MASTER

Vehicle function correctness
using mCRL2 to compare higher and lower level models

de Langen, M.A.

Award date:
2013

Link to publication
Vehicle Function Correctness
Using mCRL2 to compare higher and lower level models

Public report

Author:
M.A. de Langen

Supervisors:
prof.dr.ir. J.F. Groote
dr.ir. R.G.M. Huisman
Abstract

Software systems get larger and more complex. There are several ways to cope with the understanding and creating of these complex systems. A common development process is to firstly create a higher level specification to show the required basic behaviour. Using this specification a lower level implementation can be created. After the software is fully implemented, a valid question is “Is the implementation created according to the specification?” or “Is the behaviour of these models the same?”

In this report, the possibilities to compare higher level and lower level models are investigated. For this purpose, a translation is created from MATLAB Simulink and Stateflow models to the mCRL2 modelling language. The mCRL2 tool set provides techniques needed to compare two models to each other. Next to that, it also provides techniques to determine whether a model satisfies a set of requirements.

We performed a translation of a higher and lower level model of a cruise control system as designed by DAF Trucks N.V. Both models were verified using a number of requirements. This showed that the higher level model did not behave entirely as intended by the engineers. These failures were not found in the lower level implementation. Verification of the requirements also showed that some requirements were too weakly formulated. The models were also compared to each other to find out if they are equal. Multiple differences between higher and lower level models were found. Since we found these differences, we could state that the implementation did not show the behaviour as designed by the specification of the cruise control. The differences found were quite small and they do not show faulty behaviour which would be noticeable by the driver of the vehicle.

This report shows that comparing models is possible, but it is not trivial to do. One has to avoid that models grow too big. If these models are too big, it is impossible to compare them using limited resources. Additionally, for every difference found during comparison, an explanation had to be found. Next to that, the difference found has to be corrected in one of the models to find the next difference.

Considering the approach taken in this project, both the translation to an mCRL2 model and comparing the models is a time consuming task and requires experience with the tool set. To use the mCRL2 tool set more effectively in the automotive industry, an automatic translation from Simulink to mCRL2 models would be useful.
## Contents

1 Introduction 4

2 Preliminaries 7
   2.1 Labelled Transition Systems 7
      2.1.1 The internal action \( \tau \) 8
      2.1.2 State space explosion problem 9
   2.2 Equivalence of behaviours 9
      2.2.1 Weak trace equivalence 10
      2.2.2 Branching bisimulation 10
   2.3 mCRL2 12
   2.4 Modal \( \mu \)-calculus 13
   2.5 Stateflow/Simulink 14
      2.5.1 Notations 14
   2.6 Cruise control functionality 15
   2.7 Related work 16

3 Functional requirements 17
   3.1 Starting points 17
   3.2 Definitions 17
   3.3 Requirements 18

4 Higher level model 19
   4.1 Description 19
   4.2 Inputs and outputs 19
   4.3 Translation to mCRL2 21
      4.3.1 Actions 21
      4.3.2 Decisions 22
      4.3.3 Model 23
   4.4 Model checking 25
      4.4.1 Results 25
      4.4.2 Differences between the BTC Embedded Validator and mCRL2 26
      4.4.3 Altered requirements 27
   4.5 Evaluation 27

5 Lower level model 29
   5.1 Description 29
   5.2 Input and outputs 29
   5.3 Translation to mCRL2 29
      5.3.1 Actions 29
      5.3.2 Decisions 30
      5.3.3 Complex constructions 30
      5.3.4 Model 32
   5.4 Model checking results 34
   5.5 Evaluation 36

6 Comparing the higher and lower level models 37
6.1 Differences in requirement checks .................................................. 37
6.2 Visual comparison .......................................................................... 38
6.3 Equivalence checking ..................................................................... 40
   6.3.1 Changing the models ................................................................. 40
   6.3.2 Comparison ........................................................................... 41
6.4 Analysis ......................................................................................... 53
   6.4.1 Varying vehicle speeds .............................................................. 53
   6.4.2 Larger systems ....................................................................... 55
6.5 Evaluation ...................................................................................... 57
7 Conclusions and recommendations .................................................. 58
8 List of references ............................................................................... 59
A Requirements .................................................................................. 61
 A.1 Conversion Dutch functional requirements to English functional cruise
    control requirements .................................................................. 61
 A.2 Conversion English functional requirements to χ-calculus .......... 61
 A.3 Altered requirements .................................................................. 61
B Higher level model ........................................................................... 62
 B.1 Stateflow model .......................................................................... 62
 B.2 Inputs and outputs ...................................................................... 62
 B.3 Decisions .................................................................................... 62
 B.4 Translation to mCRL2 .................................................................. 62
 B.5 Transition renamings ................................................................... 62
 B.6 Requirement checks ................................................................... 62
   B.6.1 Altered requirements ............................................................... 62
 B.7 Faults observed for the basic requirements .................................. 62
C Lower level model ........................................................................... 63
 C.1 Simulink model ........................................................................... 63
 C.2 Inputs and outputs ...................................................................... 63
 C.3 Variables lower level model ............................................................ 63
 C.4 Decisions .................................................................................... 63
 C.5 Translation to mCRL2 .................................................................. 63
 C.6 Transition renamings ................................................................... 63
 C.7 Requirement checks ................................................................... 63
D Comparing the models .................................................................... 64
 D.1 Changes to variables ..................................................................... 64
 D.2 Traces ......................................................................................... 64
 D.3 Altered higher level model .............................................................. 64
 D.4 Notes ........................................................................................... 64
1 Introduction

Due to the increasing use of software in the products of automotive companies, software has a significant impact on product quality. Because these software systems get larger and more complex, it is increasingly hard to entirely understand and comprehend their behaviour. Unwanted behaviour is easily created, although not intended.

This document describes the findings of a project, in which behaviour of specifications and implementations is compared to each other on behalf of DAF Trucks N.V. The goal is to show equalities or inequalities between specification and implementation of a real-life software system.

Figure 1 shows a simplified DAF V-model. The V-model is a linear development process model that is used by DAF in their software development. In the first step of the development, requirements are created. These requirements describe the basic behaviour and constraints of the software. In the next step, a higher level model is constructed using the aforementioned requirements. The higher level model shows the behaviour of the software in an abstract way and is called the Generic Vehicle Function (GVF) model. When the higher level model is finished, a more detailed model is constructed: The lower level model. This model is called the Module Design Specification (MDS) model. One of the differences between the higher and the lower level model is that for instance error signals are not considered in the higher level. In the last step, the lower level model is directly implemented in an automated way on an Electronic Control Unit (ECU) within the vehicle. Every design step, has its own test procedure as can be seen in Figure 1.

![Simplified DAF V-model](image-url)

This project focusses on finding out whether it is possible to compare a lower level software model to its corresponding higher level model. The higher level model can be seen as a specification created for the lower level implementation. Hence, they should show the same behaviour. We want to verify that the behaviour specified in the higher level model is also present in the lower level model. By this, we mean that the lower level model should respond in the same way - show the same output values - as the higher level model for the same set of inputs.

Certain conclusions can be drawn after comparing both models: If two models are the same, the correctness of a functional requirement on a higher level model means correctness of the same
requirement on the corresponding lower level model. Since the higher level model is smaller than the lower level model, it would take less effort to check the requirements on the higher level model than on the lower level model. If models are not the same, one would like to know which part is not the same. For each inconsistency between specification and implementation one would like to know why this dissimilarity is present.

There are several issues in comparing two models, that need to be taken into account. The tool set that is used, generates a state space of the system. This state space consists of all possible actions that can be done within the software.

One of the most serious problems with model checking in practice is the so-called “state space explosion problem” (Section 2.1.2). When comparing two models, the entire state spaces of both models have to be compared to each other. So if the state space of a system is very large, it might be impossible to explore it entirely with available resources and memory.

In this project, the GVF model and the MDS model of a DAF cruise control are compared to each other. This project is divided in a couple of steps, as depicted in Figure 2.

The first step in this project is to create a suitable representation of the (GVF and MDS) DAF models. The choice has been made to use the mCRL2 tool set \[2\] in this project. From \[8\] it was concluded that this tool set was the most suitable tool to compare two models to each other. Since we use mCRL2, the DAF models had to be converted to a mCRL2 representation. How these higher and lower level models are translated is elaborated in Section 4.3 and Section 5.3 respectively.

The second step uses formal verification to verify DAF requirements on both models. As can be seen in the V-model (Figure 1), requirements serve as a basis for the models. Hence, checking these requirement on the models give extra information about the correctness of the models. It also gives confidence about the correctness of the translation of the DAF models to mCRL2 models. Differences between the requirement checks on the models can hint to the differences between the models. Section 6.1 elaborates on this issue. For the differences found between the requirements checks, we would like to know if these differences are also found during the actual
comparison of the models in the last step of the project. Section 4.4 shows the model checking on the higher level model and Section 5.4 shows the model checking on the lower level model.

The last step in this project compares the mCRL2 representations of the higher and lower level models. This is reported in Section 6. The objective is to reach equality between the models. Initially, an abstract version of both models is used by decreasing the number of variables in the model. This decreases the size of both the models, which makes it easier and less time-consuming to compare them to each other. If inequalities are found, the models are adjusted accordingly to make them more alike. The models are adapted to reach equality in the end. If the models are equal, the models will be made more accurate by increasing the number of variables in the model.

In Section 2 a short introduction is given to theory and tools used. This section serves as a foundation for notions used in later sections. Section 3 describes the functional requirements which are used to check the higher and lower level model of the cruise control. Section 4 describes the DAF higher level model and its translation to mCRL2. In this section the functional requirements are checked on this model. Section 5 describes the DAF lower level model and its translation to mCRL2. This model is also verified using the given functional requirements. In Section 6 the higher and lower level models are compared to each other. Comparison is based on requirement checks (Section 6.1), a visual check on the abstract models (Section 6.2) and equivalence checking using the mCRL2 tool set (Section 6.3). In this latter section, differences found are explained in detail. An analysis of the approach used is discussed in Section 6.4. Section 7 concludes the findings of this project.

In this public version of the report, the complete list of requirements, used Stateflow models and constructed mCRL2 models are omitted from Appendices A.1 - D.4 even though they are referenced to throughout this document. Next to that, some parts are omitted from the main document.
2 Preliminaries

In order to understand the issues and solutions that are discussed in the remainder of this thesis, this section gives an introduction to theory and tools used. Furthermore, this section serves as a foundation for the notions needed in later sections.

Section 2.1 gives an introduction to the notion of a Labelled Transition System (LTS), which can be used to express behaviour of a certain system. In this thesis an LTS is created for each DAF model. Two LTSs can be compared based on some kind of equivalence (Section 2.2).

A short introduction to the mCRL2 modelling language is given in Section 2.3. This modelling language is used to describe the DAF models. The mCRL2 tool set offers the ability to do requirement checks, which use modal \( \mu \)-calculus (see Section 2.4) as a language to describe behavioural properties. The mCRL2 tools set also offers the possibilities to compare two labelled transition systems.

