Development of a Cruise Control in UML using Rhapsody

a design project in order to familiarise with this new development tool

G.R. de Boer
Report of IOO

Supervisor: dr. ir. J. Broenink

March 2001
Report Nr. 002R2001
Control Laboratory
Electrical Engineering Department
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands
Summary

This project has the objective of familiarising with Rhapsody, a Unified Modeling Language (UML) based development tool. The additional goal of embedding the developed software in a TMS-320LF2407 board has appeared to be out of the scope of the project. It turned out that Rhapsody assumes a Real Time Operating System (RTOS) at the target, for which the TMS board did not qualify.

During the project, the Rapid Object-Oriented Process for Embedded Systems (ROPES) has been followed. This method prescribes ‘cyclic prototyping’, an iterative approach allowing early testing of analysis models, even though they are not complete (Douglas, 2000). To discover the possibilities of this tool, it has been chosen to use it to model a Cruise Control (CC).

The objective of getting acquainted with Rhapsody proved to be successful. A fully functional CC has been developed and comprehensive knowledge of the tool has been gathered and documented. It is recommended that further research is dedicated to improve the control law used for speed maintenance, as well as to development of a graphical user interface allowing plotting of the system output and easy event insertion. Furthermore, development of a framework for Rhapsody, allowing embedding in non-RTOS, is recommended.

Samenvatting

Dit project heeft als doel het bekend worden met Rhapsody, een ontwikkel-omgeving gebaseerd op de Unified Modeling Language (UML). Het bijkomstige doel, de ontwikkelde software te embedden in een TMS-320LF2407 board is gebleken buiten de reikwijdte van het project te vallen. Rhapsody bleek een Real Time Operating System (RTOS) als target te verwachten. Het TMS-bord voldoet niet aan die eis. Gedurende het project is de Rapid Object-Oriented Process for Embedded Systems (ROPES) methode gevolgd. Deze methode schrijft voor om cyclisch prototypes te ontwerpen; een iteratieve aanpak die vroege tests van modellen mogelijk maakt, ook als ze nog niet compleet zijn (Douglas, 2000)

Om de mogelijkheden van deze ontwikkel omgeving te ontdekken, is de omgeving ingezet om een Cruise Control (CC) te ontwikkelen.

Het doel om bekend te geraken met Rhapsody is succesvol gebleken. Een volledig werkende CC is ontwikkeld en er is veel kennis omtrent de ontwikkelomgeving vergaard en gedocumenteerd. Het wordt aanbevolen om verder onderzoek te richten op verbetering van het regelalgoritme dat de snelheid probeert vast te houden. Eveneens kan onderzoek gericht worden op het ontwerpen van een grafische User Interface om het plotten van de uitgangswaarden mogelijk te maken, evenals makkelijker event-generatie. Ook zou een framework dat embedden in een niet-RTOS mogelijk maakt, ontwikkeld kunnen worden.
## Contents

1. Introduction ......................................................................................................................................... 1

2. ROPES ................................................................................................................................................. 5

3. Analysis ................................................................................................................................................ 7
   3.1 Cruise Control .................................................................................................................................. 7
   3.2 Use Cases ......................................................................................................................................... 8
   3.3 Object Model Diagram ................................................................................................................... 11
      3.3.1 Implementation Note .................................................................................................................. 12
      3.3.2 Skills Needed ............................................................................................................................. 13

4. Design ................................................................................................................................................. 15
   4.1 State Charts .................................................................................................................................... 15
      4.1.1 Simulation and Animation ......................................................................................................... 19
      4.1.2 Implementation Note .................................................................................................................. 20
   4.2 20-SIM ............................................................................................................................................ 20

5. Implementation ................................................................................................................................. 23
   5.1 Manual Implementation of the Car model ...................................................................................... 23
   5.2 Tuning the Controller ..................................................................................................................... 25

6. Testing and Results ........................................................................................................................... 27
   6.1 Testing ............................................................................................................................................ 27
   6.2 Results ............................................................................................................................................ 28

7. Conclusions and Recommendations ................................................................................................. 31
   7.1 Conclusions .................................................................................................................................... 31
   7.2 Recommendations .......................................................................................................................... 31

Appendix A: Troubleshooting for Rhapsody ........................................................................................... 33

References ................................................................................................................................................... 35
1 Introduction

This project has the objective of familiarizing with Rhapsody; a Unified Modeling Language (UML) based development tool. It allows high level designing and structuring of embedded software. To discover the possibilities of this tool, it has been chosen to use it to model a Cruise Control (CC). CC has become very common in cars nowadays, making it very suitable to serve as an everyday-life example of a controlled system. A CC contains the basics of loop control as well as some user interfacing. The CC can be positioned in a schematic user-car system as depicted in fig. 1.1.

![User Interface > Cruise Control](image)

**Figure 1.1** A schematic representation of the position of the CC in a user-car system

Rhapsody provides for a means of defining a system with use cases, sequence diagrams, class diagrams and state charts. Use cases are specific situations the system can encounter; global sketches of the system functions. Sequence diagrams allow a systematic definition of the sequences of events the system has go through, in order to realize the desired behavior. The class structure of the model can be defined in the Object Model Diagram. Eventually, the embedded software is completed with state charts.

