see what the effect will be when the system gets more complex.

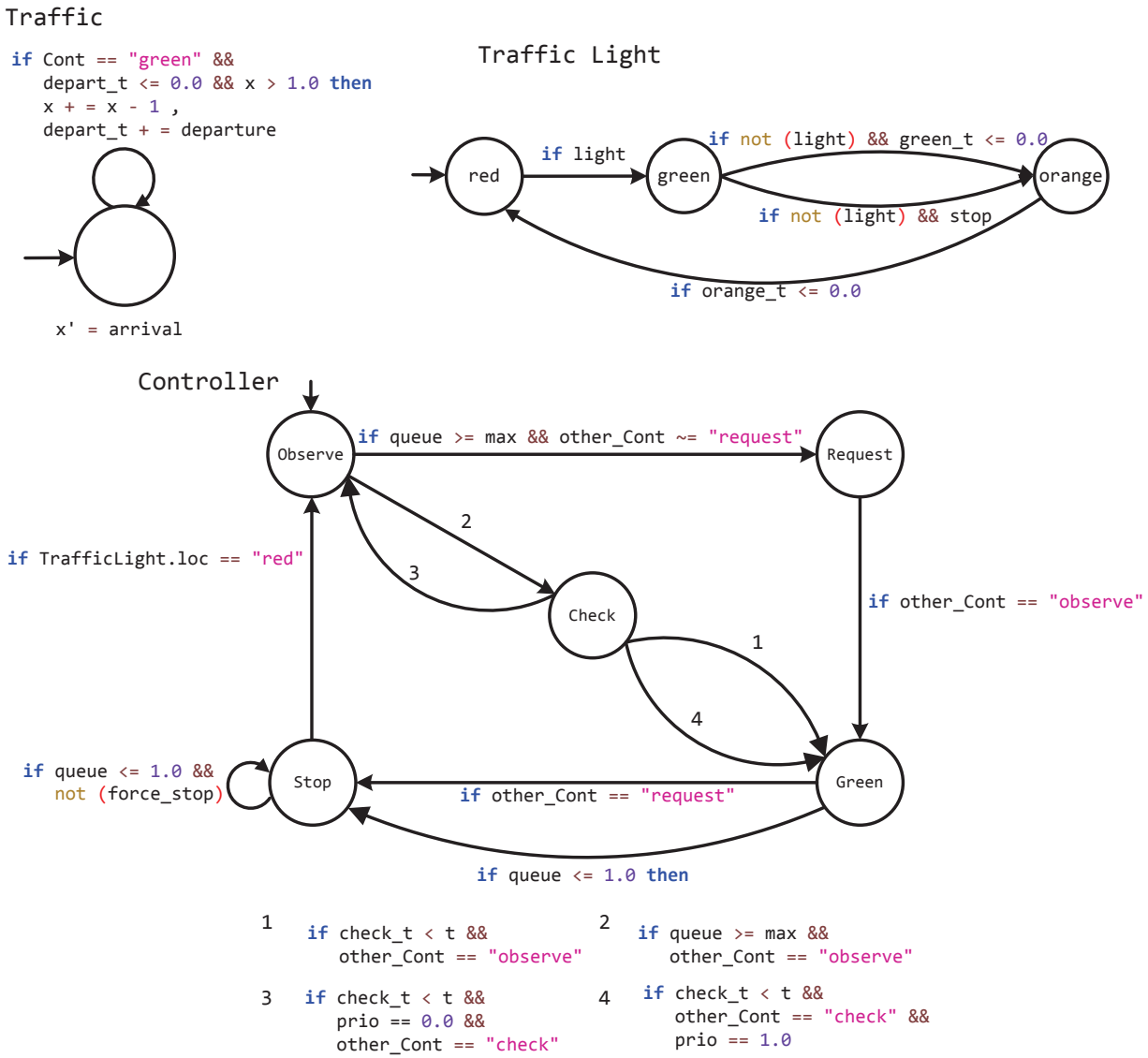


Figure 6.3: Representation by hybrid automata of the Acumen model for the traffic, traffic light and controllers

Chapter 7

Zeno Behaviour

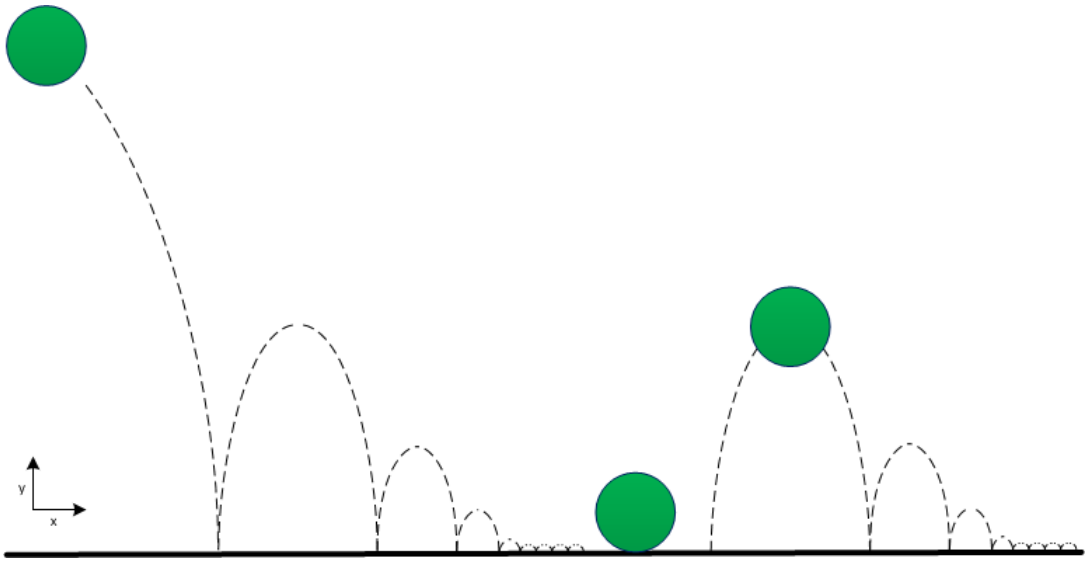


Figure 7.1: Schematic representation of the bifurcating ball example

7.1 Physical description

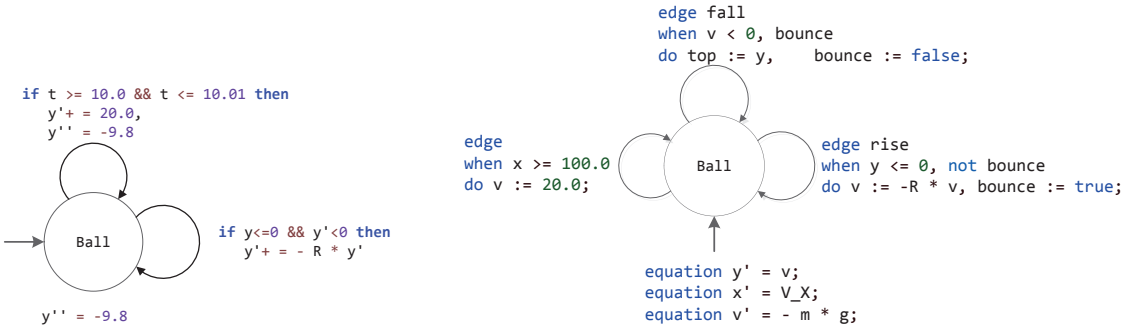
This chapter focuses on the aspect of Zeno behaviour in the bouncing ball example, discussed in Chapter 4. The expression "Zeno behaviour" [2] is derived from Zeno's paradox. Briefly, this is the phenomenon where an infinite number of events appear during a finite time. This can be an issue in modelling and simulation of hybrid systems. The example of the bouncing ball is such a system. The velocity of the ball is reduced every time it hits the ground, so the jumps decrease height. After a certain amount of time the ball is observed to just roll on, in reality. At that point the modelling tool still calculates new values for the velocity and evaluates the height of the ball over time. By definition this procedure never stops, but in modelling and simulating these systems it is undesired to get stuck in these calculations. Therefore a tool designed for modelling and simulation of hybrid systems needs to include a procedure to deal with this Zeno behaviour.

The procedures of Acumen and CIF to deal with this are investigated by modelling and simulation of a system of a bouncing ball, see Figure 7.1. This ball is initially at a certain height, where it is released and its velocity is reduced every time it hits the ground. At a given x-position, past the Zeno-point, the ball is smacked again. In case the tool deals with the Zeno behaviour and thus passes the Zeno-point, smacking the ball will be visible in the simulation. Otherwise, the program probably got stuck at the Zeno-point and never reached the x-position where the ball is smacked.

7.2 Observations

In Chapter 4, the simpler example of the bouncing ball was discussed. There it turned out the models of Acumen and CIF were quite different. In Acumen only the bouncing behaviour was modelled, while in CIF a stopping criterion was included to stop the ball from bouncing. In case of Zeno behaviour, in both tools a model is made that only contain the bouncing behaviour, as can be seen in Figure 7.2a and Figure 7.2b for respectively Acumen and CIF. Hence, every time the ball hits the ground the velocity changes sign and the value is decreased. An extra event is created in both models to confirm the velocity of the ball is assigned the value of 20.0 $\mathrm{m\,s^{-1}}$ in positive direction when it reaches x-position 10.0, and it bounces on from that situation.

The model in Acumen does not change that much with respect to the one previously discussed in Chapter 4, so the simulation result is equal as well, although now the ball get smacked at x-position 10.0. In CIF, the simulation shows different results. The ball bounces until it is near to x-position 6.0 and from there on no movement can be recognized any more, although the program is still running. Probably, this means the program got stuck at that point, the Zeno-point. Let us try to explain what underlying principles cause these differences between both tools where the model is equivalent.



(a) Acumen model

(b) CIF model

Figure 7.2: Representation by hybrid automata of the models used to investigate Zeno-behaviour

7.2.1 CIF

The general working principle of CIF with respect to simulation is already discussed in Chapter 3. Executing models in CIF is about executing enabled events one by one. Here the focus is on enabling these events and the calculation methods underlying, based on a variable time step. Let us take a look at the model for Zeno behaviour, see Appendix H. For instance, take into account edge "`rise`" that is enabled when $y < 0$ and `bounce = false`. The discrete variable bounce is not of interest in this case, because its value can only be adapted at events. Thus, the moment the value changes is known by the simulator. In case of the continuous variable this is more critical,

because the guard requires the value of y to be smaller than or equal to zero instead of exactly zero. This because evaluating continuous variables for a fixed time step can cause the simulator to lack the point in time y is exactly zero. Besides, it is really hard, if not impossible, for a computer to calculate the exact value 0.0. As discussed previously, the CIF simulator works according to a certain algorithm to solve Ordinary Differential Equations (ODE) in order to evaluate continuous variables over time. Next to this algorithm a root-finding algorithm is used. The purpose of this second algorithm is to define the exact point in time an event is enabled, based on a certain tolerance. This is done based on the following steps [1].

- The continuous variable y is evaluated over time according to the general working principle of CIF. Based on this principle the condition $y < 0$ is checked too.
- At a certain moment in time the condition $y < 0$ is fulfilled and the event is enabled.
- The root-finding algorithm tries to figure out at which exact point in time the guard has become true. It starts with defining an interval based on the results of the last two time steps, the last but one where $y > 0$ and the last where $y < 0$.
- It calculates the value of y at the middle of this interval; if this value is positive the midpoint is set to the upper boundary of the new interval, if it is negative the midpoint is set to the lower boundary of the new interval.
- In this way, each step the interval around the exact solution is bisected until the length of the interval is small enough, based on a predefined tolerance. Then it can be concluded that in that small interval the condition has become true. Subsequently, the simulator can proceed from this point in time.

This method is called the bisection method, and the root-finding algorithm in CIF works in an equivalent way. The difference is that it converges much faster to a solution. This makes it better to understand why the ball in the CIF simulation stops at the Zeno-point and does not continue to the position where it would be hit for the second time. What occurs in CIF is that when the bounces of the ball become significantly small it is probably hard for the simulator to come up with a result of the root-finding algorithm. The program continues decreasing the time step to find a sufficient interval, and because these intervals become smaller and smaller after every bounce, this process continues the rest of the simulation. For the user this is recognized as the ball standing still. The ball will never reach the point where it is smacked, because the solver keeps on calculating an infinite number of significantly small new bounces. This "situation" can be solved by introducing a stopping criteria, in this case the state "`Done`", see the model in Appendix H. Here it is defined at which tolerance it is not needed to calculate new bounces any more. Then the velocity is set constant, by assigning the value zero to the acceleration. In the simulation the ball rolls on, while in the calculations it moves up with a significantly small, constant, positive velocity. In fact the calculations in CIF are accurate and precise, but causes the program to get stuck. Therefore, a stopping criteria is used, resulting in a less accurate and less precise solution. But the model with the stopping criteria will not suffer from Zeno behaviour any more, and thus can be simulated and evaluated.