DAF uses Simulink/Stateflow to model their software for both higher and lower levels. A short introduction on this software is given in Section 2.5.

In order to understand the basic functionality of a DAF cruise control, a short description is given in Section 2.6.

Section 2.7 discusses some related work on requirements checking of DAF models. It also elaborates on some research on tools that can do requirements checking and/or compare models.

2.1 Labelled Transition Systems

Actions are the basic ingredients for models. They represent some observable atomic event and can be denoted by letters \( a, b, c \) or more descriptively by enable, disable, activate. The action activate can represent a cruise control being activated.

Actions can be parameterized with data: An action \( a \) taking data parameter \( d \) is denoted by \( a(d) \). This feature is essential in modelling reactive systems that communicate data and make decisions based on the values of this data. An action activate(40) can represent the cruise control activating and setting the speed to 40 km/h.

The order in which actions can take place is called behaviour. Behaviour is generally depicted as a labelled transition system (LTS). A labelled transition system is a directed labelled graph [7]. Labelled transition systems must have an initial state, which is expressed by a small incoming arrow. Figure 3 shows a simple LTS. It expresses the behaviour of a simplified cruise control system, which can be enabled, disabled, activated, deactivated and set the speed of the vehicle.

![Figure 3: A simplified labelled transition system of a cruise control.](image-url)
The definition of a labelled transition system that we use is the following:

**Definition 2.1. Labelled Transition System.** A labelled transition system (LTS) is a four tuple \( A = (S, \text{Act}, \rightarrow, s) \), where
- \( S \) is a set of states.
- \( \text{Act} \) is a set of actions.
- \( \rightarrow \subseteq S \times \text{Act} \times S \) is a transition relation. For \( (t, a, t') \in \rightarrow \) it is common to write \( t \xrightarrow{a} t' \).
- \( s \in S \) is the initial state.

### 2.1.1 The internal action \( \tau \)

When studying the behaviour of a system, one often wants to view the system as a black box. Only the interactions of the system with its environment are of interest. These interactions between a system and its environment is called external behaviour. Actions that happen within a system are called internal behaviour. When viewing the system as a black box, this behaviour is hidden. To still be able to model these actions, the internal action is introduced. An action is internal if we have no way of observing it directly. We use the special symbol \( \tau \) to denote any internal action. We generally assume that it is available in a labelled transition system, so \( \tau \in \text{Act} \).

An example of an internal action is a timeout timer. If the timer times out, an action is observed. But incrementing the value of the timer can be considered as an internal action, since it is not important to know the actual value of the timer.

Typical for an internal action is that if it follows another action, it is impossible to say whether it is there. Figure 4 shows two models, that cannot be distinguished. They both show an action \( a \) followed by an action \( b \) to the outside world.

![Figure 4](image)

**Figure 4:** The internal action \( \tau \) is not visible. These two LTSs cannot be distinguished.

Figure 5 shows two models, that can be distinguished. In the LTS in Figure 5a it is always possible to do an \( a \) action, as long as neither an \( a \) or a \( b \) have been done. Suppose, the actual behaviour of the system is that of the transition system in Figure 5b. After a while, if the internal action has silently happened, it is impossible to do an \( a \) action anymore. Hence, the behaviour of the two systems in Figure 5 is not the same.

In this thesis, the internal action \( \tau \) is used to hide certain actions. If an action \( a \) is not relevant within the system. For instance, when checking a requirement that does not consider action \( a \).
Action $a$ can be replaced by a $\tau$ action. This is called *hiding* action $a$.

### 2.1.2 State space explosion problem

A well known problem in the field of model checking is the *state space explosion* problem. This problem is often caused by the number of variables and their possible values in a system. Suppose a system has only one variable, a boolean called $A$. This system has two possible states: $A = \text{true}$ and $A = \text{false}$. Adding another boolean variable will give the system four possible states: $A = \text{true}$ and $B = \text{true}$, $A = \text{true}$ and $B = \text{false}$, $A = \text{false}$ and $B = \text{true}$, $A = \text{false}$ and $B = \text{false}$. A system with $n$ boolean variables has $2^n$ possible states, while a system of $n$ variables each with $Z$ allowed values will have $Z^n$ possible states.

A big state space can make it impossible to explore the entire state space with limited resources and memory. For the comparison of two models, the entire state space of both models have to be compared to each other.

In this project, models can get big and thus some precautions have to be made to avoid the state space explosion problem. Details on how this problem is avoided during our research is shown in Section 4.3.2 and Section 5.3.2.

### 2.2 Equivalence of behaviours

Comparing different models, expressed by an LTS, can be done by proving an equivalence relation between two models. An equivalence relation defines a set of conditions to be met in order for...
two LTSs to be considered the same. If two models are different, a counterexample can be shown where these conditions do not hold.

There are different kinds of equivalence relations. The equivalences that are used in this thesis are described in the next sections.

2.2.1 Weak trace equivalence

For weak trace equivalence, traces of two models are compared to each other. A trace of a model is a set of consecutive actions, which can be performed in that model. The notion of weak trace equivalence is obtained by absorbing the internal action in a trace. Formally, two processes are weakly trace equivalent, if their sets of weak traces - traces in which $\tau$-transitions are neglected - are the same [7].

The definition is the following. The symbol $\epsilon$ represents the empty trace.

**Definition 2.2. Weak trace equivalence.** Let $A = (S, Act, \rightarrow, s)$ be a labelled transition system. The set of weak traces $WTraces(t)$ for a state $t \in S$ is the minimal set satisfying:

1. $\epsilon \in WTraces(t)$.
2. If there is a state $t' \in S$ such that $t \xrightarrow{\alpha} t'$ and $\alpha \neq \tau$ and $\sigma \in WTraces(t')$, then $\alpha \sigma \in WTraces(t)$.
3. If there is a state $t' \in S$ such that $t \xrightarrow{\tau} t'$ and $\sigma \in WTraces(t')$, then $\sigma \in WTraces(t)$.

Two states $t, u \in S$ are called weak trace equivalent iff $WTraces(t) = WTraces(u)$. Two transition systems are weak trace equivalent iff their initial states are weak trace equivalent.

The two LTSs in Figure 4 are weak trace equivalent. The set of weak traces for both systems is $\{\epsilon, a, ab\}$. The two LTSs in Figure 5 are also weak trace equivalent. The set of weak traces for both systems is $\{\epsilon, a, b\}$.

2.2.2 Branching bisimulation

The notion of branching bisimulation equivalence preserves the branching structure of processes. It preserves computations together with the potentials in all intermediate states that are passed through [3]. The branching bisimulation relation is stronger than the aforementioned weak trace equivalence. This means that if two models are branching bisimulation equivalent, they are also weak trace equivalent.

The definition is the following:

**Definition 2.3. Branching bisimulation.** Consider the labelled transition system $A = (S, Act, \rightarrow, s)$. We call $R \subseteq S \times S$ a branching bisimulation relation if for all $s, t \in S$ such that $sRt$, the following conditions hold for all actions $a \in Act$:

1. If $s \xrightarrow{a} s'$, then
   - either $a = \tau$ and $s'Rt$, or
there is a sequence \( t \xrightarrow{\tau} \ldots \xrightarrow{\tau} t' \) of (zero or more) \( \tau \)-transitions such that \( sRt' \) and \( t' \xrightarrow{a} t'' \) with \( s'Rt'' \).

2. Symmetrically, if \( t \xrightarrow{a} t' \), then
   - either \( a = \tau \) and \( sRt' \), or
   - there is a sequence \( s \xrightarrow{\tau} \ldots \xrightarrow{\tau} s' \) of (zero or more) \( \tau \)-transitions such that \( s'Rt \) and \( s' \xrightarrow{a} s'' \) with \( s''Rt' \).

![Branching bisimulation diagram](image)

**Figure 6:** Branching bisimulation. Relating states with bisimulation relation \( R \).

Two states \( s \) and \( t \) are **branching bisimilar**, iff there is a branching bisimulation relation \( R \) such that \( sRt \) (Figure 6). Two labelled transition systems are **branching bisimilar** iff their initial states are branching bisimilar.

![LTS diagrams](image)

**Figure 7:** Two LTS that are weak-trace equivalent, but not branching bisimulation equivalent.

The LTSs in Figure 7 are not branching bisimulation equivalent. Figure 7a shows a model in
which after doing an \(a\) action, one can determine to do a \(b\) or a \(c\) action. In Figure 7b this choice is made before doing an \(a\) action.

### 2.3 mCRL2

The mCRL2 language and toolset are developed at the department of Mathematics and Computer Science of the Technische Universiteit Eindhoven, in collaboration with LaQuSo, CWI and the University of Twente.

mCRL2 is a formal specification language with an associated toolset. The toolset can be used for modelling, validation and verification of concurrent systems and protocols. It can also be used to show equivalence relations between models.

The toolset supports a collection of tools for linearisation, simulation, state-space exploration and generation and tools to optimise and analyse specifications. Moreover, state spaces can be manipulated, visualised and analysed.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>(a)</td>
</tr>
<tr>
<td>Sequential Composition</td>
<td>(a . b)</td>
</tr>
<tr>
<td>Alternative Composition / Choice</td>
<td>(a + b)</td>
</tr>
<tr>
<td>Recursion</td>
<td>(S = a . S)</td>
</tr>
<tr>
<td>Data</td>
<td>(c(d))</td>
</tr>
<tr>
<td>Condition</td>
<td>(d \rightarrow a \diamond b)</td>
</tr>
<tr>
<td>Summation</td>
<td>(\text{sum } d : \text{Type} . c(d))</td>
</tr>
</tbody>
</table>

*Table 1: mCRL2 process operations.*

The most important elements in the mCRL2 modelling language are processes and actions, describing the behaviour of a system. In this thesis, an mCRL2 representation is created for the DAF models. Table 1 shows the operations that are used in the representations created. The operators can be used on processes and actions. Nesting of different operators is also allowed. The behaviour of the example processes in Table 1 is as follows:

- Process \(a\) performs a single action \(a\).
- Process \(a . b\) performs an action \(a\) and after that a \(b\).
- Process \(a + b\) performs either an \(a\) action or a \(b\) action.
- Process \(S\), defined as \(S = a . S\) continuously performs action \(a\).
- Process \(c(d)\) performs action \(c\) with parameter \(d\).
- Process \(d \rightarrow a \diamond b\) performs an action \(a\) if \(d\) has the value \text{true}. Otherwise, it performs action \(b\).
- Process \(\text{sum } d : \text{Type} . c(d)\) describes the alternative composition of the actions \(c(d)\) for all possible values of variable \(d\). In the case \(d\) is of type \text{Bool} it corresponds to \(V = c(\text{true}) + c(\text{false})\). Suppose \(d\) is of type \text{Nat}, it would correspond to \(V = c(0) + c(1) + c(2) + ...\)
Processes can be combined to form a process specification. Listing 1 shows an example mCRL2 process specification. It has three processes P1, P2 and P3 and can do the actions enable, disable, activate, deactivate and set speed.