The development strategy has been taken from the Rapid Object-Oriented Process for Embedded Systems (ROPES) approach (Douglas, 2000). This method prescribes ‘cyclic prototyping’, an iterative approach allowing early testing of analysis models, even though they are not complete. The following figure depicts this cycle.

![ROPES Cycle](image)

**Figure 1.2** The ROPES cycle (Douglas, 1999, p 160)
An elaborate explanation of this method is given in Chapter 2.

Initially, the goal of the project was embedding the software in a TMS-320LF2407 programmable board, using a Code Composer. This initial project strategy is depicted schematically as in figure 1.3.

Later, this goal has been renounced because of a mismatch between the output assumptions of Rhapsody and the features of the TMS-board. Rhapsody assumes a Real Time Operating System (RTOS), and the TMS-board does not qualify for that. The project has been redirected as shown in figure 1.4.
The TMS-board could not be addressed without writing a new framework, replacing the OXF-framework Rhapsody normally includes to the design. This would be far too extensive for this assignment, so it has been decided to use a model of a car. This way, it can still be tested if a proper CC can be developed using Rhapsody.

Chapter 3 will contain the Analysis of the development process. More details on the CC are given in section 3.1. The tools and skills that were available or needed during the course of the project are reviewed in section 3.2. Besides Rhapsody and the necessary C++, these tools also comprise 20-SIM for the modeling of a car. Chapter 4 covers the Design of the CC in Rhapsody. Chapter 5 elaborates on the implementation of the CC, using the tools mentioned in section 3.2. Also it reports on the tuning of the CC (section 5.2). The testing of the CC and the results are given in chapter 6. Chapter 7 contains the conclusions and recommendations of the project.
2 ROPES

This project owes its strategy to the ROPES approach. The structure prescribed in the approach returns in the chapter-arrangement of this report. A more detailed ROPES-cycle is given in figure 2.1.

![Figure 2.1 A detailed ROPES cycle (Douglas, 1999, p 167)](image)

The primary purposes of any development process are to

(1) increase the quality of the end product
(2) improve the repeatability and predictability of the development effort
(3) decrease the effort required to develop the end product at the required level of quality (Douglas, 1999).

As can be seen, the approach breaks the software development into four primary phases: analysis, design, implementation and testing. In as much as the sub-phases apply to this development-project, they are deployed as well.

Iterative project development requires sub-goals to be recognized in the project. Every cycle, new elements have to be included in the design, making it resemble the desired design more and more. The following sub-goals in order to achieve the total design have been distinguished:
• Define the Use Cases, Sequence Charts and Class structure
• Realize data transport by the class structure
• Set a speed/Resuming a speed
• Control the Speed
• Rewrite the C++ code to C code
• Realize the embedded system on the TMS-board

the last two stages have later been replaced by:

• Model a car in 20-SIM
• Import the model in Rhapsody
• Feed the countermeasure to the car engine
• Tuning the controller, thus keeping the speed constant

The Analysis part of these stages is documented in chapter 3. The Design is covered in chapter 4. Chapter 5 will report on the Implementation and the Testing is accounted for in chapter 6. This way the ROPES structure can be recognized in the structure of the report as well.
3 Analysis

This chapter contains analysis of the CC, needed to implement it in Rhapsody. First, use cases will be found. To obtain specific use cases for a CC, as well as providing the reader with a general understanding of the working of a CC, section 1.1 gives a description of its functioning in words. Words that give notice of possible objects that have to be taken into account are underlined. Possible use cases can be recognized in the text as being bold.

3.1 Cruise Control

To keep the speed of an automobile at a certain level in the first place requires knowledge of the actual speed. In the case of a CC, the speed information is acquired using a speed sensor. This actual speed has to be compared to the desired speed in order to know whether the speed is actually being maintained. To maintain the desired speed, the CC has to be able to influence the amount of gas fed to the engine. Thus the output of the CC is a signal which gives a measure for the amount of gas. With this signal a servo can be controlled which is attached to the throttle-lever at the engine.

Pushing a button labeled ‘On/Off’ while driving can set the desired speed. The CC unit will at that moment acquire and store the actual speed. This speed will then serve as the value for the desired speed, and the CC will try to maintain that speed, continuously comparing it to the actual speed.

Next, conditions for deactivating the CC have to be specified. The driver can simply turn off the CC by pressing the ‘On/Off’ button again. More important is the deactivation of the CC when braking. This way, it is prevented that the CC will try to countermeasure the speed loss, in this case initiated (and therefore wanted) by the driver.

After deactivation of the CC the user can reset the speed at a new level, by pressing ‘On/Off’ again at a desired level of speed, or he can resume the speed level set previously.

