SYSTEM VALIDATION

Formal Validation of a Solar Car Controller

Author:
V.I.FORNADE *(0812001)*
v.i.fornade@student.tue.nl,
E.E.NIKOLOV *(0972305)*
e.e.nikolov@student.tue.nl,
S.STANIMIR *(0971290)*
s.stanimir@student.tue.nl,
H.ZHANG *(0936881)*
h.zhang.1@student.tue.nl

Supervisor:
Prof. J.F. GROOTE

Version 1.0

June 4, 2022

Contents

Chapter 1

Introduction

The goal of the project is to elaborate on the design of a controller for a solar powered car. The focus in this design is directed on the power management done by the computers inside the vehicle, and some safety factors of the system. The formal specification and the verification of the design of this controller are checked using the mCRL2 language and its accompanying tool-set.

This document is intended to aid a software engineer towards implementing a functional power management system. In Chapter 2, the description of such a system is presented. Chapter 3 defines the requirements for the solar car controller. In Chapter 4 a list of all the actions that can occur in the system are defined. The actions will be used in the formal mCRL2 specification of the controller and its requirements. Chapter 5 provides a description of the requirements written in Chapter 3 in terms of these actions. Chapter 6 covers the modal formulas that are a representation of the translated requirements, from the previous chapter, in modal μ-calculus. The last chapter will describe the checking methods that were used in order to prove that the system given in Chapter 3 is valid. Finally, Appendix A contains the *mCRL2* code of the system design, and Appendix B contains the plain-text notation of the modal formulas, which can be used in the *mCRL2* tool-set.

Chapter 2

Global Description

2.1 Overview

The power management of a solar powered car is the focus of this project. The electric motor, the battery and the solar panel are the key elements used in this system. Such a vehicle relies on the efficient use and storage of the electric energy. Even though a solar car is comprised of different other mechanical components, they are not of relevance for the computers managing the power transfer and use. Some additional features for the vehicle which are taken into account are a speedometer and cruise control system.

2.1.1 Motor

The electric motor is used to drive the vehicle, but also to recover some of the energy lost during braking. Thus, the energy can flow both to and from the electric motor. Quantifying the amount of energy necessary to move the vehicle or which is recovered from braking is dependent on the physical setup and vehicle dynamics, therefore these measurements are out of scope for this project. The motor will be powered by the user using the throttle pedal and the regenerative braking will be enabled when the user presses the brake pedal. When the cruise control is activated, the main controller will decide to either increase/decrease the power delivered to the motor, as well as engaging the regenerative braking in order to reach the desired speed.

2.1.2 Solar Panel

This device is used to harvest energy from the sun in order to power the system. The energy can flow only from it towards the motors or the battery. It will be considered within this system, for efficiency purposes, that the solar panels will first match the power demand from the motor and deliver the excessive energy towards the battery.

2.1.3 Battery

The battery is used to store the excessive energy harvested by the solar panel. Also, when the power demand from the motor will exceed the power output from the solar panel, the battery will act as a buffer and supply the remaining necessary power to the motor. It is considered that the battery can supply full power to the motor until depleted and that they can be replenished at the maximum rate given from the motor. The battery has three values denoting that it is either empty, charged or full. It also has a temperature sensor, indicating whether its temperature is dangerously high or not. If the temperature of the battery is exceeded, then the no power transfer can be driven towards the battery.

2.1.4 Cruise Control System

The vehicle is also equipped with a speedometer and a cruise control system. Thus, the user is able to observe the real-time velocity of the vehicle, but this feature is also used by the computers when the cruise control is activated. The cruise control can be activated only by the user, by pressing a dedicated button. When activated, the system will record the instantaneous velocity of the vehicle as the desired speed and it will control the power required by the motor in order to keep the speed of the vehicle close to this desired speed in time. It is assumed that the cruise control system can only be activated when the user is pressing neither the throttle nor the brake pedal, and it will only modulate the power sent to the electrical motor.

2.1.5 Monitoring System

The solar powered car is transmitting information about the system to a third party on a regular basis.

Chapter 3

Global Requirements

In this chapter the essential requirements of the solar powered car are enlisted. The goal of this section is to define characteristics of the system such that the complete and expected functionality of the vehicle is covered.

1. If the off switch is pressed:
   (a) the power exchange between the Motor, Battery and Solar Panel must be disabled.
   (b) the power exchange between the Motor, Battery and Solar Panel cannot be enabled before the on switch is pressed.
2. If the brake pedal is pressed:
   (a) power supply to the motor must be disabled.
   (b) power supply to the motor cannot be enabled before the brake pedal is released.
3. If the brake pedal is pressed, the motor must be used as a generator in order to power the battery, as long as the battery is not full nor overheating. The amount of power generated must be proportional to the current speed of the car.
4. If the battery's sensors indicate that it is full or overheating:
   (a) power supply to the battery must be disabled.
   (b) power supply to the battery cannot be enabled before the sensors indicate that the battery is no longer full nor overheating.
5. (a) Power may never be supplied to the battery before it stops supplying power to the motor.
   (b) Power may never be supplied by the battery to the motor before the solar panel and the motor stop providing power to the battery.
   (c) Power may never be supplied by the solar panel or battery to the motor before the latter stops supplying power to the battery.
   (d) Power may never be supplied by the motor to the battery before the solar panel and the battery stop supplying power to the motor.
6. (a) If the battery is empty:
      i. the power supply from the battery to the motor must be disabled.
      ii. the power supply from the battery to the motor cannot be enabled before the battery has charge.
   (b) If the solar panel output is low:
      i. the power supply from the solar panel to the battery and the motor must be disabled.

ii. the power supply from the solar panel to the battery and the motor cannot be enabled before the solar panel output rises.

7. (a) If the motor requires less power than the output of the solar panel, then the solar panel must supply the entirety of the required power.

   If in addition, the battery is not full nor overheating, the power surplus of the solar panel must be directed to the batteries.

   (b) If the motor requires more power than the output of the solar panel, then the solar panel must supply all of its power to the motor.

   If in addition, the battery is not empty, it must supply the remainder of the required power to the motor.

8. When the throttle is pressed to a certain degree, the motor's power requirement must be proportional to that degree.

9. When the cruise control is activated, the current speed of the solar car must be set as the desired speed, the speed that needs to be maintained. It can only be activated if the current speed is more than 0 and if the throttle pedal isn't pressed, except if it was subsequently released or the car was switched on.

10. When the cruise control is activated, the current speed must continuously be read and after each read:

    (a) if the current speed, subtracted from the desired speed, is more than a certain threshold, then the motor must require additional power (up to a certain maximum), unless the cruise control is deactivated.

    (b) if the desired speed, subtracted from the current speed, is more than a certain threshold, then the motor must require less power (down to a certain minimum), unless the cruise control is deactivated.

    (c) otherwise, the motor's required power must remain unchanged, unless the cruise control is deactivated.

11. When the cruise control is activated, if the throttle pedal, brake pedal, or off switch is pressed, the cruise control must be deactivated.

12. If the cruise control is activated, then it must always eventually be possible to deactivate it, as long as it hasn't been deactivated yet.

13. (a) The cruise control cannot be deactivated before it has been activated

    (b) The cruise control cannot be deactivated twice in a row without an activation between them.

14. The car must regularly stream the status of its components to a monitoring system.

Chapter 4

Interactions

In the following figures and tables, all the interactions that may occur within the system or between the system and the external environment are listed. These actions will be briefly defined in this section and they will be used in the next chapter to describe the requirements, as well as in the programming section for consistency purposes.