1 proc P1 = enable . P2;
2 P2 = disable . P1 + sum b: Bool . activate . P3(b);
3 P3(b: Bool) = b \rightarrow deactivate . P2 <> sum c: Bool . set speed . P3(c);
4
5 init P1;

Listing 1: An example process specification.

The example process specification above initializes with process P1 (line 6). The process P1 can only do an enable action, which is followed by process P2. Process P2 can do a disable action and go back to P1, an activate action followed by the P3(true) process or an activate action followed by the P3(false) process. The P3 process requires a boolean input variable b. This boolean b determines which path to choose: If true, a deactivate action is executed, followed by the P2 process. Otherwise, a set speed action is executed, followed by P3(true) or P3(false).

These examples show a minimal application of the operators used in this thesis. A more elaborate description of the syntax and the semantics of the mCRL2 modelling language can be seen in [6].

2.4 Modal $\mu$-calculus

A modal $\mu$-calculus formula is used to describe a behavioural property. Such a property can then be verified automatically to a process model described in mCRL2. The syntax of the model $\mu$-calculus formulae used in this project is given by the Backus-Naur Form (BNF) in Table 2.

| $\phi$ ::= true | false | $\phi \land \phi$ | $\phi \lor \phi$ | $[\rho]\phi$ | $\langle\rho\rangle\phi$ |
| $\rho$ ::= $\alpha$ | $\rho.\rho$ | $\rho^*$ |
| $\alpha$ ::= $\alpha \cup \alpha$ | $\alpha \cap \alpha$ | $!\alpha$ | $a(d)$ | true |

Table 2: Modal $\mu$-calculus syntax.

In the syntax description in Table 2, $\phi$ represents a property, $\rho$ represents a set of sequences of actions and $\alpha$ represents a set of actions. The actual modal $\mu$-calculus is much richer, but we will not need additional constructs for this project.

The property true holds for any state of a process, while false holds for no state. The formula $\phi_1 \land \phi_2$ is valid wherever both $\phi_1$ and $\phi_2$ hold. $\phi_1 \lor \phi_2$ is valid whenever $\phi_1$ or $\phi_2$ holds. The formula $[\rho]\phi$ describes that $\phi$ holds in all states that can be reached by a sequence in $\rho$, while $\langle\rho\rangle\phi$ describes that $\phi$ holds in some state that can be reached by a sequence in $\rho$.

To describe a sequence of actions $\rho$, operators for concatenation ($\rho.\rho$) and iteration ($\rho^*$) are available.

The connectives $\cup$ and $\cap$ in set of actions denote intersection and union of sets of actions. The notation $!\alpha$ denotes the complement of the set of actions $\alpha$ with respect to the set of all actions. An action $a$ parameterized with data element $d$ is denoted by $a(d)$. Action true describes the
presence of any action in a sequence. A more elaborate description of the modal $\mu$-calculus can be found in [5].

2.5 Stateflow/Simulink

Simulink, developed by The MathWorks, is a data flow graphical programming language tool for modeling, simulating and analyzing multidomain dynamic systems. Its primary interface is a graphical block diagramming tool and a customizable set of block libraries. It offers tight integration with the rest of the MATLAB environment.

Stateflow, is a control logic tool used to model reactive systems via state machines and flow charts within a Simulink model. Stateflow enables the representation of hierarchy, parallelism and history within a state chart. Stateflow also provides state transition tables and truth tables. Figure 8 shows an example of a Stateflow system modelling an automatic transmission system.

![Stateflow Diagram](image)

Figure 8: Example Stateflow: Control logic for an automatic transmission system.

2.5.1 Notations

A Stateflow model consists of states. Figure 8 shows several states. There are two super states gear_state and selection_state, which contain multiple substates. The substates of gear_state are first, second, third and fourth. The dashed lines around the states gear_state and selection_state show that these states execute in parallel.
Transitions can be taken from one state to another, depicted by an arrow. If multiple transitions can go from one state to another, they have a number on them. The transition with the lowest number has the highest priority. Also, transitions can have labels. Labels on these transitions have a certain function:

- \([\text{speed} < \text{down}_\text{th}]\) is a condition label. This transition can only be taken if the condition \(\text{speed} < \text{down}_\text{th}\) holds. Conditions can contain disjunctions (\(||\)) conjunctions (\(&&\)) and negations (\(\sim\)).
- \(\text{/speed} = 50\) is a assignment label. If this transition is taken, the value 50 is assigned to the variable \(\text{speed}\). This notation can not be seen in Figure 8, but it is used in the DAF Stateflow models.

2.6 Cruise control functionality

This section describes the DAF cruise control from a driver point-of-view. It elaborates on the basic behaviour of the cruise control with respect to the buttons the driver can press.

Figure 9 shows three switches to control the cruise control functionality:

1. Speed function Increment (SET +)
2. Speed function Decrement (RES -)
3. OFF switch

![Cruise control steering wheel switches.](image)

At the moment the driver presses the “Speed function Increment (SET +)” the cruise control module will set the actual speed as current vehicle speed target which will be shown on the display. If the cruise control module has already been activated before and the driver presses the “Speed function Decrement (RES -)” the cruise control module will set the previous speed target as current vehicle speed target. While the cruise control is active, this speed target can be adapted by the “Speed function Increment and Decrement switches”. When the driver presses one of those switches briefly the speed target will adapted with a predefined step. When the driver presses one of those switches for a longer time, the speed target will ramp up or down with a predefined rate. The OFF switch deactivates the cruise control.
2.7 Related work

R. Schoren [9] has shown that it is possible to perform formal verification within the automotive industry. In this research, formal verification is used on software systems at DAF Trucks N.V. to successfully show their correctness. A similar approach is used to verify requirements in this thesis.

DAF report No. 7002_51050/13-090 [8] discusses research on tools that can be used for formal verification of DAF models. Given a list of requirements of DAF, three tools were able to verify all of them. These tools are mCRL2[6], the BTC Embedded Validator[4] and the SCADE Suite Design Verifier[2]1. The mCRL2 tool set and the BTC Embedded Validator have been used in this thesis to verify requirements the higher level (GVF) and lower level (MDS) model of a DAF cruise control.

DAF report No. 7002_51050/13-090 [8] also accesses if certain tools can compare models to each other. From the list of tools investigated, only the mCRL2 tool set is capable of proving equality between models. It can compare models using different sets of equivalences (see Section 2.2) and returns a counterexample if they are not equal. The mCRL2 tool set is used in this thesis to prove equality between the higher level (GVF) and lower level (MDS) model of a DAF cruise control.

As stated in Section 2.1.2 there is a problem of the state space explosion problem. In [4], seven guidelines are presented to reduce the number of state spaces generated by the mCRL2 tool set. A useful advice for this project is the use of information polling. Information polling states that processes have to ask for information, only when it is required. A second guideline is the restricted use of data. In this case, the use of data has to be avoided whenever possible. If data is needed, one has to try to categorise it and only store those categories. For example, instead of storing a height in millimetres, one should store too_low, right_height and too_high.

16 http://www.esterel-technologies.com/products/scade-suite/add-on-modules/design-verifier
3 Functional requirements

In this section the functional requirements for the cruise control system are described. Both the higher level as the lower level model will be checked using these requirements.

The requirement checks give extra information about the correctness of both models and the correctness of the translation of the DAF models to mCRL2 models. Differences in results of these checks can hint to differences between the models. This will be discussed in Section 6.1.

Firstly, some starting points will be given. After that, some definitions will be given in Section 3.2 to clarify the requirements. Section 3.3 lists a selection of requirements used in this thesis.

3.1 Starting points

The following starting points hold for the requirements given.

• The requirements described in Section 3.3 will not be complete. Meeting all requirements in that section will not guarantee the correctness of the whole system.

• These requirements will have to be met with 100% certainty. If a requirement is not met, there is an error in the model or in the requirement.

• Timing issues have not been taken into account. One could imagine that there would be requirements about the responsiveness of the system. Thus, the assumption is made that the software system is sufficiently fast and always reacts in time.

• The cruise control will be realized by the use of electronics and software and therefore needs a power supply. We assume that all components have power at all times, hence the software will always be active. The software will also be active when one speaks of the function being off.

• The systems use continuous variables, such as the actual vehicle speed. When modeling in mCRL2, we want to avoid possible infinite numbers of values, so the assumption is made that speed is of a discrete type (SPD) and can only have a limited number of distinct values. This does not remove any complexity from the system. In fact, the same model can be used, using a SPD type with an arbitrary small step size between possible values. The values of speed are denoted as an ‘s’ followed by an integer speed value. The speeds used in the models are sX, where X is the speed in km/h between 0 and 100 with steps of 10 (s0, s10, ..., s100).

3.2 Definitions

This section describes some definitions. These definitions are useful to fully understand notions used in later sections.

• The cruise control has four main states:
  – DISABLED: The cruise control is OFF (in Dutch: “uit staan”). This means that the cruise control is disabled.
- **ENABLED**: The cruise control is ON (in Dutch: “aan staan”). In this state, the cruise control can be inactive or active.

- **INACTIVE**: The cruise control is enabled, but the speed cannot yet be set. The driver can activate the cruise control by pressing one of the activation buttons. For instance, the ‘Speed function Increment (SET +)” in Figure 9.

- **ACTIVE**: In this state, the cruise control is activated. The speed of the vehicle is determined by the cruise control. The driver can change the speed by pressing an increment, decrement, accelerate or decelerate button (see Figure 9).

  - The cruise control has an output signal `Active`, which is a boolean that sends the regulator whether or not to regulate the vehicle speed.
  - The cruise control has an output signal `SetSpd`, which is a continues variable the represents the set speed.

### 3.3 Requirements

Appendix A lists the requirements that describe the desired functionality of the cruise control in terms of inputs and outputs. These requirements are provided by DAF. Table 3 contains a selection of these requirements that are used as an example throughout the rest of this thesis. Table 4 lists this selection of requirements in English.

1. Als de functie uitstaat moet de waarde van outputsignaal `SetSpd` nul zijn (0 km/h).
5. De waarde van het outputsignaal `Active` moet false worden zodra het inputsignaal `Off Command` de waarde true heeft.
10. De waarde van outputsignaal `SetSpd` mag nooit hoger zijn dan de maximale set speed waarde.
14. Als de waarde van het inputsignaal `Increment Command` true wordt, dan moet de waarde van outputsignaal `SetSpd` verhoogd worden met een vaste waarde.

**Table 3**: A selection of the cruise control requirements.

1. If the function is disabled, the value of the output signal `SetSpd` should be zero (0 km/h).
5. The value of the output signal `Active` should be false as soon as the input signal `Off Command` has the value true.
10. The value of the output signal `SetSpd` should never be higher than the maximal set speed value.
14. If the value of the input signal `Increment Command` is true, the value of the output signal `SetSpd` should be incremented with a predefined step.

**Table 4**: The same selection of the cruise control requirements in English.
4 Higher level model

As stated in Section 1, a higher level model for a software system is constructed from a given set of requirements. At DAF, it is referred to as the Generic Vehicle Function (GVF) model. This model can be seen as a specification and using this model, a more detailed lower level model can be constructed.