Finally some additional functions can be implemented, for instance ‘CC up’ or ‘CC down’. Using these buttons, the user can fine-tune the set speed to the desired level.
The indicated objects and use cases can be translated into the following list of objects and use cases:

<table>
<thead>
<tr>
<th>Objects</th>
<th>Use Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>Acquire actual speed</td>
</tr>
<tr>
<td>Button panel</td>
<td>Store actual speed</td>
</tr>
<tr>
<td>Engine</td>
<td>Compare speed</td>
</tr>
<tr>
<td>Speed Sensor</td>
<td>Keep speed constant</td>
</tr>
<tr>
<td>Store speed</td>
<td>Activate/Deactivate CC</td>
</tr>
<tr>
<td>Calculate Difference/Throttle</td>
<td>Resume speed</td>
</tr>
<tr>
<td>Controller</td>
<td>Fine-tune speed</td>
</tr>
</tbody>
</table>

**Table 3.1** The Objects and Use Cases recognized in the Cruise Control

These use cases and objects have almost all found their way in the Use Case Diagram or Object Model Diagram. Only those that do not have to be designed in software have been omitted in the Object Model Diagram (Button Panel, Speed Sensor and the Engine (the class Car will later model the behavior of the engine in a car)).

### 3.2 Use Cases

In this modeling stage it is difficult and not yet necessary to use real sensor-signals, and producing lifelike outputs. Hence the objects found in table 3.1 reduce to:

- **GenerateSpeed** (a temporary replacement for the sensor)
- **SignalConditionerIn** (shapes the sensorreading to a usable (controllable) form)
- **ProcessUserCommand** (processes inputs from user (provided by Input), and distributes the accompanying tasks to the system)
- **StoreSpeed** (captures a sample of the actual speed when the user sets the speed)
- **Controller** (controls the speed of the car trying to maintain the setpoint)
• **SignalConditionerOut** (shapes the output of the controller to a fit range for the servo on the speed lever, or the electro-motor)
• **Display** (temporary object, sorting out the interesting signals in the system and displaying them in an orderly manner for inspection)

The first step in the design was to define the actors in a CC system as well as the Use Cases identified in the previous chapter. Second came the identification of relationships between users and Use Cases, and the generalizations that can be made between the Use Cases. This resulted in the diagram of figure 3.1.

![Figure 3.1](image.png)

**Figure 3.1**  The Use Case Diagram of the Cruise Control

As can be seen, this diagram shows the different functions the CC is capable of, as well as the locations at which the user can intervene in its functioning.

The use of this diagram is merely explanatory. No code is generated so far; the problem has only been outlined en schematized. It can be used as a reference for total overview, but also as an
illustration of how the programmer regards the problem (e.g. to the project manager). Only when a use case has the same name as a class defined in the Sequence Diagram or Object Model Diagram, Rhapsody suggests merging them. This way relations defined in the Use Case Diagram do have some effect on the code, though hardly worth mentioning.

The Sequence Diagram gives a means for sketching possible sequences of action the system can go through. One Sequence Diagram has been created, comprising the reactions of the system to each button the user can press. Again it has to be noted that this chart does not narrow down the design, it merely visualizes a possible sequence of actions. Figure 3.2 shows the Sequence Diagram.

![Sequence Diagram of the Cruise Control]

**Figure 3.2** The Sequence Diagram of the Cruise Control
As can be seen, the ProcessUserCommand class distributes the user commands to the rest of the system. The evOn and evOff to the SignalConditionerOut are necessary since the CC has to release its influence on the speed directly, when the user adjusts the speed himself.

3.3 Object Model Diagram

The Object Model Diagram is where the structure of the software is outlined. It is where the user defines the classes and their mutual relations. The following figure shows the class-structure for the CC.

Whenever one class needs data from another class for its operation, it has to be connected to that class with a relation or aggregation in the Object Model Diagram. So the Object Model Diagram is also where the infrastructure for the data-transport is defined.

In the diagram, the straight lines represent the data-flow, whereas the curved lines depict signal lines for on and off switching, event generation or monitoring, i.e. the control-flow. This is not a UML feature, but the programmer is completely free in drawing the lines. It has been chosen for clearness of the picture, in a way similar to K.C.J. Wijbrans, 1993.

![Object Model Diagram of the Cruise Control](image)

Figure 3.3 The Object Model Diagram of the Cruise Control
The ‘1-character’ on each end of the relation indicates the multiplicity of the relation, meaning the number of objects that can be made from a class.

As it became clear that the design and testing of the CC would be completely in software, and a model of a car would have to be implemented in Rhapsody, a new Object Model Diagram was created. The new diagram is given in figure 3.4. The class has been connected to Input and Output in order to close the control loop. ProcessUserCommand has also been connected to Car, since the user has to be able to address the car without mediation of the CC as well (speed up, brake, etc.). Car has a connection to Display too, since monitoring of its internal variables has to be possible.

