T U M

I N S T I T U T F Ü R I N F O R M A T I K

A Top-Down Methodology for the Development of Automotive Software

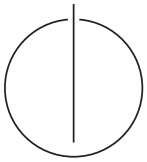
M. Feilkas, A. Fleischmann, F. Hölzl, C. Pfaller, K. Scheidemann, M. Spichkova, D. Trachtenherz

TUM-I0902
Januar 09

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

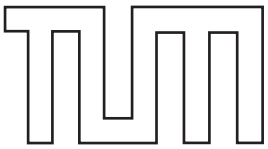
TUM-INFO-01-I0902-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München



TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Software & Systems Engineering
Prof. Dr. Dr. h.c. Manfred Broy



DENTUM

A Top-Down Methodology for the Development of Automotive Software

Authors: Martin Feilkas
Andreas Fleischmann
Florian Hölzl
Christian Pfaller
Sabine Rittmann
Kathrin Scheidemann
Maria Spichkova
David Trachtenherz

Technische Universität München - Fakultät für Informatik - Boltzmannstr. 3 - 85748 Garching

This report summarizes the main results of the project DENTUM between DENSO CORPORATION and the chair for Software & Systems Engineering at Technische Universität München. The goal of this project was to define a methodology for the model-based development of automotive systems. This methodology was evaluated by developing an Adaptive Cruise Control (ACC) system with Pre-Crash Safety (PCS) functionality.

This document explains the methodology to teach the reader the details of the work that has been done. The specifications of the ACC case study were developed at the chair for Software & Systems Engineering at Technische Universität München, the starting point for the case study was a requirement specification from DENSO CORPORATION.

Contents

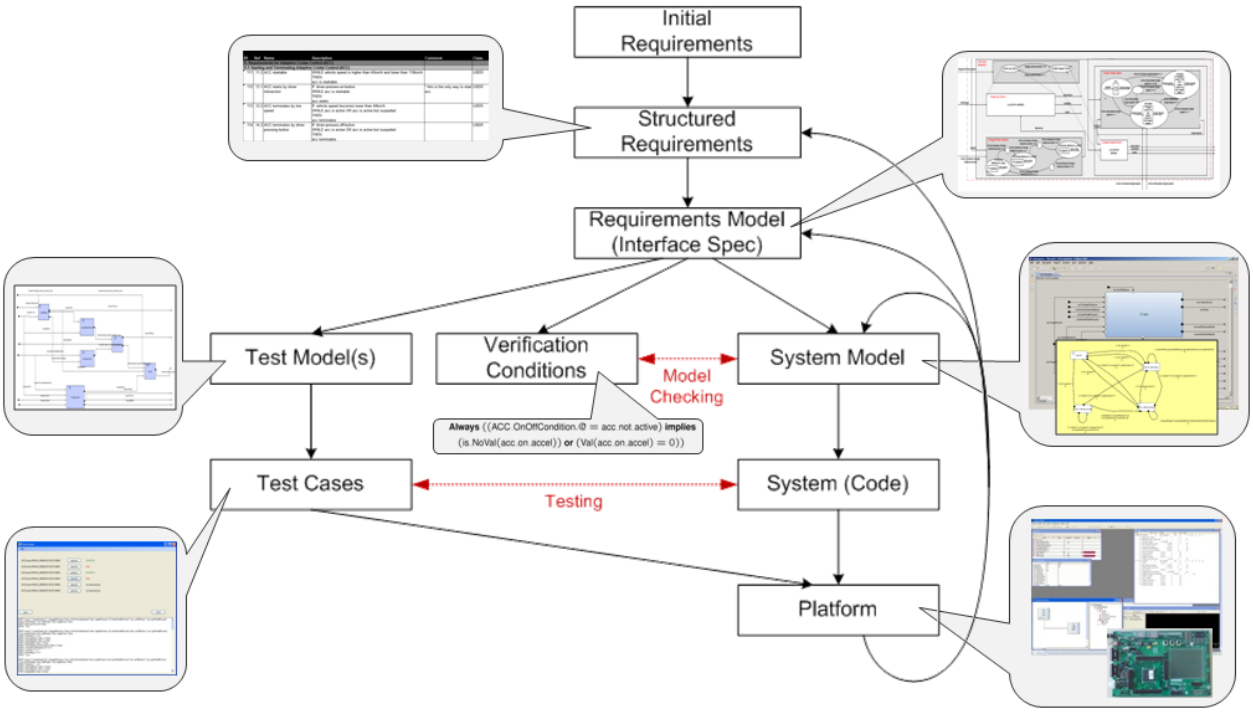


Figure 1: The process and artifacts

1 Introduction

The development of Software Systems in vehicles poses a number of challenges that can only be mastered by a more formalized development process, based on formal models. A key here is the mastering of the requirements engineering process that results in precise specifications of the functionality of systems, the decomposition of the systems into an architecture with its subsystems and finally the composition of the subsystems into a larger system which follows the verification path of the architecture.

Using the corresponding methodology presented below, the whole development process can be done in a more controlled way than it is usually today.

Figure 1 illustrates the basic structure of the process. The boxes represent the artifacts that have been developed and the arrows show from which other artifacts they were derived. The process tries to achieve system development in a top-down manner. Nevertheless, iterations are needed to cope with gain of knowledge about the system during the development process as well as modifications in the requirements.

The starting point for the case study was a document containing initial requirements given by DENSO CORPORATION. This initial requirement were then structured so that they follow specific syntactic patterns. By doing so the requirements become more homogeneous and more precise. After that the formulated textual requirements have been modeled as services that describe the functionality from the perspective of the user. This service oriented specification deals as the interface specification for the next phases of development. It also defines the first coarse-grained decomposition of the system. Based on this specification not only the implementation can be started but also the verification activities can start defining the artifacts needed to ensure the correctness of the behavior, such as the test model(s) and verification conditions. The implementation has been done using the CASE tool AutoFocus (Versions 2 and 3) using synchronous models that rely

on a global time base. In parallel a test model has been developed that implements the same functionality as the implementation model in a more abstract way. This test model is then used as an "oracle" to be able to generate test sequences with expected results and reactions of the system. Testing can be done on the level of the model by simulating the test cases as well as on the level of the deployed system. For the system tests a test driver had to be implemented. Beside the dynamic analysis using test case generation also static techniques have been used. Verification conditions have been derived directly from the formulated requirements and formulated in temporal logics. These formulas have then been checked against the implementation model.

The implementation model has been used to automatically generate C code that has been deployed onto MPC boards. An OSEK operating system has been used to execute the system periodically. The tool Vector CANoe was used to simulate the environment of the ACC and to test the implementation manually.

Although the development of the ACC case study was developed in iterations (to find out the most appropriate kind of the methodology) and did not follow a strict waterfall-like process, this document will explain the development of the final status of the artifacts and describes the transition between them as if they were never changed during the further development process. As David Parnas states in his article [PC85], this will ensure a comprehendable documentation of the development process.

The following section present the development of the ACC case study in a top-down order. Section 2 starts with describing the requirements engineering activities. After that the design and modeling method is shown in Section 3. Static verification using model checking and dynamic verification using model-based test case generation are presented in the Sections 4 and 5. Then the deployment of the completed product is described in Section 6. Finally, the last section summarizes the key ideas of the methodology and the results of the case study.

Every section first gives a motivation why this methodology has been chosen to solve the corresponding problems. After that the methodology will be presented in a general way before the concrete application to the ACC case study is given.

2 Requirements Engineering

In this section we describe the starting point of the approach: the requirements engineering (RE) activities. To that end, we give a motivation (see Section 2.1), explain the requirements engineering approach in general (see Section 2.2), and apply the approach to the case study in Section 2.3.

2.1 Motivation

Ideally, the development process starts (after the project acquisition phase) with the requirements engineering phase. The aim is to specify "what" the system-to-be is supposed to do. The requirements specification contains the system functionality that the system shall provide, constraints concerning the distribution and the technical realization, etc. Based on this information the system is developed in the subsequent design phase.

The main problems of the requirements engineering phase are the following:

- The requirements have to be specified
  - precisely (without ambiguities)

- consistently
- totally

- The requirements have to be validated (e. g. by simulation) to answer the question: "Does the requirements specification describe what I want?"
- There exists a gap between the informal requirements engineering phase (dealing with natural language) and the formal design phase (dealing with models).

Our approach aims at solving these problems. Thereby only focus on the functional requirements. This has the following reasons: Usually, requirements describing the system structure and the hardware requirements are well understood and do not pose problems. However, the functional requirements are intricate to handle. This is especially true when it comes to multi-functional systems. Multi-functional systems are characterized by a high degree of dependencies and interactions between functional entities. This kind of systems is typical for the automotive domain where the functionality is often provided by the interplay of several ECUs. The dependencies have to be captured and specified already in the requirements engineering phase.

The main contributions of our requirements engineering approach are the following:

- Precise specification of the requirements by help of pre-defined textual patterns and a method to define the vocabulary of those requirements.
- Seamless transition between the informal requirements engineering phase and the formal design phase

The result of the requirements engineering phase is a formal model of the usage behavior that can for example be checked for totality and simulated in order to validate the requirements.

Please note that non-functional requirements are also important in the requirements engineering phase but, as discussed earlier, not covered by our approach. Their handling is left to future work.

2.2 Methodology

In this section, we describe the methodology for the requirements engineering phase. It can be roughly divided into two parts:

- formulating and structuring textual requirements (see Section 2.2.1) and
- stepwise formalization of functional requirements (see Section 2.2.2).

Based on an initial set of requirements, the textually (informally) given functional requirements are structured and subsequently specified with the help of pre-defined text patterns during the first part. Furthermore, inputs and outputs of the system and system states are identified. In the second part, system services are identified and formalized step by step. The result is a formal model of the usage behavior.

2.2.1 Formulating and structuring textual requirements

In this first part, the informally given requirements are analyzed, categorized, formulated, and structured. This is done in four steps:

1. Establishing an initial template.
2. Categorizing the requirements.
3. Defining the logical interface.
4. Formulating the requirements.

In the following paragraphs, these four steps are described.

Establishing an initial template It is assumed that the requriements are gathered by an elicitation subprocess, which delivers requirements as informal text. In order to be further processed, these informally given requirements have to satisfy two initial properties: each requirement must have a unique number, and each requirement must describe only one aspect of the system's behaviour. This prerequisite is achieved by putting each requirement into a initial template which consists of the following fields:

- **Id** (mandatory): a unique identifier of the requirement, e. g. a number.
- **Name** (mandatory): some words that briefly label the requirement.
- **Description** (mandatory): the content of the requirement; initially an informally given text consisting of one or more sentences.
- **Comment** (optional): a further explanation of the requirement, which does not contain a crucial requirement content, but instead gives a reason for the requirement, or elaborates the meaning of the requirement.

These four fields are the fields needed by the formulation part. Of course, a requirements template can contain a lot more fields (e. g. author, date, priority); that doesn't affect this approach.

Categorizing the requirements Since the requirements are very heterogenous, it is not possible to structure and formulate all of them in the same way. Therefore the requirements are categorized into the following four homogeneous categories:

- **Business Requirements**: such requirements do not describe what the system is doing; instead they describe what the company wants to achieve with the product. Often business requirements appear not as individual requirements, but as rationale for user requirements. Typical business requirements are: "With this new product, the market share should be increased by 10 per cent" or "because this is a precondition for the product's usage in the European market."
- **User Requirements**: such requirements describe the system's properties and behaviour as it is percieved by the user of the overall system (e. g. driver of the car). Most of the requirements of the ACC case study are user requirements.
- **System Requirements**: such requirements describe the system's properties and behaviour as a part of the overall system that communicates with actuators, sensors, interfaces, channels, and other embedded systems.