This section examines the higher level model of the cruise control. This model is created in Stateflow. A description of this model is given in Section 4.1. This Stateflow model is translated to an mCRL2 representation (Section 4.3). In Section 4.4, the functional requirements of Section 3 are formally verified against the mCRL2 model using the mCRL2 tool set. The original model has also been verified using the BTC Embedded Validator. The differences between the outcomes of the tools are discussed in Section 4.4.2.

4.1 Description

Figure 10 shows an abstract model of the cruise control. The full Stateflow model can be seen in Appendix B.1.

Figure 10a shows the top part of the cruise control. The model initializes to the ENGINE OFF state. From this state a transition can be taken to the ENGINE RUNNING state when the engine is running. The ENGINE RUNNING block contains a DISABLED and an ENABLED block. Transitions between DISABLED and ENABLED can be taken when the enable conditions (from DISABLE to ENABLE) or disable conditions (from ENABLE to DISABLE) hold.

Figure 10b shows the content of the ENABLED block. It has an INACTIVE and an ACTIVE block.

The ACTIVE state consists of two parallel states, depicted by dashed lines. The original model (Appendix B.1) has three parallel states. In Simulink these states do not actually execute in parallel. They execute in sequence. For this abstract model, this sequence is the following:

1. CONTROL_SPEED
2. ADJUST_SPEED

The CONTROL_SPEED state initializes to the CONTROLLING state. The ACCEL_PED OVERRIDE state can be reached, when the driver accelerates using the accelerate pedal while the cruise control is active. The DRIVELINE_INTERRUPTED_DRIVER state is entered when the driveline is interrupted. Both states contain a timeout. If this timeout is reached, it triggers a disable condition (see Figure 10a) and results in disabling the cruise control.

In ADJUST_SPEED the cruise control set speed can be changed. The change of the set speed is based on the cruise control button that is pressed.

4.2 Inputs and outputs

The context of the cruise control system is defined by a number of fixed input and output signals. Table 5 shows a selection of input and output signals that are used throughout this section. The
Euro 6 Cruise Control

ENGINE OFF

[Engine running] [Engine off]

ENGINE RUNNING

DISABLED

Enable conditions

DISABLED

ENABLED

[Engine running] [Engine off]

[Enable conditions] [Disable conditions]

(a) A simplified Stateflow model showing the ENGINE OFF and ENGINE RUNNING states.

ENABLED

INACTIVE

ACTIVE

CONTROL SPEED 1

CONTROLLING

REGULATE

ACCELERATE

DECELERATE

[DecelerateCommand || AccelerateCommand]

ACCELERATE

DECELERATE

[DecelerateCommand || !AccelerateCommand]

[DecelerateCommand]

[DecelerateCommand && !AccelerateCommand]

(b) A simplified Stateflow model showing the ENABLED state.

Figure 10: A simplified Stateflow model of the higher level cruise control.

variable in italics represents a combination of different input signals, but it is modelled as one input signal. This is elaborated upon in Section 4.3.2.
<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabling_conditions</td>
<td>Active</td>
</tr>
<tr>
<td>Off Command</td>
<td>SetSpd</td>
</tr>
<tr>
<td>Set Command</td>
<td></td>
</tr>
<tr>
<td>Resume Command</td>
<td></td>
</tr>
<tr>
<td>Increment Command</td>
<td></td>
</tr>
<tr>
<td>Decrement Command</td>
<td></td>
</tr>
<tr>
<td>Accelerate Command</td>
<td></td>
</tr>
<tr>
<td>Decelerate Command</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: The input and output signals of the cruise control system used in this section.

The requirements listed in Section 4.4 describe the desired functionality in terms of these inputs and outputs.

Appendix B.2 shows the complete list of inputs and outputs.

4.3 Translation to mCRL2

In this section is explained how the cruise control system was translated from a Stateflow model to mCRL2.

4.3.1 Actions

In the mCRL2 model, a set of actions is chosen. These actions show the behaviour of the cruise control responding to certain inputs. From these actions, the outputs can be determined. These actions are the following:

- *read_env*, this action "reads" the inputs from the environment. The parameters used by this action are the inputs described in Section 4.2. After reading the inputs from the environment, the system decides what action to do next. This can be one or more of the actions listed below.

- *no_action*, this action can be observed when a component does not do any action that changes the output variables. For the input that is given to the model no response is observable to the outside world.

- *enable*, this action shows that the cruise control is enabled. This action corresponds to setting cruise control to "ON". In the Stateflow model it can be interpreted as taking the transition from DISABLED to ENABLED.

- *disable*, this action shows that the cruise control is disabled. The cruise control cannot be activated. This action corresponds to setting cruise control to "OFF". In the Stateflow model it can be interpreted as taking the transition from ENABLED to DISABLED.

- *activate*, this action shows the activation of the cruise control. This action corresponds to setting the output signal Active to true. In the Stateflow model it can be interpreted as taking the transition from INACTIVE to ACTIVE.
• deactivate, this action shows the deactivation of the cruise control. This action corresponds to setting the output signal Active to false. In the Stateflow model it can be interpreted as taking the transition from ACTIVE to INACTIVE.

• regulate_cc, this action shows that the speed of the cruise control has been regulated. This action has one parameter, s, which corresponds to setting the output signal SetSpd to the speed s.

• stop_rgl, this action shows that the cruise control does not control the speed anymore. We assume when a stop_rgl occurs, the SetSpd of the cruise control is set to 0 km/h.

<table>
<thead>
<tr>
<th></th>
<th>enable</th>
<th>activate</th>
<th>regulate(s70)</th>
<th>regulate(s50)</th>
<th>stop_rgl</th>
<th>disable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>SetSpd</td>
<td>0</td>
<td>s70</td>
<td>s50</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 11:** Actions and corresponding output signals.

The values of the output variables can be determined by inspecting a sequence of actions. After observing an activate action, it is assumed that the value of the output signal Active is true until a deactivate or disable action is observed. When a regulate_cc(s) has been seen, it is assumed the current SetSpd value is set to s, where s is an allowed cruise control speed. The SetSpd is set to s until another regulate_cc(s’) (where s’ is the new cruise control speed) is observed or a stop_rgl has been seen.

**Figure 11** shows an example. At the top of this diagram, a sequence of actions can be seen: enable.activate.regulate(s70).regulate(s50).stop_rgl.disable. The change in value of the output variables Active and SetSpd is shown below these actions. For instance, the activate action causes the output signal Active to be set to true. The regulate(s70) action sets the output signal SetSpd to s70.

### 4.3.2 Decisions

Some decisions are made that allow for effective modeling of the system:

• All states in the model have an implicit transition to the ENGINE RUNNING state. This transition can be taken when the variable Engine running is false.

  The decision is made that the engine is always running. In this way, the state ENGINE OFF can be disregarded and the Engine running variable is not considered. The initial state of the model is DISABLED.

  This results in a decreasing number of transitions in the resulting mCRL2 representation.

• The actual enable and disable conditions depend on a combination of conditions.

  The decision is made to combine the variables that occur only on transitions from DISABLED to ENABLED or ENABLED to DISABLED into one variable. This input variable is called enabling_conditions.
Modelling these inputs as one variable reduces the number of input variables in the mCRL2 model. This results in less possible transitions in the generated Labelled Transition System from this mCRL2 representation.

Appendix B.3 lists the variables that are combined.

- The states REGULATE, DECELERATE and ACCELERATE within the CONTROLLING block are not considered as different states. Being in this state does not show any behavioral difference to the outside world: These states do not change any variable or enable/disable transitions outside the CONTROLLING block.

- The two "parallel states" CONTROL_SETSPD, ADJUST_SETSPD respond to the input signals in that specific order.

4.3.3 Model

The complete mCRL2 model, containing all used sort declarations and function mappings, can be found in Appendix B.4. The basic structure of the modelled process is as follows:

```plaintext
CC_Disabled(ccss : SPD) =
  sum environment : ENV . read_env(environment) .
  (enable conditions) => enable . CC_Inactive(ccss)
  <>
  no_action . CC_Disabled(ccss)
;
CC_Inactive(ccss : SPD) =
  sum environment : ENV . read_env(environment) .
  (disable conditions) => disable . CC_Disabled(ccss)
  <>
  (activate conditions) => (activate .
    regulate_cc(calculate_new_css) .
    CC_Active(calculate_new_css, CONTROLLING))
  ) <>
  (resume conditions) => (activate . regulate_cc(ccss) . CC_Active(ccss, CONTROLLING)
  ) <>
  no_action .
  CC_Inactive(ccss)
;
CC_Active(ccss : SPD, state : STATE) =
  sum environment : ENV . read_env(environment) .
  (disable conditions) => disable . CC_Disabled(ccss)
  <>
  (deactivate conditions) => deactivate . CC_Inactive(ccss)
  <>
  CC_Control_Speed(ccss, environment)
;
CC_Control_Speed(ccss : SPD, environment : ENV, state : STATE) =
  (state => CONTROLLING) => (AccelpedFuel above CCFuel) =>
```

23
For each state in the Stateflow model (Figure 10a and Figure 10b), there is a process in the mCRL2 model. The states within the CONTROL_SPEED substate are represented as a local variable state within the CC_Active, CC_Control_Speed and CC_Adjust_Speed process.

In line 75, we see that the initial state of the cruise control is the disabled state. Here, the cruise control set speed is set to $s_0$, which represents 0 km/h. During execution, the system constantly reads the inputs from the environment (lines 2, 9, 25). As the environment is not within the system’s control, there is a read_env action for every possible combination of inputs. In the actual model, this is expressed by a sum operator for each input, but these are combined in the model above to an input environment of type ENV for reading purposes.

After getting the inputs from the environment, the system determines which transition should be taken. The expressions used in the conditions are in terms of inputs used in the read_env transition, but abstracted to for instance disable conditions and activate conditions in the description above.
The cruise control system has two outputs: \texttt{Active} and \texttt{SetSpd}. Initially, their respective values are \texttt{false} and \texttt{s0}. Taking an \texttt{activate} transition in the model corresponds to setting the \texttt{Active} output signal to \texttt{true}. A \texttt{regulate}\texttt{cc}(\texttt{ccss}) transition in the model corresponds to setting the \texttt{SetSpd} to \texttt{ccss}. When a \texttt{deactivate} or \texttt{disable} transition is taken, the \texttt{Active} output signal is set to \texttt{false}. A \texttt{stop}\texttt{rgl} transition in the model corresponds to setting the \texttt{SetSpd} to \texttt{s0}.

In the ACTIVE state (line 25 to 32) - if the disable and deactivate conditions do not hold - the transition is taken to the CC\_Control\_Speed process, which represents the \texttt{CONTROL\_SPEED} in the actual model. The CC\_Control\_Speed process can only take a transition to the CC\_Adjust\_Speed process. This ensures that the \texttt{CONTROL\_SPEED} and \texttt{ADJUST\_SPEED} execute in that specific order. The CC\_Adjust\_Speed process returns back to the CC\_Active process, where the environment is read again.