**Figure 3.4** Object Model Diagram of the CC with a model of a car

### 3.3.1 Implementation Note

When connecting classes with aggregations, data transport and event distribution can be done as soon as the ‘part of’ classes are ‘set’ in the ‘whole’ or ‘aggregated’ class. This is done in the manner of the set-functions defined in the tutorial (Ilogix, 1999, p 4-17). When a relation is used to connect the classes, Rhapsody automatically sets the connected classes. However, Rhapsody
does not set the pointers to those classes; this has to be done manually. For every connection the programmer has to write a code similar to:

```cpp
p_Class1->setItsClass2(p_Class2);
```

This code has to be added to the model in the initialization tab in Components, YourModel, Configuration, YourActiveConfiguration.

Please note that, depending on the type of the relation (one-way or symmetric), one or two pointers need to be set for each relation.

Appendix A gives a list of the most time consuming and disturbing difficulties that were encountered during the use of Rhapsody. For problem handling refer to this appendix.

### 3.3.2 Skills Needed

Implementing the CC in Rhapsody requires the user to be familiar with the program. The tutorial and reference guides are of help in gaining familiarity, although they are far from sufficient. Some of the options in the wide range of feature-windows are hardly documented, probably assuming enough programming-background for those options to become trivial. Since this background was substandard at first, a proper understanding and operation of Rhapsody advanced slowly. Knowledge of C++ is also required, since the actual function of the states, the body of the objects defined, has to be manually programmed. Maybe in-depth knowledge of C++ would have indeed made the above-mentioned options trivial, but in my case the opposite applied. The attended lectures in C++ have hardly covered object orientated programming, so most of the understanding was gained during the design. The first four weeks have mainly been dedicated to familiarizing with Rhapsody and object-orientated programming.

Regarding the scheme of fig. 1.1, the CC and User Interface will first be implemented using Rhapsody. On a very high level the programmer is able to abstract and define the system, some features only clarifying the problem to the programmer, other features generating code for the program.

Development in Rhapsody is done in three stages:

- defining Use Cases and specifying a Sequence Diagram
- creating an Object Model Diagram of the system
- designing the Statecharts

The first two stages can be recognized as the Analysis part of the ROPES approach, whereas the last stage is part of the Design.

The first stage, the Use Case and Sequence Diagram, mainly outlines the problem to the programmer and allows him to narrow down and pinpoint the exact design-problem he is dealing with. The second stage, the Object Model Diagram, creates a framework of classes and their
mutual dependencies, outlining the structure of the program and the data-flow. The third stage, the statecharts, is where the program is completed. All the different states the system can be in, have to be defined, as well as the actions that are related to those states. The programmer can choose to make one huge statechart, or define several statecharts, e.g. one for each class defined in the abstract-part. Regarding the iterative development approach of ROPES, the latter is recommended. This way gradual refining is easy to realize.

The car was to be modeled in hardware, using an electro-motor. The Rhapsody design of the CC would be embedded in the TMS-board, and the user interface would be implemented with a set of buttons. Embedding the software in the TMS-board requires some knowledge of C, since the Code Composer for the TMS-board uses C. The conversion of C++ to C however, can be done using Appendix A of J.M. van Drunen, 2000.

As became clear during the project, 20-SIM had to be put into action where the Code Composer fell short. The car now has been modeled with bond graphs. This model has been implemented in Rhapsody, thus realizing a complete user-CC-car system.
4 Design

The design in software becomes of interest, since the functionality and requirements of the CC have been analyzed. Section 4.1 gives notice of the state charts that define the desired behavior of the CC. The car model developed with 20-SIM is covered in section 4.2.

4.1 State Charts

The next stage in the UML programming process is the definition of the statecharts. The exact behavior of the CC is fixed by defining the states and transitions the CC can be in, as well as the actions that accompany them. It has been decided to assign a statechart to each class. This way it is prevented that the statecharts become complex. Furthermore, gradual refinement is possible, as stated in chapter 3. The following figures show the statecharts that have been created.

At first the statechart of the straightforward classes (SignalConditionerIn and GenerateSpeed) will be reviewed. These classes loop one single state, repeatedly executing the function defined in the body of the state. Figure 4.1 shows the form of these classes.

![State Chart of SignalConditionerIn and GenerateSpeed]

The loop time of 100 ms (tm(100)) is maintained in the whole design of the CC. This way the total loop time of the CC results in 500 ms, because a signal goes through 5 classes from input to
output (see the Object Model Diagram). That this loop-time satisfies the demands of the design will be discussed in the following section.

The following list shows the functions each of these classes loops:

- **SignalConditionerIn**
  - GetSpeed(), ShapeSpeed()
- **GenerateSpeed**
  - GenerateSpeed()

**SignalConditionerOut** has a slightly different statechart, since it has to have the ability to switch off. This is realized in the manner of figure 4.2.

