MASTER

Vehicle function correctness
using mCRL2 to verify StateFlow charts and Simulink models

Schoren, R.J.A.

Award date:
2012

Master Project CSE

Public Report

Vehicle Function Correctness

Using mCRL2 to verify StateFlow Charts and Simulink Models

by

Rob Schoren

Supervisor
prof.dr.ir. J.F. Groote

Tutor
dr.ir. R.G.M. Huisman

Eindhoven University of Technology
Department of Computer Science

On behalf of DAF Trucks N.V.

February 16, 2012

Abstract

In this report, the possibilities of performing formal verification on models used in the automotive industry are investigated. For this purpose, we found an approach to make a fitting translation from Matlab Simulink and StateFlow models to the mCRL2 modelling language, which provides the formal verification techniques needed to determine whether a model satisfies a number of requirements.

We performed such a translation on both the relatively small Cruise Control system model and the larger complete Vehicle Function Architecture as designed by DAF Trucks N.V. For both models, the system's behaviour proved not to be entirely as expected by the designers, leading to several requirements being unsatisfied. Further analysis provided insights to understand the cause of these problems. The solutions to the discovered problems generally appeared to be obvious and could be implemented practically effortlessly by the DAF Trucks N.V. designers. Remarkably, one of the proposed requirements seems to be first known example of a natural property impossible to express in a modal μ-calculus formula.

The report shows not only the potential of formal verification, but also the challenges that the automotive industry will have to overcome to use it. Composing an accurate set of unambiguous requirements turned out to be more complex than expected and requires constant reviewing and discussion. Moreover, considering the approach taken in this project, both the construction of a mCRL2 model and the verification are a time consuming task and require experience with the toolset. In order to be able to perform formal verification itself, DAF needs to gain experience in this field. For the mCRL2 toolset, there are multiple improvements possible to be used more effectively in the automotive industry, such as an automatic translation from Simulink to mCRL2 models.

Contents

1 Introduction

This document describes the findings of a project, in which the behaviour of several Vehicle Functions was modeled using the modeling language mCRL2 on behalf of DAF Trucks N.V. The project's goal is to investigate the applicability of formal verification in the automotive industry, and to help DAF making a step towards building provably correct software systems.

The vehicle function software developed at DAF Trucks N.V. is modelled in the Matlab Simulink modelling environment. This enables the designers to split the functionality into multiple components, to maximize separation of concerns. Even though this is a wise approach to modelling, the many components that communicate with each other may not always interact as expected. Currently at DAF, the models are manually reviewed and test scenarios are used to check whether the resulting behaviour is as desired. However, these test scenarios show only the behaviour of the system in that one very specific scenario, giving no guarantee that the system reacts the same to slightly different scenarios. More importantly, the test scenarios usually describe common situations, while problems primarily occur in rare cases that have been overlooked, i.e., the boundary cases. Therefore, DAF may benefit from using formal verification on their models to assure that they unquestionably satisfy important requirements.

In section 2, a short introduction is given to the mCRL2 modelling language and modal μ-calculus. Section 3 contains the results of the first phase of the project, in which a Cruise Control system was modelled based on a set of functional requirements. Section 4 describes the second phase of the project, in which the Cruise Control StateFlow model as designed by DAF was translated to mCRL2 and analyzed. The third phase of the project comprised the translation of the "Driving and Braking" Vehicle Function Architecture to mCRL2, the results of which can be found in section 5. Section 6 concludes the main document with a discussion and future recommendations. In this public version of the report, the complete lists of requirements, used Simulink / StateFlow models and constructed mCRL2 models are omitted from appendices A.1 - C.7, even though they are referenced to throughout this document.

2 Preliminaries

This section gives a short introduction to the mCRL2 modelling language, and to modal μ-calculus as a language to express behavioural properties.

2.1 mCRL2

The mCRL2 toolset is developed at the department of Mathematics and Computer Science of the Technische Universiteit Eindhoven, in collaboration with LaQuSo, CWI and the University of Twente. It is available as a free download from the mCRL2 homepage [2]. Other major toolsets are UPPAAL, FDR, CADP, Spin and νSMV.

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. The most important entities in the mCRL2 modelling language are processes, describing the behaviour of a system. A process can be specified using the following constructs:

Construct	**Example**
Action	$P \ = \ a;$
Sequential Composition	$Q \ = \ a \ . \ b;$
Alternative Composition / Choice	$R \ = \ a \ + \ b;$
Recursion	$S \ = \ a \ . \ S;$
Data	$T(d : Bool) \ = \ c(d);$
Condition	$U(d : Bool) \ = \ (d) \ \to \ a \ <> \ b;$
Summation	$V \ = \ sum \ d : Bool \ . \ c(d) \ = \ c(false) \ + \ c(true);$
Parallel composition	$W \ = \ a \vert\vert b \ = \ a.b \ + \ b.a \ + \ a \vert b;$

Table 1: mCRL2 process constructs

The example processes in Table 1 describe the following behaviour. Process P performs a single action a. Process Q first performs action a, then b. Process R performs either action a or b. Process S continuously performs action a. Process $T(d)$ performs action $c(d)$, for the same value of boolean variable d. Process $U(d)$ performs action a if $d = true$, or action b if $d = false$. Process V describes the alternative composition of the actions $c(d)$ for all possible values of boolean variable d. Process W performs actions a and b in any order (sequentially or simultaneously).

2.2 modal μ-calculus

A modal μ-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 modal μ-calculus formulae used in this project is given by the following BNF.

$\phi ::=$	$true$	$\mid$	$false$	$\mid$	$[\rho]\phi$	$\mid$	$\langle\rho\rangle\phi$	
$\rho ::=$	α	$\mid$	$\rho \mathbin{.} \rho$	$\mid$	$\rho *$			
$\alpha ::=$	$\alpha \vee \alpha$	$\mid$	$a(d)$	$\mid$	$!a(d)$	$\mid$	$true$	

Table 2: modal μ-calculus syntax

In the syntax description in Table 2, ϕ represents a property, ρ represents a sequence of actions and α represents an action. The actual μ-calculus available is much richer, but we will not need additional constructs for this project.

The property *true* holds for any model, while *false* holds for no model. The formula $[\rho]\phi$ describes that ϕ holds in all states that can be reached by a sequence ρ, while $\langle\rho\rangle\phi$ describes that ϕ holds in some state that can be reached by a sequence ρ. To describe a sequence of actions ρ, operators for concatenation and iteration are available. The presence of at least one of two actions is described by $\alpha \vee \alpha$. The presence or absence of a data parameterized action $a(d)$ is represented by $a(d)$ or $!a(d)$ respectively. Action $true$ describes the presence of any action in a sequence. A more elaborate description on the modal μ-calculus can be found in [4].

2.3 Existing Simulink/StateFlow Verification Techniques

The designers at DAF Trucks N.V. use the Matlab Simulink environment to model their vehicle functions. This section provides an overview of the verification techniques that are already available for Simulink/StateFlow models.

In the Matlab Simulink environment, there is a tool named Simulink Design Verifier available to verify a model against requirements. The Design Verifier generates a number of test cases based on a given requirement. These test cases are then simulated and it is checked whether the model behaves as expected. This approach is likely to achieve a much higher coverage than hand written test cases. However, for some requirements it may not be possible to be effectively represented by a finite number of test cases. Also, performing an actual simulation for every generated test scenario could consume a lot of time, especially when using a large and complex model.

Another existing verification tool for Simulink/StateFlow models is the BTC Embedded-Validator. It translates Simulink/StateFlow models to a mathematical representation using dSPACE TargetLink. A formal property can then be checked against this property, giving formal proof that the model does or does not meet a given requirement. This approach is similar to the one used in this project, but the main difference is that we apply some simplifications to the models (see section 3.2.3) to allow more efficient verification. The TargetLink translation provides a high level of certainty to the verification results, as no abstractions are conducted when translating to the mathematical representation. However, keeping every detail in translation may also result in the verification being very time consuming, so for a large and complex model, this approach may not be suitable.

3 Cruise Control Design

This section describes the results of a case study, in which a model of a Cruise Control system was designed 'from scratch', based on a set of requirements. With this case study being the first part of the project, we intend to gain a better understanding of designing software systems using model verification in general, and of the desired behaviour of the Cruise Control system specifically.

3.1 Functional Requirements

This section describes the desired behaviour of the Cruise Control system. First, a global description is given to clarify the scope and context of CC within its surrounding vehicle functions.

3.1.1 Global description

The Cruise Control functionality of DAF Trucks N.V. is defined by the following short global description: "The cruise control function maintains vehicle speed at the required cruise control set speed selected by the driver, without operating the accelerator pedal. This speed is maintained on condition that the engine power is sufficient to maintain the required cruise control set speed."

The context of the Cruise Control system is defined by a number of fixed input and output signals (see appendix A.1). Table 3 below contains a selection of these signals that are used as an example throughout this section.

Inputs	**Outputs**
Ignition Switch Position	Activate Governor Request
Diesel Engine State	Set Speed
Transmission State	
Vehicle Stability Control State	
Resume Request	
Set Request	
Accelerate Request	
Decelerate Request	
Set Speed Increment Request	
Set Speed Decrement Request	

Table 3: The inputs and outputs of the Cruise Control used in this section.

3.1.2 List of requirements

The requirements listed in appendix A.2 describe the desired functionality of the CC Supervisor in terms of its inputs and outputs. Table 4 below contains a selection of requirements that are used as an example throughout this section.

3. De functie moet aanstaan als de hierna genoemde inputsignalen de genoemde waarden hebben: Ignition Switch Position is M (marche), Diesel Engine State is running, Transmission State is forward gear.

11. Als de waarde van inputsignaal Vehicle Stability Control State gelijk is aan controlling, moet de waarde van outputsignaal Activate Governor Request false zijn.

16. De waarde van outputsignaal Set Speed mag nooit hoger zijn dan 85 km/h.

19. Op het true worden van de waarde van een van de volgende inputsignalen: Resume Request, Set Request, Accelerate Request, Decelerate Request, Set Speed Increment Request, Set Speed Decrement Request, mag alleen gereageerd worden als alle andere genoemde signalen false zijn.