4.1 External Components

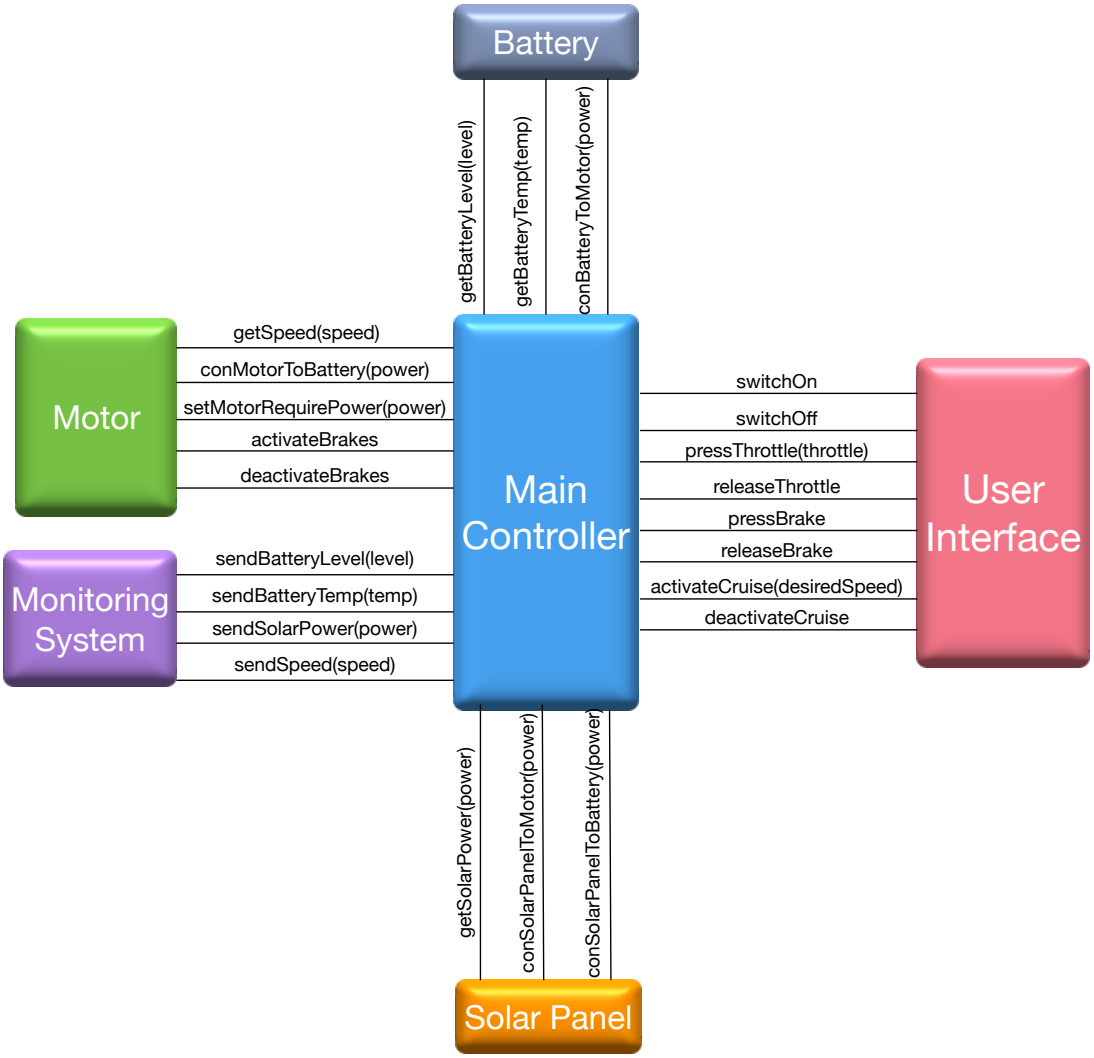

FIGURE 4.1: Main Controller External Architecture

4.2 Data types

Variable types used:

Integer numbers: limited in the interval $[0, 5]$ in order to keep the state space small
Structured types *BatteryLevel*: *Empty* | *Medium* | *Full*
Structured types *BatteryTemp*: *Normal* | *Overheating*

Constants used:

SPEED_THRESHOLD = 3
MAX = 5

Auxiliary processes:

PowerOff - contains logic for powering off the solar car
PressBrake - contains logic for what follows after pressing the brake

4.3 External actions description

User interface interactions	
A1. *switchOn*	Set the on/off switch to On
A2. *switchOff*	Set the on/off switch to Off
A3. *pressThrottle(throttle : Int)*	Press the Throttle Pedal to a specific degree
A4. *releaseThrottle*	Release the Throttle Pedal
A5. *pressBrake*	Press the Brake Pedal
A6. *releaseBrake*	Release the Brake Pedal
A7. *activateCruise(desiredSpeed : Int)*	Activate the Cruise Control system with a specific desired speed
A8. *deactivateCruise*	Deactivate the Cruise Control system
Battery	
B1. *getBatteryLevel(level : BatteryLevel)*	Indicate the current energy status of the Battery
B2. *getBatteryTemp(temp : BatteryTemp)*	Indicate the current temperature status of the Battery
B3. *conBatteryToMotor(power : Int)*	Inform the Battery to supply a specific amount of power to the Motor. Power 0 means that this connection is disabled
Motor	
C1. *getSpeed(speed : Int)*	Indicate the current speed
C2. *conMotorToBattery(power : Int)*	Inform the Motor to act as a generator and supply a specific amount of power to the Battery. Power 0 means that this connection is disabled
C3. *setMotorRequirePower(power : Int)*	Indicate that the Motor requires a specific amount of power
C4. *activateBrakes*	Activate the mechanical brakes
C5. *deactivateBrakes*	Deactivate the mechanical brakes
Solar Panel	
D1. *getSolarPower(power : Int)*	Indicate the current output power of the Solar Panel. Power 0 means that solar power output is low.
D2. *conSolarPanelToMotor(power : Int)*	Inform the Solar Panel to supply a specific amount of power to the Motor. Power 0 means that this connection is disabled
D3. *conSolarPanelToBattery(power : Int)*	Inform the Solar Panel to supply a specific amount of power to the Battery. Power 0 means that this connection is disabled
Monitoring system	
E1. *sendBatteryLevel(level : BatteryLevel)*	Send the current level status of the Battery to the Monitoring System
E2. *sendBatteryTemp(temp : BatteryTemp)*	Send the current temperature status of the Battery to the Monitoring System
E3. *sendSolarPower(power : Int)*	Send the current power output of the Solar Panel to the Monitoring System
E4. *sendSpeed(speed : Int)*	Send the current speed to the Monitoring System

TABLE 4.1: External Interactions

4.4 Internal Components

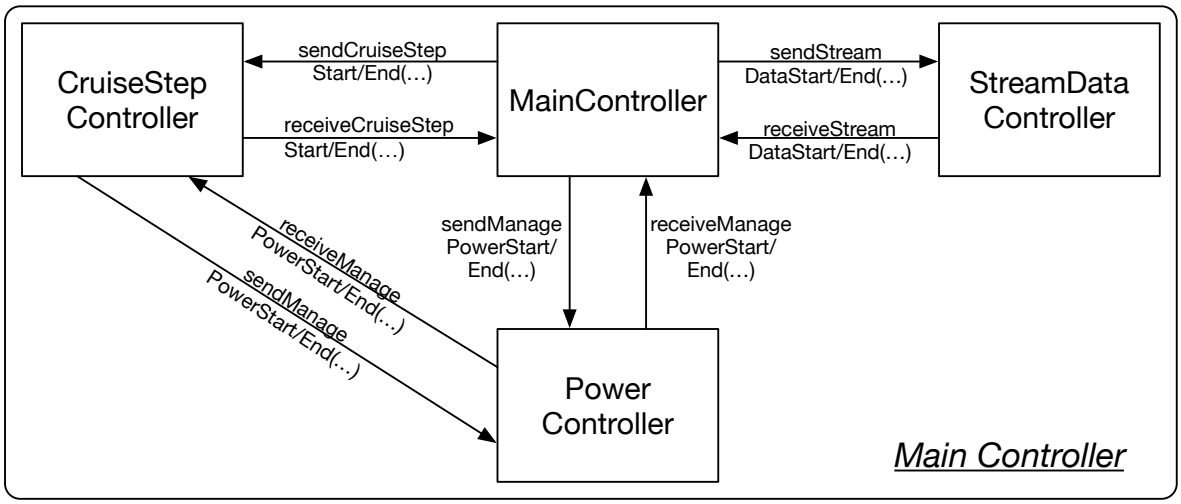

FIGURE 4.2: Main Controller Internal Architecture

The code of the system design can be found in Appendix B and it's internal architecture can be seen in figure 4.2. It contains 4 parallel controllers:

1. **MainController** - responsible for handling user input and forwarding control to the other parallel components as necessary.
2. **StreamDataController** - responsible for recording the values from the sensors for battery level and temperature, solar panel output, current speed and streaming them to the support vehicle.
3. **PowerController** - responsible for managing the power connections between the battery, the solar panel, and the motor after the required power is set by either the MainController or the CruiseStepController.
4. **CruiseStepController** - responsible for performing a single step of the cruise control. The goal of the cruise control is to maintain a specific desired speed by varying the power required by the motor on behalf of the user.