[1] a more detailed explanation of this algorithm can be found on `http://cif.se.wtb.tue.nl/tools/cif3sim/solver/basics-root.html`

7.2.2 Acumen

Based on the working principle of Acumen, discussed in Chapter 2, the simulation results can be declared. Acumen works with a fixed time step to evaluate continuous variables over time. It is based on an algorithm for solving Ordinary Differential Equations (ODE), just like CIF. At a certain point in time the simulator of Acumen reaches the "critical region", where the bounces occur during one time step. In this case the simulator recognizes that the criterion for changing the direction and value of the velocity are met every time step. Hence the simulator executes these operations. In this way the ball continues with bounces of equivalent size, and thus it continues in the x-direction as well. This means it reaches the position where the ball is smacked, and after that a new bouncing ball process starts.

The simulation presents one trace of the model and in some way neglects the Zeno behaviour of the system. This is inaccurate and therefore an option in Acumen is developed to deal with this, called the "enclosure semantics". It can be seen as a different environment in Acumen that evaluates models in a different way, namely by an enclosure. When the model is executed using the enclosure semantics the simulation is based on the idea that it produces an interval as a result. The program can ensure that this interval contains the right solution, if there exists one. It is a method that gives more certainty about the results than the traditional semantics, used by default. What happens is that the program calculates with intervals instead of values. These interval calculations are based on interval arithmetic. In short, it boils down to the fact that by calculating with intervals, a fixed time step and the fact that it needs to be ensured the solution is in the resulting interval causes the solution interval to increase over time. The calculated interval also includes incorrect values, as can be seen in Figure 7.3, but these can be deleted by introducing invariants in which physical boundaries of the problem can be defined, or these values can be neglected based on these boundaries.

In the end the enclosure semantics guarantee that, if there exists one, the solution to the problem is in the calculated interval. In case of the Zeno-point of the bouncing ball it means that it still misses the exact point of bouncing when the ball gets close to the Zeno-point, but it can ensure that the result of that bounce is in a certain interval. This argues why the program can calculate past the Zeno-point. In fact, the enclosure semantics of Acumen is not so precise, because large intervals can be the result of a simulation, but the result is accurate.

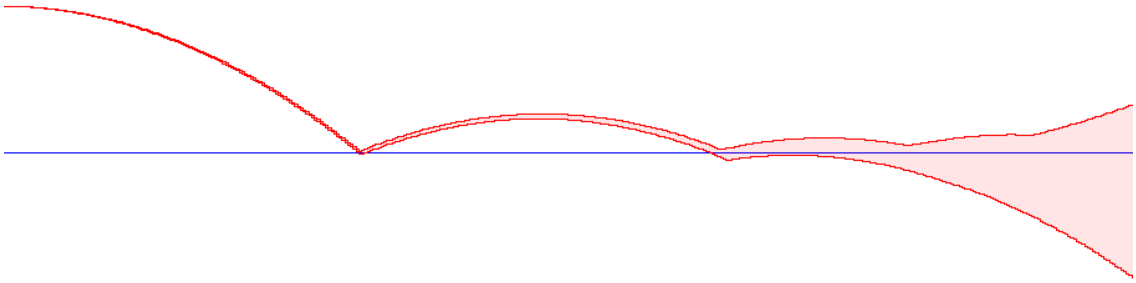


Figure 7.3: An example of the graphical results in Acumen by using the enclosure semantics for the bouncing ball example

7.3 Conclusion

It is clear that the Zeno behaviour in the bouncing ball example is a critical region of the model for both tools, CIF and Acumen. Based on the working principle of the semantics both tools deal with this property in a different way. In CIF, the model is simulated very precise and accurate. The program tries to stay as close as possible to the mathematical definition, but therefore it gets stuck in calculations. Thus, a CIF model with Zeno behaviour will normally never reach the Zeno-point. The way this is fixed in CIF, decreases the precision and accuracy of the model. The modeller needs to define a criterion where it is allowed to do not discover the Zeno behaviour any more and proceed with the simulation.

In the traditional semantics of Acumen, used by default, the program ignores the Zeno behaviour. It calculates with a fixed time step and will miss events when they appear during one time step, this is an inaccurate and imprecise method. Therefore the enclosure semantics can be used, which gives a more accurate solution through a 100 % confidence interval that contains the solution, if it exists. In this case the program gives an accurate, but still imprecise, solution.

When Zeno behaviour appears in a system it is most of the time undesired for the system to get stuck. The purpose of modelling and simulating a system is to represent the reality up to a certain level. Therefore, for each model containing Zeno behaviour a choice can be made about the accuracy and precision needed around the Zeno-point in a particular situation and based on that choice a modelling method can be used.

Chapter 8

Scaled Traffic Intersection

8.1 Physical description

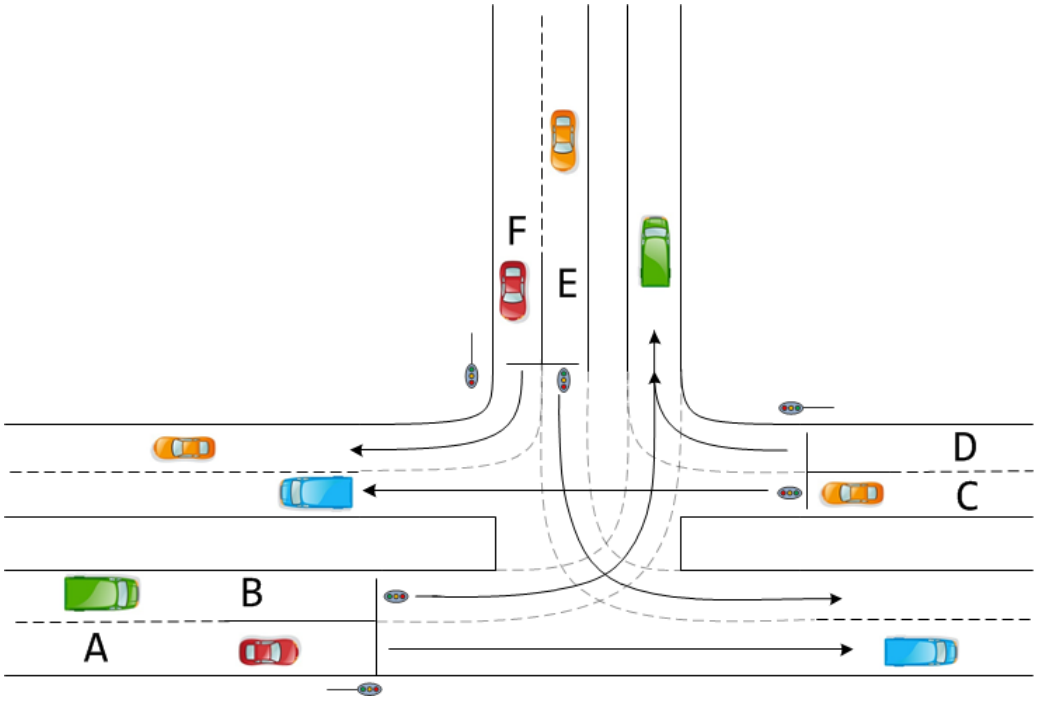


Figure 8.1: Schematic representation of a traffic intersection with six directions; A, B, C, D, E, F

In this chapter a more complex traffic intersection, according to the intersection in Chapter 6, is discovered based on scalability of communication between objects of a model. The example of the simpler intersection, with two roads crossing each other, presented the differences between both tools for modelling systems where communication is used to exclude undesired behaviour. By modelling a more complicated communicating system including equivalent behaviour the influence of scalability on the models in Acumen and CIF is investigated. The situation, see Figure 8.1, is a traffic intersection where six different directions cross each other. This intersection was also discovered in [5] with respect to optimization and implementation of networks of signalized traffic intersections. The intersection is located near Den Bosch in the Netherlands. In Figure 8.1 each

direction is characterized by a letter. Three directions intersect with one other direction (A-E, D-B and F-C) and the three other directions intersect with three directions (B-(C,D,E), C-(B,E,F) and E-(A,B,C)). In this situation at every moment in time it is allowed for only one direction per group of intersecting directions to have a green light. Next to this, there are a number of properties for the system that need to ensure the desired behaviour:

- It is not allowed for traffic lights of intersecting directions to turn green simultaneously. One of the lights can be yellow or green, but then the others are red.
- The system needs to include fairness, meaning that in case controllers of multiple intersecting direction are ready to turn green not always the same controller gets priority.
- When a certain queue is empty the corresponding traffic light turns yellow.
- A traffic light is green for a predefined amount of time, unless the corresponding queue is empty. Then the traffic light will turn red earlier because there are no cars in the queue waiting any more.
- A traffic light is always yellow for a predefined amount of time before it turns red.
- When a pre-set amount of cars is present in the queue the corresponding traffic light can turn green when the situation at the intersection permits.
- Cars arrive continuously at the queue according to a constant predefined rate all the time. They leave the queue according to a constant predefined rate only when the corresponding traffic light is green. The rates are kept constant because in Acumen it is not possible to vary these variables during simulation, while in CIF this is possible. Up to a certain level both models need to be equivalent to make a good comparison.

These properties are similar to those mentioned in Chapter 6, but the model need adaptations because of more objects are communicating. Nevertheless, with respect to scalability the concept of the controllers used for the intersection of two roads is kept equal as much as possible. In that way the consequences of scaling this model can be examined.

8.2 Acumen model compared to the CIF model

The controllers of the model for the simple intersection can be used for the directions intersecting with one other direction. But the controllers for the directions intersecting with more than one direction are adapted a little. This, because the criteria for doing a request will change and therefore are not yet ensured in the controller. These criteria are:

- Only one direction of a group of intersecting directions can be in state "`Request`".
- Next to fairness for turning green, there needs to be fairness in the model for doing a request.

In the situation for two intersecting directions these criteria are always met, because a direction could only do a request when the other direction has not. In the situation with multiple intersecting directions these criteria need to be taken into account more accurate and ensured by adapting the controllers. In what way this is done is explained below by each tool.

8.2.1 CIF

In the automaton "controller" of the CIF model with two intersecting directions it is ensured, using synchronizing events, that a controller can change state to "Request" whenever the other controller is in state "Green" or "Stop". In the more complex situation, where there are four intersecting directions, the synchronising event of doing a request is added to state "Observe", because there can be multiple directions that want to do a request while one direction has a green light. Therefore the controllers need to take into account a certain priority principle to do a request, ensuring all directions are able to do a request and therefore turn green. In Figure 8.4 a general description of the controllers used in the CIF model is presented by automata. The labels of the events correspond with the labels given in the specification, and colors are used to represent synchronising events. In the model two different controller definitions are used, in contrast to the single definition in the simple model. This, because the differences in number of intersecting directions and corresponding events cannot be included in the parameter definitions, but the layout of the automaton "controller" is similar.

The largest adaptation made to the controller of four intersecting directions is about the event "do_Request" combined with the priority principle. This event is only enabled when the current light has lowest priority of all intersecting directions that want to do a request simultaneously. The function "minimumprio" is used to check the status and priority of intersecting directions. In Figure 8.2 the description of the function can be seen.
The input arguments for the function are:

- "own_prio", the priorities of the current controller (initially 1 for every direction)
- "prios", a list of priorities of the controllers of the intersecting directions
- "cond", a list of conditional arguments of the controllers of the intersecting directions

The first two arguments are self-explanatory, but the third one needs a little explanation. These conditional arguments are defined at the top of the .cif file and are boolean variables based on the number of cars in each queue in conjunction with the state "Observe" of each controller. For instance, when the number of cars in queue B exceeds the minimum value for that direction and the controller of direction B is in state "Observe" the boolean variable "cond" for direction B is true. In any other case this value is false. These variables are used to investigate in which intersecting directions a request is enabled. The function "minimumprio" checks for every intersecting direction where a request is enabled if the corresponding priority variable has a higher value than the priority of the current direction. If this is the case for all intersecting directions the current controller can do a request and the priority is increased with the value one. This function is only used for intersecting directions that interact with more than one direction, because a direction that intersects with just one other direction uses the state "Request" to enforce a green light with respect to the intersecting direction, as explained in Chapter 6.