### 4.4 Model checking

The functional requirements have been checked in two ways. Firstly, the requirements have been checked on the mCRL2 model using the requirements described in Appendix A.2. The results are shown in Section 4.4.1. Secondly, the original Stateflow model has been tested using the BTC Embedded Validator. One difference has been found between the model checking results using the BTC Embedded Validator and the mCRL2 tool set. This is elaborated upon in Section 4.4.2. Section 4.4.3 discusses requirements that are too weak and fail for the wrong reason.

#### 4.4.1 Results

In the next table model checking results of the selection of requirements (Section 3.3) are shown. When a requirement fails, an explanation is given why this requirement fails. Some of the requirements contain multiple cases. For instance, requirement 1 is split in requirement 1a and 1b. Such a requirements holds, only if all cases of this requirement hold.

Table 7 shows the results for all requirements. Appendix B.6 shows a detailed list of results of all requirement checks.

1a. When disabled, a regulate action cannot be observed: There can be no \texttt{rgl} before \texttt{enable} when disabled. 
   This requirement holds.

1b. If the function is enabled and the cruise control set speed is set (a regulate action has been seen) and the disable command is observed, the set speed must be set to 0 km/h. So, when a \texttt{regulate} action has been seen, there must be a \texttt{stop}\texttt{rgl} directly after a \texttt{disable} action.
   This requirement does not hold. In the model no \texttt{stop}\texttt{rgl} action is present. In the original model, the only time when the \texttt{SetSpd} is set to \texttt{s0} (0 km/h) is in the ENGINE OFF state, which is disregarded in this model. Hence, the set speed will never be set to 0 km/h.
5. When an Off Command is observed and the function is active, a deactivate action must follow immediately. An activate action cannot occur directly after observing an Off Command. This requirement does not hold. A counterexample is the following:
   1. Suppose the cruise control is active. Hence, we are in some substate within the ACTIVE state.
   2. Now, the Off Command is observed and actual vehicle speed is $s_0$ (which represents 0 km/h).
   3. One of the disable conditions is that the actual vehicle speed is below 30 km/h. The cruise control will be disabled and not deactivated.

10. There can be no regulate_cc(s90) or regulate_cc(s100) This requirement holds.

14. When active and the Increment Command is true, the current value of SetSpd must be increased. This requirement does not hold. The maximum value for the speed in the mCRL2 model is $s_{80}$ (which represents 80 km/h). When the speed of the cruise control is set to $s_{80}$, the Increment Command does not set the cruise control speed to a higher value.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>mCRL2</th>
<th>BTC EV</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>2</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>4</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>5</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>6</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>7</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>8</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>9</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>10</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>11</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>12</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>13</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>14</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>15</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 7: Model checking results for the mCRL2 tool set and the BTC Embedded Validator (BTC EV).

4.4.2 Differences between the BTC Embedded Validator and mCRL2

The results of the requirement checks of the BTC Embedded Validator have been compared to the results of the mCRL2 tool set. This comparison shows only one difference. The result for
requirement 10 ("The cruise control cannot be set to a value higher than the maximal set speed value") fails using the BTC Embedded Validator, but does not fail in the mCRL2 model. This can be explained as follows:

In the mCRL2 model eleven values are used to represent the speed of the vehicle. These values are speeds between 0 to 100 with steps of 10. The BTC Embedded Validator shows that the set speed can be set higher than the maximal set speed value. The value of the maximal set speed lies between two mCRL2 speeds. The found value that exceeds the maximal set speed value lies between the same two mCRL2 speeds.

By making the step size in the representation of the speed in mCRL2 smaller, the same result as the BTC Embedded Validator can be found. This approach has been tested in mCRL2 and has shown to find the same results as the BTC Embedded Validator.

4.4.3 Altered requirements

In Section 4.4.1 we see that requirement 5 ("When the Off Command is true, the cruise control should become inactive") fails. The reason that it fails is that the cruise control is disabled instead of deactivated when an Off Command is observed. Of course, this behaviour is not wrong. A counterexample is found, because the requirement is too weak. The requirement can be altered to not consider such cases.

To show how some requirements are not strong enough, some have been altered. These are listed in Appendix A.3. If requirements still fail, the counterexamples are more meaningful than the counterexamples found in the previous sections.

The rest of this section has been removed due to confidentiality.

4.5 Evaluation

We have used two tools (mCRL2 and the BTC Embedded Validator) to verify this higher level model. They show the same results, except for one requirement. This can be solved by using more values for the speed of the vehicle in the mCRL2 representation.

The results of the requirement checks on the higher level model show that 11 out of 15 requirements fail. These failures occur for different reasons:

- Most requirements fail (6 out of 11), because the requirements are too weak. An example is shown in Section 4.4.3. This can be corrected by altering the requirements and rechecking it on the model. It can be necessary to alter and recheck some requirements multiple times to rule out cases which are not wrong. When these cases are not considered anymore, the model checking generally shows a real fault in the model or proves the requirement to hold.

- One requirement (requirement 10) shows a fault in the requirement itself. The original requirement states that the vehicle speed should not exceed the maximal set speed value. This is easily corrected by changing the requirement.

- Four requirements show a fault in the original DAF model.
- The first two faults are that the set speed is not reset to 0 km/h, when the cruise control is disabled or deactivated (requirement 1 and 6). This can be fixed by adding the assignment of 0 km/h to the set speed, when the cruise control is disabled or deactivated.

- The other fault in the DAF model is that the cruise control is activated, when the off command is pressed (requirement 5, even after it was altered).

  How this can be fixed, is elaborated upon in Section 4.4.3.

- Requirement 11 states that no action should be done, when multiple inputs are true. The model does not take this requirement into account. A possible solution is to introduce one input signal \( \text{input} \) to replace all the other input signals (except for the off command). This input should have 6 different possible values:

  1. Set
  2. Resume
  3. Increment
  4. Decrement
  5. Accelerate
  6. Decelerate

  The condition that Accelerate is true can now for instance be changed to

  \[ \text{input} = \text{Accelerate} \]

  This solution guarantees that there are no multiple inputs true at the same time, since \( \text{input} \) can only have one value of the list above.

The failure of these requirements does not counteract with the comparison process (Section 6), which is the main goal of this thesis. In fact, we will also check the requirements on the lower level model. If inconsistencies are found between requirement checks on both models, some differences between the models are found.

When the actual comparison is done, the requirement outcomes can also hint to a direction where the two models are not alike.
5 Lower level model

This section examines the lower level model as defined by DAF. The lower level model can be seen as an implementation of the cruise control. At DAF, this model is called the Module Design Specification (MDS) model and it is modelled in Simulink. A description of this model is given in Section 5.1. The inputs and outputs that are used within this section are described in Section 5.2. This Simulink model is translated in Section 5.3 to an mCRL2 representation. In Section 5.4, the functional requirements of Section 3 are formally verified using the mCRL2 tool set.

5.1 Description

This section describes an abstract version of the MDS model of the cruise control. The full MDS model can be seen in [1].

This description has been removed due to confidentiality.

5.2 Input and outputs

The context of the cruise control system is defined by a number of fixed input and output signals. Table 8 below shows a selection of input and output signals that are used throughout this section. The variable in *italics* represents a combination of different input signals, but it is modelled as one input signal. This is elaborated upon in Section 5.3.2.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver request</td>
<td>Active</td>
</tr>
<tr>
<td>Off Command</td>
<td>SetSpd</td>
</tr>
<tr>
<td><em>disable conditions</em></td>
<td></td>
</tr>
</tbody>
</table>

*Table 8: The input and output signals of the cruise control system used in this section.*

The requirements listed in Section 5.4 describe the desired functionality in terms of these inputs and outputs.

Appendix C.2 shows the complete list of inputs and outputs.

5.3 Translation to mCRL2

In this section is explained how the cruise control system was translated from a Simulink model to mCRL2.

5.3.1 Actions

In the mCRL2 model, a set of actions are chosen. These are the same as the actions chosen in the higher level model (see Section 4.3.1):
• no_action
• enable
• deactivate
• stop_rgl
• read_env

5.3.2 Decisions

Some decisions are made that allowed for effective modeling of the system:

• The decision is made that the engine is always running. When modeling the higher level model, the same decision was made in Section 4.3.2. Having both models assuming the engine is always running, will make the comparison in between both models in Section 6 more effective.

• Some (boolean) variables are introduced to represent a number of input variables. These input variables are combined to decrease the number of transitions in the mCRL2 model. Appendix C.3 lists these variables.

• The vehicle acceleration is disregarded. This variable is only used to determine if the cruise control should be disabled. It is not used in any other block of the system.

• The difference between km/h and mph is not used. The choice has been made to have a constant increase/decrease value.

• The lower level contains a block, which assures a smooth transition from the previous set speed to the new set speed. An example is the following: Suppose the cruise control is deactivated and the previous set speed is 80 km/h. The vehicle drives at 40 km/h. Now, the driver presses the RESUME button. Because of this, the cruise control has to resume from 40 km/h to 80 km/h. This block sets the speed to values between 40 and 80 and assures the vehicle accelerates in a smooth fashion. Since we are only interested in set speed values that are set, these intermediate values are not important.

• The lower level contains a block, that converts the outputs signals to signals understandable for the CAN bus. This does not effect the behaviour of the actual system and is left out.

Appendix C.4 lists the design decisions in more detail.

5.3.3 Complex constructions

In the model two complex constructions have been observed when converting it to mCRL2. These are elaborated upon in the next paragraphs.
**Determine primary retarder off** In the subsystem Determine primary retarder off the AND-OR construction in Figure 12 can be seen.

![Figure 12: AND-OR construction in Determine primary retarder off.](image)

This construction has two boolean inputs $x$ and $y$ and a boolean output $z$. It represents that $z \equiv (x \land y) \lor y$. This construction is unnecessary due to the fact that $z$ always equals $y$: $z \equiv (x \land y) \lor y \equiv y$. Consider also the truth table in Table 9.

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \land y$</th>
<th>$(x \land y) \lor y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 9: Truth table for $(x \land y) \lor y$.**

Figure 13 shows a second construction.

![Figure 13: IF construction in Determine primary retarder off.](image)

This construction has two boolean inputs $x$ and $y$ and a boolean output $z$. It represents that if $x$ holds, $z$ should become $y$ and otherwise $z$ should become false. This construction represents $z \equiv x \land y$. Consider also the truth table in Table 10.
\[
\begin{array}{c|c|c}
  x & y & \text{IF } x \text{ THEN } y \text{ ELSE false} \\
  \hline
  0 & 0 & 0 \\
  0 & 1 & 0 \\
  1 & 0 & 0 \\
  1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
  x & y & x \text{ AND } y \\
  \hline
  0 & 0 & 0 \\
  0 & 1 & 0 \\
  1 & 0 & 0 \\
  1 & 1 & 1 \\
\end{array}
\]

Table 10: Truth table for IF \( x \) THEN \( y \) ELSE false.