4.5 Internal actions description

F1.	*sendStreamDataStart*	Send a request to the StreamDataController to start streaming
F2.	*receiveStreamDataStart*	Indicates that the StreamDataController has received a request to start streaming
F3.	*sendStreamDataEnd(speed : Int, batteryLevel : BatteryLevel, batteryTemp : BatteryTemp, solarPower : Int)*	The StreamDataController indicates that it has finished streaming and indicates the streamed values
F4.	*receiveStreamDataEnd(speed : Int, batteryLevel : BatteryLevel, batteryTemp : BatteryTemp, solarPower : Int)*	A controller receives a notification that the StreamDataController has finished streaming
F5.	*sendCruiseStepStart(throttle : Int, desiredSpeed : Int, speed : Int, batteryLevel : BatteryLevel, batteryTemp : BatteryTemp, solarPower : Int)*	Send a request to the CruiseStepController to initiate a cruise step with specific parameters
F6.	*receiveCruiseStepStart(throttle : Int, desiredSpeed : Int, speed: Int, batteryLevel : BatteryLevel, batteryTemp : BatteryTemp, solarPower : Int)*	Indicates that the CruiseStepController has received a request to perform a single cruise step
F7.	*sendCruiseStepEnd(power : Int)*	The CruiseStepController indicates that it has finished with a single cruise step
F8.	*receiveCruiseStepEnd(power : Int)*	A controller receives a notification that the CruiseStepController has finished a single cruise step with the new power requirement of the motor
F9.	*sendManagePowerStart(throttle : Int, speed : Int, batteryLevel : BatteryLevel, batteryTemp : BatteryTemp, solarPower : Int)*	Send a request to the PowerController to perform a power management step
F10.	*receiveManagePowerStart(newThrottle : Int, speed : Int, batteryLevel : BatteryLevel, batteryTemp : BatteryTemp, solarPower : Int)*	Indicates that the PowerController has received a request to perform a power management step
F11.	*sendManagePowerEnd*	The PowerController indicates that it has finished with a power management step.
F12.	*receiveManagePowerEnd*	A controller receives a notification that the PowerController has finished performing a power management step
F13.	*continue*	Auxiliary action that is used to resolve deadlocks in cases where an action should be skipped
F14.	*newCycle(on : Bool, cruise: Bool, throttle : Int, desiredSpeed : Int)*	Auxiliary action that indicates the beginning of a new operation cycle of the MainController

TABLE 4.2: Internal Interactions

Chapter 5

Translation of Requirements

In this chapter, the requirements described in chapter 3 are translated in terms of the interactions defined by the previous chapter. This ought to give a better insight into the system behavior with respect to the actions that can take place.

1. If *switchOff* takes place, then:
   (a) a *conMotorToBattery(0)*, a *conBatteryToMotor(0)*, a *conSolarPanelToMotor(0)* and a *conSolarPanelToBattery(0)* must inevitably follow.
   (b) the actions *conMotorToBattery(m)*, *conBatteryToMotor(m)*, *conSolarPanelToMotor(m)* and *conSolarPanelToBattery(m)*, where $m > 0$, are not allowed before a *switchOn* happens.
2. If a *pressBrake* action takes place, then:
   (a) a *conBatteryToMotor(0)* and a *conSolarPanelToMotor(0)* must inevitably follow.
   (b) the actions *conBatteryToMotor(m)* and *conSolarPanelToMotor(m)*, where $m > 0$, are not allowed before a *releaseBrake* happens.
3. A *pressBrake* must inevitably be followed by a *conMotorToBattery(speed)*, where *getSpeed(speed)* was the last action of its kind to happen before that, as long as *getBatteryLevel(n)*, where $n \notin \{Full\}$ and *getBatteryTemp(m)*, where $m \notin \{Overheating\}$ were also the last actions of their kinds that happened prior to the *pressBrake* action.
4. If a *getBatteryTemp(Overheating)* or a *getBatteryLevel(Full)* happens, then:
   (a) a *conSolarPanelToBattery(0)* and a *conMotorToBattery(0)* must inevitably follow.
   (b) a *conSolarPanelToBattery(m)*, where $m > 0$ and a *conMotorToBattery(n)*, where $n > 0$ cannot happen before a *getBatteryTemp(Normal)* and a *getBatteryLevel(n)*, where $n \in \{Empty, Medium\}$ events take place.
5. (a) Neither a *conSolarPanelToBattery(m)*, nor a *conMotorToBattery(m)*, $m > 0$, can happen, unless preceded by a *conBatteryToMotor(0)*.
   (b) A *conBatteryToMotor(m)*, $m > 0$, cannot happen, unless preceded by both a *conSolarPanelToBattery(0)* and a *conMotorToBattery(0)*.
   (c) Neither a *conSolarPanelToMotor(m)*, nor a *conBatteryToMotor(m)*, $m > 0$, can happen, unless preceded by a *conMotorToBattery(0)*.
   (d) A *conMotorToBattery(m)*, $m > 0$, cannot happen, unless preceded by both a *conSolarPanelToMotor(0)* and a *conBatteryToMotor(0)*.

6. (a) If a *getBatteryLevel(Empty)* takes place, then:
   i. a *conBatteryToMotor(0)* must inevitably follow.
   ii. a *conBatteryToMotor(m)*, where $m > 0$ cannot follow before a *getBatteryLevel(n)*, $n \in \{Medium, Full\}$, happens.

   (b) If a *getSolarPower(0)* event takes place, then:
   i. a *conSolarPanelToMotor(0)* and a *conSolarPanelToBattery(0)* must inevitably follow.
   ii. neither a *conSolarPanelToMotor(m)*, $m > 0$, nor a *conSolarPanelToBattery(n)*, $n > 0$, can follow before a *getSolarPower(p)*, $p > 0$, happens.

7. When a *setMotorRequirePower(requiredPower)*, happens, if *getSolarPower(solarPower)*, was the last action of its kind to happen before that:

   (a) if $requiredPower \leq solarPower$, then a *conSolarPanelToMotor(requiredPower)* must inevitably follow.

   If in addition to that, *getBatteryLevel(level)* ($level \in \{Empty, Medium\}$) and *getBatteryTemp(Normal)* were the last actions of their kinds to happen prior to the *setMotorRequirePower(requiredPower)* action, then a *conSolarPanelToBattery(solarPower - requiredPower)* must also inevitably follow.

   (b) if $requiredPower \geq solarPower$, then a *conSolarPanelToMotor(solarPower)* must inevitably follow.

   If in addition to that, *getBatteryLevel(level)* ($level \in \{Medium, Full\}$) was the last action of its kind that happened prior to the *setMotorRequirePower(requiredPower)* action, then a *conBatteryToMotor(requiredPower - solarPower)* must also inevitably follow.

8. If a *pressThrottle(throttle)* happens, then a *setMotorRequirePower(power)*, where $throttle = power$, must inevitably follow.

9. An *activateCruise(desiredSpeed)* can only happen if *getSpeed(speed)* was the last action of its kind before that, where $speed = desiredSpeed$ and $speed > 0$ and there was no *pressThrottle(throttle)*, $throttle > 0$, except if it was followed by *releaseThrottle* or *switchOn*.