In this case the program deals with requests based on the priority variables every time more than one direction wants to do a request. If no other direction can do a request the function "minimumprio" returns the value zero, this indicates a request can be done without adapting the priority because there are no more directions involved. In any other case, if a controller is in state "Observe" it can turn green directly when all controllers of intersecting directions are in this state too, ensured by the synchronizing event "turn_Green".

```
func int minimumprio(int own_prio; list int prios; list bool cond):
  int i = 0;                //iteration variable
  int winning_prio = 0;     //number of intersecting directions with lower priority
  int enabled_request = 0;  //number of intersecting directions enabled to turn green

  while i <= size(prios) - 1:

      if cond[i] = true:
        enabled_request := enabled_request + 1;

        if own_prio <= prios[i]:

            winning_prio := winning_prio + 1;
            i := i + 1;
        else
            i := i + 1;
        end

      else
        i := i + 1;
      end

  end

  if enabled_request = winning_prio:
     return winning_prio;
  else
     return -1;
  end
end
```

Figure 8.2: The function "`minimumprio`" used in the CIF model

The difference between the two controller definitions used in this case is the number of events and shared variables, with respect to the priorities of each direction, in the parameter definitions. This is caused by the fact that for each intersecting direction an event starting with "`sync_`" is needed. For instance, when a road intersects with three other directions three "`sync_`" events are needed on each of the self-loops in Figure 8.4. This makes it not possible to have one controller definition for all controllers, but there is one controller definition for each number of intersecting directions.

8.2.2 Acumen

In Figure 8.5 the Acumen model can be seen represented by hybrid automata. In this model the situation where a controller needs to wait for doing a request until the light in the other direction is able to turn green, like in the CIF model, is not present for a controller of two intersecting roads. This because Acumen does not provide synchronizing events, so a request can be done at every moment if no intersecting direction had done a request yet. The controller of the light that is green at that moment will react on it by changing its state to "`Stop`" because it reads the state "`Request`" of the other controller. This means that principle does not have to be changed for the more complex situation. But the conditions for a controller to be able to do a request need to change because of the absence of synchronizing events.

The simple model dealt with this by introducing an intermediate state "`Check Green`". It prevented both directions to have a green light simultaneously as explained in Chapter 6, by using Fischer's protocol. In the complex situation the controllers of the simple system are used for the directions intersecting with just one other direction, see Figure 8.5. The controllers for the remaining directions, intersecting with three other directions, are adapted with respect to do a request. Directions that intersect are not allowed to do a request simultaneously, so to take care of this the state "`Check Request`" is introduced, see Figure 8.5. Here, equivalently to the procedure in the "`Check Green`" state it is checked if multiple intersecting directions want to do a request

simultaneously and if so the one with priority continues to the state "`Request`". The other one returns to the state "`Observe`". The priorities of each controller are defined in a similar way as in CIF. Each controller has its own priority variable and can read the priority variables of intersecting directions. Then it is decided based on the state and priority of the intersecting controllers which controller is allowed to do a request.

As can be seen in the model in Appendix I, checking the priority of different controllers takes several steps in the program. There are in both states, "`Check Green`" and "`Check Request`", three possibilities; there are other controllers in this state but the current controller gets priority, there are other controllers in this state and one of them gets priority, there are no other controllers in this state so the current controller can continue. The states "`Check Green`" and "`Check Request`" cover the behaviour of the synchronizing events in CIF, based on Fischer's protocol.

8.2.3 Scalability

The influence of scaling this system, where components communicate with each other, is compared based on the size of the models. It is obvious that no hard conclusions can be drawn from this kind of comparison, because the structure and therefore size of a model always depends on the modeller and his or her preferences with respect to modelling systems. Nevertheless, by comparing both models and looking at the size of the different components it is interesting to see what kind of differences appear due to scaling. This because both languages support a different modelling principle for communicating between components.

In Figure 8.3 the number of lines in the model needed to define certain behaviour are represented for CIF and Acumen. These are separated for templates and instantiations, where templates represent the definitions of the controllers and its behaviour and instantiations are the definitions of the controllers based on the templates by using parameter definitions.

The data is plotted with respect to the number of communicating objects, 2, 4, 6 and 8. What can be seen in the first place is that the size of all components of the models are increasing with the number of communicating objects. What is more interesting, is the size of the templates. The templates in Acumen are in general larger than the templates in CIF and they increase also much faster. A reason for this can be that the principle of Fischer's protocol needs to be implemented by the modeller itself and when scaling this, more variables are shared and thus more variables need to be checked with respect to this principle. In CIF, synchronizing events are used and thus the principles and therefore templates do not change by scaling, but the number of parameter definitions changes. This is what can be seen in the figure if the size of the instantiations are compared. In CIF, each controller uses three different types of synchronizing events (`request`, `turn_green` and `answer_request`). Each of these events is synchronized for every communicating object. For instance, a controller communicating with three other controllers needs $3 * 3 = 9$ synchronizing events for its parameter definitions next to the other events and shared priority variables. Where in Acumen only the priorities and states of the communicating controllers need to be defined in the parameter definitions. This causes the size of the instantiations in CIF to scale with the number of communicating controllers times three where in Acumen the size of the instantiations scales with the number of communicating controllers.

Next to the size of the model, which is a relative measure, a practical difference that can be noticed is that the use of defined functions like synchronizing events in CIF results in no adaptations of the model while scaling. Where in Acumen, the principle based on Fischer's protocol needs to be adapted every time because it is defined by the modeller itself. This makes it more sensitive for modelling errors or copy-page mistakes.

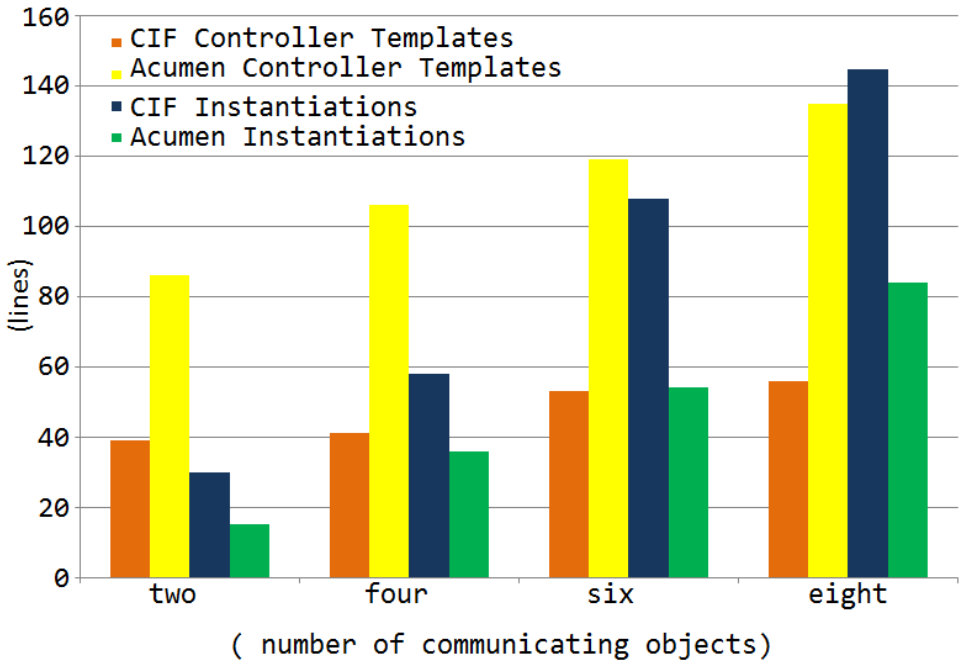


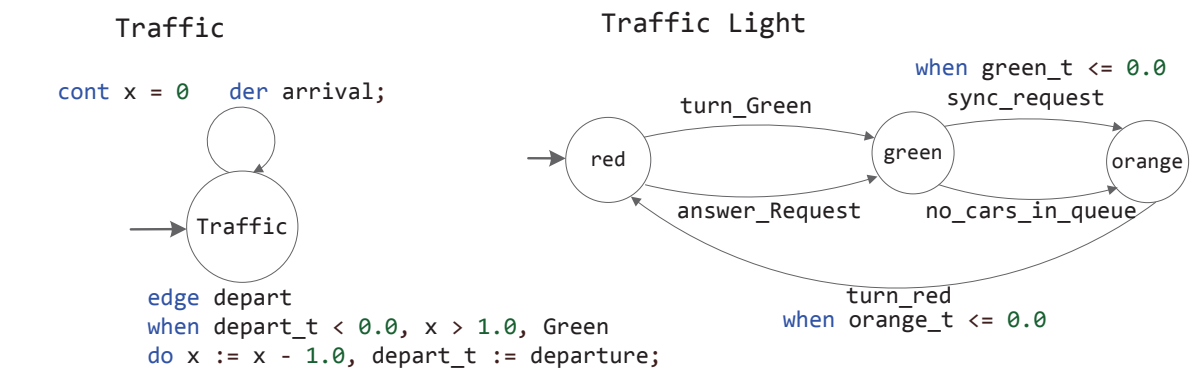
Figure 8.3: Graphical representation of the results when comparing the CIF and Acumen model to the number of lines

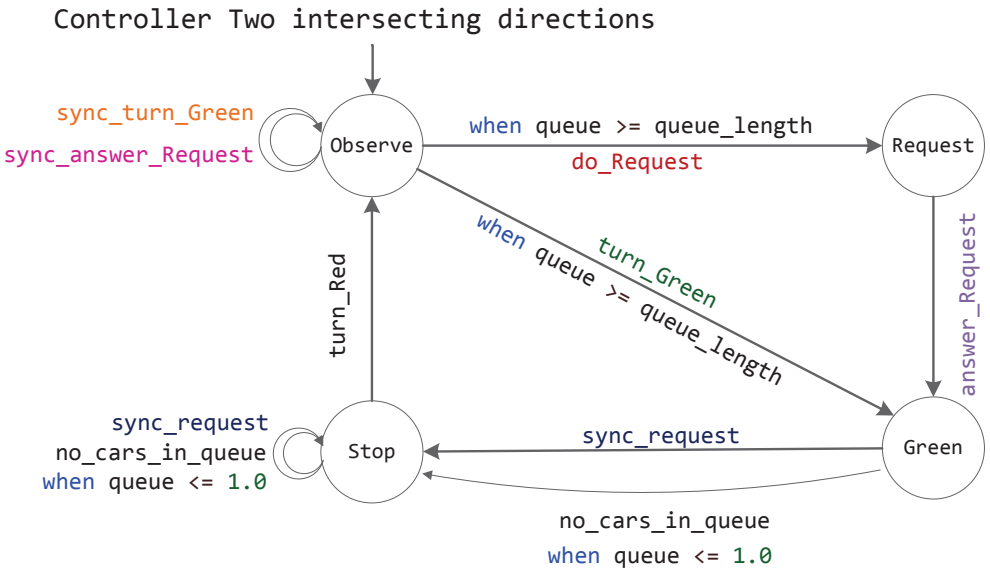
8.3 Conclusion

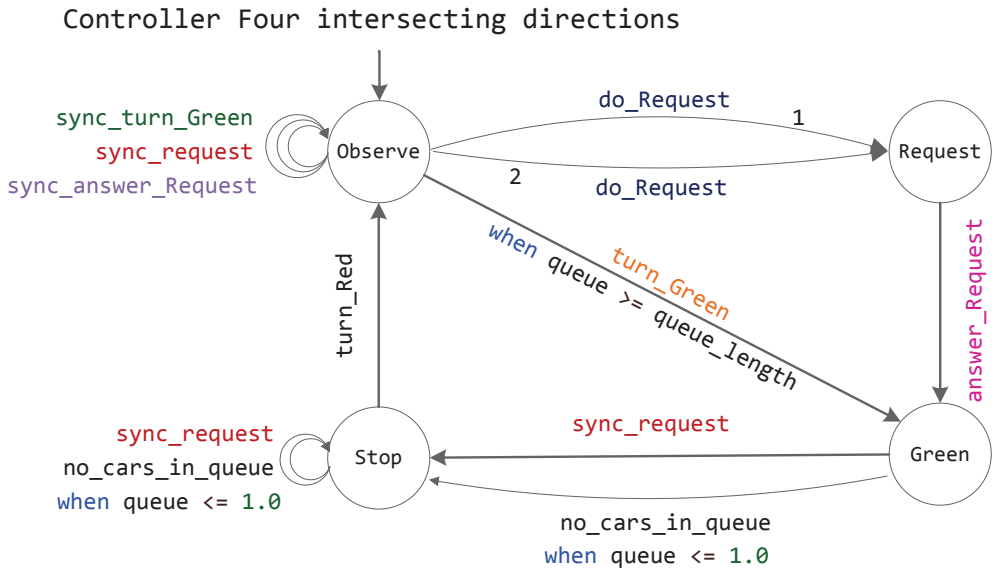
Scaling the models, for the traffic intersection example, of two communicating controllers to more communicating controllers, several adaptations were needed. This because with respect to fairness a traffic intersection of two roads is more simple than a traffic intersection with three roads or more.

Next to these adaptations, it turns out that because in Acumen no functions of the language itself can be used, and therefore a principle implemented by the modeller is needed, the model needs to be adapted every time the problem is scaled. In CIF, the use of synchronizing events does not have to be adapted when scaling. The only adaptations are the increasing number of parameter definitions, which are also increasing in the Acumen model.