Table 4: A selection of Cruise Control requirements

3.2 Designed Cruise Control Model

In this section, it is explained how the Cruise Control system was modeled in mCRL2. It contains a global description of the model and the design decisions that were made in order to be able to effectively describe the behaviour. Also, the assumptions that have been made are listed, along with a discussion on why these assumptions are valid and necessary. More information on the mCRL2 modelling language and toolset can be found in [3] and [1]. For this case study, the July 2011 (SVN revision 9551) Release version of the mCRL2 toolset has been used.

3.2.1 Description of the mCRL2 model

This global description of the constructed model gives an abstracted view on the shape of the Cruise Control process. The complete mCRL2 model, containing all used sort declarations and function mappings, can be found in appendix A.3. The basic structure of the modeled process is as follows:

```
1.  CC(cc_state:CC_State, ccss:SPD) =
2.      sum environment: ENV .
3.      read_env(environment) .
4.      (
5.          % ----------------------------------- CC Off -------------------------------------
6.          (cc_state == Off && enable_conditions) -> on . CC_Idle(ccss) <>
7.          (cc_state == Off) -> no_action . CC_Off(ccss) <>
8.          % ----------------------------------- CC Idle -------------------------------------
9.          (cc_state == Idle && !enable_conditions) -> off . CC_Off(ccss) <>
10.         (cc_state == Idle && activation_conditions) -> activate . CC_Controlling(calculate_new_ccss) <>
11.         (cc_state == Idle) -> no_action . CC_Idle(ccss) <>
12.         % ----------------------------------- CC Controlling ---------------------------------
13.         (cc_state == Controlling && !enable_conditions) -> stop_regulate . off . CC_Off(ccss) <>
14.         (cc_state == Controlling && deact_conditions) -> stop_regulate . deactivate . CC_Idle(ccss) <>
15.         (cc_state == Controlling) -> no_action . CC_Controlling(calculate_new_ccss);
16.     );
17.
18. CC_Off(ccss:SPD) = CC(Off, s0);
19. CC_Idle(ccss:SPD) = CC(Idle, ccss);
20. CC_Controlling(ccss:SPD) = regulate_cc(ccss) . CC(Controlling, ccss)
21.
22. init
23.     CC(Off, s0);
```

In line 23, the initial state of the system is defined. We see that initially *cc_state* = *Off* and *ccss* = $s0$, so the Cruise Control is Off and its set speed is 0 km/h. During execution, the system constantly reads the environment status (line 3), i.e., it reads the values of all input signals. 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 line 2 for reading purposes.

After pulling the environment status, the system determines which transition should be taken (lines 6-15). The expressions used in the conditions are in terms of the inputs used in *read_env*, but abstracted to for instance *enable_conditions* and *activation_conditions* in the description above. When a transition has been taken to a new CC state (or the same state using a *no_action*), the system ends up in one of the CC state specific processes (lines 18-20). In *CC_Off*, the current set speed is erased (so the stored *ccss* is set to *s0*). In *CC_Controlling*, a *regulate_cc* transition is taken to update the *Set_Speed* output.

The Cruise Control system has two outputs, *b_Activate_Governor* and *Set_Speed*. Initially, their respective values are *false* and *s0*. When a *regulate_cc*(*ccss*) transition is taken, this means that *b_Activate_Governor* is set to *true* and *Set_Speed* becomes *ccss*. When a *stop_regulate* transition is taken, *b_Activate_Governor* is set to *false* again and *Set_Speed* to *s0*.

3.2.2 Design decisions

During construction of the mCRL2 model, some design decisions have been made that allowed for effective modeling of the desired system:

- A common problem in model verification is the state space explosion problem. If one does not choose a careful modelling strategy, the resulting process often ends up containing an unmanageable amount of states. For instance, our Cruise Control system could contain a number of states exponential to the number of input signals modelled, if a different transition would be used for reading each input signal. In order to avoid the state space explosion problem, the choice was made to combine the reading of the values of all inputs into one transition (*read_env*). This way, it is not necessary to store all these values in the following states. Rather, after reading all inputs, it is directly determined by the model which next transition should be taken, and many of these transitions will end up in the same states. Generating a Labelled Transition System leads to a model that contains a relatively small amount of states (4538), but a large number of transitions ($4,4 * 10^7$).
- Function mappings are used to separate some of the complexity of the model from the actual process description. In this way, it was possible to keep the process descriptions small and comprehensible. Also, this separation would make it easier to adapt the model to future changes in requirements.
- In the actual Cruise Control system, the outputs are constantly set to a value. This could be modeled by adding variables in all states that show us which values the outputs currently have. However, this would cause the state space to increase significantly. The decision was made to only model changes to the outputs. Thus, when the value of an output should be changed, there is a transition in the model that sets the new value. This value then stays on the output until it is changed again. This means that when we want to use the value of an output signal to describe a requirement in modal μ-calculus, we have to refer to the last occurred transition that changed the value of that signal.

3.2.3 Assumptions

For some aspects of a Cruise Control system, such as timing and the use of continuous variables, modeling in mCRL2 can be problematic. Therefore, we made some assumptions to overcome these problems:

- For this case study, 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 arbitrarily fast and always reacts in time.

- The system, as described by the requirements, uses continuous variables such as the actual vehicle speed. When modeling in mCRL2, we want to avoid a possible infinite number 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 could be used, using a *SPD* type with an arbitrary small step size between possible values (with some effort to adapt the function mappings).
- When a driver request is active, it is important to determine whether it was already active before, or that the boolean became true at this *read_env* transition. If a request is active at two consecutive *read_env* transitions, it is assumed that it stayed active for the *Accelerate* and *Decelerate* requests. For the other requests however, it is assumed that the request has shortly been inactive between the two *read_env* transitions.

3.2.4 Visualization

In order to obtain a graphical view on the model, we constructed a Labelled Transition System from the complete mCRL2 specification (which can be found in appendix A.3). First, we translated the model to a Linear Process Specification (LPS). There are many operations available to a specification in this form.

An operation we used here, is renaming all occurrences of certain transitions that match a specific form. For instance, we renamed the *read_env*(*environment*) action to *rd_env* (which has no parameters), ensuring that all *read_env*(*environment*) transitions get the same name, regardless the value of *environment*. Then, we translated the LPS to a Labelled Transition System (LTS). This was then reduced to a minimal LTS with (bisimulation) equivalent behaviour, resulting in the LTS displayed in Figure 1.

The approach in the above description is executed using the following commands available in the (July 2011 release version of the) mCRL2 toolset:

```
mcrl22lps
lpsactionrename
lps2lts
ltsconvert -ebisim
```

Note that the rename file used for *lpsactionrename* was defined as follows:

no_action	→	*tau*
regulate_cc(*ccss*)	→	*rgl*
stop_regulate	→	*stop_rgl*
read_env(*environment*)	→	*rd_env*

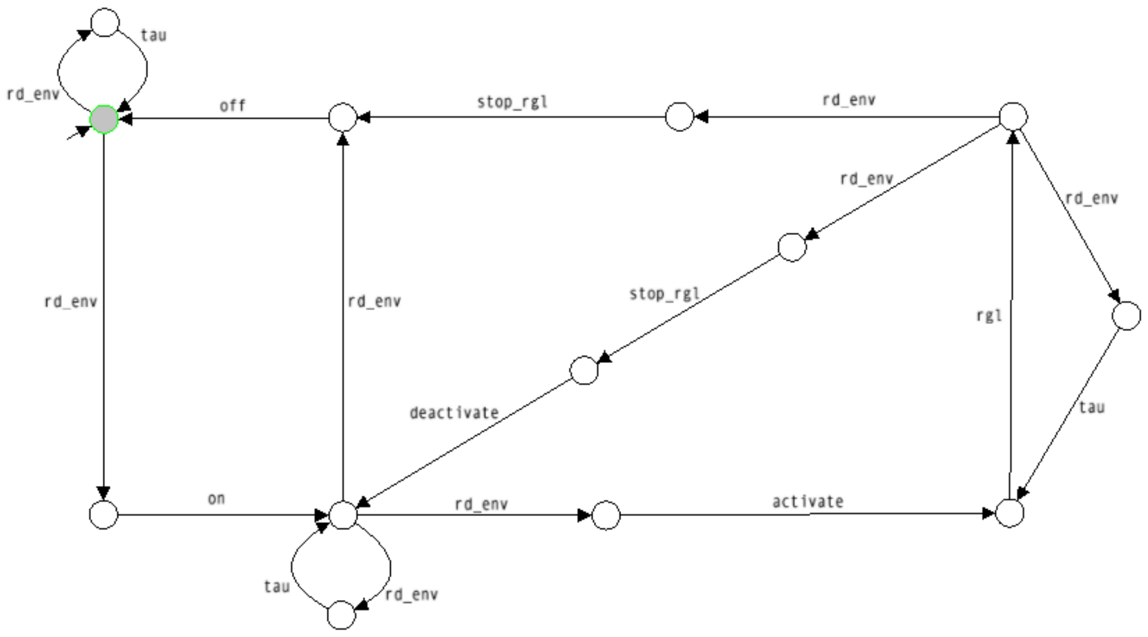


Figure 1: An LTS showing the behaviour of the model.

In Figure 1, we see the reduced LTS that was generated from the model. Representing an LTS in a graphical way often helps to gain a better understanding of the model (see section 3.3.1).

Initially, the system is in the colored grey state. Note that this state contains a loop with transitions *rd_env* and *tau*. So as long as the Cruise Control is Off, the system loops through these two states. Once the value of *environment* is such that the Cruise Control should be turned On, the *rd_env* transition to the bottomleft state is taken.

From here, there is only the possibility to take the *on* transition, leaving the Cruise Control in the Idle state. This state also contains a loop with transitions *rd_env* and *tau*, which is taken when the Cruise Control stays Idle. From here, the value of *environment* can cause the Cruise Control to be activated, taking the *activate* and *rgl* transitions.

Note that when *rgl* is performed, the outputs *b_Activate_Governor* and *Set_Speed* are updated. As long as the Cruise Control is Controlling, the system loops through the 3 transitions *rd_env*, *tau* and *rgl* on the right end of the figure. When the Cruise Control returns to the Idle or Off state, this is preceded by a *stop_rgl* transition, resetting *b_Activate_Governor* and *Set_Speed* to their initial values.

3.3 Model Checking

This section contains the results of the formal verification performed on the requirements in section 3.1. With this formal verification, it can be proved that a certain path of transitions can (or cannot) occur in a given mCRL2 model.