10. If an *activateCruise(desiredSpeed)*, $desiredSpeed > 0$, happens, and *setMotorRequirePower(power)*, was the last action of its kind to happen before that, then for each subsequent *getSpeed(speed)*:

    (a) if $desiredSpeed - currentSpeed > SPEED\_THRESHOLD$, then a *setMotorRequirePower(min(power + 1, MAX))* must inevitably follow, unless a *deactivateCruise* happens first.

    (b) if $currentSpeed - desiredSpeed > SPEED\_THRESHOLD$, then a *setMotorRequirePower(max(power - 1, 0))* must inevitably follow, unless a *deactivateCruise* happens first.

    (c) otherwise, a *setMotorRequirePower(power)* must inevitably follow, unless a *deactivateCruise* happens first.

11. After an *activateCruise(desiredSpeed)* takes place, if a *pressThrottle(throttle)*, a *releaseThrottle*, a *pressBrake* or a *switchOff* action eventually happens, then a *deactivateCruise* must inevitably follow.

12. If an *activateCruise(desiredSpeed)* takes place, then a *deactivateCruise* is always eventually possible, as long as it hasn't happened yet.

13. (a) A *deactivateCruise* cannot happen before an *activateCruise(desiredSpeed)* has happened.
    (b) Two *deactivateCruise* actions cannot happen in a row without an *activateCruise(desiredSpeed)* between them.

14. (a) If a *getBatteryLevel(level)* action takes place, then a *sendBatteryLevel(level)* must inevitably follow.
    (b) If a *getBatteryTemp(temp)* action takes place, then a *sendBatteryTemp(temp)* must inevitably follow.
    (c) If a *getSolarPower(solarPower)* action takes place, then a *sendSolarPower(solarPower)* must inevitably follow.
    (d) If a *getSpeed(speed)* action takes place, then a *sendSpeed(speed)* must inevitably follow.

Chapter 6

Modal formulas

In this chapter, the requirements described in chapter 5 are further transformed into modal formulas. Using those, the model can be verified via the mCRL2 tool-set in order to make sure that it conforms to the requirements. It should be noted that those formulas cannot be used directly by the tool. They must first be transcribed into a plain-text notation, which is semantically the same, but easier to input into a text editor. The transcribed requirements can be found in Appendix B.

1. (a) $[true^* \cdot switchOff]\mu X.([\overline{conMotorToBattery(0)}]X \wedge \langle true\rangle true) \wedge$
$[true^* \cdot switchOff]\mu X.([\overline{conBatteryToMotor(0)}]X \wedge \langle true\rangle true) \wedge$
$[true^* \cdot switchOff]\mu X.([\overline{conSolarPanelToMotor(0)}]X \wedge \langle true\rangle true) \wedge$
$[true^* \cdot switchOff]\mu X.([\overline{conSolarPanelToBattery(0)}]X \wedge \langle true\rangle true)$

   (b) $[true^* \cdot switchOff \cdot \overline{switchOn}^*]$
$\forall m : \mathbb{Z}.m > 0 \implies [conMotorToBattery(m) \cup conBatteryToMotor(m) \cup$
$conSolarPanelToMotor(m) \cup conSolarPanelToBattery(m)]false$

2. (a) $[true^* \cdot pressBrake]\mu X.([\overline{conBatteryToMotor(0)}]X \wedge \langle true\rangle true) \wedge$
$[true^* \cdot pressBrake]\mu X.([\overline{conSolarPanelToMotor(0)}]X \wedge \langle true\rangle true)$

   (b) $[true^* \cdot pressBrake \cdot \overline{releaseBrake}^*]$
$\forall m : \mathbb{Z}.m > 0 \implies [conBatteryToMotor(m) \cup conSolarPanelToMotor(m)]false$

3. $\nu X(level : BatteryLevel := Medium, temp : BatteryTemp := Normal, speed : \mathbb{Z} := 0).$
$[\forall level' : BatteryLevel, temp' : BatteryTemp, speed' : \mathbb{Z}.$
$\overline{getBatteryLevel(level') \cap getBatteryTemp(temp') \cap getSpeed(speed')}]$
$X(level, temp, speed) \wedge$
$(\forall level' : BatteryLevel.[getBatteryLevel(level')]X(level', temp, speed)) \wedge$
$(\forall temp' : BatteryTemp.[getBatteryTemp(temp')]X(level, temp', speed)) \wedge$
$(\forall speed' : \mathbb{Z}.[getSpeed(speed')]X(level, temp, speed')) \wedge$
$[pressBrake](level \not\approx Full \wedge temp \not\approx Overheating)$
$\rightarrow \mu Y.([\overline{conMotorToBattery(speed)}]Y \wedge \langle true\rangle true)$

4. (a) $[true^* \cdot getBatteryTemp(Overheating)]\mu X.([\overline{conSolarPanelToBattery(0)}]X \wedge \langle true\rangle true) \wedge$
$[true^* \cdot getBatteryTemp(Overheating)]\mu X.([\overline{conMotorToBattery(0)}]X \wedge \langle true\rangle true) \wedge$
$[true^* \cdot getBatteryLevel(Full)]\mu X.([\overline{conSolarPanelToBattery(0)}]X \wedge \langle true\rangle true) \wedge$
$[true^* \cdot getBatteryLevel(Full)]\mu X.([\overline{conMotorToBattery(0)}]X \wedge \langle true\rangle true)$

   (b) $[true^* \cdot getBatteryTemp(Overheating) \cdot \overline{getBatteryTemp(Normal)}^*]$
$\forall m : \mathbb{Z}.m > 0 \implies [conSolarPanelToBattery(m) \cup conMotorToBattery(m)]false \wedge$
$[true^* \cdot getBatteryLevel(Full) \cdot \overline{getBatteryLevel(Empty) \cup getBatteryLevel(Medium)}^*]$
$\forall m : \mathbb{Z}.m > 0 \implies [conSolarPanelToBattery(m) \cup conMotorToBattery(m)]false$

5. (a) $[\overline{conBatteryToMotor(0)}^*]$
$\forall m : \mathbb{Z}.m > 0 \implies [conSolarPanelToBattery(m) \cup conMotorToBattery(m)]false$
   (b) $[\overline{conSolarPanelToBattery(0) \cup conMotorToBattery(0)}^*]$
$\forall m : \mathbb{Z}.m > 0 \implies [conBatteryToMotor(m)]false$
   (c) $[\overline{conMotorToBattery(0)}^*]$
$\forall m : \mathbb{Z}.m > 0 \implies [conSolarPanelToMotor(m) \cup conBatteryToMotor(m)]false$
   (d) $[\overline{conSolarPanelToMotor(0) \cup conBatteryToMotor(0)}^*]$
$\forall m : \mathbb{Z}.m > 0 \implies [conMotorToBattery(m)]false$

6. (a) i. $[true^* \cdot getBatteryLevel(Empty)]\mu X.([\overline{conBatteryToMotor(0)}]X \wedge \langle true\rangle true)$
   ii. $[true^* \cdot getBatteryLevel(Empty) \cdot$
$\overline{getBatteryLevel(Medium) \cup getBatteryLevel(Full)}^*]$
$\forall m : \mathbb{Z}.m > 0 \implies [conBatteryToMotor(m)]false$
   (b) i. $[true^* \cdot getSolarPower(0)]\mu X.([\overline{conSolarPanelToMotor(0)}]X \wedge \langle true\rangle true) \wedge$
$[true^* \cdot getSolarPower(0)]\mu X.([\overline{conSolarPanelToBattery(0)}]X \wedge \langle true\rangle true)$
   ii. $[true^* \cdot getSolarPower(0)][\forall n : \mathbb{Z}.n > 0 \implies \overline{getSolarPower(n)}^*]$
$(\forall m : \mathbb{Z}.m > 0 \implies$
$[conSolarPanelToMotor(m) \cup conSolarPanelToBattery(m)]false)$