5.3.4 Model

The complete mCRL2 model, containing all used sort declarations and function mappings, can be found in Appendix C.5. The basic structure of the modelled process is as follows:

```plaintext
sort
determine_conditions:
  LOCAL # ENV -> CONDITIONS;
var
  localvars: LOCAL;
environment: ENV;
eqn
determine_conditions(localvars, environment) =
determine all conditions
;
proc
  Determine_cruise_control_conditions (ccss: SPD, localvars: LOCAL) =
    sum environment: ENV . read_env(environment).
    Determine_new_state(  
      ccss,  
      localvars,  
      determine_conditions(localvars, environment)
    );

  Determine_new_state(ccss: SPD, localvars: LOCAL, conditions: CONDITIONS) =
    (current_state == DISABLED) -> CC_Disabled(ccss, localvars, conditions) <>
    (current_state == INACTIVE) -> CC_Inactive(ccss, localvars, conditions) <>
    (current_state == ACTIVE) -> CC_Active(ccss, localvars, conditions) <>
    (current_state == INTERRUPTED) -> CC_Interrupt(ccss, localvars, conditions) <>
    (current_state == ERROR) -> CC_Not_Reliabable(ccss, localvars, conditions)
    ;

  CC_Disabled(ccss: SPD, localvars: LOCAL, conditions: CONDITIONS) =
    (error conditions active) ->
    Calculate_Target_Speed(ccss, localvars, state := ERROR) <>
    (enable conditions active & no disable conditions active) ->
    enable . Calculate_Target_Speed(ccss, localvars, state := INACTIVE) <>
    no_action . Calculate_Target_Speed(ccss, localvars, state := DISABLED)
    ;

  CC_Inactive(ccss: SPD, localvars: LOCAL, conditions: CONDITIONS) =
    (error conditions active) -> disable .
    Calculate_Target_Speed(ccss, localvars, state := ERROR) <>
    (disable conditions active) -> disable .
    Calculate_Target_Speed(ccss, localvars, state := DISABLED) <>
    (activation conditions active &
```
...
The lower level Simulink model has a number of blocks. Each block is represented by a process or by a function in the mCRL2 representation. In line 2 to 8, a function declaration for determine_conditions can be seen. This function is used in line 17.

In line 103, we see that the initial state of the cruise control is the Determine cruise control conditions block. Here, the cruise control set speed is set to \( s_0 \), which represents 0 km/h. Some local variables are initialized. Local variables are for instance timers and the current state of the cruise control.

In Determine cruise control conditions, the system reads the inputs from the environment (line 12). As the environment is not within the system’s control, there is a read_env transition for every possible combination of inputs. In the actual model, this is expressed by a sum operator for each input, but these are combined in the model above to an input environment of type ENV for reading purposes.

After reading the environment variables, the systems goes through every block in the system to determine the new outputs. From Determine cruise control conditions, a transition is taken to Determine new state. Depending on the current state of the cruise control, a transition is taken to the appropriate process. Suppose the cruise control is in the DISABLED state, a transition is taken to the CC_Disabled process.

Determining the state of the cruise control is an important part of the behaviour of the system. This is modeled by a Stateflow diagram and is represented in line 29 to 82. In the ACTIVE state (lines 48 to 66) a lot of combinations are possible. Not all are shown in this abstract model.

All processes in lines 29 to 82 make a transition to the Calculate_Target_Speed process. If the cruise control is active, the transition is taken to the Determine_speed process. This process calculates and sets the new set speed of the cruise control. If the cruise control is not ACTIVE, the transition is taken back to Determine_cruise_control_conditions.

### 5.4 Model checking results

The functional requirements have been checked on this model using the mCRL2 tool set. The results are shown in this section.

In Section 4.4 the higher level model was also verified using the BTC Embedded Validator. An attempt was made to also use this tool to verify the requirements on this model. However, the tool was not able to verify any requirement on this lower level model. This could be due to the model being too large or the machine using the tool having too little memory (8GB). The BTC Embedded Validator was not able to give any result, even when all input signals were set to a constant value.

In the next table the model checking results are shown on the selection of requirements in Section 3.3. When a requirement fails, an explanation is given why this requirement fails. Some of
the requirements contain multiple cases. These requirements hold, only if all cases of these requirements hold.

Table 12 shows the results for all requirements. Appendix C.7 shows a detailed list of the results of all requirement checks.

1a. When disabled, a regulate action cannot be observed: There can be no rgl before enable when disabled. This requirement holds.

1b. If the function is enabled and the cruise control set speed is set (a regulate action has been seen) and the disable command is observed, the set speed must be set to 0 km/h. So, when a regulate action has been seen, there must be a stop rgl directly after a disable action. This requirement holds.

5. When an Off Command is observed and the function is active, a deactivate action must follow immediately. An activate action cannot occur directly after observing an Off Command. This requirement does not hold. Suppose the cruise control is active and the Off Command is observed. If at the same time a disable condition holds (for instance: The manual brake is pressed), the cruise control will be disabled and not deactivated.

10. There can be no regulate_cc(s90) or regulate_cc(s100) This requirement holds.

14. When active and the Increment Command is true, the current value of SetSpd must be increased. This requirement does not hold. When the speed of the cruise control is set to s80, the Increment Command does not set the cruise control speed to a higher value.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>true</td>
</tr>
<tr>
<td>4</td>
<td>false</td>
</tr>
<tr>
<td>5</td>
<td>false</td>
</tr>
<tr>
<td>6</td>
<td>true</td>
</tr>
<tr>
<td>7</td>
<td>true</td>
</tr>
<tr>
<td>8</td>
<td>true</td>
</tr>
<tr>
<td>9</td>
<td>true</td>
</tr>
<tr>
<td>10</td>
<td>true</td>
</tr>
<tr>
<td>11</td>
<td>true</td>
</tr>
<tr>
<td>12</td>
<td>false</td>
</tr>
<tr>
<td>13</td>
<td>false</td>
</tr>
<tr>
<td>14</td>
<td>false</td>
</tr>
<tr>
<td>15</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 12: Model checking results.
5.5 Evaluation

The lower level model has some complex constructions. These constructions do not show faulty behaviour, but they are unnecessary or can be changed to a more understandable construct.

We have tried to verify the requirements using two tool sets: mCRL2 and the BTC Embedded Validator. However, the Embedded Validator was not able to verify any requirement on this model. The mCRL2 tool set was able to do the model checking.

The lower level model shows less failures in the requirement checks (6 out of 15). All requirements that fail, fail because the requirements are not strong enough.

Model checking on the higher level showed more failures. This hints that the models are not equal. Section 6.1 shows the differences between the requirement checks on higher and lower level model.
6 Comparing the higher and lower level models

The higher level model can be seen as a specification and the lower level can be seen as an implementation. Hence, one would like that all behaviour that is described in the higher level model, is also present in the lower level model. If this is the case, it can be stated that the models are the same. Or, that the implementation has been made according to the specification.

The higher level does not consider every detail, which is modelled in the lower level model. For instance, when an error occurs, the system goes to an error state. In this section we would like to compare the higher and lower level model to each other, with respect to the behaviour described in the higher level model. Between the behaviour of both models, an equivalence (see Section 2.2) has to be found. Some behaviour is only modelled in the lower level model. From this behaviour has to be abstracted. This is elaborated upon in Section 6.3.2.

In this section, the higher and lower level of the DAF cruise control are compared to each other in three different ways:

First, the model checking results of Section 4.4.1 (for the higher level model) and Section 5.4 (for the lower level model) are compared to each other. Section 6.1 elaborates on this.

Secondly, the basic behaviour of both models is visualized. A labelled transition system (LTS) is generated showing the abstract behaviour of both models. Section 6.2 discusses this.

The last method of comparison is to compare the actual LTSs generated from the mCRL2 representations using the mCRL2 tool set. Section 6.3 shows the results of these comparisons.

This last method is analyzed on state spaces generated by the tool in Section 6.4. This also discusses the usability of this method on larger systems.

Section 6.5 evaluates the process and the results.

6.1 Differences in requirement checks

In Section 4.4.1 and Section 5.4 requirements have been checked on the higher and lower level respectively. The results of these requirements checks can be compared to each other. Since both models should show the same behaviour, the requirement checks should also show the same results.

Table 13 compares the results of the requirement checks to each other. The requirements that show different results on the models are marked by a star (*).

This table shows that four requirement checks do not show the same results. For this behaviour, we can state that the models are not the same. For instance, both models show different behaviour when the cruise control is disabled. In the higher level model the output signal SetSpd retains the old value when disabled (failure of requirement 1) and the lower level sets the output signal SetSpd to 0 km/h.

Although this is a basic way of comparing models, it already shows some differences in behaviour between the models. In Section 6.3 these differences are also found (among others).
<table>
<thead>
<tr>
<th>Requirement</th>
<th>Higher level</th>
<th>Lower level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1*</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>3*</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>4</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>5</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>6*</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>7</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>8</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>9</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>10</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>11*</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>12</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>13</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>14</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>15</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 13: Comparing the results of the requirement checks on both models

6.2 Visual comparison

Another method of comparing two models is a visual comparison. A visual comparison can be useful to get a first impression of the two models. For both models a Labelled Transition System (LTS) has been created to show their basic behaviour. This basic behaviour can be compared using a visual inspection. Only the following actions are visible:

- enable
- disable
- activate
- deactivate
- rgl, without any parameters.

Using the following commands in the mCRL2 tool set, an LTS is constructed for each mCRL2 specification. As can be seen in the last step, the models are converted using branching bisimulation ("-ebranching-bisim").

```
mcrl22lps
lpsactionrename
lps2lts
ltsconvert -ebranching-bisim
```

The rename file for the lpsactionrename was defined as follows:

```
no_action → τ
read_env(environment) → τ
regulate_cc(s) → rgl
```
Figure 14: Comparing the abstract models.

Figure 14 shows the abstract models generated. Figure 14a shows the abstract visualization the GVF model and Figure 14b shows the visualization of the MDS model. Some \( \tau \) actions are present in both models, due to the fact that \texttt{read\_env} and \texttt{stop\_rg1} have been transformed to an internal action \( \tau \).

Both models are converted using branching bisimulation. Since the resulting LTSs are not the same, these models are not branching bisimilar. Because these abstracted models are not branching bisimilar, we can state that the original models are also not branching bisimilar.
Checking equivalence between these two models using the mCRL2 tool set shows that these two models are weak-trace equivalent. Table 14 shows which states can be related. Since the initial states of the two models can be related, the models can be considered weak-trace equivalent.

### 6.3 Equivalence checking

In this section the two models are compared using the mCRL2 tool set. The mCRL2 tool set is capable of comparing two LTSs based on an equivalence (Section 2.2). To compare the higher and lower level models, the mCRL2 models created in Section 4.3.3 and Section 5.3.4 had to be adapted. Section 6.3.1 discusses this issue.

After the models are adapted, they can be compared using any equivalence provided by the mCRL2 tool set. Section 6.3.2 shows the results of this comparison.