In appendix A.5, a translation to English and to the context of the designed mCRL2 model (see section 3.2) is given for the requirements that are listed in appendix A.2. These requirements are expressed as propositions in modal μ-calculus.

Note that some of the propositions use transition names that are not used in the process description of the mCRL2 model (but they are declared as actions). In these cases, the corresponding transitions in the model are renamed with *lpsactionrename* using the rename definitions as included in appendix A.4. In most cases, this comes down to removing all parameters of transitions (usually *read_env*) that are not used in the requirement.

Table 5 below contains the example requirements expressed in modal μ-calculus.

3.	When off, a *rd_enable*(*true*) must be directly followed by an *on*.	`[!on* . rd_enable(true) . !on] false && [true* . off . !on* . rd_enable(true) . !on] false`
11.	- As long as *ActivateGovernor* = *false*, there can be no *activate* and *rgl* directly after *rd_ebs*(*true*). - When *ActivateGovernor* = *true*, a *rd_ebs*(*true*) must be directly followed by a *stop_rgl*.	`- [!rgl* . rd_ebs(true) . activate . rgl] false` `- [true* . rgl . !stop_rgl* . rd_ebs(true) . !stop_rgl] false`
16.	There can be no *regulate_cc*(*s90*) or *regulate_cc*(*s100*).	`[true* . (regulate_cc(s90) || regulate_cc(s100))] false`
19.	After a *rd_env* with 2 or more active requests or a *rd_env* with no active requests, the same state is reached.	`‘The same state’ seems to be impossible to express.`

Table 5: Verification of the example Cruise Control requirements

3.3.1 Graphical Verification

In order to gain a better understanding on what a requirement expresses and why it is (or is not) true for a given model, it can be very useful to generate pictures of Labeled Transition Systems of the model. For some of the requirements found in Table 5, we visualized the model such that only the relevant transitions are visible.

In Figure 2, we see an LTS generated from the mCRL2 model, showing only the transitions that are relevant for requirement 3: *rd_enable*, *on* and *off*. The LTS has been reduced using branching bisimulation equivalence to obtain a manageable system. Note that we can now see clearly that the model meets requirement 3 from Table 5: when the system is off (there has been no *on*), a *rd_enable*(*true*) can only be followed by an *on* transition.

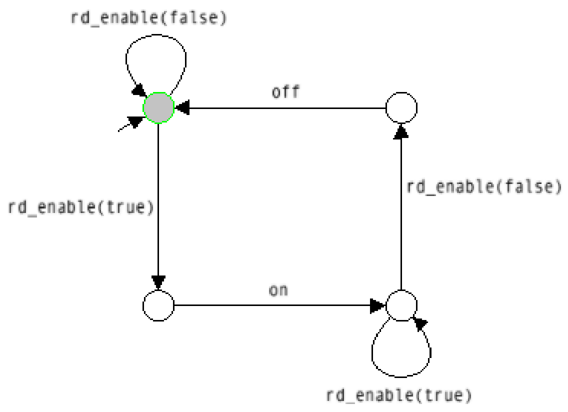


Figure 2: LTS of the model, showing only *rd_enable*, *on* and *off*.

Figure 3 shows an LTS containing the transitions that are relevant for requirement 11: *rd_ebs*, *rgl*, *stop_rgl* and *activate*, again reduced using branching bisimulation equivalence. We can now see clearly that the model meets requirement 11 from Table 5. For instance, there is only one *activate* transition in the LTS, making it easy to see that there can be no *rd_ebs*(*true*) directly before *activate*. Also, when the system is controlling (there has been an *rgl* and no following *stop_rgl*), a *rd_ebs*(*true*) can only be followed by a *stop_rgl* transition.

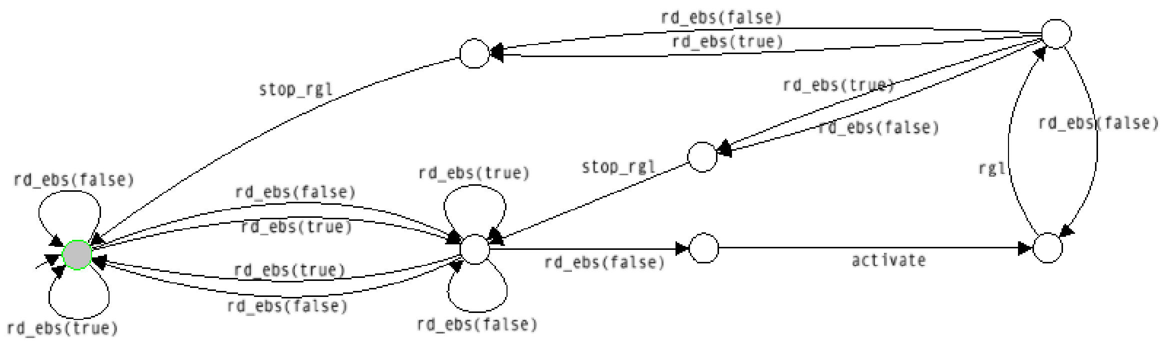


Figure 3: LTS of the model, showing only *rd_ebs*, *rgl*, *stop_rgl* and *activate*.

3.3.2 Verification Results

It seems that requirement 19 is impossible to express in modal μ-calculus. This may be the first example of a requirement that looks straightforward in natural language, but cannot be expressed in a modal formula.

For all of the other requirements, the verification resulted in *true* for the constructed mCRL2 model. Therefore, we can conclude that a Cruise Control model that provably meets these given requirements has been designed in mCRL2 successfully.

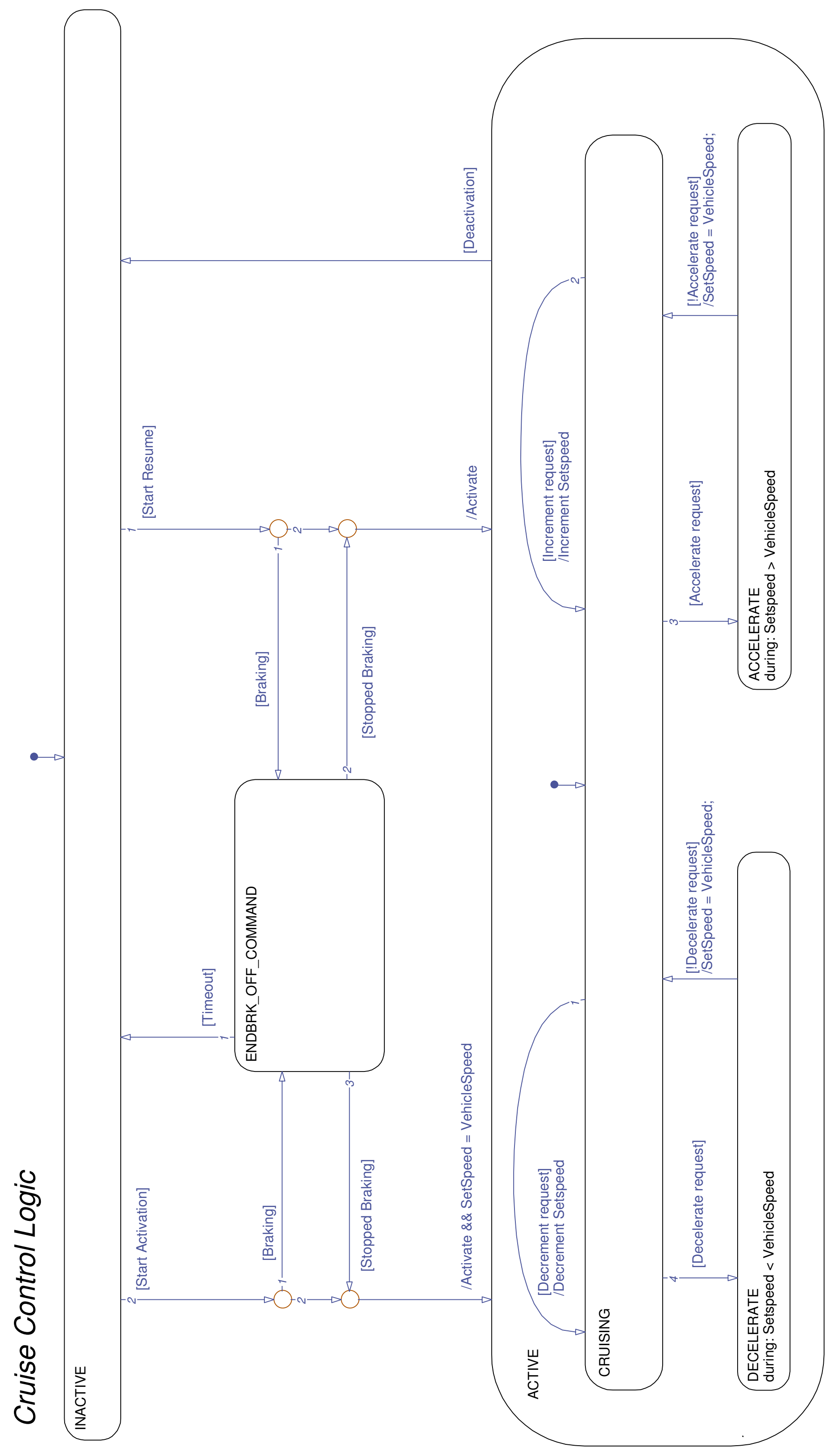
4 Translation Cruise Control StateFlow model to mCRL2

This section describes the results of a case study, in which a StateFlow model of a Cruise Control system has been translated to the modeling language mCRL2. With this case study being the second part of the project, we intend to perform formal verification on the Cruise Control system as designed at DAF.

Currently, after modelling the CC behaviour in Matlab / Simulink, several test scenarios are executed to verify that the system behaves as expected. However, this approach treats only a small fraction of all possible scenarios, and certainly does not lead to a proof that requirements are met. With this case study, we do not aim to prove that the system's behaviour is 100% correct, because this would require a complete set of requirements, which is practically impossible to obtain. However, we *can prove* whether the system satisfies several important safety requirememts in all possible scenarios.

4.1 StateFlow Model

The Cruise Control StateFlow model can be found in Figure appendix B.1. From this model, we can see the main behaviour of the system. A simplified view on the model, showing only the global behaviour, can be found in Figure 4. In this view on the model, it is not visible that the entire function can be enabled or disabled, which is a standard functionality in Simulink. Thus, if the system is disabled, it is not in any of the states visible in Figure 4. When it becomes enabled, the default transition is taken to end up in the Inactive state. After this, the system behaves according to the model, until it becomes disabled again.