7. $\nu X(level : BatteryLevel := Medium, temp : BatteryTemp := Normal,$
$solarPower : \mathbb{Z} := 0, speed : \mathbb{Z} := 0).$
$([\forall level' : BatteryLevel, temp' : BatteryTemp, solarPower' : \mathbb{Z}, speed' : \mathbb{Z}.$
$\overline{getBatteryLevel(level')} \cap \overline{getBatteryTemp(temp')} \cap$
$\overline{getSolarPower(solarPower')} \cap \overline{getSpeed(speed')}]$
$X(level, temp, solarPower, speed)) \wedge$
$(\forall level' : BatteryLevel.[getBatteryLevel(level')]X(level', temp, solarPower, speed)) \wedge$
$(\forall temp' : BatteryTemp.[getBatteryTemp(temp')]X(level, temp', solarPower, speed)) \wedge$
$(\forall solarPower' : \mathbb{Z}.[getSolarPower(solarPower')]X(level, temp, solarPower', speed)) \wedge$
$(\forall speed' : \mathbb{Z}.[getSpeed(speed')]X(level, temp, solarPower, speed'))\wedge$

   (a) $(\forall requiredPower : \mathbb{Z}.[setMotorRequirePower(requiredPower)]$
$(requiredPower \leq solarPower)$
$\rightarrow \mu Y.(\overline{conSolarPanelToMotor(requiredPower)}Y \wedge \langle true\rangle true)) \wedge$
$(\forall requiredPower : \mathbb{Z}.[setMotorRequirePower(requiredPower)]$
$((requiredPower \leq solarPower) \wedge level \not\approx Full \wedge temp \not\approx Overheating)$
$\rightarrow \mu Y.([\overline{conSolarPanelToBattery(solarPower - requiredPower)}]Y \wedge \langle true\rangle true))\wedge$
   (b) $(\forall requiredPower : \mathbb{Z}.[setMotorRequirePower(requiredPower)]$
$(requiredPower \geq solarPower)$
$\rightarrow \mu Y.(\overline{conSolarPanelToMotor(solarPower)}Y \wedge \langle true\rangle true)) \wedge$
$(\forall requiredPower : \mathbb{Z}.[setMotorRequirePower(requiredPower)]$
$((requiredPower \geq solarPower) \wedge level \not\approx Full \wedge temp \not\approx Overheating)$
$\rightarrow \mu Y.([\overline{conBatteryToMotor(requiredPower - solarPower)}]Y \wedge \langle true\rangle true))$

8. $[true^*]\forall t : \mathbb{Z}.t > 0 \implies [pressThrottle(t)]\mu X.([\overline{setMotorRequirePower(t)}] \wedge \langle true\rangle true))$

9. $\nu X(speed : \mathbb{Z} := 0, throttle : \mathbb{Z} := 0).$
$([\forall speed' : \mathbb{Z}, throttle' : \mathbb{Z}.$
$\overline{getSpeed(speed')} \cap \overline{pressThrottle(throttle')} \cap \overline{releaseThorttle} \cap \overline{switchOn})]$
$X(speed, throttle)) \wedge (\forall speed' : \mathbb{Z}.[getSpeed(speed')]X(speed', throttle)) \wedge$
$(\forall throttle' : \mathbb{Z}.[pressThrottle(throttle')]X(speed, throttle')) \wedge$
$([releaseThrottle \vee switchOn]X(speed, 0)) \wedge$
$(\forall speed' : \mathbb{Z}.[activateCruise(speed')](speed > 0 \wedge speed \approx speed' \wedge throttle \approx 0))$

10. $\nu X(desiredSpeed : \mathbb{Z} := 0, power : \mathbb{Z} := 0).$
$[\forall desiredSpeed', power' : \mathbb{Z}.$
$\overline{setMotorRequirePower(power')} \cap \overline{activateCruise(desiredSpeed')} \cap \overline{deactivateCruise}]$
$X(desiredSpeed, power) \wedge$
$(\forall desiredSpeed' : \mathbb{Z}.[activateCruise(desiredSpeed')]X(desiredSpeed', 0)) \wedge$
$[deactivateCruise]X(0, power) \wedge$
$(\forall power' : \mathbb{Z}.[setMotorRequirePower(power')]X(desiredSpeed, power')) \wedge$
$(\forall speed : \mathbb{Z}.[getSpeed(speed)](desiredSpeed > 0) \rightarrow ($

    (a) $(desiredSpeed - speed > SPEED\_THRESHOLD)$
    $\rightarrow (\mu Y.([\overline{setMotorRequirePower(min(power + 1, MAX))} \cap \overline{deactivateCruise}]Y \wedge$
    $\langle true \rangle true)) \wedge$

    (b) $(speed - desiredSpeed > SPEED\_THRESHOLD)$
    $\rightarrow (\mu Y.([\overline{setMotorRequirePower(max(power - 1, 0))} \cap \overline{deactivateCruise}]Y \wedge \langle true \rangle true)) \wedge$

    (c) $(|speed - desiredSpeed| \leq SPEED\_THRESHOLD)$
    $\rightarrow (\mu Y.([\overline{setMotorRequirePower(power)} \cap \overline{deactivateCruise}]Y \wedge \langle true \rangle true)))$

11. $[true^*]\forall n : \mathbb{Z}.[activateCruise(n)][\overline{deactivateCruise(n)}^*]$
$[(\exists m : \mathbb{Z}.pressThrottle(m)) \cup pressBrake \cup switchOff \cup releaseThrottle]$
$\mu X.([\overline{deactivateCruise}]X \wedge \langle true \rangle true)$

12. $[true^*]\forall desiredSpeed : \mathbb{Z}.(desiredSpeed > 0) \implies [activateCruise(desiredSpeed)]$
$[\overline{deactivateCruise}^*]\langle true^* \cdot deactivateCruise \rangle true$

13. (a) $[\forall desiredSpeed : \mathbb{Z}.\overline{activateCruise(desiredSpeed)}^* \cdot deactivateCruise]false$

    (b) $[deactivateCruise \cdot \forall desiredSpeed : \mathbb{Z}.\overline{activateCruise(desiredSpeed)}^* \cdot deactivateCruise]false$

14. (a) $[true^*]\forall n : BatteryLevel.[getBatteryLevel(n)]\mu X.([\overline{sendBatteryLevel(n)}]X \wedge \langle true \rangle true)$

    (b) $[true^*]\forall n : BatteryTemp.[getBatteryTemp(n)]\mu X.([\overline{sendBatteryTemp(n)}]X \wedge \langle true \rangle true)$

    (c) $[true^*]\forall n : \mathbb{Z}.[getSolarPower(n)]\mu X.([\overline{sendSolarPower(n)}]X \wedge \langle true \rangle true)$

    (d) $[true^*]\forall n : \mathbb{Z}.[getSpeed(n)]\mu X.([\overline{sendSpeed(n)}]X \wedge \langle true \rangle true)$

Chapter 7

System Verification

The model that was designed needs to be verified against the modal formulas from Chapter 6 in order to determine whether or not it meets the global requirements. This is achieved by using the *mCRL2* verification tool-set, which requires the modal formulas to be transcribed into a computer friendly plain-text notation. These transcriptions can be found in Appendix B.

7.1 Verification environment

The following environment is used to verify the model:

1. The tool-set applied for verifying the design requirement is *mCRL2*: 201409.1.13218 (Release).
2. Verification System:

Processor:	Intel® CoreTM i7-4710HQ @ 2.50Ghz
Internal memory (RAM):	16 GB
Operating System:	Windows 10

7.2 Verification Steps

In the process of verification, several helper scripts were developed around the *mCRL2* tool-set in order to simplify the repetitive execution of verification steps. They can be found at https://github.com/e-nikolov/mcrl2-helper. In this chapter however, the verification steps will be discussed in their native *mCRL2* form in order to make them more basic and accessible.