![Figure 4.2](image)

**Figure 4.2** The State Chart of SignalConditionerOut

**SignalConditionerOut** executes the operations GetOutput() and ShapeOutput().

A look at **ProcessUserCommand** shows that the whole system can be mastered by inserting 5 events in this class:

- **evSetSpeed** (the CC is activated to maintain a new speed (the ‘On/Off’ button) or deactivated)
- **evRestoreSpeed** (the CC is activated to maintain previous speed)
- **evCCUp** (the set speed is increased by 1 km/h)
- **evCCDown** (the set speed is decreased by 1 km/h)
- **evBrake** (the brake is used, and the CC is deactivated)

Figure 4.3 shows how these events are included in the statechart of **ProcessUserCommand**.
The class ProcessUserCommand is where the entire system is coordinated. The other classes are invoked from here. Entering the Idle state deactivates the Controller and the SignalConditionerOut, exiting the Idle state activates it. Entering the Store state generates an event in the class StoreSpeed (evStoreSpeed()). It can be seen that evSetSpeed() results in entering the Store state, and also in returning to the idle state (when evSetSpeed() reoccurs within 100 ms, since after that time the Controlling state is entered). This has been done for the sake of completeness, since it could occur that the user accidentally presses the On/Off button and corrects that within 100 ms. The controlling state does not contain code, since exiting the idle state already started the controller.

It can be seen that (when in the Controlling state) the event evCCUp() and evCCDown() result in evIncrease() and evDecrease() in the class StoreSpeed. Pressing ‘On/Off’ (evSetSpeed()), hitting the brake or adjusting the speed in another way, results in returning to the state idle. It has to be noted that de CC has to react to accelerating just as to hitting the brake, or even clutching. As soon as the driver wants to adjust the speed in any way, the CC has to deactivate. So pressing either one of these levers results in the same event: evBrake().

Figure 4.3 The State Chart of ProcessUserCommand
The StoreSpeed class has been defined as follows. The evStoreSpeed(), generated in ProcessUserCommand, results in taking a sample from the speed in SignalConditionerIn and storing it in the variable SetSpeed with the following function (the operation GetSpeed(), invoked in the state On):

\[
\text{SetSpeed} = \text{itsSignalConditionerIn} -> \text{GiveSpeed}();
\]

The class SignalConditionerIn reacts to this call by returning the actual speed:

\[
\text{Return Speed};
\]

![The State Chart of StoreSpeed](image)

**Figure 4.4** The State Chart of StoreSpeed

The reactions to evIncrease() and evDecrease() are respectively SetSpeed++ and SetSpeed--. This way the driver can tune the set speed by 1 km/h.

Finally the main function of the CC will be shown: the controller. Every 100 ms the controller refreshes the values it needs for speed-control: Speed and SetSpeed (in the controller this is called SetPoint). With these values, it continuously loops a cycle of comparing and controlling
(for now no real countermeasure is necessary, since we do not know what engine we are going to control). The control-law used in the controller is a proportional one. As soon as the car model is implemented in Rhapsody, this control-law will be tuned to control the speed of the car. Figure 4.5 shows the state chart of the controller. It can be seen that this class contains three parallel processes, each with a loop time of 100 ms.

![State Chart of Controller](image)

**Figure 4.5**  The State Chart of Controller

This way a simple Cruise Control has been realized. The next section will give some comments on the animation of the system (one of the features of Rhapsody).

### 4.1.1 Simulation and Animation

When simulating the model, Rhapsody allows insight via a graphical way in the operation of the system, by means of animation. One can choose to animate any statechart of the system, as well as a sequence diagram. This displays the sequence of the events the system goes through during
animation. This way, one can quickly see if and how the system functions and check if that corresponds with the demands.

Simulating and animating a model can be done as described in the tutorial. However, since the tutorial does not provide for comprehensive explanation of the instructions, some remarks regarding indistinct issues are given here.

- In the folder Component/Configuration, the ‘initialization-tag’ shows all the constructed classes, with a check-box. Only the checked classes will be included in the animation. If a class has a function whatsoever in the animation, check its check-box. In general one can simply check all check-boxes, to avoid failing animations.
- Animation is only possible when the simulation is running.
- When the initial state of a class invokes other classes (e.g. generating an event), be sure to start with a few milliseconds delay, before entering the initial state. This way, indefinite errors are prevented, caused by an incomplete data-structure.

4.1.2 Implementation Note

The 40 ms. before entering the on state in ProcessUserCommand has been done because the startup state, idle, directly inserts events into another class (Controller and SignalConditionerOut). They first have to be constructed before they can be addressed. Without this delay, a memory exception occurs, caused by an incomplete data-structure.

A presence-check should be implemented, holding the program until the addressed class is present (only for a limited time, since it could occur that the class does not exist due to caused by other than startup problems)

4.2 20-SIM

Since controlling the speed of a real electro-motor is out of the picture, the design of the CC will now be tested in software. 20-SIM has been used to create a model of a car and engine, and to generate C++ code for that model so that it can be implemented in Rhapsody. The model and the reductions that were applicable will be discussed in this section.

At first a complete picture of the relevant parts of the system of a car and engine have been modeled. Relevant are the dynamic properties of the engine as well as the weight and total friction of the car. However, in this stage of the design it is of little importance whether we use a
combustion engine or an electrical engine. For now, we will apply a simple modulated effort source. This resulted in the bond graph model of figure 4.6.