Cruise Control Logic
INACTIVE
[Start Activation]
[Start Resume]
[Braking]
[Stopped Braking]
[Timeout]
ENDBRK_OFF_COMMAND
/Activate && SetSpeed = VehicleSpeed
/Activate
[Deactivation]
ACTIVE
CRUISING
[Decrement request] /Decrement Setspeed
[Increment request] /Increment Setspeed
[Decelerate request]
[Accelerate request]
[!Decelerate request] /SetSpeed = VehicleSpeed;
[!Accelerate request] /SetSpeed = VehicleSpeed;
DECELERATE during: Setspeed < VehicleSpeed
ACCELERATE during: Setspeed > VehicleSpeed

Figure 4: A simplified version of the Cruise Control StateFlow model.

The StateFlow model uses preprocessed input signals to determine the activation and deactivation conditions. These signals are not available as such for the Cruise Control system, but they are calculated using the “External Activation / Deactivation Conditions” Simulink block in appendix B.1. Since both signals are dependent on the same set of input signals, but not each others negation, this complexity is also taken into account for the translation to an mCRL2 model (see section 4.3).

4.2 Functional Requirements

The translation of the StateFlow model to mCRL2 results in another mCRL2 Cruise Control model, which should satisfy the same requirements as the model constructed in section 3. However, the context of the two models is not identical, so the original list of requirements cannot be used unaltered. Based on the input and output signals defined in section 3.1, but adapted to the StateFlow model’s inputs and outputs, Table 6 contains the resulting list of signals used to express the functional requirements.

Inputs	**Outputs**
b_Enabled	Activate Governor Request
DNR Switch Position	Set Speed
Other Vehicle Function Active	
Vehicle Speed	
Brake Stalk Active	
Resume Request	
Off Request	
Driver Request	

Table 6: The list of inputs and outputs

The requirements listed in appendix B.2 describe the desired functionality of the CC Supervisor in terms of these inputs and outputs, based on the requirements defined in section 3.1. Table 7 below contains a selection of requirements that are used as an example throughout this section.

3.	De functie moet enabled zijn als de hierna genoemde inputsignalen de genoemde waarden hebben: *b_Enabled* is *true*, DNR Switch Position is D.
11.	Als de waarde van inputsignaal Other Vehicle Function Active gelijk is aan *true*, moet de waarde van outputsignaal Activate Governor Request *false* zijn.
16.	De waarde van outputsignaal Set Speed mag nooit hoger zijn dan 85 km/h.

Table 7: A selection of Cruise Control requirements

4.3 Translation to mCRL2

In this section, it is explained how the Cruise Control system was translated from a StateFlow model to mCRL2. More information on the mCRL2 modelling language and toolset can be found in [3] and [1]. For this case study, the July 2011 (SVN revision 9551) Release version of the mCRL2 toolset has been used.

The global description of the constructed model below gives an abstracted view on the shape of the Cruise Control process. The complete mCRL2 model, containing all used sort declarations and function mappings, can be found in appendix B.3. The basic structure of the modeled process is as follows:

```
CC_Disabled =
    sum environment: ENV . read_env(environment) . (
        (enable conditions) -> enable . CC_Init <>
        no_action . CC_Disabled);

CC_Init = CC_Inactive(s0);

CC_Inactive(ccss: SPD) =
    sum environment: ENV . read_env(environment) . (
        (disable conditions) -> disable . CC_Disabled <>
        (activation conditions && Resume Request) -> CC_Res1(ccss, brake stalk) <>
        (activation conditions && Other Request)  -> CC_Set1(ccss, brake stalk, current speed) <>
        no_action . CC_Inactive(ccss));

CC_Res1(ccss: SPD, brake stalk: Bool) = (
        (brake stalk) -> CC_BrkOff(true, 0, ccss) <>
        CC_Res2(ccss));

CC_Set1(ccss: SPD, brake stalk: Bool, current speed:SPD) = (
        (brake stalk) -> CC_BrkOff(false, 0, ccss) <>
        CC_Set2(current speed));

CC_BrkOff(activated_by_resume: Bool, timer: Nat, ccss: SPD) =
    sum environment: ENV . read_env(environment) . (
        (disable conditions) -> disable . CC_Disabled <>
        (!(brake stalk) && activated_by_resume) -> CC_Res2(ccss) <>
        (!(brake stalk) && !activated_by_resume) -> CC_Set2(current speed) <>
        (timer > max waiting time) -> CC_Inactive(ccss) <>
        no_action . CC_BrkOff(activated_by_resume, (timer + 1), ccss));

CC_Res2(ccss: SPD) =
    activate . regulate_cc(ccss) . CC_Activated(ccss);

CC_Set2(current speed: SPD) =
    activate . regulate_cc(current speed) . CC_Activated(current speed);

CC_Activated(ccss: SPD) =
    sum environment: ENV . read_env(environment) . (
        (disable conditions) -> disable . CC_Disabled <>
        (deactivation conditions) -> stop_regulate . deactivate . CC_Inactive(ccss) <>
        (Driver Request == Decrement) -> regulate_cc(calculate_new_ccss) . CC_Activated(calculate_new_ccss) <>
        (Driver Request == Increment) -> regulate_cc(calculate_new_ccss) . CC_Activated(calculate_new_ccss) <>
        (Driver Request == Accelerate) -> CC_Accelerate(ccss) <>
        (Driver Request == Decelerate) -> CC_Decelerate(ccss) <>
        no_action . CC_Activated(ccss));

CC_Accelerate(ccss: SPD) =
     sum environment: ENV . read_env(environment) . (
        (disable conditions) -> disable . CC_Disabled <>
        (deactivation conditions) -> stop_regulate . deactivate . CC_Inactive(ccss) <>
        (Request != Accelerate) -> regulate_cc(current_speed) . CC_Activated(current_speed) <>
52.         regulate_cc(calculate_new_ccss) . CC_Accelerate(calculate_new_ccss));
53.
54. CC_Decelerate(ccss: SPD) =
55.     sum environment: ENV . read_env(environment) . (
56.         (disable conditions) -> disable . CC_Disabled <>
57.         (deactivation conditions) -> stop_regulate . deactivate . CC_Inactive(ccss) <>
58.         (Request != Decelerate) -> regulate_cc(current_speed) . CC_Activated(current_speed) <>
59.         regulate_cc(calculate_new_ccss) . CC_Decelerate(calculate_new_ccss));
60.
61. init
62.     CC_Disabled;
```

Note that for each state in the StateFlow model (Figure 4), there is a process in the mCRL2 model. The 'Active' superstate in the StateFlow model is not explicitly translated, but it is represented by its substates, as the system can be in only one of these exclusive substates at a given time.

In line 62, we see that initially the Cruise Control is disabled. During execution, the system constantly reads the environment status (lines 2, 9, 24, 38, 48, 55). 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 above model to an input *environment* of type *ENV* for reading purposes.

After pulling the environment status, the system determines which transition should be taken. The expressions used in the conditions are in terms of the inputs used in *read_env*, but abstracted to for instance *disable conditions* and *activation conditions* in the description above. In line 28, we see that the system returns to the Inactive state after some maximum waiting time. In the used mCRL2 model, the constant *max waiting time* has been set to the value 2.

The Cruise Control system has two outputs, Activate Governor Request and Set Speed. Initially, their respective values are *false* and *s0*. When a *regulate_cc*(*ccss*) transition is taken, this means that *b_Activate_Governor* is set to *true* and *Set_Speed* becomes *ccss*. When a *stop_regulate* transition is taken, *b_Activate_Governor* is set to *false* and *Set_Speed* to *s0*.

From the complete model (which can be found in appendix B.3), we constructed a reduced bisimulation equivalent Labelled Transition System using the following commands available in the mCRL2 toolset: *mcrl22lps*, *lpsactionrename*, *lps2lts*, *ltsconvert -ebisim*, resulting in the LTS displayed in Figure 5. Note that the rename file used for *lpsactionrename* was defined as follows:

no_action	$\rightarrow$	*tau*
regulate_cc(*ccss*)	$\rightarrow$	*rgl*
stop_regulate	$\rightarrow$	*stop_rgl*
read_env(*environment*)	$\rightarrow$	*rd_env*

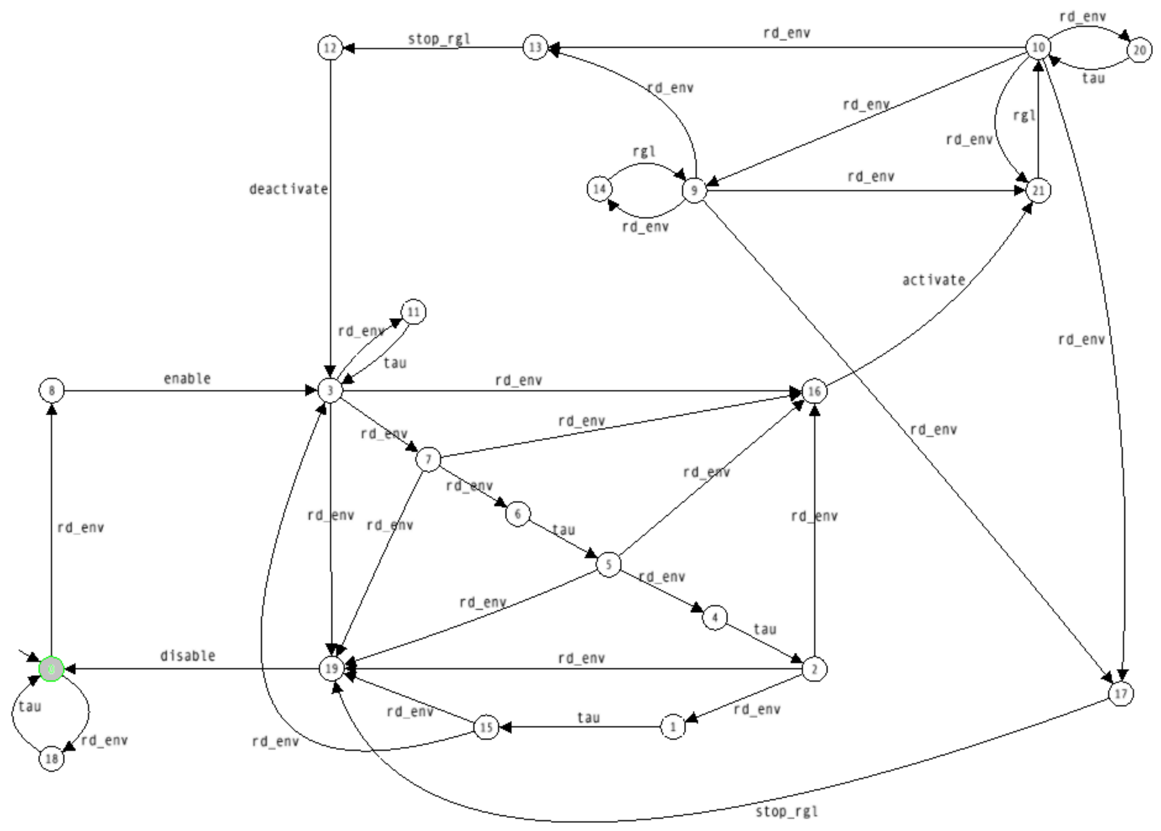


Figure 5: An LTS showing the behaviour of the model.