- **Process Requirements**: such requirements do not describe what the system is doing; instead they describe how the system is supposed to be developed, e. g. the application of standards, laws, and development techniques (e. g. testing and verification directives).

For each category, specific formulation rules can be defined. For the category of user requirements, those rules have been defined (see next paragraphs). Hence, in the following, only **functional user requirements** will be further processed by applying those rules. All other requirements are also important, but their formalization is a task for future work; until then, those requirements will just stay untransformed in the initial template.

Defining the logical interface An informally given textual requirement consists of a set of words, which can be distinqued into two categories:

- Content words (**actions**): these are system-specific words (e. g. "driver presses on/off button" or "acc terminates"). The set of all content words forms the **logical interface** of the system: with what means does the system react, and what are the relevant events a system reacts to.
- Relation words (**keywords**): these are domain-independend words (e. g. "if", "then"). These words form relationships between the content words, and therefore define the **behaviour and properties** of the system.

In this step, the requirements are analyzed, and the actions (content words) are identified and extracted. The extracted actions are put into tables with the following fields:

- **Name**: the name of an action is identical with the action itself; it must be unique. Usually, in a specification document there are different ways used to express the same meaning, e. g. "system decelerates" and "system brakes" might mean the same. Whenever this is the case, one expression has to be chosen and all others discarded. As a result, for each meaning there is exactly one expression. While resolving these multiple expressions, the requirements engineer often has to ask, if two expressions really mean the same thing: these questions often lead to a better understanding of the logical interface of the system.
- **Input/output**: it has to be defined, whether an action describes an event, to which the system reacts (input), or a means, with which the system reacts (output).
- **Type**: an action can be one of the following types:
  - *Event*: an event describes a moment, in which the system realizes something or does something; an event has no duration. Examples are: a button gets pressed, another car appears within sensor range.
  - *State*: a state describes a time period, in which something is. Examples are: a button is pressed, another car is within sensor range.
  - *Activity*: an activity describes a time period, in which the system does something. Examples are: acc accelerates, acc warns driver.

  It is crucial to decide what type a content word is; in today's practice this distinction often is not clear. The clarification of the type is a big factor in better understanding the logical interface of a system.
- **State space**: states and events are structured within a table by sorting them to an appropriate state space. For example, the states "acc is active" and "acc is not active" belong to the same state space. Within a state space, certain rules apply:

- all states of a state space are disjunct.
- the system must be in exactly one state of a state space all the time (hence, a state space must be completed, so that this rule always is true).

- **Variables**: within actions there sometimes are variables that they reference; e. g. the action "driver-increases-target-speed" references the variable "target-speed". In order to better understand dependencies between actions, which can be recognized by their reference to the same variables, such variables are explicitly noted.

Formulating the requirements In the previous step, the system-specific content words have beed extracted from the requirements sentences and a logical interface has been defined. Now, in the last step of this phase, the requirements themselves are *formulated*. There, two rules apply:

- Only the standardized words from the logical interface can be used.
- Sentences can be formed according to requirements patterns only.

The following requirements patterns are defined:

- Reaction Behaviour Patterns
  - Direct Reaction of System: IF an event occurs WHILE a situation (optional) THEN the system reacts with another event
  - Direct Change of System: IF an event occurs WHILE a situation (optional) THEN a system state changes
  - Prohibition of Reaction: IF an event occurs WHILE a situation (optional) THEN an activity or event is forbidded
- Situation Behaviour Patterns
  - Situation Behaviour: WHILE a situation THEN an activity is performed
  - Behaviour Restrictions: WHILE a situation THEN an activity is restricted
  - Prohibition of Behaviour: WHILE A situation THEN an activity or event isd forbidden
- Invariant Patterns are directly formulated within the action tables.

The requirements are *structured* according to their preconditions (that is the list of situations belonging to a requirement). This structure is used as a starting point for the service tree in the next part. The logical requirements interface is used by the next part and extended to the logical syntax interface. And the formulated requirements are the foundation for defining the behaviour of the services. How this is done will be explained in the next paragraph.

The ideas presented in this section are explained in more detail in [Fle08].

2.2.2 Stepwise formalization of functional requirements

Based on the results of the previous step, the requirement will now be formalized step by step. The ideas presented in this section are explained in detail in [Rit08].

Identification of atomic services As the overall system functionality can be quite comprehensive, we structure it by *services*. A service is a *partial piece of behavior* which relates system inputs to system outputs. Thus, a service describes a piece of usage behavior which can be observed at the system boundaries. The system functionality is comprised of single services which collaboratively establish the overall functionality. By services, the usage behavior of a system is structured.

First, we identify the so-called *atomic services* which are the "smallest" services which are visible to and can by accessed by the user. A service corresponds to a use case which describes how to use a system for a specific purpose by means of interaction patterns. Examples for atomic services are "open the power window", "turn on the radio", or "move the seat toward the steering wheel".

Usually, there is no 1:1 mapping between functional requirements and services and that it requires genuine design work to identify the services out of the requirements.

Some of the atomic services need to operate on peristent data. For example, the speed control of the ACC functionality needs to save the desired speed at which the car is supposed to go. Therefore, we also informally specify this data. As services might operate on the same data[1] we list all persistent data in a repository. The informal service specifications then refer to entries of this list.

The atomic services and the persistent data are only specified *informally* at this stage of the methodology. We suggest to use tables for their specification (see Tables 1 and 2).

Table 1: Specification of atomic services

Req	Service name	Abbreviation	Textual service description	Data ref.
...	...	...	...	...

Table 2: Specification of persistent data

Abbreviation	Name	Description
...	...	...

Logical syntactic system interface As mentioned above, services define pieces of the usage behavior by mapping system inputs to system outputs. The inputs and output were already identified (see Section 2.2.1). At this place, we specify the inputs and outputs more precisely by adding more information. Although the inputs and output of the system are still considered at a logical level (e. g. we still abstract from technical signals and refer to the meaning instead), we assign data types and data ranges to the inputs and outputs, respectively.

During modeling the functionality formally (see Section 2.2.2) not all the previously inputs and outputs might be needed. Furthermore, several inputs might be integrated to another input. Therefore, in this step we only specify the inputs and outputs which we actually need within the formal models.

As notational technique, we suggest to use tables in order to specify the logical inputs and outputs, respectively. Table 3 shows an example. For each input/output, the requirements are listed which motivate the introduction of the input/output. Furthermore a name and an abbreviation are given.

[1] In that case one might need to solve conflicts if more than one services want to write the same data concurrently.

A data type is also introduced. However, the data type does not have to be the data type which is needed during the implementation. Rather should it be chosen in a way which is appropriate for modeling the functionality on the logical level.

Table 3: Specification of logical (input and output) actions

Reqs	Abbreviation	Name/discription	Data type
...	...	...	...

Identification of service relationships So far, we have captured the single atomic services in a modular fashion. In this step, we relate the services with each other. As mentioned above, multi-functional systems are intricate to develop as they are - per definition - characterized by a high degree of interaction between functionalities. Therefore, we explicitly capture the various dependencies, i. e. the *service relationships*, between them. As from our perspective, the system is a black box behavior, only relationships which are observable at the system boundaries, are taken into consideration. For example, the interruption of a service, but not the call of a sub functionality.

First, we structure the overall usage behavior *hierarchically* by so-called *vertical service relationships* (also called sub service relationships). Graphically the result is a tree in which the nodes are services and the edges represent the sub service relationship. The leaves of the tree are the atomic services identified in Section 2.2.2; the root is the overall system behavior. Note that our notion of service is scalable. A partial behavior can both be a small, atomic service and a more comprehensive functionality. For example, the ACC functionality is a service again which is comprised of the sub services Follow-up Control and Constant Speed Control. We call this ordering of the system services according to the sub service relationship the *service hierarchy*.

Next, we enrich the service hierarchy by the dependencies between the services, the so-called *horizontal service relationships*. For example, either the Follow-up Control or the Constant Speed Control is executed. Thus we have an XOR relationship between these (hierarchically decomposed) services. Each horizontal service relationship is informally specified. For example, the specification of the XOR relationship contains the information that the Follow-up Control is executed if no vehicle is detected and that otherwise the Constant Speed Control is executed.

The result is a table containing the informal specification of the horizontal service specifications and the so-called *service graph* which is the service hierarchy plus the dependencies.

Formal specification of atomic services In the previous steps we have modeled the usage behavior informally. In this step, we get formal. We specify the atomic services (i. e. the leaves of the service hierarchy/graph) formally. To that end, we make use of AutoFOCUS Mode Diagrams. Mode Diagrams specify the system behavior depending on the mode (state) the system is in. Mode diagrams consist of modes (states) and transitions between these modes. Transitions are labeled by predicates overn the system inputs and outputs and define mode transitions. For each mode, either a Mode Diagram is given in turn (i. e. the modes can be decomposed hierarchically), or the behavior is described by means of a data flow network. Entities in Mode Diagrams can make use of modes again.

Combination of services on basis of the service relationships So far we have only modeled the atomic services of the service hierarchy/graph formally. In this step, we combine the atomic services to more comprehensive services until the overall usage behavior is obtained. Doing this we make use of a bottom up approach concerning our service hierarchy.

During the combination process, the horizontal services relationships (i. e. the dependencies between the services) have to be taken into account. For example, if there exists an XOR relationship between services, they can be combined using modes. The XOR related behavior is specified in a (separate) mode, respectively. The mode transitions are given by the information which is specified for the XOR relationship.

Result of the requirements engineering phase As mentioned above, the result of the requirements engineering phase is a formal model of the usage functionality.

2.3 Application to the ACC Case Study

In the following sections of this chapter we are demonstrating the first two parts of methodology by the means of the case study of an adaptive cruise control (ACC). This case study resembles a real system; please note, that for confidentiality reasons specific values have been replaced by constants or are blackened.