Thus, including defined functions in the language itself can help to make scaling of model components easier and less sensitive for mistakes. Although, the modeller preferences also affects scalability results.








```
1 when queue >= queue_length,
       minimumprio(prio, [prio1, prio2], [cond1, cond2]) > 0
  do prio := prio + 1

2 when queue >= queue_length,
       minimumprio(prio, [prio1, prio2], [cond1, cond2]) = 0
```

Figure 8.4: Representation by hybrid automata of the CIF model for the traffic intersection

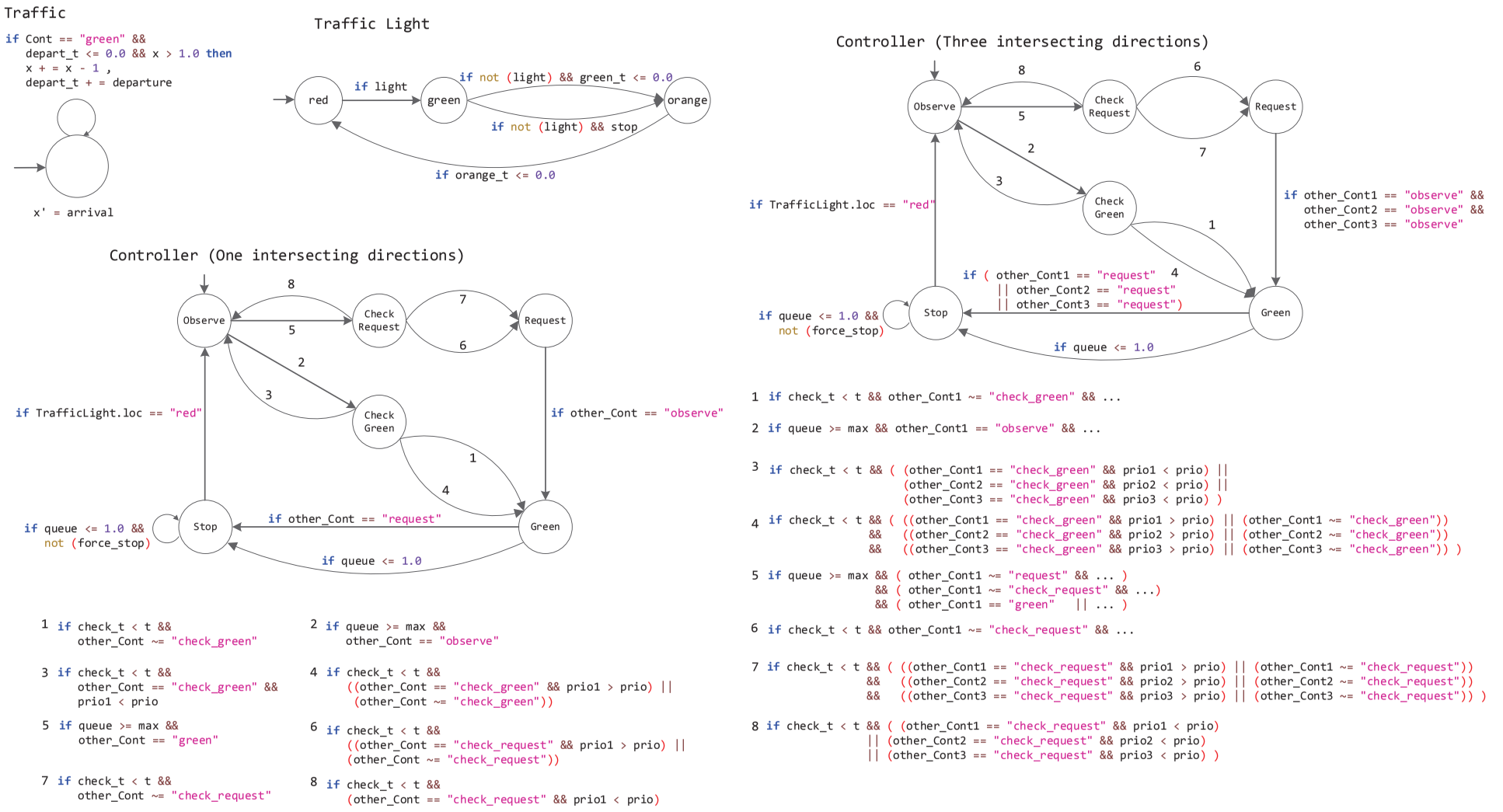


Figure 8.5: Representation by hybrid automata of the Acumen model for the traffic intersection

Chapter 9

Conclusions and Further Research

9.1 Conclusions

This research treated a partial comparison between Acumen and CIF as tools for modeling and analysis of Cyber-Physical Systems. A few characteristic properties of Cyber-Physical Systems were investigated, namely, hybrid behaviour, zeno-behaviour, dynamical creation and termination of objects and communication.

While modelling and simulating the systems, each containing one of the typical properties, differences and similarities between Acumen and CIF were recognized. Most of the differences could be traced back to the design principles and the underlying ideas of the languages.

The emphasis of this comparison was on the structure of the models, the working principle of the simulators, the use of different types of data and the results of the simulation of models. In the following sections the most significant observations concerning the similarities and the differences between Acumen and CIF recognized during this project are stated.

Structure of models

The models in Acumen consist of objects created based on a template consisting of an `initially` section and an `always` section. In what way the behaviour of a certain object is described in the always sections is left to the modeller, and is not bounded by any restrictions with respect to the structure of a model. Therefore, a modeller has a lot of possibilities to implemented certain behaviour or principles into the model. The creation of objects based on templates influences the hierarchy in the model as well. This causes restrictions on the modelling possibilities, so it needs to be taken into account when the model is built.

In CIF, a model consist of automata and/or groups, where groups are commonly used as collections of data. Automata are the more important objects in a model in combination with the locations and edges they contain. Based on these three types of components, the structure of a CIF model is predefined. It is necessary for a modeller to define the system in terms of automata, locations and edges. Besides, in CIF it is much easier to use variables at different places of the model because there is a global read, local write principle.

Working principle of the simulator

With respect to the working principle of the simulator of Acumen and CIF, there are typical differences. In Acumen active assignments, discrete or continuous, are collected and executed simultaneously. Where in CIF, enabled events are listed and executed one at a time based on a certain criteria. It is important to keep the semantics of the language in mind when you model a system with a certain language.

Next to this, the simulator of Acumen works with a constant time-step while the CIF simulator uses variable time-step and a root-finding algorithm to determine the exact point where events are enabled. At this point CIF is more precise than Acumen, because it tries to approximate the behaviour of the system as much as possible. But there is a chance this precision can cause problems (zeno-behaviour) in the models, where the less precise but still accurate principle of Acumen does not cause problems.

Another interesting difference with respect to the working principle of the simulators is the fact that Acumen's traditional semantics is deterministic where CIF features non-deterministic behaviour. Using Acumen's enclosure semantics makes it possible to analyse non-deterministic systems as well.

Types of data

Acumen and CIF support three categories of variables; continuous variables, discrete variables and algebraic variables. A remarkable difference between the languages with respect to data categories and data types (real, integer, string etc.) is that CIF requires initialization of each variable by category and type and this is fixed for the complete model. In Acumen, the program recognizes the category and type of each variables individually based on the initialization in the initially section, and these are not fixed. This greater freedom in using variables can be useful, but can also cause mistakes and a decrease in the general readability of the system.

Results of the simulation

Graphical results, numerical results and visualisation, all three are possible to obtain in Acumen and CIF. The numerical results in both languages contain the values of each variable at every time-step given in a table. The graphical results are also similar and can be used to get a better overview of what happens with the value of certain variables during simulation. The visualisation of both tools is more interesting. Acumen provides these results, where CIF makes use of .svg files.

In Acumen the options to visualise your model are much more limited compared to CIF. Using the mathematical structures a lot of shapes can be represented but it is a devious way of defining the visualisation. The use of a .svg file in CIF is more straightforward to use because you have as much freedom as writing with a pencil on a piece of paper by creating the shapes of the model.

General Comparison

In general, based on the experience gained during this project the Acumen languages approaches more the physical dynamics of Cyber-Physical Systems, where CIF is better suited for the software and networking dynamics of these kind of systems. This certainly does not mean CIF cannot

cover physical dynamics or Acumen cannot cover software and networking dynamics. This general observation is based on the fact that Acumen has much more freedom in defining the behaviour of objects and does not contain much predefined structures. This gives the modeller a lot of freedom to describe the system the way he or she likes it. While in CIF each model need to have the structure of automata, locations and edges and hence, the modeller is forced to divide a system into these components. Besides that, CIF contains a lot of predefined structures which can be used easily based on their principles. Therefore, CIF gives a more abstract interpretation of models where Acumen seems to be more free.

9.2 Further Research

Based on the results of this research and its restriction of considering only a few properties of Cyber-Physical Systems it can be interested to investigate other properties of these kind of systems as well. For instance, asynchronous communication can be a useful property to compare because that eliminates the synchronizing events in some case and also appears in many practical cases. Besides, it could be interesting to investigate other analysis techniques than simulation.

Another interesting option is to use this project to compare other languages with respect to each other. Especially, if languages are under development they can be compared with languages designed for an equivalent purpose. This can help to come up with functions or applications that are not yet included in the developed language. It does not mean these need to be copied, but it can help to explore the application on demand in a certain research area.

Bibliography

[1] H. Beohar, D.E. Nadales Agut, D.A. van Beek, and P.J.L. Cuijpers. Hierarchical states in the compositional interchange format. Technical report, Eindhoven University of Technology, Department of Mechanical Engineering, 2010. Seventh Research Framework Programme of the European Commission, INFSO-ICT-224249.

[2] P.J.L. Cuijpers, M.A. Reniers, and A.G. Engels. Beyond zeno-behaviour. Technical report, Eindhoven University of Technology, Department of Computer Science, 2001. Computer Science Reports 0104.

[3] Department of Mechanical Engineering Systems Engineering, Group. *Compositional Interchange Format 3.0*, 2014. Eindhoven University of Technology (TU/e), `http://cif.se.wtb.tue.nl/`, viewed at: *2014-10-25*.

[4] Walid Taha. *Acumen 2014 Reference Manual*. Halmstad University, Department of Computer Science, 2014. `http://bit.ly/Acumen-manual-2014`, viewed at: *2014-11-05*.

[5] J.W.C.M. van der Vleuten. Networks of signalized traffic intersections, optimization and implementation. Master's thesis, Eindhoven University of Technology, Manufacturing Networks Group, 2014.

[6] J.J. Vereijken. Fischer's protocol in timed process algebra. Technical report, Eindhoven University of Technology, Department of Computing Science, 1995. in: A. Ponse, C. Verhoef, S.F.M. van Vlijmen (Eds.), Algebra of Communicating Processes.

Appendix A

Model Bouncing Ball Acumen

```
model BB() =
initially
  t    = 0 , t' = 1 ,
  R    =  0.5 ,
  y    = 20 , y' = 0 , y'' = 0 ,
  _3D  =  ()
always
  claim y >= 0 ,
  _3D = (Box center=(0,0,0) size=(20.0,2.0,0.05) color=yellow rotation=(0,0,0),
         Sphere center=(t,0,y + 0.275) size=0.25 color=blue rotation=(0,0,0)) ,
  t' = 1 ,

  if y<=0 && y'<0 then
    y' + = - R * y'
  else
    y''=-9.8

model Main (simulator) =
initially
  a  =  create BB()
always
```

Appendix B

Model Bouncing Ball CIF

```
svgfile "bouncing_ball.svg";

const real g     = 9.8;          // gravity
const real m     = 1.0;          // mass
const real MAX_Y = 20.0;         // maximum/initial vertical position of ball
const real V_X   = 10.0;         // constant horizontal velocity
const real R     = 0.5;          // speed reduction on bounce
const real E     = 0.0001;       // epsilon distance (stop height)

automaton bouncing_ball:
  cont x = 0.0,
       y = MAX_Y,
       v = 0.0;

  equation y' = v;
  equation x' = V_X;

  location fall:
    initial;
    equation v' = -m * g;
    edge when y <= 0 do v := -R * v goto rise;

  location rise:
    equation v' = -m * g;
    edge when v <= 0, y >= E goto fall;
    edge when v <= 0, y <  E goto done;

  location done:
    equation v' = 0.0;
end

svgout id "time" text value fmt("Time: %.1f", time);

svgout id "ball_grp" attr "transform"
value fmt("translate(%s,%s)",
          bouncing_ball.x,
          scale(bouncing_ball.y, 0, MAX_Y, 0, -200));
```