From Figure 5, it is hard to comprehend the entire behaviour of the Cruise Control system. However, it is possible to recognize several processes from the mCRL2 process description above. In Table 8, these states and corresponding processes are listed.

State	mCRL2 Process
0	CC_Disabled
3	CC_Inactive
7, 6, 5, 4, 2, 1, 15	CC_BrkOff (*max waiting time* = 2)
16	CC_Res2, CC_Set2
10	CC_Activated
9	CC_Accelerate, CC_Decelerate

Table 8: The list of LTS states and corresponding mCRL2 processes.

4.4 Model Checking

This section contains the results of the formal verification performed on the requirements in section 4.2. With this formal verification, it can be proved that a certain path of transitions can (or cannot) occur in a given mCRL2 model.

In appendix B.5, a translation to English and to the context of the designed mCRL2 model (see section 4.3) is given for the requirements that are listed in appendix B.2. These requirements are expressed as propositions in modal μ-calculus.

Note that some of the propositions use transition names that are not used in the process description of the mCRL2 model (but they are declared as actions). In these cases, the corresponding transitions in the model are renamed with *lpsactionrename* using the rename definitions as included in appendix B.4. In most cases, this comes down to removing all parameters of transitions (usually *read_env*) that are not used in the requirement.

Table 9 below contains the example requirements expressed in modal μ-calculus.

3.	When disabled, a *rd_enable*(*true*) must be followed directly by an *enable*.	`[!enable* . rd_enable(true) . !enable] false &&`
		`[true* . disable . !enable* . rd_enable(true) . !enable] false`

11. **X**	- As long as *ActivateGovernor* = *false*, there can be no *activate* and *rgl* directly after *rd_oth*(*true*).
	- When *ActivateGovernor* = *true*, a *rd_oth*(*true*) must be directly followed by a *stop_rgl*.
		`- [!rgl* . rd_oth(true) . activate . rgl] false`
		
		`- [true* . rgl . !stop_rgl* . rd_oth(true) . !stop_rgl] false`

16. **X**	There can be no *regulate_cc*(*s90*) or *regulate_cc*(*s100*).	`[true* . (regulate_cc(s90) || regulate_cc(s100))] false`

Table 9: Verification of the example Cruise Control requirements.

4.4.1 Verification Results

Some of the requirements in appendix B.5 are marked with an **X**, indicating that the verification resulted in *false* for the constructed mCRL2 model, so these requirements are *not* met by the model. For the other requirements, the verification resulted in *true*. We have constructed a counterexample for the unsatisfied requirements, e.g. a trace of transitions possible in the model which contradicts the requirement. The unsatisfied requirements and corresponding counterexamples are listed and discussed in appendix B.6. The unsatisfied requirements from our example selection are listed below.

11. Counterexample: 1.*rd_oth*(*false*), 2.*enable*, 3.*rd_oth*(*false*), 4.*rd_oth*(*true*), 5.*activate*, 6.*rgl*

    The system checks whether another vehicle function is active in CC_Inactive, but if it is in the CC_BrkOff state for some time after that before actual activation, it is possible that one of the other vehicle functions has already become active.

16. Counterexample: 1.*rd_env*, 2.*enable*, 3.*rd_env*, 4.*activate*, 5.*regulate_cc*(*s80*), 6.*rd_env*, 7.*regulate_cc*(*s80*), 8.*rd_env*, 9.*regulate_cc*(*s100*)

    The first five transitions describe enabling and activation of the CC. Now assume that the current vehicle speed is 100 km/h (or "s100"). If the Driver Request is "Accelerate" (in transition 6), the system goes to the CC_Accelerate state. Then, if the Driver Request is quickly changed back to "No_Rq" in transition 8, the Set Speed is set to the current vehicle speed (which can still be "s100") and used in transition 9.

4.4.2 Graphical Verification

In the same way we performed a graphical verification on requirement 3 in section 3.3.1, we generated an LTS from the mCRL2 model in Figure 6. We can now see clearly that this model also meets requirements 3 (from Table 9): when the system is disabled (there has been no *enable*), a *rd_enable*(*true*) can only be followed by an *enable* transition.

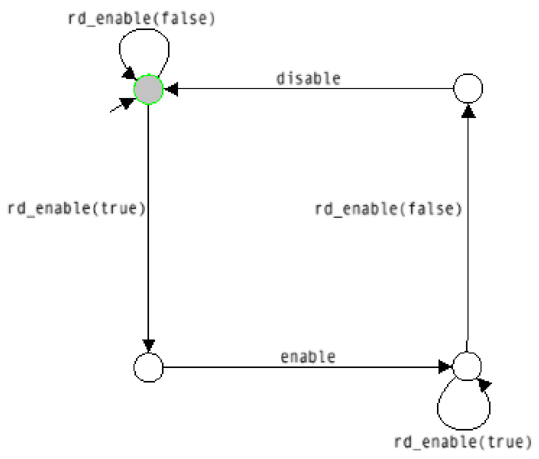


Figure 6: LTS of the model, showing only *rd_enable*, *enable* and *disable*.

4.4.3 Matlab / Simulink Validation

For most of the unsatisfied requirements, the constructed counterexamples give us a clear view on why the requirement is not met for both the mCRL2 model and the StateFlow model. However, in case of requirements 21-24, it is not trivial to see whether the behaviour in the counterexamples is also possible in the original StateFlow model.

More specifically, the following scenario is possible in the mCRL2 model. Let's assume the Driver Request is "Decelerate", and the Cruise Control is in the CC_Decelerate state. Then, the Driver Request becomes "Increment", causing the Cruise Control to go to the CC_Activated state. If the Driver Request is then immediately set to "No_Rq", the Cruise Control will not increment the Set Speed anymore, even though the Driver Request has been "Increment" (which is the reason that requirement 23 is not satisfied).

In order to determine whether this behaviour is also possible in the StateFlow model, we set up a simulation in Matlab. Most of the inputs of the Cruise Control system are set to a fixed value (in general a value that allows the Cruise Control to enable/activate). The only input that is variable, is the Driver Request.

First, we simulate a scenario to show the normal behaviour for an "Increment" Driver Request. The results of this simulation are displayed in part A of Figure 7. For the first second of the simulation, the Driver Request is set to "No_Rq". Then, during the next second, it is set to "Decelerate". This causes the Cruise Control to activate and the Set Speed to become 50 km/h (which is the fixed value for the actual vehicle speed) shortly. As the Driver Request is "Decelerate" for a while, the Cruise Control Set Speed is then set to 47 km/h (the actual vehicle speed minus the Decelerate offset). When the Driver Request is set back to "No_Rq", the Cruise Control reaches the Cruising state and the Set Speed becomes 50 km/h again. Then, at $2,5$ seconds, the Driver Request is set to "Increment" and immediately back to "No_Rq". From the slight ascent in Figure 7, we can see that the Set Speed is now increased to 50,5 km/h.

Now, we want to find out whether it is possible that the system ignores the "Increment" signal, if we do not give it enough time to return to the Cruising state. In part B of Figure 7, we see that the Driver Request is set from "Decelerate" to "Increment" at $2,0$ seconds, and then immediately to "No_Rq". Note that the Set Speed is now set to 50 km/h, but never incremented to 50,5 km/h. This shows us that the behaviour of the mCRL2 model can also occur in the StateFlow model.

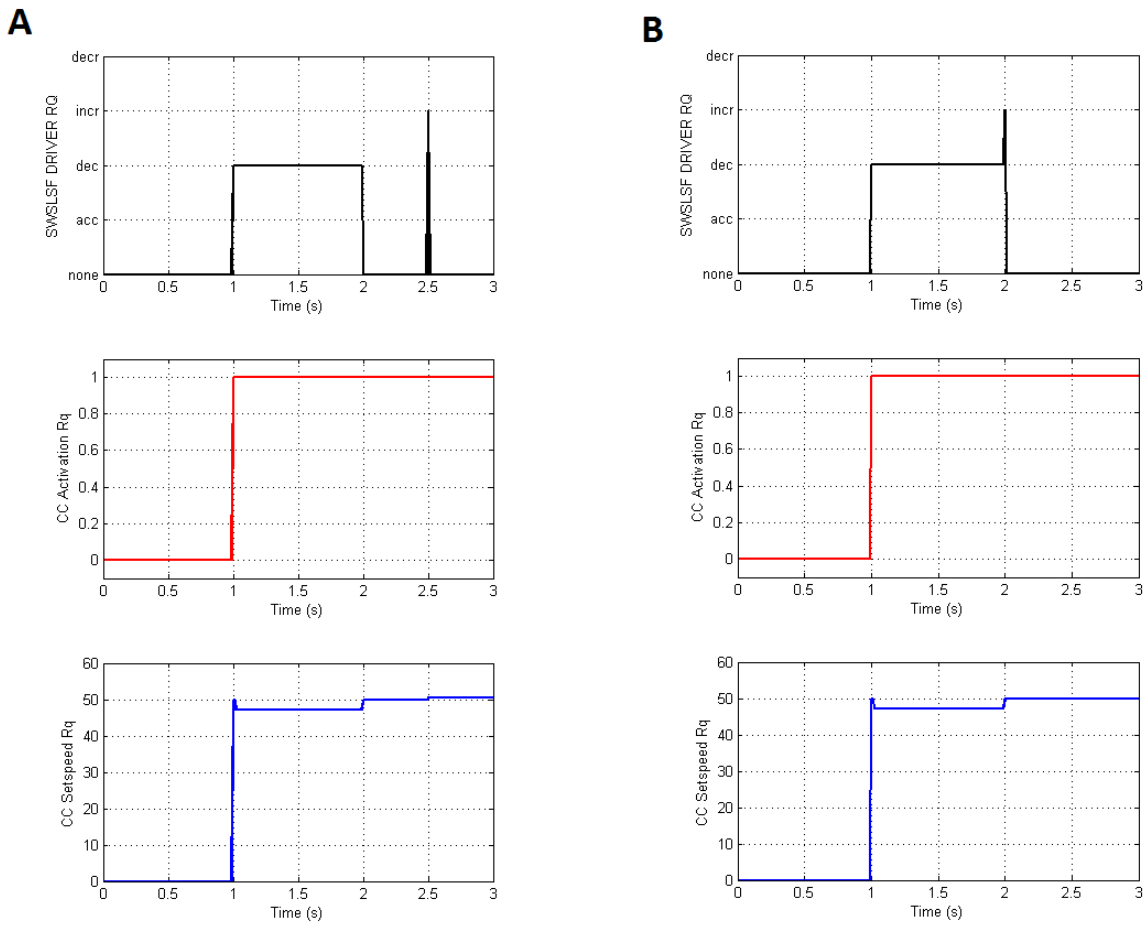


Figure 7: The results of the Matlab simulations.