2.3.1 Formulating and structuring textual requirements

Original requirements Starting point for formulating the requirements is the list of the originally given informal requirements as follows (Table 4):

Table 4: List of the originally given informal requirements

Number	Requirements
(1)	Select target vehicle. Probability is calculated by radar data. Condition: Within the defined range, High probability, Most close range vehicle.
(2)	Select target vehicle. Reject Parking cars.
(3)	Follow-up control of ACC system starts in case of target vehicle exists. Distance between two cars is controlled with in the target. Target acceleration is decided on deviation of distance and relative speed. Target acceleration is conveyed from Drive assist ECU to fusion ECU. Fusion ECU provides request to engine and brake components.
(4)	Follow-up control of ACC system starts in case of target vehicle exists. Distance between two cars is configurable depend on vehicle speed.
(5)	Constant speed control of ACC system starts in case of target vehicle NOT exists. Vehicle speed is controlled for target speed.
(6)	Target speed is configurable by Driver operation. Target speed is defined with cruise configuration switch. [Target speed] Initial condition : current speed, Increment : $+ChangeSpeed_{const}$ km/h, Decrement : $-ChangeSpeed_{const}$ km/h
(7)	Following distance is configurable by Driver operation. Following distance is changed by Setup SW. (3 Stage setup: Long → Middle → Short → Long → ...)
(8)	ACC vehicle speed is in the defined range. ACC vehicle setting speed range: $AccRangeMin_{const}$ $AccRangeMax_{const}$ km/h
(9)	ACC acceleration is in the defined range. [ACC target acceleration range] Lower limit: $AccMin_{const}$, Upper limit: $AccMax_{const}$
(10)	Put ACC slowdown limit warning in action, when acceleration speed of ACC control is under lower limit. In case of ACC target acceleration is under limit, ACC slowdown limit warning is commanded.
(11)	ACC will be able to start, if vehicle speed is within the specified speed frame. ACC control will be able to start, if vehicle speed is between $MinAccSpeed_{const,1}$ km/h and $MaxAccSpeed_{const}$ km/h.
(12)	When vehicle speed is under defined range, ACC control is terminated. ACC control will be terminated, if vehicle speed is less than $MinAccSpeed_{const,2}$ km/h.
(13)	ACC control will start by drive SW operation. ACC control will start in case of pushing [SET SW] by driver.
(14)	ACC control will be terminated by drive SW operation. ACC control will be terminated in case of pushing [CANCEL SW] by driver.
(15)	Select PCS target. Set PCS target when scanned object has a high probability of collision. (Embedded on radar function)

(Table continued on the next page.)

(Table of previous page continued.)

Number	Requirements
(16)	Put PCS warning in action, when collision time is short. In case of collision time is shorter then defined range, PCS warning is in action.
(17)	When collision time is further short, PCS brake assist is set for ready condition. In case of collision time is shorter than defined range, PCS brake assist is set for ready condition.
(18)	When collision time is further short, PCS seat belt control is set for ready condition. In case of collision time is shorter then defined range, PCS seat belt control is set for ready condition.
(19)	When collision time is further short, PCS brake control is executed. In case of collision time is shorter than defined range, PCS brake control is executed.
(20)	When vehicle speed is in defined range, PCS comes in. When vehicle speed is over $MinAccSpeed_{const,3}$ km/h, PCS comes in.
(19)	Drive's brake operation has a priority to ACC control. When Drive operate brake pedal, ACC control is terminated.
(20)	Driver's acceleration operation has a priority to ACC control. When Driver operate acceleration pedal, vehicle speed is able to speed up. (Embedded on engine ECU)
(21)	Driver's acceleration operation has a priority to ACC control. When Driver operate acceleration pedal, ACC brake control is suspended.
(22)	Compare request level of speed reduction between Driver's brake operation and PCS brake control. PCS brake control is continued when Driver operates brake pedal.
(23)	PCS brake control has a priority to Driver's accelerator pedal operation. During PCS brake control is in execution, Driver's acceleration request is turned down. (Embedded on engine ECU)
(24)	Compare request level of speed reduction between ACC control and PCS control. Strong request has a priority. Brake control has to be active if PCS brake control is NOT active.
(25)	Compare request level of speed reduction between ACC control and PCS control. Strong request has a priority. Engine throttle has to be closed if PCS Control is active.
(26)	ACC Control should be canceled when PCS brake control is operated.

Initial template In the first step (see section 2.2.1), the list of informally given requirements is put into an initial template. Therefore:

- It is taken care that the identifying number for each requirement is unique (in the originally given requirements, the numbers 19 and 20 are used several times).
- A field for a name is added to each requirement. This field can be left blank for now, but has to be filled with an appropriate name during the next steps; at the latest when the requirements are finally formulated (fourth step), a name has to be filled in.
- The requirements themselves are moved into the description fields; requirements, which contain more than one aspect of a system description, are split into several requirements.

By taking care that each requirement has a unique number, and by sometimes splitting one requirement into several requirements, the need of a new numbering of the requirements arises. Hence, the requirements get new identifying numbers. In order to be able to trace the requirements to their origins, another field is added, **OrigNumber**, which contains the original number of the original requirement that served as the source for the requirement.

The result of this first step is a table (table 5) that contains all requirements and that satisfies the rules of the initial template:

Table 5: Informally given requirements are put into an initial template

Number	OrigNumber	Name	Requirement
(1.1)	(1)	...	Select target vehicle. Condition: Within the defined range, High probability, Most close range vehicle.
(1.2)	(1)	...	Probability is calculated by radar data.
(2)	(2)	...	Select target vehicle. Reject Parking cars.
(3.1)	(3)	...	Follow-up control of ACC system starts in case of target vehicle exists.
(3.2)	(3)	...	Distance between two cars is controlled with in the target.
(3.3)	(3)	...	Target acceleration is decided on deviation of distance and relative speed.
(3.4)	(3)	...	Target acceleration is conveyed from Drive assist ECU to fusion ECU. Fusion ECU provides request to engine and brake components.
(4.1)	(4)	...	Follow-up control of ACC system starts in case of target vehicle exists. .
(4.2)	(4)	...	Distance between two cars is configurable depend on vehicle speed.
(5)	(5)	...	Constant speed control of ACC system starts in case of target vehicle NOT exists. Vehicle speed is controlled for target speed.
(6.1)	(6)	...	Target speed is configurable by Driver operation.
(6.2)	(6)	...	Target speed is defined with cruise configuration switch.
(6.3)	(6)	...	[Target speed] Initial condition : current speed, Increment : $+ChangeSpeed_{const}$ km/h, Decrement : $-ChangeSpeed_{const}$ km/h
(7.1)	(7)	...	Following distance is configurable by Driver operation. Following distance is changed by Setup SW. (3 Stage setup: Long $rightarrow$ Middle → Short → Long → ...)
(8)	(8)	...	ACC vehicle speed is in the defined range. ACC vehicle setting speed range: $AccRangeMin_{const}$ $AccRangeMax_{const}$km/h
(9)	(9)	...	ACC acceleration is in the defined range. [ACC target acceleration range] Lower limit: $AccMin_{const}$, Upper limit: $AccMax_{const}$
(10)	(10)	...	Put ACC slowdown limit warning in action, when acceleration speed of ACC control is under lower limit. In case of ACC target acceleration is under limit, ACC slowdown limit warning is commanded.
(11)	(11)	...	ACC will be able to start, if vehicle speed is within the specified speed frame. ACC control will be able to start, if vehicle speed is between $MinAccSpeed_{const,1}$ km/h and $MaxAccSpeed_{const}$ km/h.

(Table continued on the next page.)

(Table of previous page continued.)

Number	OrigNumber	Name	Requirement
(12)	(12)	...	When vehicle speed is under defined range, ACC control is terminated. ACC control will be terminated, if vehicle speed is less than $MinAccSpeed_{const,2}$ km/h.
(13)	(13)	...	ACC control will start by drive SW operation. ACC control will start in case of pushing [SET SW] by driver.
(14)	(14)	...	ACC control will be terminated by drive SW operation. ACC control will be terminated in case of pushing [CANCEL SW] by driver.
(15.1)	(15)	...	Select PCS target. Set PCS target when scanned object has a high probability of collision.
(15.2)	(15)	...	(Embedded on radar function)
(16)	(16)	...	Put PCS warning in action, when collision time is short. In case of collision time is shorter then defined range, PCS warning is in action.
(17)	(17)	...	When collision time is further short, PCS brake assist is set for ready condition. In case of collision time is shorter than defined range, PCS brake assist is set for ready condition.
(18)	(18)	...	When collision time is further short, PCS seat belt control is set for ready condition. In case of collision time is shorter then defined range, PCS seat belt control is set for ready condition.
(19)	(19)	...	When collision time is further short, PCS brake control is executed. In case of collision time is shorter than defined range, PCS brake control is executed.
(20)	(20)	...	When vehicle speed is in defined range, PCS comes in. When vehicle speed is over $MinAccSpeed_{const,3}$ km/h, PCS comes in.
(21)	(19)	...	Drive's brake operation has a priority to ACC control. When Drive operate brake pedal, ACC control is terminated.
(22.1)	(20)	...	Driver's acceleration operation has a priority to ACC control. When Driver operate acceleration pedal, vehicle speed is able to speed up.
(22.2)	(20)	...	(Embedded on engine ECU)
(23)	(21)	...	Driver's acceleration operation has a priority to ACC control. When Driver operate acceleration pedal, ACC brake control is suspended.
(24)	(22)	...	Compare request level of speed reduction between Driver's brake operation and PCS brake control. PCS brake control is continued when Driver operates brake pedal.
(25.1)	(23)	...	PCS brake control has a priority to Driver's accelerator pedal operation. During PCS brake control is in execution, Driver's acceleration request is turned down.
(25.2)	(23)	...	(Embedded on engine ECU)

(Table continued on the next page.)

(Table of previous page continued.)

Number	**OrigNumber**	**Name**	**Requirement**
(26)	(24)	...	Compare request level of speed reduction between ACC control and PCS control. Strong request has a priority. Brake control has to be active if PCS brake control is NOT active.
(27)	(25)	...	Compare request level of speed reduction between ACC control and PCS control. Strong request has a priority. Engine throttle has to be closed if PCS Control is active.
(28)	(26)	...	ACC Control should be canceled when PCS brake control is operated.

Categorized requirements In the second step (see section 2.2.1), all given requirements are analyzed, whether they are business, user, system or process requirements. In the ACC case study, **all** given requirements are **user requirements**, except from the following (Table 6):

Table 6: System requirements that are not forther processed

Number	OrigNumber	Name	Requirement
(1.2)	(1)	ACC target calculation	Probability is calculated by radar data.
(3.4)	(3)	ACC target acceleration	Target acceleration is conveyed from Drive assist ECU to fusion ECU. Fusion ECU provides request to engine and brake components.
(15.2)	(15)	PCS target selection	The selection of the PCS target is embedded on radar function.
(22.2)	(20)	Location of Priority Decision	That the Driver's acceleration operation has a priority to ACC control is embedded on engine ECU.
(25.2)	(23)	Location of Priority Decision	That PCS brake control has a priority to Driver's accelerator pedal operation is embedded on engine ECU.
(29)	—	Force of PCS brake	The force of the PCS brake depends on the vehicle-speed and the distance to the pcs-object.

The listed **system requirements** (Table 6) are not formalized in the requirements phase and hence are passed through to the next phase as a list as it is. All **user requirements** (which are the majority of the given requirements) are further processed within the requirements phase (see the next steps: third and fourth step).

Logical interface The previous two steps produced intermediate results. The now following steps produce the output of the first requirements part which will be used by the second requirements part. In this third step now (see also section 2.2.1), the logical interface is extracted from the requirements. This is done by going through the requirements one by one, extracting the relevant content words (actions), and sorting them into a structured table. Within the table, missing actions are added. The logical interface consists of three parts:

- A table containing the states and events, sorted into a structure of state spaces.
- A table containing the activities.
- A table containing the variables.