Appendix C

Model Bifurcating Ball Acumen

```
class Ball(mode, x,x',x'', y,y',y'', m)
private
_3D := [];
end
 _3D = [["Sphere",[x,0,y],0.05,blue,[0,0,0]]];

  switch mode
    case "Fly"
      if y < 0  && y' < 0
        if m > 0.2
          create Ball("Bounce", x, x'+0.2, 0, y, y', 0, m/2);

          create Ball("Bounce", x, x'-0.2, 0, y, y', 0, m/2);

          terminate self
        else
          mode := "Bounce"
        end
      else
        y'' = -9.8;
        x'' = 0
      end
    case "Bounce"
      y := 0;
      x' :=  0.8 * x';
      y' := -0.8 * y';
      mode := "Fly"
  end
end

class Main(simulator)
  private
    mode := "Init";
    _3D := [];

  end
        simulator.endTime = 20.0;

  switch mode
    case "Init"

      create Ball("Fly", 0,0,0, 10.0,0,1.0, 10.0);

      mode := "Persist"
    case "Persist"

  end
end
```

Appendix D

Model Bifurcating Ball CIF

```
svgfile "bouncing_ball_split.svg";

const real R     = 0.8;
const real g     = 9.8;
const real m     = 1.0;

automaton def First_Ball (alg real start_x, x_, start_y, y_, mass):
    cont x      = start_x,
         y      = start_y,
         der_x  = x_,
         der_y  = y_;

    equation x'     = der_x;
    equation der_x' = 0;

    disc real m = mass;

    event bounce;

    location Start:
    initial;
        equation y'     = der_y;
        equation der_y' = -9.8;

        edge bounce     when y < 0, y' < 0                      goto Terminate;

    location Terminate:
        equation y'     = 0.0;
        equation der_y' = 0.0;

        svgout id "ball_grp" attr "transform" value fmt("translate(%s,%s)",x,scale(y,0,20.0,0,-200));

        svgout id "ball_grp" attr "visibility" value if Terminate: "hidden" else "visible" end;
end

automaton def Balls (alg int nr; event bounce; alg real prev_x, x_, y_, prev_m):
    cont x      = -10.0,
         y      = -10.0,
         der_x  = 0.0,
         der_y  = 0.0;

    equation der_x' = 0.0;

    event stop, finish;

    disc real m;

    location Stop:
        initial;
        equation x'     = 0.0;
        equation y'     = 0.0;
        equation der_y' = 0.0;

        edge bounce     do x := prev_x, y := 0.0, der_x := R * x_, der_y := -R * y_, m := prev_m / 2   goto Fly;

    location Fly:
        equation x'     = der_x;
        equation y'     = der_y;
        equation der_y' = -9.8;

        edge stop       when y < 0, y' < 0, m >= 0.2              goto Stop;
        edge finish     when y < 0, y' < 0, m <  0.2              goto Stop;

        svgcopy id "ball_grp" post <string>nr;

        svgout id "ball_grp"+<string>nr attr "transform" value fmt("translate(%s,%s)",x,scale(y,0,20.0,0, -200));

        svgout id "ball_grp" + <string>nr attr "visibility" value if Stop: "hidden" else "visible" end;

end
```

```
b1 : First_Ball(300.0, 0.0,10.0,1.0,10.0);

b2 : Balls(1, b1.bounce, b1.x, b1.x'+10.2, b1.y', b1.m);
b3 : Balls(2, b1.bounce, b1.x, b1.x'-10.2, b1.y', b1.m);

b21 : Balls(3, b2.stop, b2.x, b2.x'+10.2, b2.y', b2.m);
b22 : Balls(4, b2.stop, b2.x, b2.x'-10.2, b2.y', b2.m);
b31 : Balls(5, b3.stop, b3.x, b3.x'+10.2, b3.y', b3.m);
b32 : Balls(6, b3.stop, b3.x, b3.x'-10.2, b3.y', b3.m);

b211 : Balls(7, b21.stop, b21.x, b21.x'+10.2, b21.y', b21.m);
b212 : Balls(8, b21.stop, b21.x, b21.x'-10.2, b21.y', b21.m);
b221 : Balls(9, b22.stop, b22.x, b22.x'+10.2, b22.y', b22.m);
b222 : Balls(10, b22.stop, b22.x, b22.x'-10.2, b22.y', b22.m);
b311 : Balls(11, b31.stop, b31.x, b31.x'+10.2, b31.y', b31.m);
b312 : Balls(12, b31.stop, b31.x, b31.x'-10.2, b31.y', b31.m);
b321 : Balls(13, b32.stop, b32.x, b32.x'+10.2, b32.y', b32.m);
b322 : Balls(14, b32.stop, b32.x, b32.x'-10.2, b32.y', b32.m);

b2111 : Balls(15, b211.stop, b211.x, b211.x'+10.2, b211.y', b211.m);
b2121 : Balls(16, b212.stop, b212.x, b212.x'+10.2, b212.y', b212.m);
b2211 : Balls(17, b221.stop, b221.x, b221.x'+10.2, b221.y', b221.m);
b2221 : Balls(18, b222.stop, b222.x, b222.x'+10.2, b222.y', b222.m);
b3111 : Balls(19, b311.stop, b311.x, b311.x'+10.2, b311.y', b311.m);
b3121 : Balls(20, b312.stop, b312.x, b312.x'+10.2, b312.y', b312.m);
b3211 : Balls(21, b321.stop, b321.x, b321.x'+10.2, b321.y', b321.m);
b3221 : Balls(22, b322.stop, b322.x, b322.x'+10.2, b322.y', b322.m);
b2112 : Balls(23, b211.stop, b211.x, b211.x'-10.2, b211.y', b211.m);
b2122 : Balls(24, b212.stop, b212.x, b212.x'-10.2, b212.y', b212.m);
b2212 : Balls(25, b221.stop, b221.x, b221.x'-10.2, b221.y', b221.m);
b2222 : Balls(26, b222.stop, b222.x, b222.x'-10.2, b222.y', b222.m);
b3112 : Balls(27, b311.stop, b311.x, b311.x'-10.2, b311.y', b311.m);
b3122 : Balls(28, b312.stop, b312.x, b312.x'-10.2, b312.y', b312.m);
b3212 : Balls(29, b321.stop, b321.x, b321.x'-10.2, b321.y', b321.m);
b3222 : Balls(30, b322.stop, b322.x, b322.x'-10.2, b322.y', b322.m);

b21111 : Balls(31, b2111.stop, b2111.x, b2111.x'+10.2, b2111.y', b2111.m);
b21211 : Balls(32, b2121.stop, b2121.x, b2121.x'+10.2, b2121.y', b2121.m);
b22111 : Balls(33, b2211.stop, b2211.x, b2211.x'+10.2, b2211.y', b2211.m);
b22211 : Balls(34, b2221.stop, b2221.x, b2221.x'+10.2, b2221.y', b2221.m);
b31111 : Balls(35, b3111.stop, b3111.x, b3111.x'+10.2, b3111.y', b3111.m);
b31211 : Balls(36, b3121.stop, b3121.x, b3121.x'+10.2, b3121.y', b3121.m);
b32111 : Balls(37, b3211.stop, b3211.x, b3211.x'+10.2, b3211.y', b3211.m);
b32211 : Balls(38, b3221.stop, b3221.x, b3221.x'+10.2, b3221.y', b3221.m);
b21121 : Balls(39, b2112.stop, b2112.x, b2112.x'+10.2, b2112.y', b2112.m);
b21221 : Balls(40, b2122.stop, b2122.x, b2122.x'+10.2, b2122.y', b2122.m);
b22121 : Balls(41, b2212.stop, b2212.x, b2212.x'+10.2, b2212.y', b2212.m);
b22221 : Balls(42, b2222.stop, b2222.x, b2222.x'+10.2, b2222.y', b2222.m);
b31121 : Balls(43, b3112.stop, b3112.x, b3112.x'+10.2, b3112.y', b3112.m);
b31221 : Balls(44, b3122.stop, b3122.x, b3122.x'+10.2, b3122.y', b3122.m);
b32121 : Balls(45, b3212.stop, b3212.x, b3212.x'+10.2, b3212.y', b3212.m);
b32221 : Balls(46, b3222.stop, b3222.x, b3222.x'+10.2, b3222.y', b3222.m);
b21112 : Balls(47, b2111.stop, b2111.x, b2111.x'-10.2, b2111.y', b2111.m);
b21212 : Balls(48, b2121.stop, b2121.x, b2121.x'-10.2, b2121.y', b2121.m);
b22112 : Balls(49, b2211.stop, b2211.x, b2211.x'-10.2, b2211.y', b2211.m);
b22212 : Balls(50, b2221.stop, b2221.x, b2221.x'-10.2, b2221.y', b2221.m);
b31112 : Balls(51, b3111.stop, b3111.x, b3111.x'-10.2, b3111.y', b3111.m);
b31212 : Balls(52, b3121.stop, b3121.x, b3121.x'-10.2, b3121.y', b3121.m);
b32112 : Balls(53, b3211.stop, b3211.x, b3211.x'-10.2, b3211.y', b3211.m);
b32212 : Balls(54, b3221.stop, b3221.x, b3221.x'-10.2, b3221.y', b3221.m);
b21122 : Balls(55, b2112.stop, b2112.x, b2112.x'-10.2, b2112.y', b2112.m);
b21222 : Balls(56, b2122.stop, b2122.x, b2122.x'-10.2, b2122.y', b2122.m);
b22122 : Balls(57, b2212.stop, b2212.x, b2212.x'-10.2, b2212.y', b2212.m);
b22222 : Balls(58, b2222.stop, b2222.x, b2222.x'-10.2, b2222.y', b2222.m);
b31122 : Balls(59, b3112.stop, b3112.x, b3112.x'-10.2, b3112.y', b3112.m);
b31222 : Balls(60, b3122.stop, b3122.x, b3122.x'-10.2, b3122.y', b3122.m);
b32122 : Balls(61, b3212.stop, b3212.x, b3212.x'-10.2, b3212.y', b3212.m);
b32222 : Balls(62, b3222.stop, b3222.x, b3222.x'-10.2, b3222.y', b3222.m);
```

```
b211111 : Balls(63, b21111.stop, b21111.x, b21111.x'+10.2, b21111.y', b21111.m);
b212111 : Balls(64, b21211.stop, b21211.x, b21211.x'+10.2, b21211.y', b21211.m);
b221111 : Balls(65, b22111.stop, b22111.x, b22111.x'+10.2, b22111.y', b22111.m);
b222111 : Balls(66, b22211.stop, b22211.x, b22211.x'+10.2, b22211.y', b22211.m);
b311111 : Balls(67, b31111.stop, b31111.x, b31111.x'+10.2, b31111.y', b31111.m);
b312111 : Balls(68, b31211.stop, b31211.x, b31211.x'+10.2, b31211.y', b31211.m);
b321111 : Balls(69, b32111.stop, b32111.x, b32111.x'+10.2, b32111.y', b32111.m);
b322111 : Balls(70, b32211.stop, b32211.x, b32211.x'+10.2, b32211.y', b32211.m);
b211211 : Balls(71, b21121.stop, b21121.x, b21121.x'+10.2, b21121.y', b21121.m);
b212211 : Balls(72, b21221.stop, b21221.x, b21221.x'+10.2, b21221.y', b21221.m);
b221211 : Balls(73, b22121.stop, b22121.x, b22121.x'+10.2, b22121.y', b22121.m);
b222211 : Balls(74, b22221.stop, b22221.x, b22221.x'+10.2, b22221.y', b22221.m);
b311211 : Balls(75, b31121.stop, b31121.x, b31121.x'+10.2, b31121.y', b31121.m);
b312211 : Balls(76, b31221.stop, b31221.x, b31221.x'+10.2, b31221.y', b31221.m);
b321211 : Balls(77, b32121.stop, b32121.x, b32121.x'+10.2, b32121.y', b32121.m);
b322211 : Balls(78, b32221.stop, b32221.x, b32221.x'+10.2, b32221.y', b32221.m);
b211121 : Balls(79, b21112.stop, b21112.x, b21112.x'+10.2, b21112.y', b21112.m);
b212121 : Balls(80, b21212.stop, b21212.x, b21212.x'+10.2, b21212.y', b21212.m);
b221121 : Balls(81, b22112.stop, b22112.x, b22112.x'+10.2, b22112.y', b22112.m);
b222121 : Balls(82, b22212.stop, b22212.x, b22212.x'+10.2, b22212.y', b22212.m);
b311121 : Balls(83, b31112.stop, b31112.x, b31112.x'+10.2, b31112.y', b31112.m);
b312121 : Balls(84, b31212.stop, b31212.x, b31212.x'+10.2, b31212.y', b31212.m);
b321121 : Balls(85, b32112.stop, b32112.x, b32112.x'+10.2, b32112.y', b32112.m);
b322121 : Balls(86, b32212.stop, b32212.x, b32212.x'+10.2, b32212.y', b32212.m);
b211221 : Balls(87, b21122.stop, b21122.x, b21122.x'+10.2, b21122.y', b21122.m);
b212221 : Balls(88, b21222.stop, b21222.x, b21222.x'+10.2, b21222.y', b21222.m);
b221221 : Balls(89, b22122.stop, b22122.x, b22122.x'+10.2, b22122.y', b22122.m);
b222221 : Balls(90, b22222.stop, b22222.x, b22222.x'+10.2, b22222.y', b22222.m);
b311221 : Balls(91, b31122.stop, b31122.x, b31122.x'+10.2, b31122.y', b31122.m);
b312221 : Balls(92, b31222.stop, b31222.x, b31222.x'+10.2, b31222.y', b31222.m);
b321221 : Balls(93, b32122.stop, b32122.x, b32122.x'+10.2, b32122.y', b32122.m);
b322221 : Balls(94, b32222.stop, b32222.x, b32222.x'+10.2, b32222.y', b32222.m);
b211112 : Balls(95, b21111.stop, b21111.x, b21111.x'-10.2, b21111.y', b21111.m);
b212112 : Balls(96, b21211.stop, b21211.x, b21211.x'-10.2, b21211.y', b21211.m);
b221112 : Balls(97, b22111.stop, b22111.x, b22111.x'-10.2, b22111.y', b22111.m);
b222112 : Balls(98, b22211.stop, b22211.x, b22211.x'-10.2, b22211.y', b22211.m);
b311112 : Balls(99, b31111.stop, b31111.x, b31111.x'-10.2, b31111.y', b31111.m);
b312112 : Balls(100, b31211.stop, b31211.x, b31211.x'-10.2, b31211.y', b31211.m);
b321112 : Balls(101, b32111.stop, b32111.x, b32111.x'-10.2, b32111.y', b32111.m);
b322112 : Balls(102, b32211.stop, b32211.x, b32211.x'-10.2, b32211.y', b32211.m);
b211212 : Balls(103, b21121.stop, b21121.x, b21121.x'-10.2, b21121.y', b21121.m);
b212212 : Balls(104, b21221.stop, b21221.x, b21221.x'-10.2, b21221.y', b21221.m);
b221212 : Balls(105, b22121.stop, b22121.x, b22121.x'-10.2, b22121.y', b22121.m);
b222212 : Balls(106, b22221.stop, b22221.x, b22221.x'-10.2, b22221.y', b22221.m);
b311212 : Balls(107, b31121.stop, b31121.x, b31121.x'-10.2, b31121.y', b31121.m);
b312212 : Balls(108, b31221.stop, b31221.x, b31221.x'-10.2, b31221.y', b31221.m);
b321212 : Balls(109, b32121.stop, b32121.x, b32121.x'-10.2, b32121.y', b32121.m);
b322212 : Balls(110, b32221.stop, b32221.x, b32221.x'-10.2, b32221.y', b32221.m);
b211122 : Balls(111, b21112.stop, b21112.x, b21112.x'-10.2, b21112.y', b21112.m);
b212122 : Balls(112, b21212.stop, b21212.x, b21212.x'-10.2, b21212.y', b21212.m);
b221122 : Balls(113, b22112.stop, b22112.x, b22112.x'-10.2, b22112.y', b22112.m);
b222122 : Balls(114, b22212.stop, b22212.x, b22212.x'-10.2, b22212.y', b22212.m);
b311122 : Balls(115, b31112.stop, b31112.x, b31112.x'-10.2, b31112.y', b31112.m);
b312122 : Balls(116, b31212.stop, b31212.x, b31212.x'-10.2, b31212.y', b31212.m);
b321122 : Balls(117, b32112.stop, b32112.x, b32112.x'-10.2, b32112.y', b32112.m);
b322122 : Balls(118, b32212.stop, b32212.x, b32212.x'-10.2, b32212.y', b32212.m);
b211222 : Balls(119, b21122.stop, b21122.x, b21122.x'-10.2, b21122.y', b21122.m);
b212222 : Balls(120, b21222.stop, b21222.x, b21222.x'-10.2, b21222.y', b21222.m);
b221222 : Balls(121, b22122.stop, b22122.x, b22122.x'-10.2, b22122.y', b22122.m);
b222222 : Balls(122, b22222.stop, b22222.x, b22222.x'-10.2, b22222.y', b22222.m);
b311222 : Balls(123, b31122.stop, b31122.x, b31122.x'-10.2, b31122.y', b31122.m);
b312222 : Balls(124, b31222.stop, b31222.x, b31222.x'-10.2, b31222.y', b31222.m);
b321222 : Balls(125, b32122.stop, b32122.x, b32122.x'-10.2, b32122.y', b32122.m);
b322222 : Balls(126, b32222.stop, b32222.x, b32222.x'-10.2, b32222.y', b32222.m);


svgout id "time" text value fmt("Time: %.1f", time);
```