![Bond Graph Model](image)

**Figure 4.6** Bond graph model of a car and engine

Realistic parameters of the electro-motor have been found in Breedveld and Van Amerongen, 1994. When we regard the transfer function of the engine (Eq. 4.1) the time constant of the engine can be found.

\[
H(s) = \frac{\omega(s)}{u(s)} = \frac{K_T}{R_e + R_m + s(R_m J_m + R_m L) + s^2 L J_m} = \frac{1}{K_T} \frac{1}{1 + s R_m J_m K_T^{-2}} \quad \text{(Eq. 4.1)}
\]

Applying the parameters found in Breedveld and Van Amerongen, 1994, pg. 143, table 19.1, the resulting time constant of the engine is roughly in the range of $10^{-4}$ s. For a car of about 1000 kg, an approximation of the time constant can be made by the following imaginary experiments.

Imagine a car of 1000 kg. released at a speed of 100 km/h, and try to predict the time the car needs to reach $1/e \approx 0.37$ times that speed. This can be anywhere between 30 and 90 seconds, or maybe even more. Now imagine the same car speeding from stand to 100 km/h, and predicting when it will reach $1-1/e \approx 0.63$ times that speed. This will occur somewhere between 5 and 10 seconds. We will assume a worst case, and therefore use the smallest time constant of 5 s.

This second experiment is not completely valid, since most cars do not saturate at a speed of 100 km/h, but go beyond that speed. However, it does indicate that the time constant of the engine is exceeded by far by the time constant of the car. Hence the time constant of the engine is of no
significance compared to this time. Reducing the model using these considerations, results in the bond graph model of figure 4.7.

Figure 4.7  A simple bond graph representation of a car and engine

When applying the time constant of 5 s, the resistance of the car follows from

\[
\tau = \frac{1}{RC} = \frac{I}{R} \iff R = \frac{I}{\tau} = \frac{1000}{5} = 200 \text{ Ns/m} \quad \text{(Eq. 4.2)}
\]

This model can quite adequately function as a representation of the engine-driven car. The next chapter will elaborate on the implementation of this model into Rhapsody.
5 Implementation

Since no software is actually being embedded in this project, the implementation reduces to inserting the 20-SIM model in Rhapsody. This implementation into Rhapsody turned out to be problematic since some included C++ library files contained a ‘name clash’. Rhapsody does not allow variables and classes having the same name, which occurs in e.g. stdio.h. This made manual implementation inevitable. How this has been done, is described in section 5.1. Tuning of the controller is reported in section 5.2.

5.1 Manual Implementation of the Car model

Analyzing the 20-SIM generated code and its equations, the following set of model equations emerged.

\begin{align*}
\text{<initialization> //set all the variables zero the first time} \\
\text{impulse} &= \text{impulse} + h\cdot\text{force} \quad (\text{Eq. 5.1}) \\
\text{speed} &= \frac{\text{impulse}}{\text{mass}} \quad (\text{Eq. 5.2}) \\
\text{rforce} &= \text{resistance}\cdot\text{speed} \quad (\text{Eq. 5.3}) \\
\text{force} &= \text{eforce} - \text{rforce} \quad (\text{Eq. 5.4})
\end{align*}

The explicit Euler integration method has been applied as is used in eq. 5.1, with \text{eforce} the force produced by the engine and \text{rforce} the force resulting from the friction of the car. These equations will be looped continuously in the model of the car. The loop time will be 100 ms. just as the rest of the system. When regarding the design demand of a sample time of at least ten times the time constant of the model (Van Amerongen and De Vries, 1999), this demand has been satisfied, since the time constant of the car has been chosen 5 s. This is exactly ten times the loop time of the controller, as this was set to 0.5 s. (refer to chapter 4).

Now, realistic values of the parameters of the car have to be found, in order to realize a life-like model. The model in 20-SIM has been applied in order to find realistic parameters for the model of the car. First of all, the time constant of the car has been determined. When accelerating the car up to 100 km/h, the time constant is the time at which the speed is 1/e times 100 km/h (63 km/h).
The following figure shows the car speeding up to 100 km/h. with a resistance of 200 Ns/m and a force of 5600 N. This shows that the time constant is approximately 5 s, which corresponds with the calculations in chapter 4.

![Speeding up to 100 km/h](image)

**Figure 5.1** determination of the time constant of the car

The maximum force the car can deliver, when we assume the resistance of the car to be 200 Ns/m, can be found when we suppose that the car has a maximum velocity of approximately 180 km/h. The model in 20-SIM shows that this speed is reached with a force of 10,000 N. This is assumed to be the maximum force the engine can deliver.