An *mCRL2* model (e.g. *solar-car.mcrl2*) can be verified against a modal formula (e.g. *req1.mcf*) using the following steps:

1. The *mCRL2* code is transformed into a Linear Process Specification (LPS), in a **.lps* file, via the *mcl22lps* tool:

```
$ mcrl22lps solar-car.mcrl2 solar-car.lps -lstack
```

2. The Linear Process Specification is transformed into a Labeled Transition System (LTS), in a **.lps* file, via the *lps2lts* tool:

```
$ lps2lts solar-car.lps solar-car.lts
```

3. The state space of the LTS is reduced by using a divergence preserving branching bisimilarity using signature refinement reduction via the *ltsconvert* tool:

```
$ ltsconvert --equivalence=dpbranching-bisim-sig \
      solar-car.lps solar-car.lts
```

4. The reduced LTS, combined with a modal formula (e.g. *req1.mcf*) is used in order to generate a Parameterised Boolean Equation System (PBES), in a **.pbes* file, via the *lts2pbes* tool:

```
$ lts2pbes --formula=req1.mcf solar-car.lts \
solar-car.pbes
```

5. Finally, a Boolean value, which states whether or not the formula holds for the model, is generated by solving the PBES via the *pbes2bool* tool:

```
$ pbes2bool solar-car.pbes
```

7.3 Verification results

All of the modal formulas used for the validation of the model, produced TRUE as a result, which means that the model satisfies all of the global requirements.

Appendix A

mCRL2 code

This appendix contains the mCRL2 code of the solar-car model.

solar-car.mcrl2 .

```
%------------------------ Sort declarations -------------------------------------%

sort BatteryLevel = struct Empty  | Medium | Full;
sort BatteryTemp  = struct Normal | Overheating;

%------------------------ Map declarations --------------------------------------%

map MAX : Int;
eqn MAX = 5;

map SPEED_THRESHOLD : Int;
eqn SPEED_THRESHOLD = 3;


%------------------------ Action declarations -----------------------------------%

% Input actions

act switchOn, switchOff;
    pressThrottle : Int;
    releaseThrottle;
    pressBrake, releaseBrake;

    activateCruise : Int;
    deactivateCruise;

% Battery actions

    getBatteryLevel : BatteryLevel;
    getBatteryTemp : BatteryTemp;
    conBatteryToMotor : Int;

% Motor actions

    getSpeed : Int;
    conMotorToBattery : Int;
    setMotorRequirePower : Int;

% Solar Panel actions
    getSolarPower : Int;
    conSolarPanelToMotor : Int;
    conSolarPanelToBattery : Int;

% Mechanical brakes

    activateBrakes, deactivateBrakes;

% Monitoring system

    sendBatteryLevel : BatteryLevel;
    sendBatteryTemp : BatteryTemp;
    sendSolarPower : Int;
    sendSpeed : Int;
```

```
54
55  % Internal actions
56
57      sendStreamDataStart, receiveStreamDataStart, startStreamData;
58      sendStreamDataEnd, receiveStreamDataEnd, endStreamData :
59          Int # BatteryLevel # BatteryTemp # Int;
60
61      sendCruiseStepStart, receiveCruiseStepStart, startCruiseStep :
62          Int # Int # Int # BatteryLevel # BatteryTemp # Int;
63      sendCruiseStepEnd, receiveCruiseStepEnd, endCruiseStep : Int;
64
65      sendManagePowerStart, receiveManagePowerStart, startManagePower :
66          Int # Int # BatteryLevel # BatteryTemp # Int;
67      sendManagePowerEnd, receiveManagePowerEnd, endManagePower;
68
69      continue;
70
71      newCycle : Bool # Bool # Int # Int;
72
73
74  %------------------------ Process declarations -------------------------------------%
75
76  %----------------------- MainController Processes --------------%
77
78
79  proc MainController(on : Bool, cruise : Bool, throttle : Int, desiredSpeed : Int) =
80      (on)
81      ->  (
82              newCycle(on, cruise, throttle, desiredSpeed)
83              . sendStreamDataStart
84              . sum speed : Int, batteryLevel : BatteryLevel,
85                    batteryTemp : BatteryTemp, solarPower : Int
86              . receiveStreamDataEnd(speed, batteryLevel, batteryTemp, solarPower)
87              .
88              (
89                  (batteryLevel == Full || batteryTemp == Overheating)
90                  ->  conMotorToBattery(0) . conSolarPanelToBattery(0)
91                  <>  continue
92              )
93              .
94              (
95                  (batteryLevel == Empty)
96                  ->  conBatteryToMotor(0)
97                  <>  continue
98              )
99              .
100             (
101                 (solarPower == 0)
102                 ->  conSolarPanelToMotor(0) . conSolarPanelToBattery(0)
103                 <>  continue
104             )
105             .
106             (
107                 (cruise == true)
108                 ->  sendCruiseStepStart(throttle, desiredSpeed, speed,
109                                         batteryLevel, batteryTemp, solarPower)
110                     . sum newThrottle : Int . receiveCruiseStepEnd(newThrottle)
111                     . MainController(true, true, newThrottle, desiredSpeed)
112
113                 + (throttle > 0)
114                 ->  releaseThrottle . ((cruise) -> deactivateCruise <> continue)
115                     . sendManagePowerStart(0, speed, batteryLevel, batteryTemp, solarPower)
116                         . receiveManagePowerEnd
117                         . MainController(true, false, 0, 0)
118
119                 + sum newThrottle : Int . (newThrottle >= 1 && newThrottle <= MAX)
120                 ->  (
121                         pressThrottle(newThrottle)
122                         . ((cruise) -> deactivateCruise <> continue)
123                         . sendManagePowerStart(newThrottle, speed,
124                                                batteryLevel, batteryTemp, solarPower)
125                         . receiveManagePowerEnd
```

```
126                         . MainController(true, false, newThrottle, 0)
127                     )
128
129                 +   (cruise == false && throttle == 0 && speed != 0)
130                 ->  activateCruise(speed) . MainController(true, true, 0, speed)
131
132                 + PowerOff . ((cruise) -> deactivateCruise <> continue)
133                 . MainController(false, false, 0, 0)
134
135                 + PressBrake(speed, batteryLevel, batteryTemp, solarPower, false)
136                 . ((cruise) -> deactivateCruise <> continue)
137                 . MainController(true, false, throttle, 0)
138             )
139         )
140
141     <> switchOn . MainController(true, false, 0, 0);
142
143
144 proc PowerController =
145     sum throttle : Int, speed : Int, batteryLevel : BatteryLevel,
146         batteryTemp : BatteryTemp, solarPower : Int
147     . receiveManagePowerStart(throttle, speed,
148                               batteryLevel, batteryTemp, solarPower)
149     . setMotorRequirePower(throttle)
150     .
151     (
152         (throttle == 0)
153         ->  (
154                 (batteryLevel == Full || batteryTemp == Overheating)
155                 ->  conSolarPanelToMotor(0) . conBatteryToMotor(0) . conMotorToBattery(0)
156                 <>  conSolarPanelToMotor(0) . conBatteryToMotor(0) . conMotorToBattery(speed)
157                     . conSolarPanelToBattery(solarPower)
158             )
159         <>  (
160                 (solarPower > throttle)
161                 ->  conMotorToBattery(0) . conBatteryToMotor(0)
162                     . conSolarPanelToMotor(throttle)
163                     .
164                     (batteryLevel == Full || batteryTemp == Overheating)
165                     ->  conSolarPanelToBattery(0)
166                     <>  conSolarPanelToBattery(solarPower - throttle)
167                 <>  conSolarPanelToBattery(0) . conMotorToBattery(0)
168                     . conSolarPanelToMotor(solarPower)
169                     .
170                     (batteryLevel == Empty)
171                     ->  conBatteryToMotor(0)
172                     <>  conBatteryToMotor(throttle - solarPower)
173             )
174     )
175     . sendManagePowerEnd
176     . PowerController;
177
178
179 proc CruiseStepController =
180     sum throttle : Int, desiredSpeed : Int, currentSpeed : Int, batteryLevel : BatteryLevel,
181         batteryTemp : BatteryTemp, solarPower : Int
182     . receiveCruiseStepStart(throttle, desiredSpeed, currentSpeed,
183                              batteryLevel, batteryTemp, solarPower)
184     .
185     (
186         (desiredSpeed - currentSpeed > SPEED_THRESHOLD)
187         ->  sendManagePowerStart(min(throttle + 1, MAX),
188                                  currentSpeed, batteryLevel, batteryTemp, solarPower)
189             . receiveManagePowerEnd . sendCruiseStepEnd(min(throttle + 1, MAX))
190
191         <>  (currentSpeed - desiredSpeed > SPEED_THRESHOLD)
192             ->  sendManagePowerStart(max(throttle - 1, 0),
193                                      currentSpeed, batteryLevel, batteryTemp, solarPower)
194                 . receiveManagePowerEnd . sendCruiseStepEnd(max(throttle - 1, 0))
195             <>  sendManagePowerStart(throttle, currentSpeed,
196                                      batteryLevel, batteryTemp, solarPower)
197                 . receiveManagePowerEnd . sendCruiseStepEnd(throttle)
```

```
        ) . CruiseStepController;


proc DataStreamController =
    receiveStreamDataStart
    . sum speed : Int . (speed >= 0 && speed <= MAX)
    ->  getSpeed(speed) . sendSpeed(speed)

    . sum batteryLevel : BatteryLevel
    .   getBatteryLevel(batteryLevel) . sendBatteryLevel(batteryLevel)

    . sum batteryTemp : BatteryTemp
    .   getBatteryTemp(batteryTemp) . sendBatteryTemp(batteryTemp)

    . sum solarPower : Int . (solarPower >= 0 && solarPower <= MAX)
    ->  getSolarPower(solarPower) . sendSolarPower(solarPower)

    . sendStreamDataEnd(speed, batteryLevel, batteryTemp, solarPower)
    . DataStreamController;



% ---------------- Auxiliary Processes -------------- %

proc PowerOff = switchOff . conMotorToBattery(0) . conSolarPanelToMotor(0)
                . conSolarPanelToBattery(0) . conBatteryToMotor(0);

proc PressBrake(speed : Int, batteryLevel : BatteryLevel,
                batteryTemp : BatteryTemp, solarPower : Int, braking : Bool) =
                (!braking)
                ->  pressBrake
                    . sendManagePowerStart(0, speed, batteryLevel, batteryTemp, solarPower)
                    . receiveManagePowerEnd
                    . activateBrakes
                    . PressBrake(speed, batteryLevel, batteryTemp, solarPower, true)
                <>  releaseBrake . conMotorToBattery(0) . deactivateBrakes;


%------------------------ Initialization -------------------------------------%

init hide({continue},
  allow(
    {
        switchOn, switchOff, pressThrottle, pressBrake, releaseBrake, activateCruise,
        deactivateCruise, getBatteryLevel, getBatteryTemp, conBatteryToMotor, getSpeed,
        conMotorToBattery, setMotorRequirePower, getSolarPower, conSolarPanelToMotor,
        conSolarPanelToBattery, activateBrakes, deactivateBrakes, sendBatteryLevel,
        sendBatteryTemp, sendSolarPower, sendSpeed, startStreamData, endStreamData,
        startCruiseStep, endCruiseStep, startManagePower, endManagePower, releaseThrottle,
        newCycle, continue

    },
    comm(
      {
        sendStreamDataStart     | receiveStreamDataStart    -> startStreamData,
        sendStreamDataEnd       | receiveStreamDataEnd      -> endStreamData,
        sendCruiseStepStart     | receiveCruiseStepStart    -> startCruiseStep,
        sendCruiseStepEnd       | receiveCruiseStepEnd      -> endCruiseStep,
        sendManagePowerStart    | receiveManagePowerStart   -> startManagePower,
        sendManagePowerEnd      | receiveManagePowerEnd     -> endManagePower
      },
      MainController(false, false, 0, 0) || CruiseStepController ||
      DataStreamController || PowerController
  )));
```