The states and events extracted from the requirements (classified, normalized, and structured) are (Table 7):

Table 7: Logical interface: States and Events

Input/Output	StateSpace	Action	ActionType	Variables	Rules
input (environment)	target vehicle	no target vehicle	state	-	-
input (environment)	target vehicle	target vehicle gets detected	event	-	-
input (environment)	target vehicle	target vehicle exists	state	-	-
input (environment)	target vehicle	target vehicle disappears	event	-	-
input (driver)	on/off button	driver-presses-on-button	event	-	-
input (driver)	on/off button	driver doesn't-press-on-button	state	-	-
input (driver)	on/off button	driver doesn't-press-off-button	state	-	-
input (driver)	on/off button	driver-presses-off-button	event	-	-
input (driver)	changing target-speed	driver-increases-target-speed	event	target-speed	-
input (driver)	changing target-speed	driver-decreases-target-speed	event	target-speed	-
input (driver)	changing following-distance	driver-presses-change-distance-button	event	following-distance	-
input (vehicle)	speed analysis	vehicle speed becomes lower than $MinAccSpeed_{const,2}$ km/h	event	vehicle-speed	-
input (vehicle)	speed analysis	vehicle speed becomes higher than $MinAccSpeed_{const,3}$ km/h	event	vehicle-speed	-

(Table continued on the next page.)

(Table of previous page continued.)

Input/Output	**StateSpace**	**Action**	**ActionType**	**Variables**	**Rules**
input (vehicle)	speed analysis	vehicle speed is higher than $MinAccSpeed_{const,3}$ km/h	state	vehicle-speed	-
input (vehicle)	speed analysis	vehicle speed becomes higher than $MaxAccSpeed_{const}$ km/h	event	vehicle-speed	-
input (vehicle)	speed analysis	vehicle stops	event	vehicle-speed	EQUALS vehicle-speed is 0 km/h
input (vehicle)	speed analysis	vehicle speed is higher than $MinAccSpeed_{const,1}$ km/h and lower than $MaxAccSpeed_{const}$ km/h	state	vehicle-speed	-
input (acc)	target speed analysis	target-speed-is-equalbigger-$AccRangeMax_{const}$ km/h	state	target-speed	-
input (acc)	target speed analysis	target-speed-is-equalsmaller-$AccRangeMin_{const}$ km/h	state	target-speed	-
input (target vehicle)	distance-to-object-analysis	collision-time becomes smaller than limit	event	collision-time, collision-time-limit	-
input (target vehicle)	distance-to-object-analysis	collision-time is smaller than limit	event	collision-time, collision-time-limit	-
output (acc)	startable/not startable	acc becomes startable	event	-	-

(Table continued on the next page.)

(Table of previous page continued.)

Input/Output	**StateSpace**	**Action**	**ActionType**	**Variables**	**Rules**
output (acc)	startable/not startable	acc is startable	state	-	-
output (acc)	startable/not startable	acc becomes not-startable	event	-	-
output (acc)	startable/not startable	acc is not-startable	state	-	-
output (acc)	activation/termination	acc starts	event	-	-
output (acc)	activation/termination	acc is active	state	-	-
output (acc)	activation/termination	acc terminates	event	-	-
output (acc)	activation/termination	acc is not active	state	-	-
output (acc)	active acc modes	acc is in follow-up-control	state	-	-
output (acc)	active acc modes	acc starts constant-speed-control	event	-	-
output (acc)	active acc modes	acc is in constant-speed-control	state	-	-
output (acc)	active acc modes	acc starts follow-up-control	event	-	-
output (acc)	setting target-speed	acc sets target-speed to current-speed	event	target-speed, current-speed	-
output (acc)	setting target-speed	acc increases target-speed with $+ChangeSpeed_{const}$ km/h	event	target-speed	-
output (acc)	setting target-speed	acc decreases target-speed with $-ChangeSpeed_{const}$ km/h	event	target-speed	-
output (acc)	change following distance	acc sets following distance to "long"	event	following distance	-
output (acc)	change following distance	following-distance is "long"	state	following distance	-

(Table continued on the next page.)

(Table of previous page continued.)

Input/Output	**StateSpace**	**Action**	**ActionType**	**Variables**	**Rules**
output (acc)	change following distance	acc sets following distance to "middle"	event	following distance	-
output (acc)	change following distance	following-distance is "middle"	state	following distance	-
output (acc)	change following distance	acc sets following distance to "short"	event	following distance	-
output (acc)	change following distance	following-distance is short"	state	following distance	-
output (pcs)	active/not active	pcs is active	state	-	-
output (pcs)	active/not active	pcs is not active	state	-	-

The activities extracted from the requirements are (Table 8):

Table 8: Logical interface: Activities

Input/Output	Actor	Action	Influences	Rules	Comments
output	acc	search for target vehicle	target vehicle exists	REQUIRES acc is active	continously searches for and selectes target vehicle
output	acc	control-following-distance-to-target-vehicle	vehicle speed, vehicle acceleration	REQUIRES acc is in follow-up-control	-
output	acc	adjust-vehicle-speed-to-target-speed	vehicle speed, vehicle acceleration	REQUIRES acc is in constant-speed-control	-
output	acc	keep-vehicle-speed-under-target-speed	vehicle speed, vehicle acceleration	-	-
output	acc	acc-ignores-user-input-except-on-button	-	-	-
output	acc	acc-does-not-break-or-accelerate-or-warn	-	-	-
output	pcs	search for pcs-object	-	REQUIRES pcs is active	continously searches for and selects pcs object
output	pcs	warns-driver-about-small-collision-time	-	REQUIRES pcs is active	-
output	pcs	rewind-seatbelt	-	REQUIRES pcs is active	-
output	pcs	release-seatbelt	-	REQUIRES pcs is active	-
output	pcs	execute-brake	vehicle speed, vehicle acceleration	REQUIRES pcs is active	-

The variables extracted from the states, events, and activities, are (Table 9):

Table 9: Logical interface: Variables

Variable	**Definition**	**Range**
Variable	**Definition**	**Range**
Target Vehicle	a vehicle (1) within range (2) within the same lane (with a high probability) (3) if several possibilities: choose closest (4) parking cars must be ignored	Object
Vehicle Speed	actual speed (current speed) of vehicle	-100 km/h up to +400 km/h
Target Speed	desired speed of vehicle	$AccRangeMin_{const}$ km/h up to $AccRangeMax_{const}$ km/h
Vehicle Acceleration	actual acceleration of vehicle	-
Following Distance	desired distance to target vehicle	small, medium, long (depends on vehicle speed)
PCS Object	pcs-target is a scanned object with a high probability of collision	Object
Collision-Time	estimated time till crushing into pcs-object	time in millisec between 0 and 10000
Collision-Time-Limit	minimal allowed collision time	time in millisec (e.g. 10 millisec)

Note that the range of the variables must be specified in more detail on the systems level.

Formulated and structured requirements In the fourth and last step (see section 2.2.1), the requirements themselves are formulated using the defined patterns of keywords and using the normalized content words (actions) from the logical interface. Also, the requirements are structured according to the preconditions within the patterns. Since the previous requirement lists (initial template, categorization) were intermediate results to transform informal requirements into formulated ones, the following list is also the place where change requests and new requirements are dealt with. The result is the following list of requirements (Table 10):

Table 10: Formulated requirements

Number	OldNumber	Name	Requirement
			1. Requirements for Adaptive Cruise Control (ACC)
			1.1 Starting and Terminating Adaptive Cruise Control (ACC)
111	11.2	ACC startable	WHILE vehicle speed is higher than $MinAccSpeed_{const,1}$ km/h and lower than $MaxAccSpeed_{const}$ km/h THEN: acc is startable
111b	—	ACC not startable	WHILE vehicle speed is lower than $MinAccSpeed_{const,1}$ km/h or higher than $MaxAccSpeed_{const}$ km/h THEN: acc is not startable
112	13.1	ACC starts by driver interaction	IF driver-presses-on-button WHILE acc is startable THEN acc starts
113	13.1	ACC starts by driver interaction only	WHILE driver-doesn't-press-on-button THEN acc doesn not start
114	12.2	ACC terminates by low speed	IF vehicle speed becomes lower than $MinAccSpeed_{const,2}$ km/h WHILE acc is active THEN acc terminates
115	14.2	ACC terminates by driver pressing button	IF driver-presses-off-button WHILE acc is active THEN acc terminates
116	21.2	ACC terminates by driver brake pedal	IF driver operates brake pedal WHILE acc is actived THEN acc terminates
117	23.2	ACC break overruled by driver acceleration	WHILE driver-presses-acceleration-pedal WHILE acc is active THEN acc MUST NOT break
			1.2 General Behaviour while ACC is active
121	3.1	Follow-Up-Control Condition	WHILE acc is active AND target vehicle exists THEN: acc is in follow-up-control
122	5.1	Constant-Speed-Control Condition	WHILE acc is active AND no target vehicle THEN: acc is in constant-speed-control
123	—	Target-Speed is always Maximum Speed	WHILE acc is active AND WHILE target-speed-is-set THEN acc MUST keep-vehicle-speed-under-target-speed
124	6.4	Incrementing Target-Speed	IF driver-increases-target-speed WHILE acc is active THEN acc increases target-speed with $+ChangeSpeed_{const}$ km/h
125	—	Maximal Target-Speed	IF driver-increases-target-speed WHILE acc is active AND target-speed-is-equalbigger-$AccRangeMax_{const}$ km/h THEN target-speed is set to $AccRangeMax_{const}$ km/h

(Table continued on the next page.)

(Table of previous page continued.)

Number	OldNumber	Name	Requirement
126	6.5	Decrementing Target-Speed	IF driver-decreases-target-speed WHILE acc is active THEN acc decreases target-speed with $-ChangeSpeed_{const}$ km/h
127	—	Minimal Target-Speed	IF driver-decreases-target-speed WHILE acc is active AND target-speed-is-equalsmaller-$AccRangeMin_{const}$ km/h THEN target-speed is set to $AccRangeMin_{const}$ km/h
			1.3 Behaviour in Follow-Up-Control
131	3.2	Follow-Up-Control Behaviour	WHILE acc is active AND acc is in follow-up-control THEN: acc controls-following-distance-to-target-vehicle
132	7.3	Initial Following-Distance	IF acc starts follow-up-control THEN acc sets following distance to middle
133	7.3	Changing Following-Distance (long to middle)	IF driver-presses-change-distance-button WHILE acc is in follow-up-control AND following-distance is long THEN acc sets following distance to middle
134	7.3	Changing Following-Distance (middle to short)	IF driver-presses-change-distance-button WHILE acc is in follow-up-control AND following-distance is middle THEN acc sets following distance to short
135	7.3	Changing Following-Distance (short to long)	IF driver-presses-change-distance-button WHILE acc is in follow-up-control AND following-distance is short THEN acc sets following distance to long
			1.4 Behaviour in Constant-Speed-Control
141	5.2	Constant-Speed-Control Behaviour	WHILE acc is in constant-speed-control THEN: adjust-vehicle-speed-to-target-speed
142	6.3	Initial Target-Speed	IF acc starts constant-speed-control THEN acc sets target-speed to current-speed
			1.5 General Behaviour of ACC when not active
151	—	Behaviour when not active	WHILE the acc is not active THEN acc-ignores-user-input-except-on-button AND acc-does-not-break-or-accelerate-or-warn
			2. Requirements for Pre-Crush Control (PCS)
			2. 1 Starting and Terminating Pre-Crush-Control (PCS)
211	20.2	PCS Active-Condition	WHILE vehicle speed is higher than $MinAccSpeed_{const,3}$ km/h THEN: pcs is active
212	—	PCS Not-Active-Condition	WHILE vehicle speed is lower than $MinAccSpeed_{const,3}$ km/h THEN: pcs is not active

(Table continued on the next page.)

(Table of previous page continued.)