If we index the throttle to be fed to the car from 0 to 100, 100 has to correspond with the maximum force, so incrementing the throttle by 1 means incrementing the force by:

\[ 100 = 10000 \text{ N} \rightarrow 1 = 10000 / 100 \text{ N} = 100 \text{ N} \]  

(Eq. 5.6)

Finally, the external forces (wind, slopes) have to be inserted in the model. It has been chosen to alter the `eforce` with a step size of 1000. This way, every step results in visible speed. With these values an estimation of the time constant of a car, based on this first order model follows from eq. 5.8.
5.2 Tuning the Controller

Now that a model of the car has been derived and included in Rhapsody, the CC needs to control the speed of the car model. Since the control-law applied in the controller is a proportional one, its K-factor will have to be optimized. For convenience it has been chosen to tune this factor empirically. Within three attempts the K-factor has been found to be 35, or 36 at the most. It has been set to 35 since 38 resulted in instable behavior, and we wish to have stability with a margin.
6  Testing and Results

Now that a completely functional system has been developed, it has to be tested. However, since the output of the model during animation is hardly presentable when it comes to reactions in time, presentation of the test results is quite problematic. Rhapsody only prompts numbers for the values of some variables. A proper plot function for graphical presentation of the results has not been developed yet. Therefore testing and report of results can be done only in written form, without supporting illustrations. Section 6.1 will elaborate on the sequence of events fed to the system in order to test its behavior, and section 6.2 will report on the reactions of the system. Only when relevant, a semi-screendump (handwritten) of the output of Rhapsody is included for illustration.

Note that, in order to show km/h, the internal variable Speed has been multiplied by 3.6 in the class Display.

Testing

All the vital functions the CC has to possess, have been programmed and have been tested. First the model of the car will be checked. It has to accelerate and decelerate like a normal car, without the speed of the car becoming negative as a result of the negative force of the brakes. A second test is, when driving at normal speed, setting the speed and observing how the set speed is maintained. Now the speed has been set and the controller controls the speed, any speed adjustment done by the user has to result in a nullification of the speed, without losing the set speed. This has been tested, once by increasing the speed to fast driving and once by braking. Next the user can resume the previously set speed. This has been tested by first driving normal again and then pressing ‘Resume Speed’.

Also, the external force can be adjusted, representing weather conditions or slopes in the road. This has been done, once increasing the resisting force to 1000 N and once by decreasing the force to -1000 N.

Finally, the set speed can be tuned by 1 km/h. This has been tested by increasing the set speed to +3 km/h and decreasing it to –3 km/h. The next list displays the sequence of events fed to the system.

- evDriveNormal()
- evBrake()
- evDriveNormal()
• evSetSpeed()
• evDriveFast()
• evDriveNormal()
• evSetSpeed()
• evBrake()
• evDriveNormal()
• evResumeSpeed()
• evResistanceUp() (3 times, since on event results in +1000 N)
• evResistanceDown() (6 times in order to reach an external force of −3000 N, starting at the +3000 from the previous events)
• evCCUp() (3 times to reach +3 km/h)
• evCCDown() (6 times to reach −3 km/h, after the 3 times CCUp())

6.2 Results

The foregoing list will be extended with the results of the system to the test sequence. As mentioned, in some cases a semi-screendump will show the reactions of the system. In the results, Speed is the actual speed of the car, SetPt is the set speed of the CC, Resist is the external resistance force effecting the car and Cthrot is the correcting throttle calculated by the CC in order to maintain the set speed.

<table>
<thead>
<tr>
<th>Speed</th>
<th>SetPt</th>
<th>Resist</th>
<th>Cthrot</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

• evDriveNormal() the car reaches a speed of approximately 90 km/h

• evBrake() the speed reduces to zero in approximately two seconds

• evDriveNormal() the car reaches 90 km/h again
• **evSetSpeed()**
  
  the CC sets the actual speed of the car and the correcting throttle is activated in order to maintain that speed (note that the correction throttle is non-zero, due to the subtraction of two numeric numbers)

<table>
<thead>
<tr>
<th>Speed</th>
<th>SetPt</th>
<th>Resist</th>
<th>Cthrot</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>90</td>
<td>0</td>
<td>-6.57e-7</td>
</tr>
</tbody>
</table>

• **evDriveFast()**
  
  the car reaches a speed of approximately 135 km/h and the correcting throttle is set to zero.

<table>
<thead>
<tr>
<th>Speed</th>
<th>SetPt</th>
<th>Resist</th>
<th>Cthrot</th>
</tr>
</thead>
<tbody>
<tr>
<td>134.998</td>
<td>90</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

• **evDriveNormal()**
  
  the car reaches 90 km/h again

• **evSetSpeed()**
  
  the speed is set to the actual speed again

• **evBrake()**
  
  the speed reduces to zero and the correcting throttle is set to zero

• **evDriveNormal()**
  
  the car reaches 90 km/h again

• **evResumeSpeed()**
  
  the previously set speed is maintained again

• **evResistanceUp()**
  
  the resistance is set to 3000 N (notice the increased correcting throttle and the speed not exactly reaching the set speed, which is a property of a proportional controller)

<table>
<thead>
<tr>
<th>Speed</th>
<th>SetPt</th>
<th>Resist</th>
<th>Cthrot</th>
</tr>
</thead>
<tbody>
<tr>
<td>87.081</td>
<td>90</td>
<td>3000</td>
<td>28.3784</td>
</tr>
</tbody>
</table>

• **evResistanceDown()**
  
  the resistance is set to –3000 N (the correcting throttle now is negative too and the speed is slightly higher than the set speed)
the linearity of the model can be recognized in the exactly equal magnitude of the correcting throttle.