Appendix E

Model Traffic Intersection Acumen

```
model Light(light, stop, xpos_light, ypos_light) =
initially
  loc  =  "red" ,
  a  =  red ,
  b  =  white ,
  c  =  white ,
  _3D  =  () ,

  yellow_t  =  0.0 , yellow_t'  =  0.0 , yellow_time  =  2.0 ,
  green_t  =  0.0 , green_t'  =  0.0 , green_time  =  5.0
always
  yellow_t' = -1.0 ,
  green_t'  = -1.0 ,

  _3D + = (Cylinder center=(xpos_light,0,ypos_light+1) radius=  0.45 length=0.1  color=a rotation=(0,0,0),
           Cylinder center=(xpos_light,0,ypos_light)   radius=  0.45 length=0.1  color=b rotation=(0,0,0),
           Cylinder center=(xpos_light,0,ypos_light-1) radius=  0.45 length=0.1  color=c rotation=(0,0,0),
            Cylinder center=(xpos_light,0.01,ypos_light-1.5) radius=  0.65 length=0.1  color=(0,0,0)
            rotation=(0,0,0),
           Cylinder center=(xpos_light,0.01,ypos_light+1.5)  radius=  0.65 length=0.1  color=(0,0,0)
           rotation=(0,0,0),
           Box      center=(xpos_light,0.01,ypos_light) size=(1.3,0.1,3.0) color=(0,0,0) rotation=(0,0,0)
           ),

  match loc with [
    "red" ->
      a + = red ,
      b + = white ,
      c + = white ,
                      if light then
                         green_t + = green_time ,
                         loc + = "green"
                      noelse
  | "yellow" ->
      a + = white ,
      b + = yellow ,
      c + = white ,
                      if yellow_t <= 0.0 then
                         loc + = "red"
                      noelse
  | "green" ->
      a + = white ,
      b + = white ,
      c + = green ,
                      if not (light) && green_t <= 0.0 then
                         yellow_t + = yellow_time ,
                         loc + = "yellow"

                      else if not (light) && stop then
                              yellow_t + = yellow_time ,
                              loc + = "yellow"
                      noelse
                  ]

model Traffic(arrival, departure, Cont, xpos_queue, ypos_queue) =
initially
  x  =  0.0 , x'  =  0.0 ,  depart_t  =  departure , depart_t'  =  0.0 , x  =  0 ,
  _3D  =  ()
always
  x' = arrival ,
  depart_t' = -1.0 ,

  _3D + = (Text     center=( xpos_queue,0,ypos_queue)      size= 1 content="queue =",
           Text     center=( xpos_queue+4,0,ypos_queue)    size= 1 content= x
           ),

      if Cont == "green" && depart_t <= 0.0 && x > 1.0 then
              x + = x - 1 ,
              depart_t + = departure
```

```
          noelse

model Controller(max, other_Cont, arrival, departure, prio, xpos_light, ypos_light, xpos_queue, ypos_queue) =
initially
  t  =  0.0 , t'  =  0.0 ,
  loc    =  "observe" ,
  queue  =  0 ,
  TrafficLight    =  create Light  (false, false, xpos_light, ypos_light) ,
  Traffic_queue   =  create Traffic(arrival, departure, "red", xpos_queue, ypos_queue) ,
  light_on = false ,
  force_stop = false ,
  check_t  =  0.0
always
  t' = 1.0 ,
  TrafficLight.light   = light_on ,
  TrafficLight.stop    = force_stop ,
  Traffic_queue.Cont   = TrafficLight.loc ,
  queue = Traffic_queue.x ,

  match loc with [
    "observe" ->
                  if queue >= max && other_Cont == "green" then
                     loc + = "request"

                  else if queue >= max && other_Cont == "observe" then
                          check_t + = t ,
                          loc + = "check"
                  noelse
  | "check" ->
                  if check_t < t && other_Cont == "observe" then
                     check_t + = 0.0 ,
                     light_on + = true ,
                     loc + = "green"

                  else if check_t < t && other_Cont == "check" && prio == 1.0 then
                          check_t + = 0.0 ,
                          light_on + = true ,
                          loc + = "green"

                  else if check_t < t && other_Cont == "check" && prio == 0.0 then
                          check_t + = 0.0 ,
                          loc + = "observe"
                  noelse
  | "request" ->
                     if other_Cont == "observe" then
                        light_on + = true ,
                        loc + = "green"
                    noelse
  | "green" ->
                           if other_Cont == "request" then
                             light_on + = false ,
                             loc + = "stop"

                           else if queue <= 1.0 then
                                   light_on + = false ,
                                   force_stop + = true ,
                                   loc + = "stop"
                            noelse
  | "stop" ->
                           if TrafficLight.loc == "red" then
                              loc + = "observe" ,
                              force_stop + = false

                           else if queue <= 1.0 && not (force_stop) then
                                   force_stop + = true
                               noelse
              ]
```

```
model Main(simulator) =
initially
  t  =  0.0 , t'  =  0.0 ,
  Controller_North  =  create Controller(5,"observe", 0.5, 0.1, 1.0,  0.0, 3.0,  -2.0, 6.0) ,
  Controller_West   =  create Controller(20,"observe", 0.5, 0.1, 0.0, -6.0, 0.0,  -8.0, 3.0) ,
  _3DView = () ,
  _3D = ()
always
  t' = 1.0 ,
   _3DView = ((-3,-50,0), (-3,0,0)),
  simulator.endTime  = 100.0 ,
  Controller_North.other_Cont = Controller_West.loc ,
  Controller_West.other_Cont = Controller_North.loc
```

Appendix F

Model Traffic Intersection CIF

```
svgfile "TrafficLight_crossing.svg";
import "minimum_prio.cif";

const real yellow_time      = 2.0;
const real green_time       = 5.0;

alg int min_queue_north = 1;
alg int min_queue_west = 1;

alg bool cond_north = traffic_north.x >= min_queue_north;
alg bool cond_west  = traffic_west.x >= min_queue_west;

automaton def Light(alg string dir; event turn_Green1, turn_Green2, no_cars):
    cont yellow_t der -1.0;
    cont green_t  der -1.0;

    event turn_Red, turn_Yellow;

    location Red:
        initial;
        edge turn_Green1                          do green_t := green_time    goto Green;
        edge turn_Green2                          do green_t := green_time    goto Green;

    location Green:
        edge turn_Yellow     when green_t <= 0.0   do yellow_t := yellow_time  goto Yellow;
        edge no_cars                              do yellow_t := yellow_time  goto Yellow;

    location Yellow:
        edge turn_Red        when yellow_t <= 0.0                            goto Red;

    svgout id "red_" + <string>dir      attr "fill" value if Red:    "red"    else "white" end;
    svgout id "green_" + <string>dir    attr "fill" value if Green:  "green"  else "white" end;
    svgout id "orange_" + <string>dir   attr "fill" value if Yellow: "orange" else "white" end;

    svgout id "orange_time_" + <string>dir  text value fmt("%.2f",  max(0,yellow_t));

    svgout id "green_time_" + <string>dir   text value fmt("%.2f",  max(0,green_t));

end



automaton def Traffic(alg string dir; alg bool Green; alg real arrival, departure):
    cont x = 0   der arrival;
    cont depart_t = departure der -1.0;

    event depart;

    location Traffic:
        initial;

        edge depart   when depart_t < 0.0, Green, x >= 1.0    do x := x - 1.0, depart_t := departure;

   svgout id "queue_" + <string>dir  text value fmt("queue: %.1f", x);

end
```

```
automaton def Controller ( alg real queue; alg int queue_length;
                           event sync_turn_Green, sync_answer_Request, do_Request, sync_request,
                                 turn_Green, answer_Request, turn_Red):
    event no_cars_in_queue;

    location Observe:
        initial;
        edge sync_turn_Green;
        edge sync_answer_Request;

        edge do_Request                when queue >= queue_length          goto Request;

        edge turn_Green                when queue >= queue_length          goto Green;

    location Request:
        edge answer_Request                                                goto Green;

    location Green:
        edge sync_request                                                  goto Stop;
        edge no_cars_in_queue          when queue <= 1.0                   goto Stop;

    location Stop:
        edge sync_request;
        edge turn_Red                                                      goto Observe;
        edge no_cars_in_queue          when queue <= 1.0;
end

event   turn_Green_North, answer_Request_North, Request_North,
        turn_Green_West,  answer_Request_West,  Request_West;

controller_North: Controller(traffic_north.x, min_queue_north, turn_Green_West,  answer_Request_West,
                  Request_North, Request_West,  turn_Green_North, answer_Request_North,
                  light_north.turn_Red);

controller_West:  Controller(traffic_west.x,  min_queue_west,  turn_Green_North, answer_Request_North,
                  Request_West,  Request_North, turn_Green_West,  answer_Request_West,
                  light_west.turn_Red);

light_north: Light("north", turn_Green_North, answer_Request_North, controller_North.no_cars_in_queue);

light_west : Light("west" , turn_Green_West, answer_Request_West, controller_West.no_cars_in_queue);

traffic_north: Traffic("north", light_north.Green,  1, 0.5);
traffic_west : Traffic("west" , light_west.Green,   1, 0.5);

svgout id "time"               text value fmt("Time: %.2f", time);
```