Concluding, we have validated that this aspect of the behaviour of the mCRL2 model is equivalent to that of the StateFlow model: when the inputs of the system are such that a transition should be taken, the system takes only one transition and then evaluates the values of the inputs again, unlike taking transitions until the system is in a 'steady' state before evaluating input values again.

More specifically, we have validated that requirements 21-24 are indeed not satisfied by the StateFlow model.

4.5 Solution Proposal

It was shown in section 4.4 that the original DAF StateFlow model does not satisfy all requirements. In this section, we propose some adaptations to the original model, such that it meets all requirements.

4.5.1 Adaptations

In order to obtain a StateFlow model that meets all the requirements, we made some adaptations to the original model. The StateFlow model that is the resulting solution, along with an explanation of the changes, can be found in appendix B.7.

As an example, Figure 8 shows the proposed solution that makes sure that the model satisfies requirement 11. There are additional guards on the outgoing transitions from the ENDBRK_OFF_COMMAND state to ensure that the upcoming activation can only occur if the activation conditions remain true.

4.5.2 Translation of solution to mCRL2

In order to show that the StateFlow model that we propose as a solution, is indeed a solution, we performed the verification on the adapted model. This means that it is also translated to an mCRL2 model, which can be found in appendix B.8. The model is, naturally, very similar to that presented in section 4.3.

4.5.3 Verification Results

The mCRL2 model in appendix B.8 has been tested to the same set of requirements (appendix B.5). The verification indicated that the model satisfies all requirements, which shows us that the solution proposed is a correct solution.

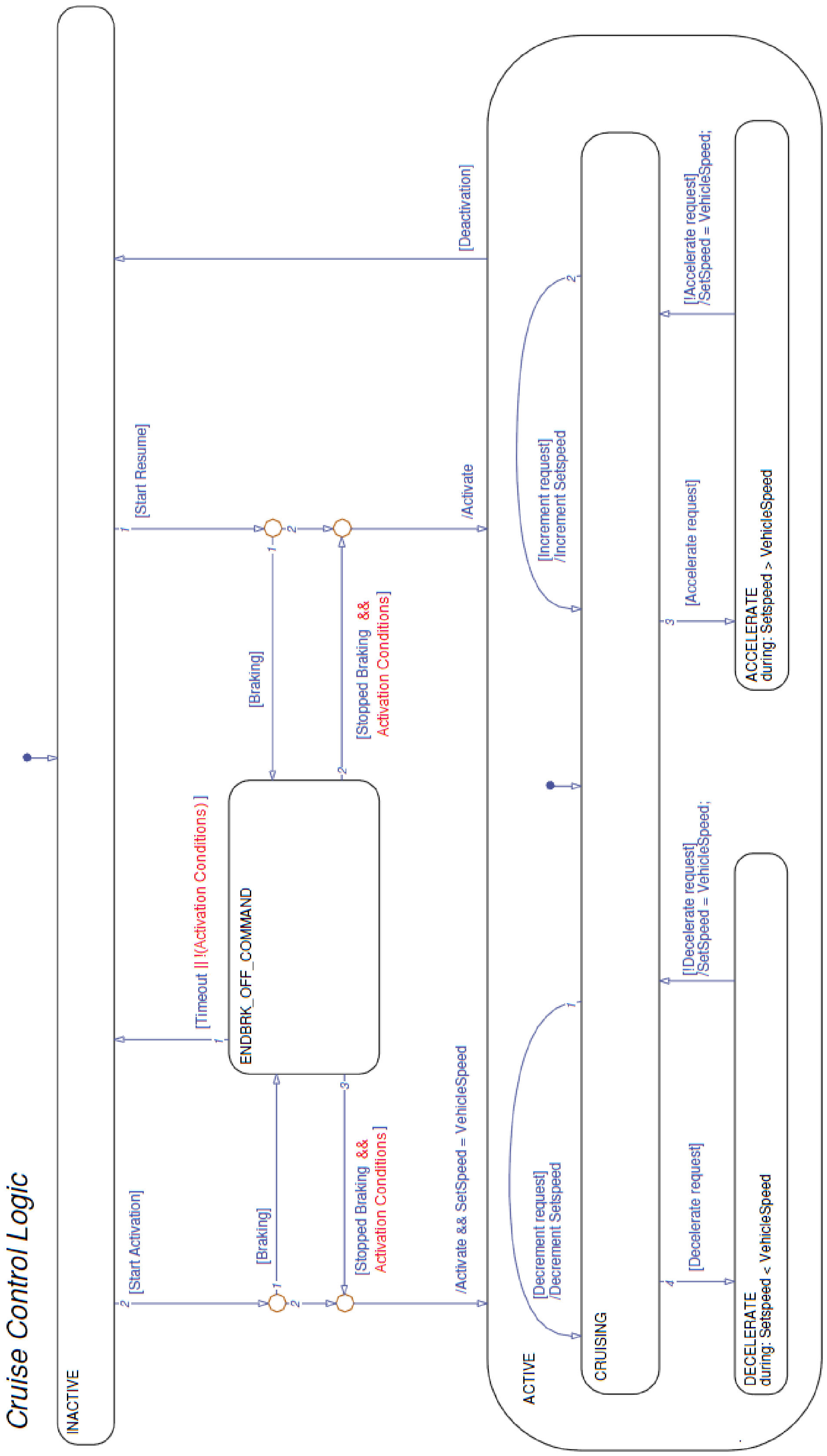

Cruise Control Logic
INACTIVE
ACTIVE
CRUISING
ACCELERATE
during: Setspeed > VehicleSpeed
DECELERATE
during: Setspeed < VehicleSpeed
ENDBRK_OFF_COMMAND
[Deactivation]
[Start Resume]
/Activate
[Braking]
[Stopped Braking && Activation Conditions]
[Timeout || !(Activation Conditions)]
[Start Activation]
/Activate && SetSpeed = VehicleSpeed
[Increment request] /Increment Setspeed
[Decrement request] /Decrement Setspeed
[!Accelerate request] /SetSpeed = VehicleSpeed;
[Accelerate request]
[!Decelerate request] /SetSpeed = VehicleSpeed;
[Decelerate request]

Figure 8: A simplified version of proposed solution to the Cruise Control StateFlow model.

5 Translation Vehicle Function Architecture to mCRL2

This section describes the results of a case study, in which a Matlab / Simulink model of a Vehicle Function Architecture has been translated to the modeling language mCRL2. With this case study being the third and final part of the project, we intend to perform formal verification on the Vehicle Function Architecture (VFA) that manages "Driving and Braking" as designed at DAF. The VFA is a system that contains several parallel components, one of which is the Cruise Control system that was also investigated during this project (see section 4).

5.1 Matlab / Simulink Model

The global view on the VFA Simulink model and a more detailed view on a selection of important components of the system can be found in appendix C.2. A simplified view on the VFA is provided in Figure 9 below, showing the division of the VFA into components.

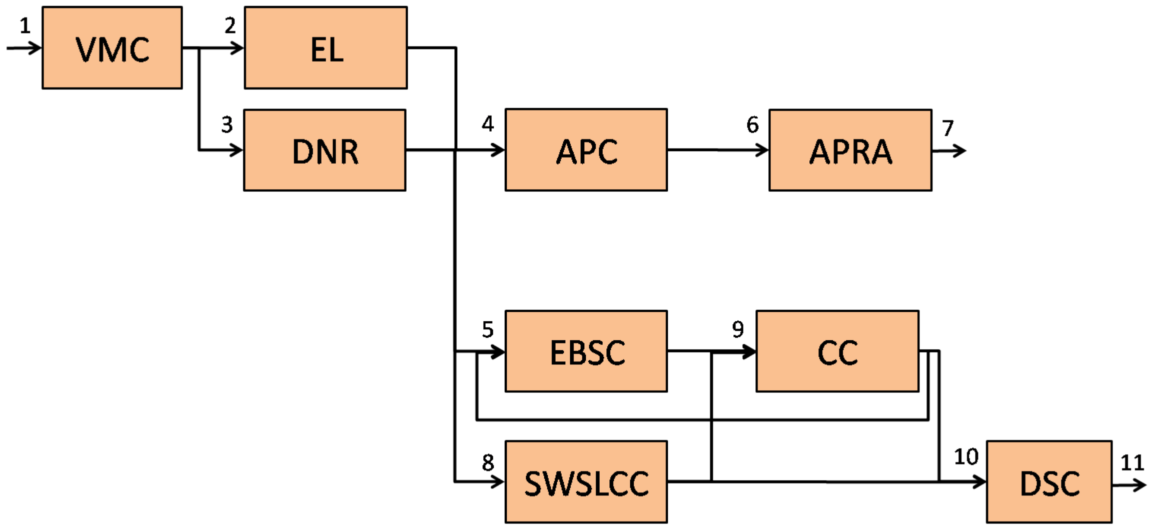


Figure 9: A simplified version of the VFA Simulink model.