#### 6.3.1 Changing the models

Before comparing the higher and lower level model, a difference between them has already been observed and adapted. This difference is the following:

Both models have different input signals as can be seen in Section 4.2 and Section 5.2. Also, the lower level model has more input signals than the higher level model.

Since these signals are not the same, comparing the models is not yet useful. The models are different, because they use different inputs. Hence, the input signals of both models had to be compared to each other. For nearly every input in one model, a matching signal in the other model was found.

One example of different input signals is the following: The higher level model uses a different variable for each input of the driver. These are the Set Command, Resume Command, Accelerate Command, Decelerate Command, Increment Command and the Decrement Command. This also showed a failure in model checking with requirement 11 (Section 4.4.1). In the lower level these signals are represented by only one input variable, which can have these different commands as its value. This difference has also been found when model checking with requirement 11 (Section 6.1). To overcome this issue, the higher level model has been adapted to have the same input

<table>
<thead>
<tr>
<th>GVF</th>
<th>MDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>10,17</td>
</tr>
<tr>
<td>3</td>
<td>13,15</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 14: States that are weak-trace equivalent in Figure 14
signal as the lower level model. This also fixes the failure of requirement 11 in the higher level model.

Some input signals could be disregarded. For instance, the lower level takes error signals into account. These kind of signals are not considered in the higher level model. Hence, the error signals are disregarded when comparing the models. This means that the behaviour of the lower level model with respect to these error signals is not considered.

Appendix D.1 lists all the input signals that are connected and which are disregarded.

6.3.2 Comparison

In the first attempt to compare the models, the vehicle speed has been set to a constant value. Setting the speed to a constant value reduces the model sizes and therefore it takes less time to compare them to each other. The chosen vehicle speed is 50 km/h. This speed allows for enabling and activating the cruise control. It is also roughly in the middle of the allowed set speed for the cruise control.

Section 6.2 shows that the abstracted models are not branching bisimilar. So the original models are also not branching bisimilar. However, that section also showed that the abstracted models are weak-trace equivalent. Hence, the choice has been made to initially check for differences between the models using weak-trace equivalence.

Using the following commands in the mCRL2 tool set, an LTS is constructed for each mCRL2 specification. The branching bisimulation conversion (the last command) reduces the number of states and transitions.

```
mcrl22lps
lps2lts
ltsconvert -ebranching-bisim
```

After these commands are executed for both models, they are compared using the following command:

```
ltscompare -eweak-trace.
```

This last command shows a counterexample in the form of a witness trace if the models are different. This trace shows a sequence of actions in the model. It shows behaviour, which is not present in one of the two models.

For every difference found between the models, the higher level model is adapted. This model is easier to understand, since is it much smaller and less complex than the lower level model. The goal is to reach equality in the end. After we reach equality, all differences can be shown in the resulting altered higher level model.

The differences found between both models are elaborated upon in the next sections. In these sections, the Stateflow model of the higher level model (Section 4.1, Figure 10) is often referred to explaining the differences.

Appendix D.2 lists the traces of the differences found. Appendix D.4 lists some extra notes.
Difference 1

In the lower level model, the cruise control set speed is constantly set to 0 km/h, when the cruise control is not active. In the higher level model, this is not the case. After the cruise control set speed has been set in the higher level model, the set speed is never set to 0 km/h (except when the engine is turned off). When checking requirement 1 and 6 (Section 4.4.1), this has already been shown. Section 6.3 also showed this difference between both models.

Change The change made to the lower level is to hide the stop_rgl transition. By hiding this transition, the stop_rgl transition is changed to a $\tau$-transition and can be seen as an internal action.
To fully understand the difference between the two models, an understanding is needed how the activation conditions are calculated in the lower level model. Figure 15 shows a simplified view of the subsystem Determine Set Activation, which determines the activation conditions in the lower level model. It has a flip flop, which can be set and reset. This flip flop is set, when the (boolean) activation conditions are true. This means that, for instance, the SET button is pressed. The flip flop can be reset by a couple of conditions. This subsystem has one output Activation conditions active, which is a boolean that states if the cruise control should activate. This boolean output is only true, when the flip flop is set and the primary retarder is inactive.

The rest of this section has been removed due to confidentiality.
Difference 3

In the higher level model, the condition to be disabled by the driveline is stronger than the condition in the lower level model. The condition on the disable transition in the higher level model is

\[
\text{Driveline interrupted} \land \land \text{Driveline timer timeout reached}
\]

The state DRIVELINE_INTERRUPTED_DRIVER in the higher level model has a timer that registers the time that the driveline has been interrupted. If a timeout is reached, the cruise control is disabled. In the lower level model, the cruise control is disabled immediately when the driveline is interrupted.

Change The driveline timer has been removed in the higher level model.

Note that removing the timer makes the state DRIVELINE_INTERRUPTED_DRIVER unnecessary. The only transition to this state requires the driveline to be interrupted. Now, the cruise control is immediately disabled when this happens. Hence, this state is never reached.
Figure 16: The DECELERATE activation condition in the subsystem Determine Set Activation.

Figure 16 shows a part of the Determine Set Activation subsystem. This figure shows the activation conditions (see Figure 15), which set the flip flop to true. One can see that the DECELERATE command has an extra restriction to activate the cruise control. This figure shows that the DECELERATE command only sets the flip flop to true, when the previous cruise control set speed value is larger than 0. In other words: The cruise control set speed should have already been set. In the higher level model, the DECELERATE command can activate the cruise control without this restriction.

Change In the higher level model, this extra condition is added: The previous cruise control set speed value should be at larger than 0 when the Decelerate Command is pressed to activate the cruise control.
Difference 5

This text has been removed due to confidentiality.
Difference 6

This text has been removed due to confidentiality.
Difference 7

This text has been removed due to confidentiality.
In this case, the primary retarder has been set to false.

![Figure 17: Trace visualized: A SET command follows immediately after a RESUME command within the delay time of the primary retarder deactivated variable. The lower level responds to SET instead of RESUME.](image)

Figure 17 visualizes this witness trace. The steps are explained as follows. We assume the cruise control is already enabled (thus in the INACTIVE state):

1. A SET command is observed. Both models go to the ACTIVE state immediately after the delay that is present in both models. It sets the cruise control set speed to the actual vehicle speed.

2. An deactivate condition is observed. Both models go to the INACTIVE state.

3. A RESUME command is observed. In the higher level model, a delay was added. It now has to wait one time stamp. In the lower level model, the resume flip flop has the value true, but the primary retarder deactivated signal is delayed. Hence, the value of the resume conditions is false.

4. The higher level model resumes the cruise control (due to the RESUME command in the previous step) and sets the speed to the previous set speed. The SET command is also observed. In the lower level model, the activation flip flop and the resume flip flop are now set to true. The value of primary retarder deactivated is set to true after the delay of one step. Both flip flop are true in the lower level model. The activation flip flop has a higher priority than the resume flip flop, hence the cruise control speed is set to the current vehicle speed in the lower level model.

This scenario is not possible in an actual DAF truck. This is due to the SET and RES buttons that are used in the vehicle, which are shown in Section 2.6 in Figure 9. However, if the hardware buttons change in the future, this scenario might occur.

Change We want the higher level to show the behaviour of the lower level model. So, this
behaviour is added in the delay state. If a SET command directly follows the RESUME command, the cruise control sets the speed to the vehicle speed. Figure 18 shows this addition.

Figure 18: Delay changed to cope with a Set command directly after receiving a Resume command.
Equality

With the changes made in the previous sections the models reach weak-trace equivalence. Note that the value of the primary retarder is not considered anymore. In both models the value of the primary retarder is set to false. This value has been disregarded, since it causes very different behaviour in both models. Changing the higher level model according to the lower level behaviour would change the higher level model too much to still have a model close to the original DAF model.

Adding possible vehicle speeds to the maximum of the 11 speeds of the original mCRL2 model, does not violate the weak-trace equivalence. Hence, no extra differences are found.

The resulting models have been checked on equality using branching bisimulation. These two models are also branching bisimilar. This can be explained by the functionality that is added to the higher level model. The branching structure, or "the moment of choice" is the same for both models. The approach used to model each DAF model can be used to explain why the models are branching bisimulation equivalent now: Both model first "read the environment". Based on the inputs that are read from this environment, a choice is made to change outputs variables and internal states. This moment of choice is the same in both models, hence they are branching bisimulation equivalent.

Figure 19 shows the altered higher level model. Note that a delay is not shown in this altered model. This delay should be added before entering the ACTIVE state in Figure 19b. The variable SetSpdMem has been added to remember the old cruise control set speed. This allows the output SetSpd to be set to 0, when the cruise control is deactivated or disabled. The state DRIVELINE_INTERRUPTED_DRIVER has been removed. This state is not reachable anymore, since the cruise control is disabled when the driveline is interrupted. This disable condition is shown in the transition from ENABLED to DISABLED in Figure 19a. A condition is added when a DECELERATE button is pressed. This can only activate the cruise control, when the previous cruise control set speed is higher than 0 km/h.

Appendix D.3 shows how the original DAF model is altered for all differences.
(a) A simplified Stateflow model showing the altered ENGINE OFF and ENGINE RUNNING states.

(b) A simplified Stateflow model showing the altered ENABLED state.

Figure 19: A simplified Stateflow model of the higher level cruise control.
6.4 Analysis

This section shows an analysis of the approach to compare two models of the cruise control software investigated. Firstly, Section 6.4.1 investigates how the created mCRL2 models react on different numbers of allowed vehicle speeds. Section 6.4.2 shows an analysis on how this approach of comparing is applicable on larger systems.

6.4.1 Varying vehicle speeds

To investigate how the mCRL2 models respond to the number of possible vehicle speeds, several state spaces have been generated for these models using a different number of vehicle speeds. This gives an insight on how these particular models respond to a change of total number of inputs. The vehicle speed is used, because the allowed number of speeds is easily changed.

The models that have been examined are those that were prepared for comparison. These are the models where the input signals are the same as discussed in Section 6.3.1. Also, the higher level model after comparison has been investigated.

The number of allowed vehicle speeds were varied from 1 to 11 possible speeds. For \( n \) possible speeds, the set of allowed speeds is defined by

\[
\{10 \cdot (i - 1)|1 \leq i \leq n\}
\]

So, for one allowed speed, this set contains \( \{0\} \). For two allowed speeds the set contains \( \{0,10\} \).

Note that for the three allowed speeds, the values of the speed can vary between 0 and 20 km/h. This cannot enable the cruise control, because a vehicle speed of 30 km/h or higher is required. Hence, little behaviour is expected between one to three allowed vehicle speeds.

![Changes in the number of states](image)

(a) Changes in the number of states.

![Changes in the number of transitions](image)

(b) Changes in the number of transitions.

**Figure 20:** Higher level model - Before comparison.

Figure 20 shows the results on the analysis of the higher level model before comparison.