Number	OldNumber	Name	Requirement
			2.2 Arbitration of PCS and Driver
221	24.2	PCS-Brake undisturbed by Driver-Brake	IF driver brakes WHILE pcs-brake-control-is-executed THEN pcs-brake-control-execution-continues
222	25.2	PCS-Brake overrules Driver-Acceleration	IF driver accelerates WHILE pcs-brake-control-is-executed THEN driver's acceleration request is turned-down
			2.3 Stages of Pre-Crush-Behaviour
231	16.2	PCS warning	IF collision-time becomes smaller than limit WHILE pcs is active THEN pcs warns-driver-about-small-collision-time
232	17.2	PCS brake	IF collision-time becomes smaller than limit WHILE pcs is active THEN pcs executes-break
233	18.2	PCS seat belt	IF collision-time becomes smaller than limit WHILE pcs is active THEN pcs rewinds-seatbelt
236	19.2	PCS brake-end	IF collision-time becomes larger than limit OR vehicle-stops WHILE pcs execute-brake THEN pcs ends-pcs-brake-control-execution AND pcs releases-seatbelt
			3. Arbitration of PCS and ACC
311	28	PCS-Brake suspends ACC	IF pcs-executes-brake WHILE acc-is-active THEN acc terminates

The three tables that form the logical interface, and the list of formulated requirements serve as the starting point for the next phase: the formalization and modelling of requirements with services.

2.3.2 Stepwise formalization of functional requirements

In this step of the requirements engineering methodology, we formalize the functional requirements of the ACC case study.

Identification of atomic services For the case study, we obtain the examplatory atomic services shown in Table 11. Furthermore, the data that has to be exchanged between services is shown in Table 12.

Table 11: Specification of atomic services

Req	Service name	Abbr.	Textual service description	Data ref.
...	ACC on, off, suspended condition	ACC on off suspended	The ACC is set to active if the current speed is greater than $MinAccSpeed_{const,1}$ and less than $MaxAccSpeed_{const}$ km/h and the the respective on-button is pressed. If the ACC is active, it can be suspended either if the acceleration pedal is pressed or if the pre-crush system suspends it. It can be set to not active, either if the driver presses the off button, the current speed is less than $MinAccSpeed_{const,2}$ km/h, greater than $MaxAccSpeed_{const}$ km/h, or if the brake pedal is pressed.	-
...	Follow Up Control	FUC	The system keeps a pre-defined distance to the target vehicle. The distance depends on the chosen mode (short, middle, long).	distance-to-be
...	Change Follow Distance	CFD	The distance of the car to the target vehicle can be adjusted by the driver. The available modes are: "short", "medium", "long". Depending on the mode, the distance (depending on the current vehicle speed) is calculated differently. If the ACC starts the follow up control behavior, the mode is set to "middle". By pressing the distance button (again and again) the mode can be changed to "short", "long", "medium", "short"...	distance-to-be
...	Change Target Speed	CTS	The user can change the target speed (ranging from $AccRangeMin_{const}$ to $AccRangeMax_{const}$ km/h) by pressing the increase/decrease button. If the increase/decrease button is pressed, the target speed is increased/decreased by $ChangeSpeed_{const}$ km/h.	target-speed
...	Constant Speed Control	CSC	If the car is in the constant speed control mode, it automatically keeps the target speed.	target speed
...	Collision Time Smaller Than Limit	CTSTL	If the collision time is smaller than a calculated limit, the brakes are executed, the seatbelt is rewinded and a warning is given. If the car stops, the seatbelt is released.	-

(Table continued on previous page.)

(Table of previous page continued.)

Req	**Service name**	**Abbr.**	**Textual service description**	**Data ref.**
...	Collision Time Not Smaller Than Limit	CTNSTL	If the collision time is not smaller than a calculated limit, the PCS does nothing.	-
...	PCS on & off	PCS on off	The PCS is only active if the current speed is higher than $MinAccSpeed_{const,3}$ km/h.	-

Table 12: Specification of persistent data

Abbreviation	Name	Description
distance-to-be	distance mode	The driver can chose the distance mode (”short”, ”medium”, ”long”) for the Follow Up Control.
target-speed	Target speed	The target speed as chosen by the driver.

Logical syntactic interface In this step we list the logical inputs and outputs as needed for the formal modeling of the usage behavior. Note that the process of identifying the logical inputs and outputs is iterative and intermingled with the modeling of the system behavior.

Identification of service relationships In this step, we identify the service relationships of our case study. To that end, we first structure the services hierarchically (service hierarchy) by vertical relationships and then enrich the service hierarchy by horizontal dependencies. The resulting service graph is shown in Figure 2.

The overall usage behavior is comprised of the 'ACC functionality' and the 'PCS functionality'. The 'ACC functionality' can be again decomposed into the functionality to turn it on and off the core behavior. The core behavior is made up of the atomic services 'Follow Up Control', 'Constant Speed Control', 'Change Follow Distance', and 'Change Target Speed'. We introduced the service 'ACC core behavior' to be able to turn the four atomic services on and off at once. As far as the 'PCS functionality' is concerned, we analogously decompose it into a service which is responsible to turn it on and off ('PCS on off') and the remaining functionality. The latter can be distinguished into two services which take place if the collision time is smaller or not small than a given limit.

With help of the 'ACC on off suspended' ('PCS on off') service, the 'ACC core behavior' ('PCS core behavior') can be turned on and off, therefore we introduce an 'enable' and a 'disable' relationship. The 'Follow Up Control' and the 'Constant Speed Control' exclude each other. This is represented by the 'XOR' relationship. Analogously, we have an 'XOR' relationship between the sub services of the 'PCS core behavior'. If the collision time is smaller than a given limit, the ACC functionalities are suspended. This is indicated by the 'suspend' relationship. Furthermore, we have two data dependencies in our service graph. The distance mode ('distance-to-be') is determined by the 'Change Follow Distance' service and read by the 'Follow Up Control' service. The target speed ('target-speed') is set by the service 'Change Target Speed' and read by the 'Constant Speed Control' service.

Note that the requirements also specify that the PCS has a higher priority than the driver (concerning braking and accelerating). In order to model this formally, the braking and accelerating by the driver has to be modeled, too. Therefore, this is omitted in the remainder.

Table 13: Specification of logical inputs

Reqs	Abbreviation	Name of action	Data type
...	vspeed	Current speed of the car in km/h.	Integer
...	vehicle-stops	Indicates if the vehicle is stopping.	Bool
...	driver-presses-on-button	Set to TRUE if the driver presses the on button of the ACC.	Bool
...	driver-presses-off-button	Set to TRUE if the driver presses the off button of the ACC.	Bool
...	brake-pedal-pressed	Set to TRUE if the break pedal is pressed.	Bool
...	acc-pedal-pressed	Set to true if the acceleration pedal is pressed.	Bool
...	driver-presses-change-distance-button	By pressing this button, the driver can change the distance mode (short, medium, large). (If the driver presses the button, the value is set to TRUE. If the button is not pressed, the value is FALSE.)	Bool
...	collision-time-is-smaller-than-limit	The collision time is smaller than a calculated limit.	Bool
...	target-vehicle-exists	Set to TRUE if a target vehicle is detected.	Bool
...	dist-target	Depicts the distance to a target object in meter. The change of variable value (from 0 to a value or from a value to 0) depicts the appearance (leaving) of the object. (The distance is needed for the caluculation of the collision time.)	Real
...	driver-decreases-target-speed	By pressing the decrease speed button (set to true), the driver decreases the target speed by $ChangeSpeed_{const}$ km/h.	Bool
...	driver-increases-target-speed	By pressing the increase speed button (set to true), the driver increases the target speed by $ChangeSpeed_{const}$ km/h.	Bool
...	...	...	...

Table 14: Specification of logical outputs

Reqs	Abbreviation	Name of action	Data type
...	accel	Set to TRUE if the car is accelerating.	Bool
...	slow-dowm	Set to TRUE if the car is slowing down.	Bool
...	constant	Set to TRUE if the car is driving at constant speed.	Bool
...	execute-brake	Set to TRUE if the car is breaking.	Bool
...	rewind-seatbelt	Informs the seatbelt system to rewind the seat belt if set to TRUE.	Bool
...	release-seatbelt	Informs the seatbelt system to release the seat belt if set to TRUE.	Bool
...	warns-driver-about-small-collision-time	Set to TRUE if a warning about a collision is issued.	Bool