Appendix G

Model Zeno-Behaviour Ball Acumen

```
model Ball()=
initially
  t = 0, t' = 1, R = 0.5, y = 20, y' = 0, y'' = 0,
  _3D = ()
always
_3D = (Box center=(10.0,0,0) size=(20.0,2.0,0.05) color=yellow rotation=(0,0,0),
       Sphere center=(t,0,y + 0.275) size= 0.25 color=blue rotation=(0,0,0)
       ),

  t' = 1,

  if t >= 10.0 && t <= 10.01 then
     y'+ = 20.0,
     y'' = -9.8
  else
       if y<=0 && y'<0 then
             y'+ = - R * y'
       else
             y'' = -9.8

model Main (simulator)=
initially
  a = create Ball()
always
```

Appendix H

Model Zeno-Behaviour Ball CIF

```
svgfile "bouncing_ball.svg";

const real MAX_Y = 20.0;
const real V_X   = 10.0;
const real R     = 0.5;
const real E     = 0.0001;
const real g     = 9.8;
const real m     = 1.0;

automaton Ball:
  cont x = 0.0,
       y = MAX_Y,
       v = 0.0;

  equation y' = v;
  equation x' = V_X;
  equation v' = - m * g;

  disc real top = MAX_Y;
  disc bool bounce = false;

  event rise, fall;

  location ball:
    initial;
    edge rise when y < 0, not bounce    do v := -R * v, bounce := true;
    edge fall when v < 0, bounce        do top := y,    bounce := false;
    edge      when x >= 100.0           do v := 20.0;

end

svgout id "time" text value fmt("Time: %.1f", time);

svgout id "ball_grp" attr "transform"
value fmt("translate(%s,%s)", Ball.x, scale(Ball.y, 0, MAX_Y, 0, -200));
```

Appendix I

Model Complex Traffic Intersection Acumen

```
model Light(light, stop, xpos_light, ypos_light) =
initially
  loc  =  "red" ,
  a  =  red ,
  b  =  white ,
  c  =  white ,
  _3D  =  () ,

  yellow_t  =  0.0 , yellow_t'  =  0.0 , yellow_time  =  2.0 ,
  green_t  =  0.0 , green_t'  =  0.0 , green_time  =  5.0
always
  yellow_t' = -1.0 ,
  green_t'  = -1.0 ,

  _3D + = (Cylinder center=(xpos_light,0,ypos_light+1)      radius=  0.45 length=0.1  color=a   ,
           Cylinder center=(xpos_light,0,ypos_light)        radius=  0.45 length=0.1  color=b   ,
           Cylinder center=(xpos_light,0,ypos_light-1)      radius=  0.45 length=0.1  color=c   ,
           Cylinder center=(xpos_light,0.01,ypos_light-1.5) radius=  0.65 length=0.1  ,
           Cylinder center=(xpos_light,0.01,ypos_light+1.5) radius=  0.65 length=0.1  ,
           Box      center=(xpos_light,0.01,ypos_light)     size=(1.3,0.1,3.0)
           ),

  match loc with [
    "red" ->
      a + = red ,
      b + = white ,
      c + = white ,
                    if light then
                       green_t + = green_time ,
                       loc + = "green"
                    noelse
  | "orange" ->
      a + = white ,
      b + = yellow ,
      c + = white ,
                    if yellow_t <= 0.0 then
                       loc + = "red"
                    noelse
  | "green" ->
      a + = white ,
      b + = white ,
      c + = green ,
                    if not (light) && green_t <= 0.0 then
                       yellow_t + = yellow_time ,
                       loc + = "orange"

                    else if not (light) && stop then
                            yellow_t + = yellow_time ,
                            loc + = "orange"
                         noelse
                ]

model Traffic(arrival, departure, Cont, xpos_queue, ypos_queue) =
initially
  x  =  0.0 ,   x'  =  0.0 ,  depart_t  =  departure , depart_t'  =  0.0 ,  x  =  0 ,
  _3D  =  ()
always
  x' = arrival ,
  depart_t' = -1.0 ,

  _3D + = (Text     center=( xpos_queue,0,ypos_queue)      size= 1 content="queue =",
           Text     center=( xpos_queue+4,0,ypos_queue)    size= 1 content= x
           ),

      if Cont == "green" && depart_t <= 0.0 && x > 1.0 then
              x + = x - 1 ,
              depart_t + = departure
      noelse
```

```
model Controller_two_directions(max, prio, other_Cont, arrival, departure, prio1, xpos_light, ypos_light,
xpos_queue, ypos_queue) =
initially
  t  =  0.0 , t'  =  0.0 ,
  loc    =  "observe" ,
  queue  =  0 ,
  TrafficLight    =  create Light  (false, false, xpos_light, ypos_light) ,
  Traffic_queue   =  create Traffic(arrival, departure, "red", xpos_queue, ypos_queue) ,
  light_on = false ,
  check_t  =  0.0 ,
  count_green = 0 ,
  force_stop = false ,
  _3D  =  ()
always
  t' = 1.0 ,
  TrafficLight.light   = light_on ,
  TrafficLight.stop    = force_stop ,
  Traffic_queue.Cont   = TrafficLight.loc ,
  queue = Traffic_queue.x ,

  _3D + = (Text     center=( xpos_light,0,ypos_light-3)    size= 1 content= count_green
           ),

  match loc with [
   "observe" ->
                   if queue >= max && other_Cont == "observe" then
                      check_t + = t ,
                      loc + = "check_green"

                   else if queue >= max && other_Cont ~= "request" then
                           check_t + = t ,
                           loc + = "check_request"
                   noelse

  | "check_green" ->

                   if check_t < t && other_Cont == "check_green" && prio1 < prio then
                           loc + = "observe"

                   else if check_t < t && other_Cont ~= "check_green" then
                           light_on + = true ,
                           count_green + = count_green + 1 ,
                           loc + = "green"

                   else if check_t < t && ((other_Cont == "check_green" && prio1 > prio) ||
                                                                   (other_Cont ~= "check_green")) then
                           light_on + = true ,
                           count_green + = count_green + 1 ,
                           prio + = prio + 6 ,
                           loc + = "green"

                   noelse

  | "check_request" ->

                   if check_t < t && (other_Cont == "check_request" && prio1 < prio) then
                           loc + = "observe"

                   else if check_t < t && other_Cont ~= "check_request" then
                           loc + = "request"

                   else if check_t < t && ((other_Cont == "check_request" && prio1 > prio) ||
                                                                   (other_Cont ~= "check_request")) then
                           prio + = prio + 6 ,
                           loc + = "request"

                   noelse
```

```
 | "request" ->
                      if other_Cont == "observe" then
                         light_on + = true ,
                         count_green + = count_green + 1 ,
                         loc + = "green"
                       noelse
  | "green" ->
                            if other_Cont == "request" then
                              light_on + = false ,
                              loc + = "stop"

                            else if queue <= 1.0 then
                                    light_on + = false ,
                                    force_stop + = true ,
                                    loc + = "stop"
                             noelse
  | "stop" ->
                            if TrafficLight.loc == "red" then
                               loc + = "observe" ,
                               force_stop + = false

                            else if queue <= 1.0 && not (force_stop) then
                                    force_stop + = true
                                 noelse
                ]

model Controller_four_directions(max, prio, other_Cont1, other_Cont2, other_Cont3, arrival, departure,
                                 prio1, prio2, prio3, xpos_light, ypos_light, xpos_queue, ypos_queue) =
initially
  t  =  0.0 , t'  =  0.0 ,
  loc    =  "observe" ,
  queue  =  0 ,
  TrafficLight    =  create Light  (false, false, xpos_light, ypos_light) ,
  Traffic_queue   =  create Traffic(arrival, departure, "red", xpos_queue, ypos_queue) ,
  light_on = false ,
  check_t  =  0.0 ,
  count_green = 0 ,
  force_stop = false ,
  _3D  =  ()
always
  t' = 1.0 ,
  TrafficLight.light   = light_on ,
  TrafficLight.stop    = force_stop ,
  Traffic_queue.Cont   = TrafficLight.loc ,
  queue = Traffic_queue.x ,

  _3D + = (Text     center=( xpos_light,0,ypos_light-3)    size= 1 content= count_green
           ),

  match loc with [
    "observe" ->

if queue >= max && other_Cont1 == "observe" && other_Cont2 == "observe" && other_Cont3 == "observe" then
                     check_t + = t ,
                     loc + = "check_green"

else if queue >= max && ( other_Cont1 ~= "request" && other_Cont2 ~= "request" && other_Cont3 ~= "request")
                     && ( other_Cont1 ~= "check_request" && other_Cont2 ~= "check_request" &&
                                                                          other_Cont3 ~= "check_request")
                   && ( other_Cont1 == "green"   || other_Cont2 == "green"   || other_Cont3 == "green"  ) then
                          check_t + = t ,
                          loc + = "check_request"
noelse
```

```
 | "check_green" ->

if check_t < t && ( (other_Cont1 == "check_green" && prio1 < prio) ||
                    (other_Cont2 == "check_green" && prio2 < prio) ||
                           (other_Cont3 == "check_green" && prio3 < prio) ) then
                         loc + = "observe"

else if check_t < t && other_Cont1 ~= "check_green" && other_Cont2 ~= "check_green" &&
                                                                        other_Cont3 ~= "check_green" then
                         light_on + = true ,
                         count_green + = count_green + 1 ,
                         loc + = "green"

else if check_t < t && ( ((other_Cont1 == "check_green" && prio1 > prio) || (other_Cont1 ~= "check_green"))
                    &&   ((other_Cont2 == "check_green" && prio2 > prio) || (other_Cont2 ~= "check_green"))
                    &&   ((other_Cont3 == "check_green" && prio3 > prio) || (other_Cont3 ~= "check_green")) )
then
                         light_on + = true ,
                         count_green + = count_green + 1 ,
                         prio + = prio + 6 ,
                         loc + = "green"
noelse

  | "check_request" ->

if check_t < t && ( (other_Cont1 == "check_request" && prio1 < prio) ||
                    (other_Cont2 == "check_request" && prio2 < prio) ||
                           (other_Cont3 == "check_request" && prio3 < prio) ) then
                         loc + = "observe"

else if check_t < t && other_Cont1 ~= "check_request" && other_Cont2 ~= "check_request" && other_Cont3 ~=
"check_request" then
                         loc + = "request"

else if check_t < t && ( ((other_Cont1 == "check_request" && prio1 > prio) || (other_Cont1 ~= "check_request"))
                  &&   ((other_Cont2 == "check_request" && prio2 > prio) || (other_Cont2 ~= "check_request"))
                  &&   ((other_Cont3 == "check_request" && prio3 > prio) || (other_Cont3 ~= "check_request")) )
then
                         prio + = prio + 6 ,
                         loc + = "request"
noelse

  | "request" ->
                     if other_Cont1 == "observe" && other_Cont2 == "observe" && other_Cont3 == "observe" then
                        light_on + = true ,
                        count_green + = count_green + 1 ,
                        loc + = "green"
                     noelse
  | "green" ->
                     if ( other_Cont1 == "request" || other_Cont2 == "request" || other_Cont3 == "request")
                     then
                        light_on + = false ,
                        loc + = "stop"

                     else if queue <= 1.0 then
                             light_on + = false ,
                             force_stop + = true,
                             loc + = "stop"
                           noelse
  | "stop" ->
                     if TrafficLight.loc == "red" then
                        loc + = "observe" ,
                        force_stop + = false

                     else if queue <= 1.0 && not (force_stop) then
                             force_stop + = true
                          noelse
                ]
```

```
model Main(simulator) =
initially
  t  =  0.0 , t'  =  0.0 ,
  Controller_A  =  create Controller_two_directions  (1, 1, "observe", 1, 0.01, 0,   -7.0, -6.0, -10.0, 9.0) ,
  Controller_D  =  create Controller_two_directions  (1, 2, "observe", 1, 0.01, 0,    5.0,  3.0, -10.0, 6.0) ,
  Controller_F  =  create Controller_two_directions  (1, 3, "observe", 1, 0.01, 0,   -1.0,  7.0, -10.0, 4.0) ,

 Controller_B  =  create Controller_four_directions(1, 4, "observe", "observe", "observe", 1, 0.01, 0,  0, 0,
                                                                                  -5.0, -4.0, -10.0, 8.0) ,

 Controller_C  =  create Controller_four_directions(1, 5, "observe", "observe", "observe", 1, 0.01, 0,  0, 0,
                                                                                  7.0,  1.0, -10.0, 7.0) ,

 Controller_E  =  create Controller_four_directions(1, 6, "observe", "observe", "observe", 1, 0.01, 0,  0, 0,
                                                                                  1.0,  7.0, -10.0, 5.0) ,

  _3D  =  () , _3DView = ()
always
  t' = 1.0 ,
  simulator.endTime  = 100.0 ,
  simulator.timeStep = 0.01 ,
   _3DView = ((-3,-50,0), (-3,0,0)),
  _3D + = ( Box      center= (1,0,-7)     size=(10,0.1,0.1),
            Cone     center = (6,0,-7)    radius = 0.4       length = 1           rotation = (pi/2,-pi/2,0) ,

            Box      center= (-0.25,0,-6) size=(7.5,0.1,0.1)       ,
            Box      center= (3.5,0,0)    size=(12,0.1,0.1)                       rotation=(0,pi/2,0) ,
            Cone     center = (3.5,0,6)   radius = 0.4       length = 1           rotation = (pi/2,0,0) ,

            Box      center= (1,0,0)      size=(10,0.1,0.1)       ,
            Cone     center = (-4,0,0)    radius = 0.4       length = 1           rotation = (pi/2,pi/2,0) ,

            Box      center= (3.125,0,2)  size=(1.25,0.1,0.1)      ,
            Box      center= (2.5,0,5)    size=(6,0.1,0.1)         rotation=(0,pi/2,0) ,
            Cone     center = (2.5,0,8)   radius = 0.4       length = 1              rotation = (pi/2,0,0) ,

            Box      center= (-1,0,3)     size=(3,0.1,0.1)         rotation=(0,pi/2,0) ,
            Box      center= (-3,0,1.5)   size=(4,0.1,0.1)         ,
            Cone     center = (-5,0,1.5)  radius = 0.4       length = 1              rotation = (pi/2,pi/2,0) ,

            Box      center= (1,0,0)      size=(9,0.1,0.1)         rotation=(0,pi/2,0) ,
            Box      center= (3.5,0,-4.5) size=(5,0.1,0.1)         ,
            Cone     center = (6,0,-4.5)  radius = 0.4       length = 1              rotation = (pi/2,-pi/2,0)
          ),

  Controller_A.other_Cont = Controller_E.loc ,   Controller_A.prio1 = Controller_E.prio ,

  Controller_D.other_Cont = Controller_B.loc ,   Controller_D.prio1 = Controller_B.prio ,

  Controller_F.other_Cont = Controller_C.loc ,   Controller_F.prio1 = Controller_C.prio ,

  Controller_B.other_Cont1 = Controller_D.loc , Controller_B.prio1 = Controller_D.prio ,
  Controller_B.other_Cont2 = Controller_C.loc , Controller_B.prio2 = Controller_C.prio ,
  Controller_B.other_Cont3 = Controller_E.loc , Controller_B.prio3 = Controller_E.prio ,

  Controller_C.other_Cont1 = Controller_F.loc , Controller_C.prio1 = Controller_F.prio ,
  Controller_C.other_Cont2 = Controller_B.loc , Controller_C.prio2 = Controller_B.prio ,
  Controller_C.other_Cont3 = Controller_E.loc , Controller_C.prio3 = Controller_E.prio ,

  Controller_E.other_Cont1 = Controller_A.loc , Controller_E.prio1 = Controller_A.prio ,
  Controller_E.other_Cont2 = Controller_B.loc , Controller_E.prio2 = Controller_B.prio ,
  Controller_E.other_Cont3 = Controller_C.loc , Controller_E.prio3 = Controller_C.prio
```