- **evCCUp()**

  the set speed is increased by 3 km/h (the resistance has been maintained, to show the subsequent correcting throttle change as a result of the set speed increase)

<table>
<thead>
<tr>
<th>Speed</th>
<th>SetPt</th>
<th>Resist</th>
<th>Cthrot</th>
</tr>
</thead>
<tbody>
<tr>
<td>94.627</td>
<td>92.999</td>
<td>-3000</td>
<td>-26.802</td>
</tr>
</tbody>
</table>

The increase of the setpoint is not exactly 3 km/h since it is an increase of, 0.2777 m/s, which introduces an round off error.

- **evCCDown()**

  the set speed is decreased by 6 km/h (3 km/h lower that the initially set speed)

<table>
<thead>
<tr>
<th>Speed</th>
<th>SetPt</th>
<th>Resist</th>
<th>Cthrot</th>
</tr>
</thead>
<tbody>
<tr>
<td>90.082</td>
<td>87.001</td>
<td>-3000</td>
<td>-29.955</td>
</tr>
</tbody>
</table>

As can be seen from the test results, the CC functions similar to a real-life CC. It can be questioned if the linear model of the car is lifelike enough, since no real car is completely linear. However, these considerations already exceed the scope of this project. After all, it was not the intent to construct a very accurate model, but merely to demonstrate the possibility of modeling with Rhapsody.
7 Conclusions and Recommendations

The work on this project has lead to the following conclusions and recommendations.

Conclusions

- Developing a fully functional CC in Rhapsody has proved that the initial objective of getting acquainted with the new tool has been successful.

- The tool has been mastered, and comprehensive knowledge of the tool has been gathered and documented.

- Embedding the software in the TMS-320LF2407 board has appeared to be out of the scope of the project.

- UML in Rhapsody provides for a structured method of developing object-oriented software.

Recommendations

- A framework can be constructed enabling the output of Rhapsody to be embedded on non-Real Time Operating Systems.

- For testing means, a Graphical User Interface can be developed. First of all to make insertion of events more easy and second to plot the output in orderly charts. This way the reactions of the system can easily be made visible.

- The control law in the CC can be improved to better maintain the set speed. This however is dependent on the properties of the object to be controlled, so specific controllers have to be developed for specific cases, or a learning controller has to be designed.

- Currently, the classes do not check for the presence of the classes they address, at startup resulting in memory violations. Research could be dedicated to finding a way to evade these violations by constructing a ‘presence-check’.
Appendix A: Troubleshooting for Rhapsody

Since many problems were encountered in the development process using Rhapsody, some vital problems and their solutions have been listed below. This to prevent new developments to be as slow and difficult as this one.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>how does a class ‘see’ an other class?</td>
<td>In case of an aggregation, follow the ‘setClass’-approach given in the tutorial, where the ‘whole’ class sets its ‘part of’ classes.</td>
</tr>
<tr>
<td></td>
<td>In case of a relation, set the pointers of the classes in the Components – YourComponent - Configuration – YourConfiguration – Initialisation tab, in the following manner: p_Class1-&gt;setItsClass2(p_Class2); Note that this has to be done for both directions of the relation. If both classes can ‘see’ eachother, both classes have to set their pointers.</td>
</tr>
<tr>
<td>how do I transport data?</td>
<td>using a syntax with a form like this: Variable=itsClass-&gt;GetVariable(); The operation GetVariable() has a body similar to: return Variable;</td>
</tr>
<tr>
<td>a memory exception occurs when I want to animate</td>
<td>One of the initial states in your design addresses classes that have not been constructed yet. Try a delay of some tens of ms. before entering that state.</td>
</tr>
<tr>
<td></td>
<td>Your pointers have not been set correctly. Follow the instructions above.</td>
</tr>
<tr>
<td>Rhapsody says one of my syntaxes does not point to a class/struct/union, which it most definitely does</td>
<td>Check if the operation with the concerning syntax has the ‘static’ box checked. Uncheck it. In my case this occurred most when this class was receiving data and had this checkbox checked.</td>
</tr>
<tr>
<td></td>
<td>Check the multiplicity of the relation or aggregation between</td>
</tr>
</tbody>
</table>
the classes. If that has not been set to 1 or *, that was the problem.

Note that in the OMD, at least one of the classes has to be set active. In case of aggregation this is the ‘whole’ class.

- Rhapsody has some sort of problem with one of my classes (I don’t remember the exact prompt)
- A class can not contain a variable with a name equal to that of the class. Rhapsody will prompt that as soon as it occurs.
  But it also does not allow a variable receiving data from another class having the same name as that other class (e.g.: output-itsoutput->GiveOutput()). Check your model for these type of identical names.

- The output of my system seems random. It does nothing I have programmed.
- Check if your variables are set to static. If not: do so.

These are the major difficulties encountered in this project. Hopefully this summation can speed up further development.
References