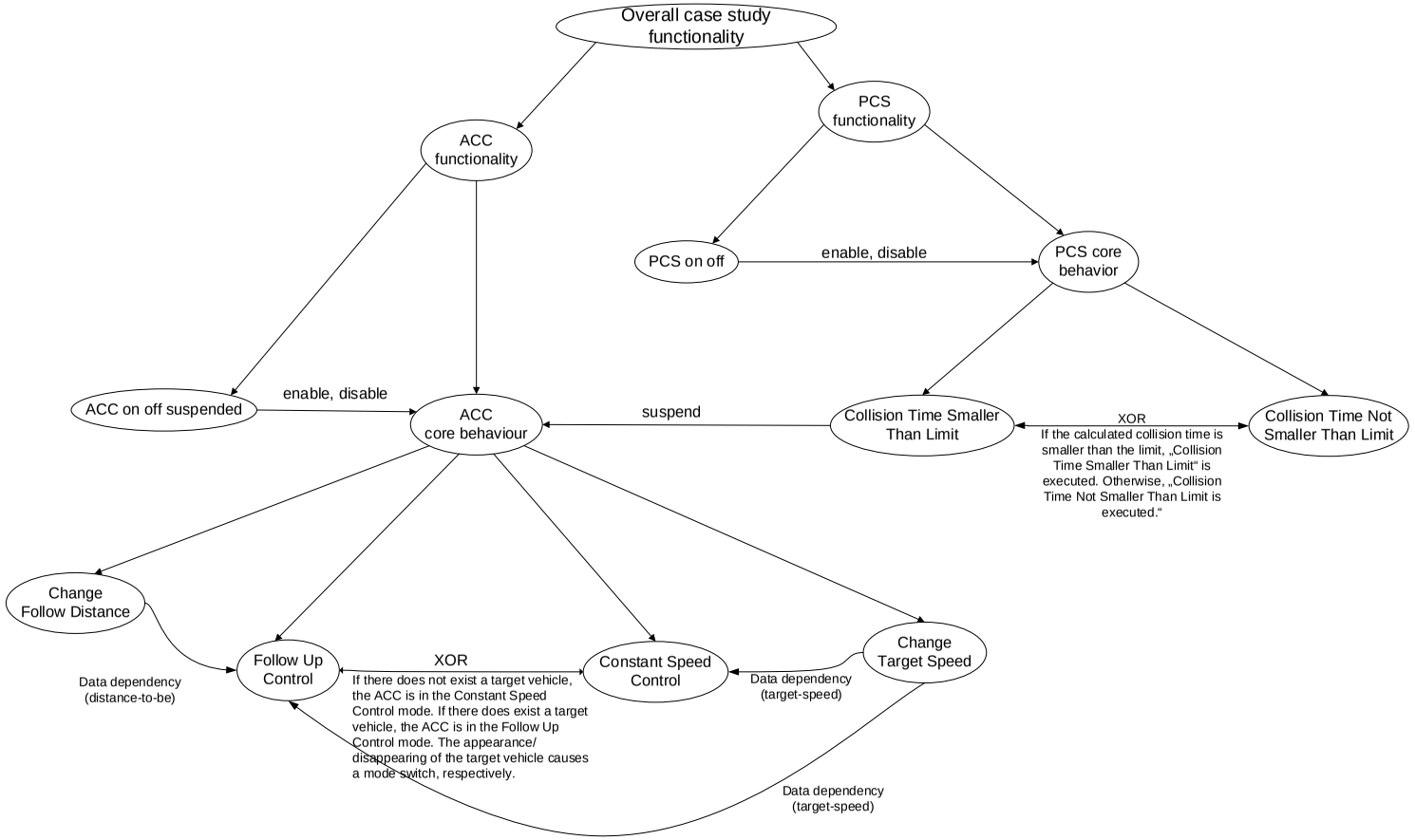


Figure 2: Service hierarchy

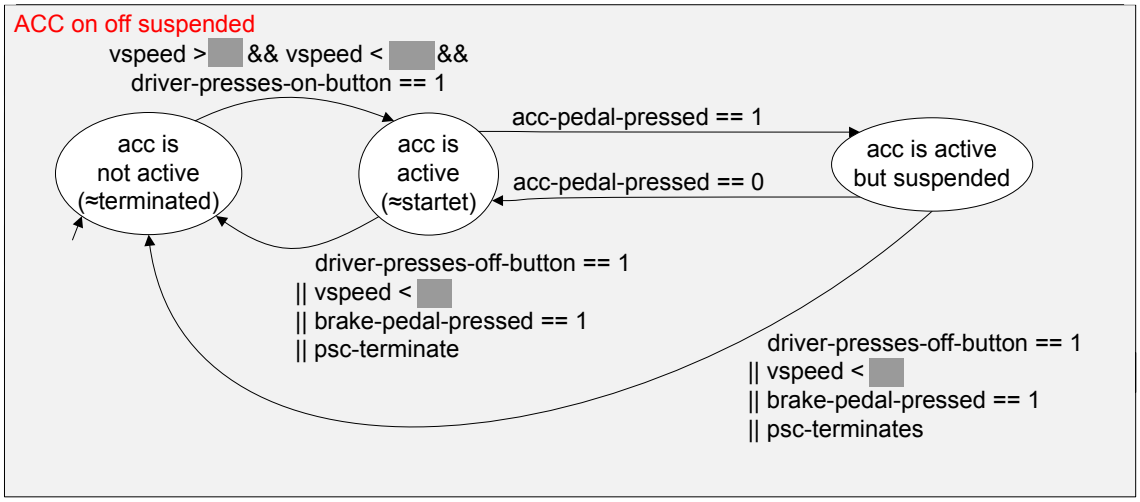


Figure 3: Formal specification of 'ACC on off suspended'

Formal specification of atomic services The atomic services (contained in Table 11) are now formalized by means of AutoFOCUS Mode Diagrams. As services are partial pieces of behavior, one might not want to specify the output on each channel. Therefore, if no output is specified, no restriction is made for this output channel.

The 'ACC on off suspended' functionality is given in Figure 3. Basically, it is comprised of three states (modes) and the transitions between the modes as specified in the requirements.

The 'Change Follow Distance' service is given in Figure 4. If the driver presses the distance mode button, the modes are entered sequentially. In each mode, the respective distance (which the car is supposed to have to the detected vehicle) is sent to the 'Follow Up Control'.

The formalization of the other atomic services are omitted at this place but can be seen in Figure 5.

Combination of services on basis of the service relationships So far we have formally modeled the atomic services of the case study. Now, we combine the formal model on basis of the service relationship. Figure 5 shows the result. The 'ACC core behavior' is only executed if the ACC is active. Therefore, the behavior is assigned to the mode 'acc is active'. Furthermore, the 'ACC core behavior' is comprised of the formal specifications of the four sub services 'Follow Up Control', 'Change Follow Distance', 'Change Target Speed', and 'Constant Speed Control'. The mode automaton assures that the 'Follow Up Control' and the 'Constant Speed Control' mutually exclude each other (depending on if a target vehicle is detected or not). The other two services ('Change Follow Distance' and 'Change Target Speed') are combined in parallel.

As far as the PCS is concerned, its core behavior ('PCS core behavior') is only performed in the mode 'pcs is active'. The XOR relationship between the services 'Collision Time Smaller Than Limit' and 'Collision Time Not Smaller Than Limit' is realized by a mode automaton. By sending the internal action 'psc-terminate' to the ACC (to be more precise to the 'ACC on off suspend' service), the ACC is terminated if the PCS is executing the brake.

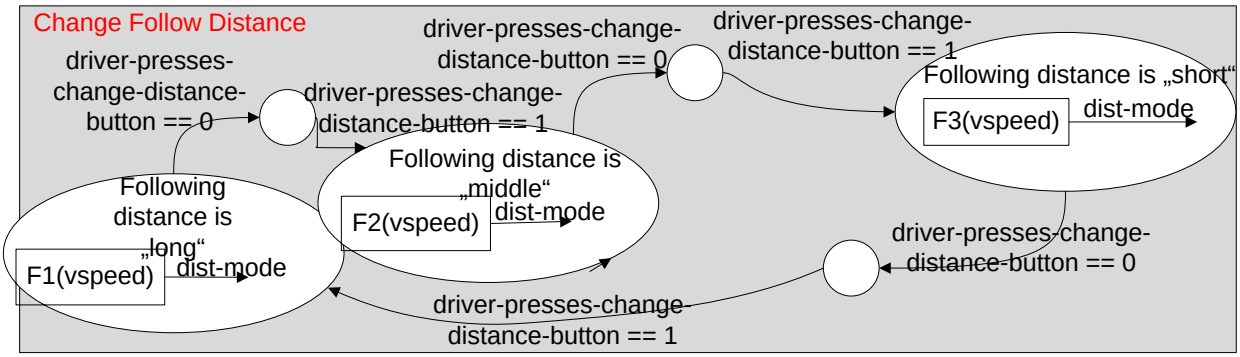


Figure 4: Formal specification of 'Change Follow Distance'

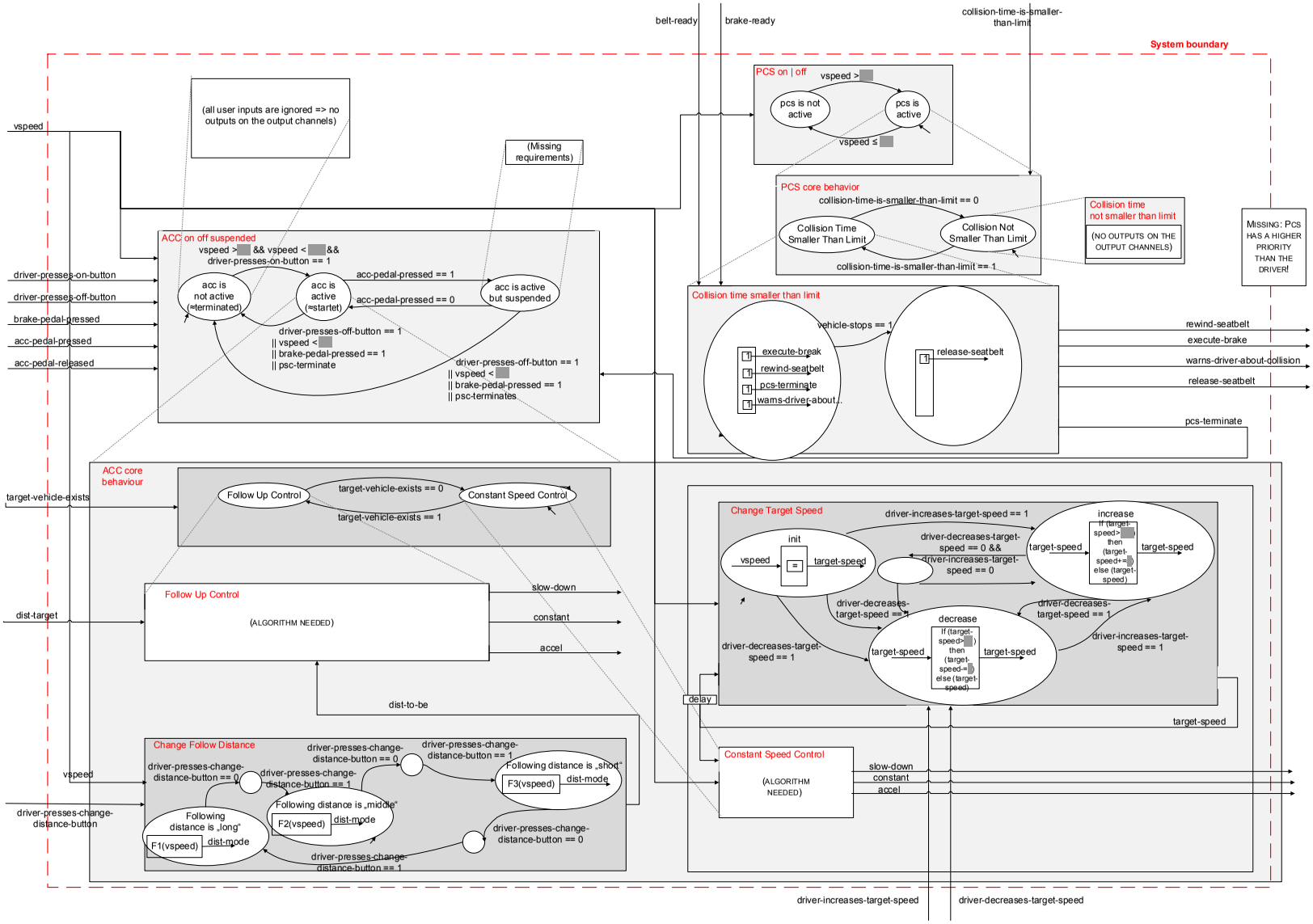


Figure 5: Formal specification of the overall usage behavior

Result of the requirements engineering phase The result of the requirements engineering phase is a formal model of the overall usage behavior. It is already shown in Figure 5.

3 Modeling in AutoFOCUS

In this section, we describe the next step towards the system implementation. We develop a logical model of the system that can be simulated to do a first step in quality assurance. This model will be the central artifact for the subsequent development. It is the basis for generating executable code. Furthermore prallel activities described in subsequent chapters parts of the model will be inspected and verified using model-checking techniques, other parts on the contrary will be validated using testing techniques.

3.1 Motivation

While the service architecture is the first step towards an implementation, we now refine this specification further. We develop a logical model of the system under development. This model abstracts from the concrete execution environment, i.e. the hardward structures, like ECUs and busses, and the basic software parts, like the operating system and its processes. For example, the logical model uses data types independently of their later representation on some given execution environment, e.g. we use integer values without taking care of their later precision (int32, int64, ...).

The model developed here is a complete specification in the sense that each component of the system is completely specified. This specification comprises each component's syntactical interface and its behavior (its semantical interface).

3.2 Methodology

The methodology for developing the logical model of the system is divided into the three following steps:

- **Data definition**: basic user data types and functions must be defined before they can be used in component interfaces and behavior.
- **System structure**: the static structure of the system is decribed by a set of communicating components. Each component has a syntactic interface for reading (the set of input ports), and writing data (the set of output ports). Communication pathes are described by directed channels that connect ports. A component can be divided into sub-components. Its behavior is then defined by parallel composition of the sub-components' behaviors.
- **Component behavior**: Components that are not decomposed any further must have a behavior specification describing its output reactions on some given input. This specification resembles a general automaton description consisting of a set of data state variables, a set of control states and transitions describing the components reaction on some input by defining the output and the successor states of the data variables and the control state.