The VFA model contains 9 Simulink blocks, that are connected to each other as displayed in Figure 9. Most of these blocks also have VFA inputs and/or outputs, but these are omitted from the above figure, because it is only intended to show the interaction between the several components. The arrows are labelled with numbers to denote the order in which the communication actions are executed in the mCRL2 translation presented in section 5.3.

Some of the Simulink blocks have trivial behaviour, such as the Vehicle Mode Control (VMC), which simply converts its input signal to an output signal that has a slightly different type. However, the behaviour of other blocks is modeled using multiple layers of hierarchy and StateFlow diagrams, such as the Accelerator Pedal Control (APC) and Downhill Speed Control (DSC).

5.2 Functional Requirements

In order to define a list of requirements for formal verification, we consider the VFA to be one complete system having a set of inputs and output signals. The requirements will be only in terms of these inputs and outputs, without considering the internal structure of the VFA system. These inputs and output signal are listed in appendix C.1. Table 10 below contains a selection of these signals that are used as an example throughout this section.

Inputs	**Outputs**
VehModeSw_Pos	*AccelPedReqArb_SetSpeed*
DNRSwitch_Pos	*DSC_SetSpeed*
	CC_DrCtrlLogic_SetSpeed

Table 10: The inputs and outputs of the VFA used in this section.

The requirements listed in appendix C.3 describe the desired functionality of the VFA in terms of its inputs and outputs. This set of requirements is by no means a complete list that fully defines the expected behaviour, but it merely states a number of safety requirements that are most important to satisfy. Table 11 below contains a selection of these requirements, using only the input and output signals from Table 10.

1.	Als de *VehModeSw_Pos* NIET gelijk is aan *VEHMOD_DRIVE*, dan moet de waarde van *AccelPedReqArb_SetSpeed* gelijk zijn aan 0 km/h.
9.	Outputsignaal *DSC_SetSpeed* moet altijd minimaal *DSC_CC_OFFSET* hoger zijn dan *CC_DrCtrlLogic_SetSpeed*.
11.	Wanneer inputsignaal *DNRSwitch_Pos* ongelijk is aan *SWITCH_D*, dan moet de waarde van outputsignaal *DSC_SetSpeed* altijd gelijk zijn aan 255 km/h.

Table 11: A selection of VFA requirements

5.3 Translation to mCRL2

In this section, it is explained how the Vehicle Function Architecture was translated from a Matlab / Simulink model to mCRL2. More information on the mCRL2 modelling language and toolset can be found in [3] and [1]. For this case study, the July 2011 (SVN revision 9551) Release version of the mCRL2 toolset has been used.

The global description of the constructed model below gives an abstracted view on the shape of the VFA process. The complete mCRL2 model, containing all used sort declarations and function mappings, can be found in appendix C.4. The basic structure of the modeled process is as follows:

```
% Vehicle Mode Control
VMC = comm_1 . VMC_Logic . comm_2 . comm_3 . VMC;

% Enabling Logic
EL = comm_2 . EL_Logic . comm_4 . comm_5 . comm_6 . comm_8 . comm_9 . comm_10 . EL;

% DNR Switch Logic
DNR = comm_3 . DNR_Logic . comm_4 . comm_9 . comm_10 . DNR;

% Accelerator Pedal Control
APC = comm_4 . APC_Logic . comm_6 . APC;

% Accelerator Pedal Request Arbiter
APRA = comm_6 . APRA_Logic . comm_7 . APRA;

% Endurance Brake Stalk Control
EBSC = comm_5 . EBSC_Logic . comm_6 . comm_9 . EBSC;

% Cruise Control Logic
CC = comm_5 . comm_9 . CC_Logic . comm_10 . CC;

% Steering Wheel Switch Logic Cruise Control
SWSLCC = comm_8 . SWSLCC_Logic . comm_9 . comm_10 . SWSLCC;

% Downhill Speed Control Logic
DSC = comm_10 . DSC_Logic . comm_11 . DSC;

% Synchronization
SYNC = comm_1 . comm_2 . comm_3 . comm_4 . comm_5 . comm_6 . comm_7 . comm_8 . comm_9 . comm_10 . comm_11 . SYNC;

init
    allow(
        {cmnc_1, cmnc_2, cmnc_3, cmnc_4, cmnc_5, cmnc_6, cmnc_7, cmnc_8, cmnc_9, cmnc_10, cmnc_11},

    comm({  comm_1 | comm_1 -> cmnc_1,
            comm_2 | comm_2 | comm_2 -> cmnc_2,
            comm_3 | comm_3 | comm_3 -> cmnc_3,
            comm_4 | comm_4 | comm_4 | comm_4 -> cmnc_4,
            comm_5 | comm_5 | comm_5 | comm_5 -> cmnc_5,
            comm_6 | comm_6 | comm_6 | comm_6 | comm_6 -> cmnc_6,
            comm_7 | comm_7 -> cmnc_7,
            comm_8 | comm_8 | comm_8 -> cmnc_8,
            comm_9 | comm_9 | comm_9 | comm_9 | comm_9 | comm_9 -> cmnc_9,
            comm_10 | comm_10 | comm_10 | comm_10 | comm_10 | comm_10 -> cmnc_10,
            comm_11 | comm_11 -> cmnc_11},

            VMC || EL || DNR || APC || APRA || EBSC || CC || SWSLCC || DSC || SYNC
    ));
```

Note that in the mCRL2 model (line 47), there are 10 processes initialized to run in parallel. These consist of 9 processes that correspond to the 9 Simulink blocks in the original VFA model, and 1 process that handles the synchronization of the communications. In the above simplified model, we abstracted from the fact that the actions (*comm_1* to *comm_11*) performed by the processes contain a number of data variables. These variables make sure that, if a component uses an input signal, it uses the same value as the corresponding output signal of the component that outputs this signal.

The processes are in general of the following form:

1. First, they perform an action to read their input signals;
2. then, some calculations are done to determine the values for their output signals;
3. and finally, they perform one or multiple actions to communicate the output signals to other components that use these signals as their inputs.

An exception to this general form is the Cruise Control process, which already has to send some output signals (in action *comm_5*) before it had the chance to read its input signals (action *comm_9*). This is necessary because there is a loop between the EBSC and CC blocks (see Figure 9). For the first occurrence of *comm_5*, the default values for these output signals are used. After that, the output signals of *comm_5* are dependent on the input signals received with *comm_9* and have the same values as those in *comm_10*.

The Synchronization (SYNC) process makes sure that all the communication actions are performed in the correct order, without having any influence on the data that is communicated. Without the Synchronization process, it would be possible that some actions are performed too soon or too late. For instance, if the VMC has read its inputs in a *comm_1* action, and sent its outputs in the *comm_2* and *comm_3* actions, it could already perform the next *comm_1* action without waiting for the other processes to finish. In order to avoid this kind of confusing behaviour and keep the state space as small as possible, the Synchronization process is used to enforce the same trace of actions in every cycle. The result of generating an LTS from the mCRL2 model (after renaming all actions to remove the data values and performing a branching bisimulation equivalence reduction) is displayed in Figure 10:

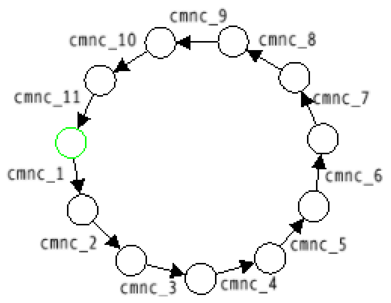


Figure 10: An LTS showing the behaviour of the VFA model.

5.3.1 Alternative Translation Approach

During the case study, we also attempted to take an alternative approach to translating the Simulink model to mCRL2. In the model described above, every component only communicates with the components it shares signals with in the Simulink model. The alternative translation approach however, combines all these transitions into one single communication action *comm_all* that contains all data variables. In this way, all components simultaneously read their input signals and write their output signals. Based on the values of the inputs signals, the next values for the output signals are determined. While the mCRL2 model in section 5.3 uses 11 transitions to complete a simulation step, the alternative model performs a simulation step for every component with every *comm_all* transition (see Figure 11).

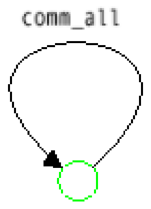


Figure 11: An LTS showing the behaviour of the alternative VFA model.

Using this translation approach, the construction of the mCRL2 model requires a smaller effort and results in a small and readable model. We realize that the resulting model contains a relatively small number states and a large number of transitions. It is hard to foresee how this influences the performance of the tools in the mCRL2 toolset. Unfortunately, it turned out to be impossible to perform a state space exploration or verify a property within reasonable time. This indicates that it is very important to choose an appropriate structure when modelling a complex system and further research on this topic could help us comprehend the effects of our choices.

5.4 Model Checking

This section contains the results of the formal verification performed on the requirements in section 5.2. With this formal verification, it can be proved that a certain path of transitions can (or cannot) occur in a given mCRL2 model.

In appendix C.6, a translation to English and to the context of the designed mCRL2 model (see section 5.3) is given for the requirements that are listed in appendix C.3. These requirements are expressed as propositions in modal μ-calculus.

Note that some of the propositions use transition names that are not used in the process description of the mCRL2 model (but they are declared as actions). In these cases, the corresponding transitions in the model are renamed with *lpsactionrename* using the rename definitions as included in appendix C.5. In most cases, this comes down to removing all parameters of transitions (usually *read_env*) that are not used in the requirement.

Table 12 below contains the example requirements expressed in modal μ-calculus.