Figure 20a shows how the number of generated states change, when allowing more vehicle speeds. The line shows the number of states compared to the number of allowed vehicle speeds. Two extra lines are shown, which have a growth rate of \( x^2 \). For 1, 2 and 3 allowed vehicle speeds, there are only 2 states in the model. This was expected, as explained earlier. These two states are shown in
Figure 21. Suppose we have initial state \( s \) and another state \( t \). State \( s \) has an action \texttt{read\_env} (with an arbitrary set of parameters) to \( t \) and state \( t \) has an outgoing action \texttt{no\_action} to \( s \). The cruise control is not enabled by these speeds. From the point where 4 speeds are allowed, the line seems to grow with a rate of \( x^2 \), until 9 allowed speeds. The line offsets after 9 allowed speeds. This can be explained by the speeds 90 and 100 being added. These speeds cannot be set by the cruise control. Hence, these speeds do not introduce as much extra behaviour as the previous cases.

Figure 20b shows how the number of generated transitions change. This model shows little increase in transitions for 1,2 and 3 allowed vehicle speeds. Again, since the cruise control is not enabled in this case, there is little behaviour to be shown. From 4 to 9 allowed vehicle speeds, the growth of the line is \( x^2 \). After 9 allowed speeds, a small change can be seen again. This can be explained by the fact that the speeds 90 and 100 are added, which do not add extra behaviour to the system.

Figure 22 shows the results on the analysis of the lower level model. Figure 22a shows how the number of states change by adding allowed vehicle speeds. Figure 22b shows the changes in the number of transitions. As discussed earlier, allowing 1 to 3 or more than 9 vehicle speeds shows a nod in the plotted line. This can be explained by the cruise control not being enabled for 1 to 3 allowed speeds and allowing more speeds than 9 does not introduce more behaviour. For both pictures, it seemed there was a linear growth between 4 and 9 allowed vehicle speeds. However, adding a linear line shows that the growth is larger dan linear from 4 to 9 allowed vehicle speeds.

The results for the higher level model (after comparison) can be seen in Figure 23. Firstly, Figure 23b shows a growth rate of \( x^2 \) again for the number of transitions compared to the allowed vehicle speeds. This rate is quite similar to the growth of as shown in the higher level model before comparison (Figure 20b). It shows less transitions, which can be explained by some
transitions being removed in the model during comparison. For instance, the transition to the DRIVELINE_INTERRUPTED_DRIVER state.

Figure 23 shows a linear line, which is different from the other models. The line is equal to the linear line. Also, this adapted higher level model contains a lot less states than the higher level model before comparison. This can be explained by all the changes that are made in this model. Next to that, the third parallel state within the ACTIVE state has been removed. Removing this extra state causes the growth rate of this line to change.

*Due to confidentiality, this case is not further explained in this document.*

### 6.4.2 Larger systems

In this thesis, only the higher and lower level model of the cruise control vehicle function were compared to each other. Since there are a lot more software components within a vehicle, we would like to know if this comparison is also applicable on a larger system. For instance, a software model that combines multiple vehicle functions.

In [9] a higher level model of a DAF cruise control system was investigated. This is not the same model as investigated in this thesis. In his research, a Vehicle Function Architecture (VFA) was also checked. This VFA combines different vehicle function components, including the cruise control component. An estimation is made on the size a lower level model VFA. For this, we compare states spaces from [9] to the state spaces generated in this thesis.

Table 15 shows some information about the mCRL2 representation of several software models investigated in thesis and in [9]. The number of inputs is shown for each software model. Since the inputs of the higher and lower level model in this thesis were matched to each other, the number of inputs for these models is the same. The number of combinations that can be created from these inputs are also shown in this table. Since all combinations of inputs are inspected during model checking and comparing models, this is a important number. The VFA is modelled in a different way. The inputs (and combinations) are spread throughout its mCRL2 representation. These numbers cannot be easily found. The number of states and transitions generated by the mCRL2 tool set are shown for all models.
One can see that the cruise control in [9] has less inputs and less possible combinations of those inputs than the DAF cruise control investigated in this thesis. Therefore, less states and transitions are generated by the mCRL2 tool set. The number of states in the lower level model cruise control investigated in this thesis is 2.807 times as much as the states in its corresponding higher level model. The number of transitions is roughly twice as much. The difference between the number of states and transitions of the higher and lower level model can be explained by the way they are modelled. The lower level model has more internal (local) variables. These variables are for instance the current state of the cruise control and the flip flops for resume and activation. These internal variables do not appear in the higher level model. The current state of the higher level model is modelled by separate processes.

In [9] a Vehicle Function Architecture (VFA) that manages “Driving and Braking” was investigated. This VFA was translated to mCRL2 and this model has 625.935 states and 146.996.525 transitions. Suppose the lower level representation of this VFA would scale the same way as investigated cruise control system in this thesis. The resulting lower level model would have 1.756.999.545 states and 317.218.501 transitions. For this statement to hold, we have to assume that the complexity between higher and lower level for the VFA is the same as the cruise control model in this thesis.

Suppose we also take the growth of complexity of the cruise control in account. The higher level model of the cruise control in this thesis has 91 times as many states as the cruise control in [9]. It has 130 times as many transitions. If we assume that the total functionality of the VFA has grown with a linear rate, the total number of states would be 159.886.958.595, and the total number of transitions 41.238.405.130. This number of quite high and lies on the edge of what is possible to work with using the techniques and machines available.

This is of course a rough estimation and further research would be needed to get exact numbers.

A side note for this analysis is the following. Since the VFA combines several vehicle functions, it is sufficient to compare each higher level vehicle function to its lower level implementation. If all separate higher and lower level vehicle functions are equal, one could assume the higher and lower level VFA models are also equal.

Suppose one wants to check requirements on the lower level model of a VFA. An approach for this is to combine the requirement checking for each vehicle function separate as discussed in [9] with the comparing of models as discussed in this thesis. For each vehicle function, the higher and

<table>
<thead>
<tr>
<th>Software model</th>
<th>Inputs</th>
<th>Combinations</th>
<th>States</th>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cruise control of [9]</td>
<td>8</td>
<td>5280</td>
<td>471</td>
<td>850.390</td>
</tr>
<tr>
<td>DAF cruise control (higher level)</td>
<td>13</td>
<td>112.640</td>
<td>42.633</td>
<td>110.879.409</td>
</tr>
<tr>
<td>DAF cruise control (lower level)</td>
<td>13</td>
<td>112.640</td>
<td>119.653.080</td>
<td>239.275.698</td>
</tr>
<tr>
<td>Vehicle Function Architecture (higher level)</td>
<td>-</td>
<td>-</td>
<td>625.935</td>
<td>146.996.525</td>
</tr>
<tr>
<td>Vehicle Function Architecture (lower level, estimated)</td>
<td>-</td>
<td>-</td>
<td>159.886.958.595</td>
<td>41.238.405.130</td>
</tr>
</tbody>
</table>

Table 15: Information on the software models investigated and created.
lower levels are compared. If they are all the same, the behaviour of the higher and lower level VFA should be the same. Hence, model checking can be done on the higher level model VFA.

6.5 Evaluation

Comparing the models showed 8 differences in the end. These differences have been verified on the original DAF models. This method showed that it is possible to compare two real-life models to each other. The approach used is quite time-consuming, since the differences have to explained and verified. After that, the models have to be adapted accordingly to find the next difference.

Analysis has shown how the models react on different numbers of allowed vehicle speeds. The differences in speeds that cannot enable the cruise control (0, 10 and 20 km/h) or cannot be set by the cruise control (90 and 100 km/h) show a noticeable difference in number of states and transitions generated. The adapted higher level model (after comparison) showed a different graph for the generated number of states. This can be explained by a lot of functionality that was removed from this model, to make it match the behaviour of the lower level model. The growth of the models is quadratic in the number of vehicle speeds allowed. This shows that to have a discrete type for the vehicle speed (Section 3.2) has been a good decision.

Analysis showed that the number of inputs and the possible combinations of those has a large effect on the eventual state spaces of the models. An estimation is given for a possible state space of a lower level Vehicle Function Architecture. It is still a rough estimation and more research could be done in this area. The size found for the state space of the VFA is quite high and lies on the edge of what is possible to work with using the techniques and machines available.
7 Conclusions and recommendations

In the first part of the project, a Stateflow model of the higher level cruise control was translated to mCRL2. Model checking was performed on this translation with a set of requirements provided by DAF. This verification showed that 11 out of 15 cases violate a requirement. Most violations were caused by requirements being too weak. Four requirements showed an actual fault in the original higher level model. These faults could easily be corrected in the higher level model.

In the second part of the project, a Simulink model of the lower level cruise control was translated to mCRL2. Again, this model was formally verified using the same set of requirements as in the first part of the project. This verification showed that 6 out of 15 cases violate a requirement. All of them were caused by requirement being too weak. Hence, the fault found in the higher level model are not present in the actual implementation.

In the third and last part of the project, the higher and lower level model have been compared to each other. Eight differences were found. These have been verified on the original Simulink / Stateflow models. The differences found were quite small and they do not show faulty behaviour which would noticeable by the driver of the vehicle.

With this project, we intended to investigate the possibility to compare two real-life software models to each other. The higher level and lower level model of the a cruise control were examined. With software becoming an increasingly prominent part of vehicles, it also tends to grow larger and more complex. In order to obtain a high software quality in these vehicles, designers need new techniques to prove that their product performs as required. During this project, some problems in real-life systems have shown the power of formal verification. Next to that, the formal comparison of the models showed that the implementation did not fully behave as specified by the higher level specification. These findings show that formal techniques to verify and compare system are quickly becoming more important in the automotive industry. The approach used during this project was to manually translate Stateflow and Simulink models to mCRL2, which is very time consuming, error-prone and requires an extensive amount of knowledge of the mCRL2 modelling language. For future research, it would be interesting to investigate the possibility to automate the translation from Stateflow models to mCRL2. This would make the tasks performed in this project less time consuming and less error-prone. The main challenge here will be to find a translation scheme that preserves the original behaviour, but also produces mCRL2 models that allow for comparing and verification.

Formal comparison techniques - as used in the last section of this document - would be useful to show equality between the higher and lower level model in the design process. However, in order to fully realize this, a solution would have to be found to automate this process, or another method to make it less time-consuming.
8 List of references


Appendix

Confidentiality

Note that this is the public version of the document. This document omits the complete lists of requirements, used Simulink models and constructed mCRL2 models. All appendices are excluded from this version of the document.
A Requirements

A.1 Conversion Dutch functional requirements to English functional cruise control requirements

A.2 Conversion English functional requirements to $\mu$-calculus

A.3 Altered requirements
B Higher level model

B.1 Stateflow model

B.2 Inputs and outputs

B.3 Decisions

B.4 Translation to mCRL2

B.5 Transition renamings

B.6 Requirement checks

B.6.1 Altered requirements

B.7 Faults observed for the basic requirements
C  Lower level model

C.1  Simulink model
C.2  Inputs and outputs
C.3  Variables lower level model
C.4  Decisions
C.5  Translation to mCRL2
C.6  Transition renamings
C.7  Requirement checks
D Comparing the models

D.1 Changes to variables

D.2 Traces

D.3 Altered higher level model

D.4 Notes