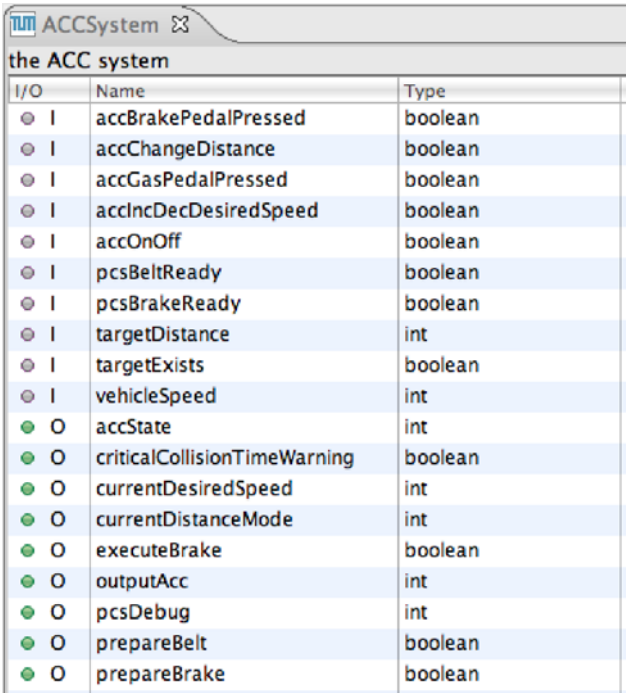


Figure 6: External interface of the ACC system

3.3 Application to the ACC Case Study

In this section, we apply the methodology described before to the adaptive cruise control case study. We will depict the different steps with pictures from AutoFOCUS 3, a tool that is built upon precise semantics and offers the necessary capabilities to build the logical model. Note, that other modeling tools could be used instead. However, a thorough aligment of the tool's capablities to the necessary steps defined by the methodology is needed.

3.3.1 Data definition

For the given case study, we only need *boolean* and *integer* values, so we can skip the data definition part.

3.3.2 Black-box system specification

First, we define the external syntactic interface of the complete system. Based on the service architecture specification we obtain the external interface depicted in Fig. 6. Note, that some signals in from the service architecture have been merged into one signal in the logical model.

The design rationale for the inputs of the logical system model is shown in the following table. State signals are assumed to be always present and of the defined type, while event signals may also be absent, thus not having a value of the given type. Think of event signals implicitly having a special element that represents the emtpy signal message.

Input	State/Event	Rationale
accBrakePedalPressed	State	true, if brake pedal is pressed
accChangeDistance	State	true, if change distance button is pressed
accGasPedalPressed	State	true, if gas pedal is pressed
accIncDecDesiredSpeed	Event	true, increases the desired speed setting; false, decreases the desired speed setting by $ChangeSpeed_{const}$km/h.
accOnOff	State	true, turns the ACC on and off; false, holds current state
pcsBeltReady	State	true, means belt pre-load device is ready
pcsBrakeReady	State	true, means brake system is ready
targetDistance	State	distance to the followed target vehicle
targetExists	State	true, if a vehicle is detected in front
vehicleSpeed	State	the current vehicle speed

Output	State/Event	Rationale
accState	State	the current state of the ACC; Off = 1; On = 2; Suspended = 3
criticalCollisionTimeWarning	State	true, if collision time is critical
currentDesiredSpeed	State	the current desired speed
currentDistanceMode	State	the currently selected distance mode
executeBrake	State	true, if the brake shoud be executed
outputAcc	State	the current acceleration requested by the ACC
pcsDebug	State	debug signal of PCS subsystem; Off = 1; Non-Critical = 2; Critical, preparing = 3; Critical, executing brake = 5
prepareBelt	State	true, if the belt must be prepared for emergency brake
prepareBrake	State	true, if the brake must be prepared for emergency brake

3.3.3 System structure decomposition

After having defined the external interface of the system, we can start to build the system structure by introducing new components. Thus we build a hierarchical system description consisting of components and sub-components.

There is no fixed rule to what extent the hierarchical decomposition should be carried out. It mostly depends on the experience of the engineer doing the design. However, decomposition should lead to a easily understandable model. A good rule of thumb is to give a rationale for every component that is part of the system design. We will show the decomposition in the following and explain our design decisions with such rationales.

Fig. 7 shows the top-level decomposition of the ACC system. We separate the pre-crash system (PCS) from the adaptive cruise control (ACC). We also introduce a channel from the PCS to the ACC in order to allow the PCS to suspend the ACC when collision avoidance or emergency braking is active.

The PCS component is not decomposed further, but has its behavior implemented directly by an automaton. The ACC system is decomposed into a core component, which handles the computation

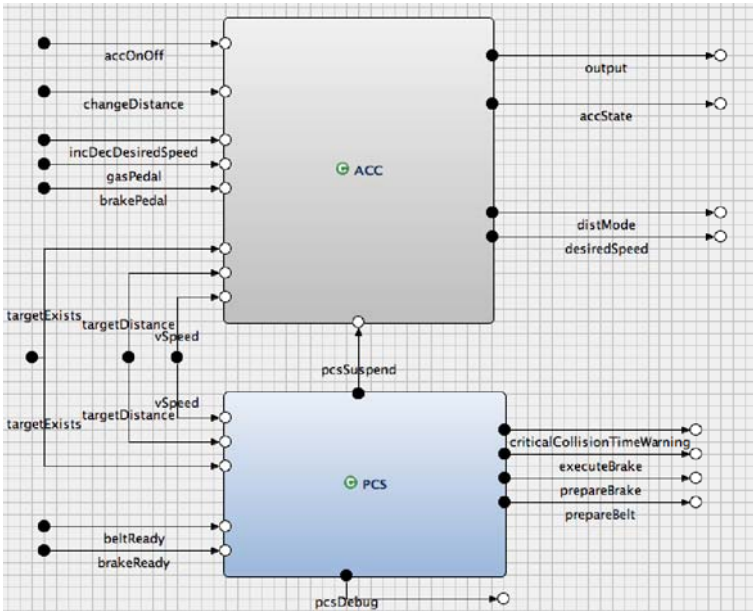


Figure 7: Internal structure of the complete system

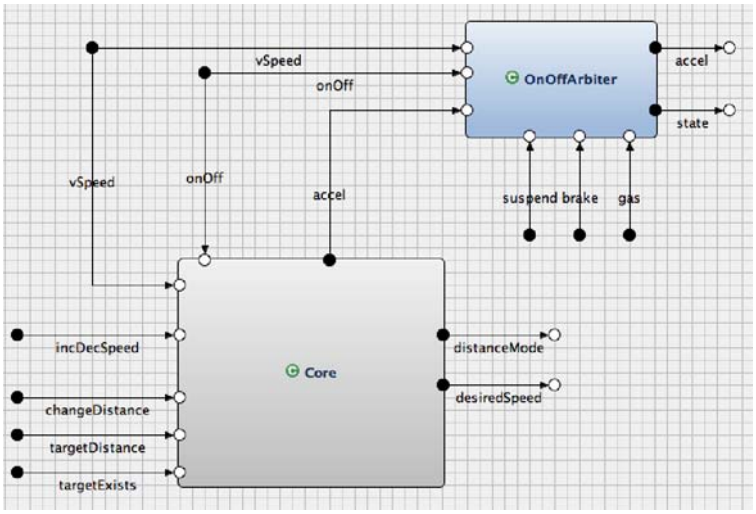


Figure 8: Internal structure of the ACC component

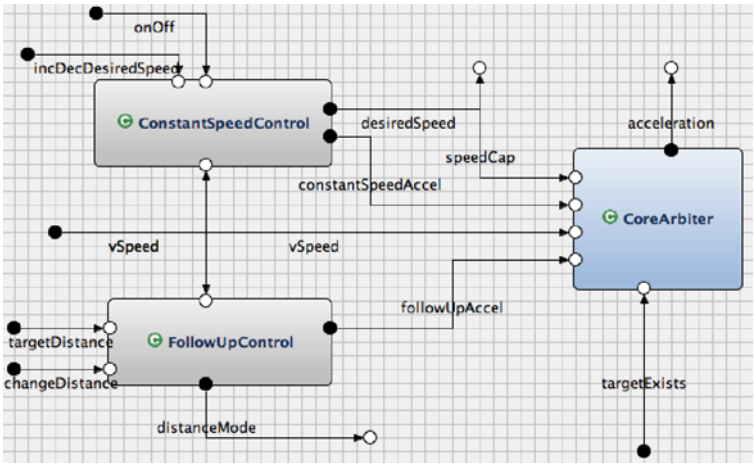


Figure 9: Internal structure of the Core component

modes, and an arbiter component (OnOffArbiter) that forwards the calculation results depending on the current state of the ACC. The design rationale, here, is to separate safety critical OnOffArbiter component from the computation related system parts, e.g. the ACC core component. The arbiter component will be verified using modelchecking techniques (see section 4).

Fig. 9 shows the decomposition of the ACC core component. We have introduced another arbiter that decides which computation result is forwarded depending on the current ACC mode. If the ACC is in follow-up mode, i.e. there is a target vehicle detected, the computation result of the Follow-up Control component is forwarded. Otherwise, if the ACC is in constant speed mode, the computation result of the Constant Speed component is forwarded to become the ACC core component's acceleration value.

The system structure introduced so far, follows a simple layer scheme using the arbiters as filters. Both acceleration computations are carried out in parallel, then the first arbiter decides whether the ACC mode is follow-up or constant-speed. Afterwards the second arbiter (OnOffArbiter) takes care of the enablement state of the whole ACC system. This architecture is not the most efficient one, because computations are carried out although the results might not be needed, i.e. filtered by one of the arbiters. However, we gain a system structure where each component has a defined role.

There are of course more sofisticated system architectures. We might for example send enablement signals from the arbiters to the computation components, to avoid unnecessary computation. However, this increase in efficiency leads to are more complex design, which also reduces understandability of our model.

The ACC system modes *follow-up* and *constant-speed* are implemented by the FollowUp and the ConstantSpeed component, respectively. Fig. 10 shows the ConstantSpeed component. It consists of two sub-components: the DesiredDistance component translates user commands for setting the desired distance mode into a concrete distance (measured in meters) depending on the current vehicle speed. The DistanceControl component compares the desired distance with current distance to the preceding vehicle and computes the acceleration necessary to reach or hold the desired distance.