Appendix B

Plain-Text Notation

This appendix contains the plain text notation of the modal formulas, copied directly from the .mcf files.

req1.mcf .

```
true

%%%% 1 %%%%     %true
%%%% a %%%%     %true
    && ([true* . switchOff] mu X. ([!conMotorToBattery(0)] X && <true> true))
    && ([true* . switchOff] mu X. ([!conBatteryToMotor(0)] X && <true> true))
    && ([true* . switchOff] mu X. ([!conSolarPanelToMotor(0)] X && <true> true))
    && ([true* . switchOff] mu X. ([!conSolarPanelToBattery(0)] X && <true> true))
%%%% b %%%%     %true
    && ([true* . switchOff . (!switchOn)*]
        forall  m: Int . val(m > 0)
            => [(conMotorToBattery(m) ||
                conBatteryToMotor(m) ||
                conSolarPanelToMotor(m) ||
                conSolarPanelToBattery(m))] false)
```

req2.mcf .

```
true

%%%% 2 %%%%     %true
%%%% a %%%%     %true
    && ([true* . pressBrake] mu X . ([!conBatteryToMotor(0)] X && <true> true))
    && ([true* . pressBrake] mu X . ([!conSolarPanelToMotor(0)] X && <true> true))

%%%% b %%%%     %true
    && ([true* . pressBrake . (!releaseBrake)*]
        forall m : Int . val(m > 0)
            => [(conBatteryToMotor(m) || conSolarPanelToMotor(m))] false)
```

req3.mcf .

```
true

%%%% 3 %%%%     %true
&&  (
        nu X(level : BatteryLevel = Medium, temp : BatteryTemp = Normal, speed : Int = 0) .
            ([forall level' : BatteryLevel, temp' : BatteryTemp, speed' : Int .
                !getBatteryLevel(level') && !getBatteryTemp(temp') && !getSpeed(speed')]
                X(level, temp, speed)) &&
            (forall level' : BatteryLevel . [getBatteryLevel(level')]
                X(level', temp, speed)) &&
            (forall temp' : BatteryTemp   . [getBatteryTemp(temp')]
                X(level, temp', speed)) &&
            (forall speed' : Int          . [getSpeed(speed')]
                X(level, temp, speed')) &&
            [pressBrake] (val(level != Full && temp != Overheating)
                => (mu Y . ([!conMotorToBattery(speed)] Y && <true> true)))
    )
```

req4.mcf .

```
true

%%%% 4 %%%%     %true
%%%% a %%%%     %true
    && ([true* . (getBatteryTemp(Overheating))]
        mu X . ([!(conSolarPanelToBattery(0))]  X && <true> true))
    && ([true* . (getBatteryTemp(Overheating))]
        mu X . ([!(conMotorToBattery(0))]       X && <true> true))
    && ([true* . (getBatteryLevel(Full))]
        mu X . ([!(conSolarPanelToBattery(0))]  X && <true> true))
    && ([true* . (getBatteryLevel(Full))]
        mu X . ([!(conMotorToBattery(0))]       X && <true> true))

%%%% b %%%%     %true
    && ([true* . getBatteryTemp(Overheating) . !(getBatteryTemp(Normal))*]
        forall m : Int . val(m > 0)
        => [conSolarPanelToBattery(m) || conMotorToBattery(m)] false)

    && ([true* . getBatteryLevel(Full) .
        !(getBatteryLevel(Empty) || getBatteryLevel(Medium))*]
        forall m : Int . val(m > 0)
        => [conSolarPanelToBattery(m) || conMotorToBattery(m)] false)
```

req5.mcf .

```
true

%%%% 5 %%%%     %true
%%%% a %%%%     %true
    && ([!(conBatteryToMotor(0))*]
      forall m : Int . val(m > 0)
        => [conSolarPanelToBattery(m) || conMotorToBattery(m)] false)

%%%% b %%%%     %true
    && ([!(conSolarPanelToBattery(0) || conMotorToBattery(0))*]
      forall m : Int . val(m > 0)
        => [conBatteryToMotor(m)] false)

%%%% c %%%%     %true
    && ([!(conMotorToBattery(0))*]
      forall m : Int . val(m > 0)
        => [conSolarPanelToMotor(m) || conBatteryToMotor(m)] false)

%%%% d %%%%     %true
    && ([!(conSolarPanelToMotor(0) || conBatteryToMotor(0))*]
      forall m : Int . val(m > 0)
        => [conMotorToBattery(m)] false)
```