Appendix J

Model Complex Traffic Intersection CIF

```
svgfile "TrafficLight_crossing_scale.svg";
import "minimum_prio.cif";

const real yellow_time      = 2.0;
const real green_time       = 5.0;

alg int min_queue_A = 1;
alg int min_queue_B = 1;
alg int min_queue_C = 1;
alg int min_queue_D = 1;
alg int min_queue_E = 1;
alg int min_queue_F = 1;

alg bool cond_A = traffic_A.x >= min_queue_A and controller_A.Observe;
alg bool cond_B = traffic_B.x >= min_queue_B and controller_B.Observe;
alg bool cond_C = traffic_C.x >= min_queue_C and controller_C.Observe;
alg bool cond_D = traffic_D.x >= min_queue_D and controller_D.Observe;
alg bool cond_E = traffic_E.x >= min_queue_E and controller_E.Observe;
alg bool cond_F = traffic_F.x >= min_queue_F and controller_F.Observe;

automaton def Light (alg string dir; event turn_Green1, turn_Green2, no_cars):
    cont yellow_t der -1.0;
    cont green_t  der -1.0;

    event turn_Red, turn_Yellow;

    location Red:
        initial;
        edge turn_Green1                          do green_t := green_time    goto Green;
        edge turn_Green2                          do green_t := green_time    goto Green;

    location Green:
        edge turn_Yellow     when green_t <= 0.0   do yellow_t := yellow_time  goto Yellow;
        edge no_cars                              do yellow_t := yellow_time  goto Yellow;

    location Yellow:
        edge turn_Red        when yellow_t <= 0.0                            goto Red;

    svgout id "red_" + <string>dir      attr "fill" value if Red:    "red"    else "white" end;
    svgout id "green_" + <string>dir    attr "fill" value if Green:  "green"  else "white" end;
    svgout id "orange_" + <string>dir   attr "fill" value if Yellow: "orange" else "white" end;

    svgout id "orange_time_" + <string>dir  text value fmt("%.2f",  max(0,yellow_t));

    svgout id "green_time_" + <string>dir   text value fmt("%.2f",  max(0,green_t));

end

automaton def Traffic(alg string dir; alg bool Green; alg real arrival, departure):
    cont x = 0   der arrival;
    cont depart_t = departure der -1.0;

    event depart;

    location Traffic:
        initial;

    edge depart   when depart_t < 0.0, Green, x >= 1.0     do x := x - 1.0, depart_t := departure;

svgout id "queue_" + <string>dir  text value fmt("queue: %.1f", x);

end
```

```
automaton def Controller_onecross (alg string dir; alg real queue; alg int queue_length;
                                   event sync_turn_Green,  sync_answer_Request, do_Request,
                                       turn_Green, answer_Request, sync_request, turn_Red):
    disc int count_green = 0;
    event no_cars_in_queue;

    location Observe:
        initial;
        edge sync_turn_Green;
        edge sync_request;
        edge sync_answer_Request;

        edge do_Request        when queue >= queue_length     goto Request;

        edge turn_Green        when queue >= queue_length     do count_green := count_green + 1
                                                                  goto Green;

    location Request:
        edge answer_Request                   do count_green := count_green + 1   goto Green;

    location Green:
        edge sync_request                                                        goto Stop;
        edge no_cars_in_queue          when queue <= 1.0                         goto Stop;

    location Stop:
        edge sync_request;
        edge turn_Red                  goto Observe;
        edge no_cars_in_queue          when queue <= 1.0;

    svgout id "state_"       + <string>dir  text value <string>self;
    svgout id "count_green_" + <string>dir  text value fmt("# green: %d", count_green);

end

automaton def Controller_threecrosses (alg string dir; alg real queue;
                                       alg int queue_length, own_prio, prio1, prio2;
                                       event sync_turn_Green1, sync_turn_Green2, sync_turn_Green3,
                                       sync_answer_Request1, sync_answer_Request2,
                                       sync_answer_Request3, do_Request, turn_Green,
                                       answer_Request, sync_request1, sync_request2, sync_request3,
                                       turn_Red; alg bool cond1, cond2):

    disc int count_green = 0, prio = own_prio;
    event no_cars_in_queue;

    location Observe:
        initial;
        edge sync_turn_Green1,      sync_turn_Green2,       sync_turn_Green3;
        edge sync_request1,         sync_request2,          sync_request3;
        edge sync_answer_Request1,  sync_answer_Request2,   sync_answer_Request3;

        edge do_Request
            when queue >= queue_length, minimumprio(prio, [prio1, prio2], [cond1, cond2]) > 0
                                        do prio := prio + 1              goto Request;

        edge do_Request
            when queue >= queue_length, minimumprio(prio, [prio1, prio2], [cond1, cond2]) = 0
                                                                         goto Request;

    edge turn_Green    when queue >= queue_length   do count_green := count_green + 1   goto Green;
```

```
    location Request:
        edge answer_Request           do count_green := count_green + 1       goto Green;

    location Green:
        edge sync_request1, sync_request2, sync_request3                      goto Stop;
        edge no_cars_in_queue       when queue <= 1.0                         goto Stop;

    location Stop:
        edge sync_request1, sync_request2, sync_request3;
        edge turn_Red               goto Observe;
        edge no_cars_in_queue       when queue <= 1.0;

    svgout id "state_"       + <string>dir  text value <string>self;
    svgout id "count_green_" + <string>dir  text value fmt("# green: %d", count_green);
end

event   turn_Green_A, answer_Request_A, Request_A, turn_Green_B, answer_Request_B, Request_B,
        turn_Green_C, answer_Request_C, Request_C, turn_Green_D, answer_Request_D, Request_D,
        turn_Green_E, answer_Request_E, Request_E, turn_Green_F, answer_Request_F, Request_F;

controller_A:
Controller_onecross       ("A", traffic_A.x,  min_queue_A,  turn_Green_E, answer_Request_E,
                           Request_A, turn_Green_A, answer_Request_A, Request_E, light_A.turn_Red);
controller_D:
Controller_onecross       ("D", traffic_D.x,  min_queue_D,  turn_Green_B, answer_Request_B,
                           Request_D, turn_Green_D, answer_Request_D, Request_B, light_D.turn_Red);
controller_F:
Controller_onecross       ("F", traffic_F.x,  min_queue_F,  turn_Green_C, answer_Request_C,
                           Request_F, turn_Green_F, answer_Request_F, Request_C, light_F.turn_Red);
controller_B:
Controller_threecrosses   ("B", traffic_B.x,  min_queue_B,  1, controller_C.prio,
                           controller_E.prio, turn_Green_C, turn_Green_D, turn_Green_E,
                           answer_Request_C,  answer_Request_D,  answer_Request_E, Request_B,
                           turn_Green_B, answer_Request_B, Request_C, Request_D, Request_E,
                           light_B.turn_Red, cond_C, cond_E);
controller_C:
Controller_threecrosses   ("C", traffic_C.x,  min_queue_C,  1, controller_B.prio,
                           controller_E.prio, turn_Green_B, turn_Green_E, turn_Green_F,
                           answer_Request_B,  answer_Request_E,  answer_Request_F, Request_C,
                           turn_Green_C, answer_Request_C, Request_B, Request_E, Request_F,
                           light_C.turn_Red, cond_B, cond_E);
controller_E:
Controller_threecrosses   ("E", traffic_E.x,  min_queue_E,  1, controller_B.prio,
                           controller_C.prio, turn_Green_A, turn_Green_B, turn_Green_C,
                           answer_Request_A,  answer_Request_B,  answer_Request_C, Request_E,
                           turn_Green_E, answer_Request_E, Request_A, Request_B, Request_C,
                           light_E.turn_Red, cond_B, cond_C);

light_A: Light ("A", turn_Green_A, answer_Request_A, controller_A.no_cars_in_queue);
light_B: Light ("B", turn_Green_B, answer_Request_B, controller_B.no_cars_in_queue);
light_C: Light ("C", turn_Green_C, answer_Request_C, controller_C.no_cars_in_queue);
light_D: Light ("D", turn_Green_D, answer_Request_D, controller_D.no_cars_in_queue);
light_E: Light ("E", turn_Green_E, answer_Request_E, controller_E.no_cars_in_queue);
light_F: Light ("F", turn_Green_F, answer_Request_F, controller_F.no_cars_in_queue);
traffic_A: Traffic("A", light_A.Green,  1, 0.5);
traffic_B: Traffic("B", light_B.Green,  1, 0.5);
traffic_C: Traffic("C", light_C.Green,  1, 0.5);
traffic_D: Traffic("D", light_D.Green,  1, 0.5);
traffic_E: Traffic("E", light_E.Green,  1, 0.5);
traffic_F: Traffic("F", light_F.Green,  1, 0.5);
svgout id "time"               text value fmt("Time: %.2f", time);
```