1.	The first *cmnc_7* after *cmnc_1_A* or *cmnc_1_St* must be *cmnc_7_setspeed_none*.	`[true* . (cmnc_1_A || cmnc_1_St) . (!cmnc_7_setspeed_none)* . cmnc_7_setspeed_other] false`
9. **X**	For each *cc_spd* : *SPD* and *dsc_spd* : *SPD* such that *cc_spd* has a higher value than *dsc_spd*, a *cmnc_10_setspeed*(*cc_spd*) may never be directly followed by a *cmnc_11_setspeed*(*dsc_spd*).	`[true* . cmnc_10_setspeed(sLow) . cmnc_11_setspeed(sNeg)] false && [true* . cmnc_10_setspeed(sLow) . cmnc_11_setspeed(sNone)] false && [true* . cmnc_10_setspeed(sMid) . cmnc_11_setspeed(sNeg)] false && [true* . cmnc_10_setspeed(sMid) . cmnc_11_setspeed(sNone)] false && [true* . cmnc_10_setspeed(sMid) . cmnc_11_setspeed(sLow)] false && [true* . cmnc_10_setspeed(sHigh) . cmnc_11_setspeed(sNeg)] false && [true* . cmnc_10_setspeed(sHigh) . cmnc_11_setspeed(sNone)] false && [true* . cmnc_10_setspeed(sHigh) . cmnc_11_setspeed(sLow)] false && [true* . cmnc_10_setspeed(sHigh) . cmnc_11_setspeed(sMid)] false`
11. **X**	The first *cmnc_11* after *cmnc_3_dnr_other* must be *cmnc_11_setspeed_high*.	`[true* . cmnc_3_dnr_other . (!cmnc_11_setspeed_high)* . cmnc_11_setspeed_other] false`

Table 12: Verification of the example VFA requirements

5.4.1 Verification Results

Some of the requirements in appendix C.6 are marked with an **X**, indicating that the verification resulted in *false* for the constructed mCRL2 model, so these requirements are *not* met by the model. For the other requirements, the verification resulted in *true*. The unsatisfied requirements are listed and discussed in appendix C.7. A short discussion for the unsatisfied requirements from our example selection is listed below.

9. This requirement is violated because on activation of the Downhill Speed Control, the *DSC_SetSpeed* is initialized to the actual vehicle speed, independent of whether the Cruise Control was also active. During this simulation step, the *DSC_SetSpeed* can be lower than the *CC_DrCtrlLogic_SetSpeed*, which already violates the requirement, even though the *DSC_SetSpeed* will be adjusted to a value higher than *CC_DrCtrlLogic_SetSpeed* during the next simulation step.

11. This requirement is not satisfied because the DNR Switch Logic can output that the *Allowed_DNR_Switch_Pos* is *D*, while the switch has just been moved to another position. This can only occur if *DNRSwPos* changes at the same simulation step that a certain transition in the StateFlow diagram in the DNR Switch Logic is taken. This makes it an improbable scenario, but also a dangerous one, as it could cause unexpected behaviour for many of the components in the system, as the *Allowed_DNR_Switch_Pos* signal is used by most of them. This also holds for the Downhill Speed Control, which is why this requirement is not satisfied. If the DSC was already active when the above scenario occurs, it will not know that the *DNRSwPos* has changed and *DSC_SetSpeed* does not change to *sHigh*.

5.4.2 Graphical Verification

In Figure 12, we see an LTS generated from the mCRL2 model, showing only the transitions that are relevant for requirement 1: *cmnc_1* and *cmnc_7*. The LTS has been reduced using branching bisimulation equivalence to obtain a manageable system. Note that we can now see clearly that the model meets requirement 1 from Table 12: every transition labelled *cmnc_1_A* or *cmnc_1_St* enters the state labelled 7. From here, *cmnc_7_setspeed_none* is the only possible next transition, which makes it easy to see that there can never be a *cmnc_7_setspeed_other* in between.

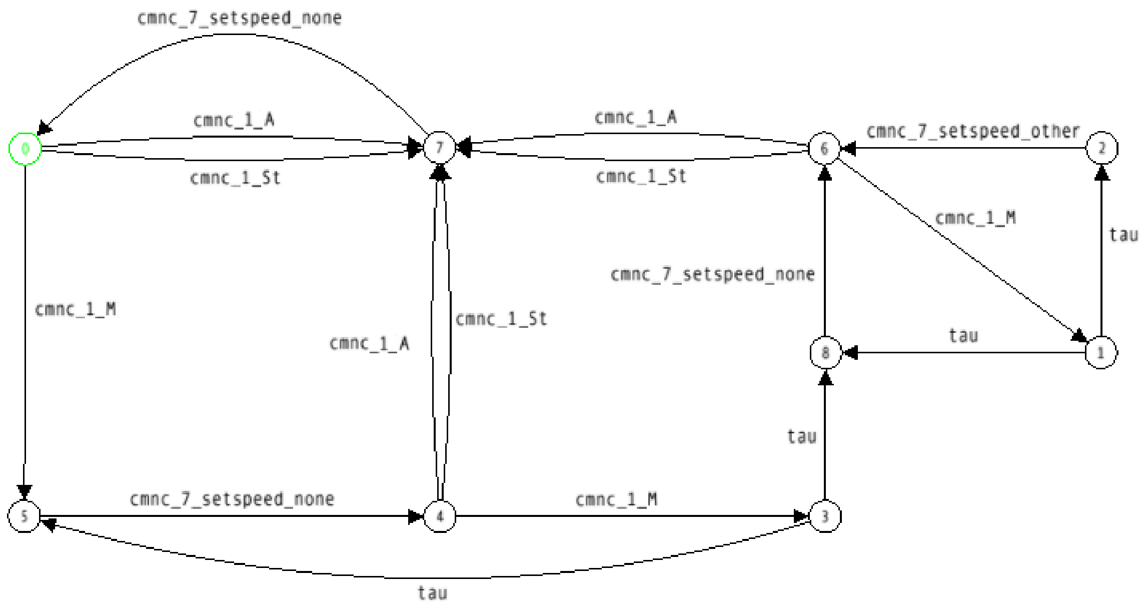


Figure 12: LTS of the model, showing only *cmnc_1*, *cmnc_7* and *tau*.

6 Conclusions

From the first part of the project, in which a model of a Cruise Control system was constructed in mCRL2, we learned that for designing a system, using only a set of high-level requirements is not enough. It is practically impossible to make a complete set of requirements that fully describes the desired behaviour of the system. However, using the requirements combined with a good understanding of the desired system, a high quality model can be constructed. Verifying the requirements while designing helps finding problems as soon as design errors are made. Note that not only the system model can be erroneous, but there may also be requirements that turn out to be too strong or too weak for our needs. When we finally end up with a system that meets all requirements, we have confidence in both the requirements' and the system's quality. A remarkable fact is that one of the proposed requirements seems to be first known example of a natural property impossible to express in a modal μ-calculus formula.

In the second part of the project, a StateFlow model of the Cruise Control system was translated to mCRL2. Even though these two modelling languages are rather dissimilar, a straightforward translation approach was found to ensure minimal behavioural differences between the models. Performing formal verification with a set of requirements showed that the translation was accurate, as every scenario that violated a requirement in the mCRL2 model turned out to be a possible scenario in the original StateFlow model as well.

The third and final part of the project featured the translation of a larger and more complex model to mCRL2. The Vehicle Function Architecture system contains several Matlab Simulink components that run in parallel and communicate with each other. The most challenging aspect about this part of the project was finding a translation approach that is not only as straightforward as possible, but also results in a model that allows performing efficient operations using the mCRL2 toolset. Once a fitting translation was obtained, the formal verification performed on the mCRL2 model proved to be valuable once again. Several problems were revealed that would have been very hard to find by reviewing or using test scenarios.

With this project we intended to investigate the potential and necessity of using formal verification in the automotive industry in general, and for DAF Trucks N.V. in particular. With software systems becoming an increasingly prominent part of vehicles, they also tend to grow larger and more complex. In order to obtain the software quality needed to guarantee the safety a vehicle obviously requires, designers need new techniques to prove that their product doubtlessly performs as required. During this project, the exposure of several problems in real-life models showed the power of formal verification. In a direct sense, this has led to several adjustments to the Simulink / StateFlow models by the DAF designers. But more importantly, it shows that formal verification is quickly becoming indispensable in the automotive industry. However, the approach used during this project, i.e. manually translating Simulink and StateFlow models to mCRL2, is very time consuming and requires extensive knowledge of the mCRL2 modelling language and modal μ-calculus, which may be major drawbacks for DAF.

Therefore, we recommend some topics for future research that can bring the automotive industry another step closer to using formal verification techniques effectively. First, it would be interesting to investigate the possibility of automating the translation from Simulink and

StateFlow models to mCRL2. This would make formal verification a substantially less time consuming task, but it would still require a good understanding of both mCRL2 and the modal μ-calculus. The main challenge here will be finding a translation scheme that not only preserves the original model's behaviour, but also produces mCRL2 models that allow efficient verification. Finally, a profound investigation of other available verification tools can show whether these are more suitable for Simulink and StateFlow models. While these tools may provide a more intuitive way of constructing requirements, they may lack expressiveness in comparison to the modal μ-calculus leaving some requirements impossible to verify. Another important factor would be the efficiency of the verification; it is crucial to understand how the size and complexity of the used models affects the performance.

References

[1] The mCRL2 Documentation. `http://www.mcrl2.org/mcrl2/wiki/index.php/Documentation`.

[2] The mCRL2 homepage. `http://www.mcrl2.org`.

[3] J. F. Groote, Aad H. J. Mathijssen, Y. S. Usenko, M. A. Reniers, and Muck J. Weerdenburg. The formal specification language mCRL2. *Methods for Modelling Software Systems (MMOSS)*, (06351), 2007.

[4] J.F. Groote and R. Mateescu. Verification of temporal properties of processes in a setting with data. *AMAST'98*, 1548:74–90, 1999.

Confidentiality

Note that this is the public version of the document in which the complete lists of requirements, used Simulink / StateFlow models and constructed mCRL2 models are omitted from the following appendices.

A Cruise Control Design Appendix

A.1 Context of the Cruise Control Supervisor

A.2 Functional Requirements for Designed Cruise Control

A.3 Designed Cruise Control mCRL2 Model

A.4 Transition Renamings

A.5 Functional Requirements for Designed Cruise Control expressed in modal μ-calculus

B Cruise Control Translation Appendix

B.1 Cruise Control StateFlow model

B.2 Functional Requirements for Translated Cruise Control

B.3 Translation Cruise Control to mCRL2 model

B.4 Transition Renamings

B.5 Functional Requirements for Translated Cruise Control expressed in modal μ-calculus

B.6 Counterexamples Unsatisfied Requirements Translation Cruise Control

B.7 Proposed Solution to Cruise Control StateFlow model

B.8 Translation Solution Cruise Control to mCRL2 model

C Vehicle Function Architecture Translation Appendix

C.1 Context of the VFA system

C.2 Vehicle Function Architecture Simulink Model

C.3 Functional Requirements for VFA

C.4 Translation VFA to mCRL2 model

C.5 Transition Renamings

C.6 Functional Requirements for VFA expressed in modal μ-calculus

C.7 Analysis Unsatisfied Requirements Translation VFA