req6.mcf .

```
true

%%%% 6a %%%     %true
%%%% i %%%%     %true
    && ([true* . getBatteryLevel(Empty)] mu X . ([!(conBatteryToMotor(0))] X && <true> true))

%%% ii %%%%     %true
    && ([true* . getBatteryLevel(Empty)
        . !(getBatteryLevel(Medium) || getBatteryLevel(Full))*]
        (forall m : Int . val(m > 0) => [conBatteryToMotor(m)] false))

%%%% 6b %%%     %true
%%%% i %%%%     %true
    &&  ([true* . getSolarPower(0)]
            ((mu X . ([!conSolarPanelToMotor(0)] X && <true> true)) &&
            (mu X . ([!conSolarPanelToBattery(0)] X && <true> true)))
        )

%%% ii %%%%     %true
    &&  ([true* . getSolarPower(0)]
        [(forall n : Int . val(n > 0) => !getSolarPower(n))*]
        (forall m : Int . val(m > 0)
        => [conSolarPanelToMotor(m) || conSolarPanelToBattery(m)] false))
```

req7.mcf .

```
true

%%%% 7 %%%%     %true
&&  (
        nu X(level : BatteryLevel = Medium, temp : BatteryTemp = Normal,
             solarPower : Int = 0, speed : Int = 0) .
            ([forall level' : BatteryLevel, temp' : BatteryTemp,
                     solarPower' : Int, speed' : Int .
                !getBatteryLevel(level') && !getBatteryTemp(temp') &&
                !getSolarPower(solarPower') && !getSpeed(speed')]
                X(level, temp, solarPower, speed)) &&
            (forall level' : BatteryLevel . [getBatteryLevel(level')]
                X(level', temp, solarPower, speed)) &&
            (forall temp' : BatteryTemp   . [getBatteryTemp(temp')]
                X(level, temp', solarPower, speed)) &&
            (forall solarPower' : Int . [getSolarPower(solarPower')]
                X(level, temp, solarPower', speed)) &&
            (forall speed' : Int . [getSpeed(speed')]
                X(level, temp, solarPower, speed')) &&

%%%% a %%%%     %true
            (forall requiredPower : Int . [setMotorRequirePower(requiredPower)]
                (val(requiredPower <= solarPower)
                => mu Y . ([!conSolarPanelToMotor(requiredPower)] Y && <true> true))) &&
            (forall requiredPower : Int . [setMotorRequirePower(requiredPower)]
                (val(requiredPower <= solarPower && level != Full && temp != Overheating)
                => mu Y . ([!conSolarPanelToBattery(solarPower - requiredPower)] Y &&
                <true> true))) &&

%%%% b %%%%     %true
            (forall requiredPower : Int . [setMotorRequirePower(requiredPower)]
                (val(requiredPower >= solarPower)
                => mu Y . ([!conSolarPanelToMotor(solarPower)] Y && <true> true))) &&
            (forall requiredPower : Int . [setMotorRequirePower(requiredPower)]
                (val(requiredPower >= solarPower && level != Empty)
                => mu Y . ([!conBatteryToMotor(requiredPower - solarPower)] Y &&
                <true> true)))
    )
```

req8.mcf .

```
true

%%%% 8 %%%%     %true
&&  ([true*] forall t : Int . val(t > 0) => [pressThrottle(t)]
    mu X . ([(!setMotorRequirePower(t))] X && <true> true))
```

req9.mcf .

```
true

%%%% 9 %%%% %true
    && nu X(speed : Int = 0, throttle : Int = 0) .
        ([forall speed' : Int, throttle' : Int .
            (!getSpeed(speed') && !pressThrottle(throttle') && !releaseThrottle && !switchOn)]
            X(speed, throttle)) &&
        (forall speed' : Int . [getSpeed(speed')] X(speed', throttle)) &&
        (forall throttle' : Int . [pressThrottle(throttle')] X(speed, throttle')) &&
        ([releaseThrottle || switchOn] X(speed, 0)) &&
        (forall speed' : Int . [activateCruise(speed')]
            val(speed > 0 && speed == speed' && throttle == 0))
```

req10.mcf .

```
true

%%%% 10 %%%%    %true
    &&  (
            nu X(desiredSpeed : Int = 0, power : Int = 0) .
                ([forall desiredSpeed', power' : Int .
                    !setMotorRequirePower(power') && !activateCruise(desiredSpeed') &&
                    !deactivateCruise]
                    X(desiredSpeed, power)) &&
                (forall desiredSpeed' : Int .
                    [activateCruise(desiredSpeed')]
                    X(desiredSpeed', 0)) &&
                ([deactivateCruise] X(0, power)) &&
                (forall power' : Int . [setMotorRequirePower(power')]
                    X(desiredSpeed, power')) &&
                (
                    forall speed : Int . [getSpeed(speed)]
                    (   val(desiredSpeed > 0)
                        => (
%%%% a %%%%     %true
                                    (val(desiredSpeed - speed > SPEED_THRESHOLD)
                                        => (mu Y . ([!setMotorRequirePower(min(power + 1, MAX)) &&
                                                    !deactivateCruise]
                                                    Y && <true> true))) &&
%%%% b %%%%     %true
                                    (val(speed - desiredSpeed > SPEED_THRESHOLD)
                                        => (mu Y . ([!setMotorRequirePower(max(power - 1, 0)) &&
                                                     !deactivateCruise]
                                                     Y && <true> true))) &&
%%%% c %%%%     %true
                                    (val(abs(speed - desiredSpeed) <= SPEED_THRESHOLD)
                                        => (mu Y . ([!setMotorRequirePower(power) &&
                                                    !deactivateCruise]
                                                    Y && <true> true)))
                                )
                    )
                )
        )
```

req11.mcf .

```
true

%%%% 11 %%%%    %true
    && ([true*] forall n : Int . [activateCruise(n)] [!(deactivateCruise)*]
        [(exists m : Int . pressThrottle(m)) || pressBrake || switchOff || releaseThrottle]
        mu X . ([!deactivateCruise] X && <true> true))
```

req12.mcf .

```
true

%%%% 12 %%%%    %true
&& ([true*]
    forall desiredSpeed : Int . val(desiredSpeed > 0)
    => [activateCruise(desiredSpeed) . !deactivateCruise*] <true* . deactivateCruise> true)
```

req13.mcf .

```
true

%%%% 13 %%%%    %true
%%%% a  %%%%    %true
&&  ([forall desiredSpeed : Int . !activateCruise(desiredSpeed)* . deactivateCruise] false)

%%%% b  %%%%    %true
&& ([deactivateCruise . forall desiredSpeed : Int . !activateCruise(desiredSpeed)*
    . deactivateCruise] false)
```

req14.mcf .

```
true

%%%% 14 %%%%    %true
%%%% a  %%%%    %true
    && ([true*] forall n : BatteryLevel . [getBatteryLevel(n)]
        mu X . ([!sendBatteryLevel(n)] X && <true> true))

%%%% b  %%%%    %true
    && ([true*] forall n : BatteryTemp . [getBatteryTemp(n)]
        mu X . ([!sendBatteryTemp(n)] X && <true> true))

%%%% c  %%%%    %true
    && ([true*] forall n : Int . [getSolarPower(n)]
        mu X . ([!sendSolarPower(n)] X && <true> true))

%%%% d  %%%%    %true
    && ([true*] forall n : Int . [getSpeed(n)]
        mu X . ([!sendSpeed(n)] X && <true> true))
```

Appendix C

Changelog

This appendix contains the modifications of the document.

v1.0

1. added a changelog as an appendix
2. chapter 4: added subsections for each part in this chapter.
3. chapter 4, page 10: added a short description of the responsibilities of each component in the Main Controller Internal Architecture.
4. chapter 6, page 17: modified the chapter description.
5. chapter 7, page 21: modified the chapter description and corrected some spelling/grammar mistakes.