Fig. 11 shows the internal structure of the constant speed control. The architecture is quite similar to that of the follow-up control. The DesiredSpeed component reacts on user input, like enabling the ACC or increasing or decreasing the desired speed, and forwards the desired speed value to the

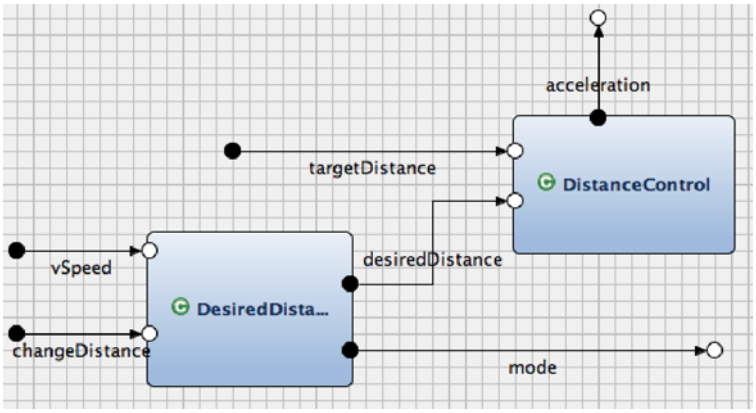


Figure 10: Internal structure of the Followup Control component

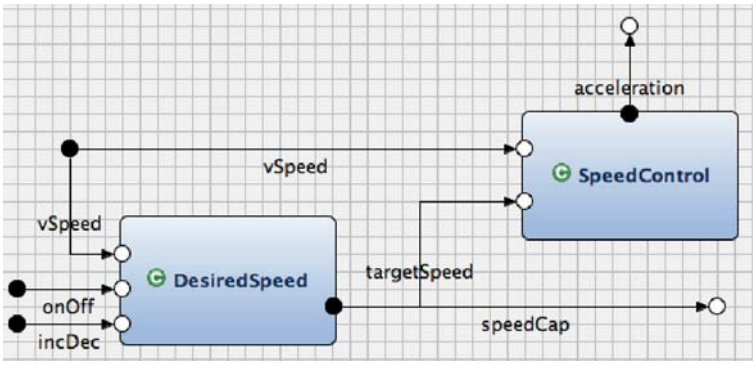


Figure 11: Internal structure of the Constant Speed component

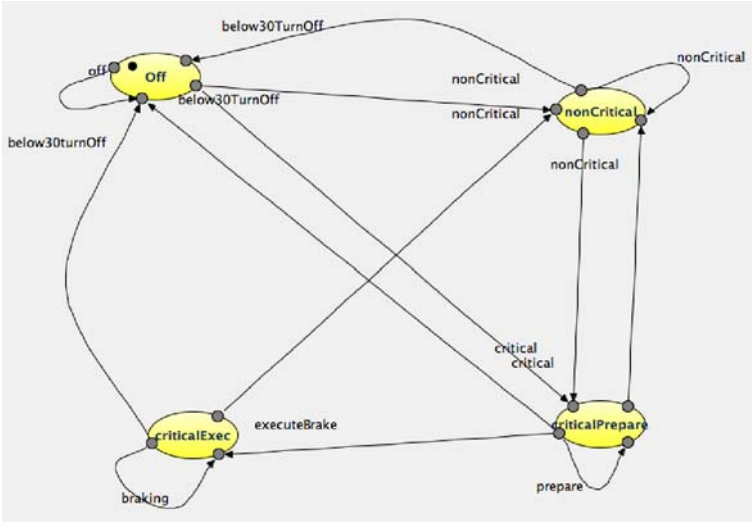


Figure 12: Automaton specification of the PCS component

SpeedControl component. The SpeedControl component does the actual control computation using the current vehicle speed and the desired speed. The output of this component is the acceleration necessary to run the vehicle at the desired speed.

Now, we have seen the complete system architecture of the ACC system. Each component has been assigned a specific task. We can characterize the different components by some informal properties. The PCS component and the afrbiter componenst are the most critical components w.r.t. safety of the ACC system. Therefore, they were designed to be verified in order to ensure their functionality. The DesiredDistance component and the DesiredSpeed component are typical reactive components connected to the user interface. The DistanceControl component and the SpeedControl component are typical control components solving the control task of the ACC in its different modes. These to should be tested thoroughly to ensure that the control carried out, corresponds to the desired properties of the ACC control.

3.3.4 Component behavior specifications

In the following, we describe the behavior of all the atomic component. The behavior automaton of each component consists of a set of data state variables, a set of control states (depicted as yellow ovals), and a set of transitions connecting control states via connector points. Note, that AutoFOCUS 3 basically provides hierarchical structuring of states into sub-states and dividing transitions into transition segments. However, we do not use this feature here.

In the following, we show the state transition diagrams of each atomic component and give a rational for each state and the most important transitions.

Fig. 12 shows the behavior automaton of the pre-crash system component. While the current vehicle speed is below $MinAccSpeed_{const,3}$, the component is in *Off* state: it does not check for imminent crashs. If the vehicle speed is above $MinAccSpeed_{const,3}$and there is either no vehicle in front or it is out of critical range, the component stays in the *nonCritical* state. If the vehicle speed is greater than $MinAccSpeed_{const,3}$and the preceding vehicle is too close, the PCS component prepares the emergency braking (state *criticalPrepare*) by issuing the prepare signals for the brake and the belt, and issuing the critical collision time warning. As soon as both belt and brake are ready the PCS component transits to the *criticalExecute* state and fires the brake.

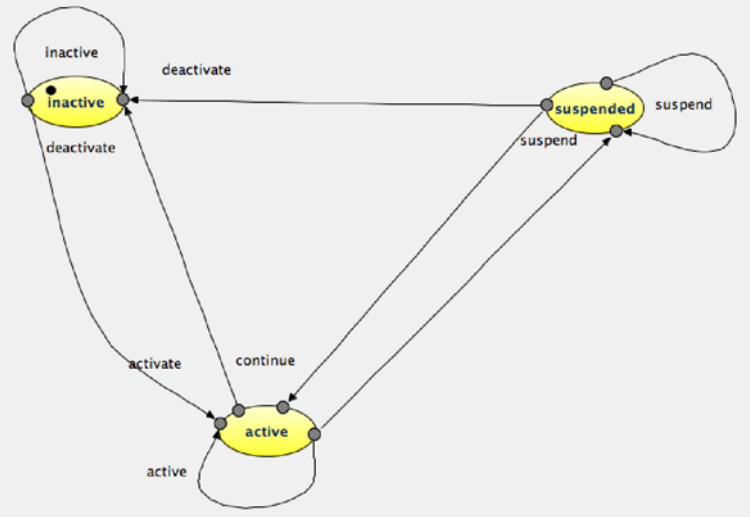


Figure 13: Automaton specification of the On-Off Arbiter component

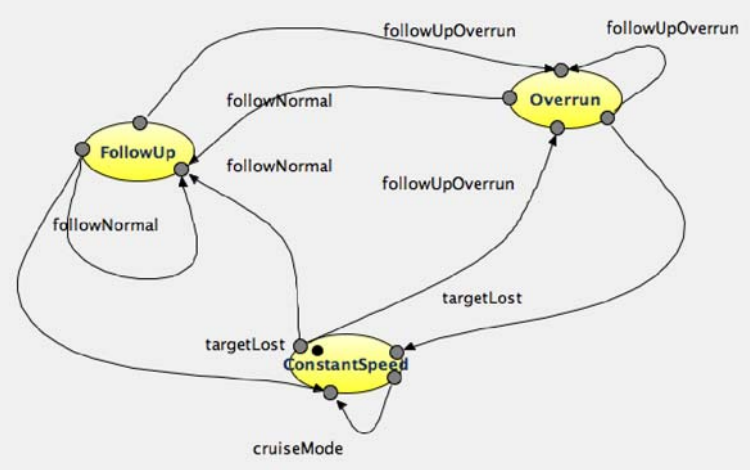


Figure 14: Automaton specification of the Core Arbiter component

Fig. 13 shows the behavior automaton of the on-off-arbiter component. While the vehicle speed is below $MinAccSpeed_{const,1}$or the brake is pressed, the ACC system is in the state *inactive*. While in this state it forwards an acceleration request of 0. Pressing the On button transits the arbiter into the *active* state, if the vehicle speed is greater then $MinAccSpeed_{const,1}$and the brake is not operated. Once the ACC is active, either pressing the gas pedal or if the PCS signals a suspend, the arbiter component transits to the *suspended* state. While suspended an acceleration of 0 is output. If active or suspended the ACC is deactivated if the vehicle speed drops below $MinAccSpeed_{const,2}$, the user turns the ACC off, or if the brake is pressed.

Fig. 14 shows the automaton specification of the CoreArbiter component. Depending on the existence of a preceding vehicle the arbiter forwards the computation results of the constant speed control or the follow-up control, respectively. The *Overrun* state is entered whenever the follow-up acceleration would lead to a vehicle speed greater then the current constant speed limit. Here, the acceleration provided by the follow-up control is simply ignored, i.e. the output is zero. Thus we never drive faster than the constant speed setting permits, although we might loose contact to the preceding vehicle.

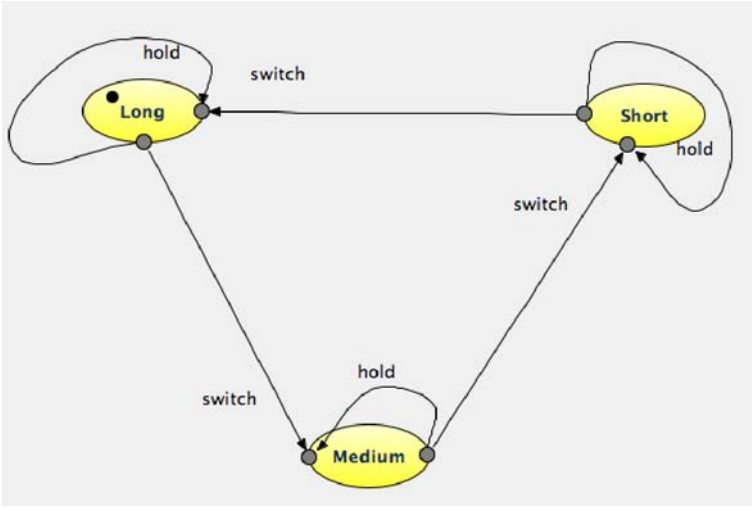


Figure 15: Automaton specification of the Desired Distance component

Fig. 15 shows the automaton specification of the DesiredDistance component. This component has three states, corresponding to the three distance modes that are available. In each of these states the actual desired distance is calculated depending on the current vehicle speed. For the moment, we use three quarter of the current speed in state *Long*, half of the current speed in state *Medium*, and one third in state *Short*. As the transition arrows of the *switch* transitions suggest, the user can cycle through the distance modes from Long over Medium to Short.

Fig. 16 shows the automaton specification of the DistanceControl component. This specification resembles a basic control algorithm. If the current distance to the preceding vehicle is more than our desired distance we accelerate. Furthermore, we decelerate if the current distance is smaller than the desired distance. Note, that we use simple hysteresis of 2 meters for the *TooFar* state and 1 meter for the *TooClose* state[2]. This avoids state jitter for small deviations of desired distance and actual distance signals.

Fig. 17 shows the automaton specification of the DesiredSpeed component. We use a data state variable *_speed* to remember the current speed setting. Whenever the ACC is turned on, we store the current vehicle speed (rounded to a multiple of $ChangeSpeed_{const}$km/h) in this variable. When the user increases the speed setting, this remembered speed is increased in steps of $ChangeSpeed_{const}$km/h, but no further than $MaxAccSpeed_{const}$km/h. Decreasing works accordingly, never decreasing below $MinAccSpeed_{const,1}$km/h. As long as no interaction happens, the *cruise* transition executed, thus outputting the current value of the *_speed* variable.

Fig. 18 shows the automaton specification of the SpeedControl component. This is again a basic control function that outputs an acceleration value depending on the relation of the current desired speed and the actual vehicle speed. Again, we use a hysteresis (here: 2 km/h in both directions) to avoid state jitter.

[2]Note, that these values have been chosen without reference to a real implementation, i.e. these values might be chosen differently for a real system.

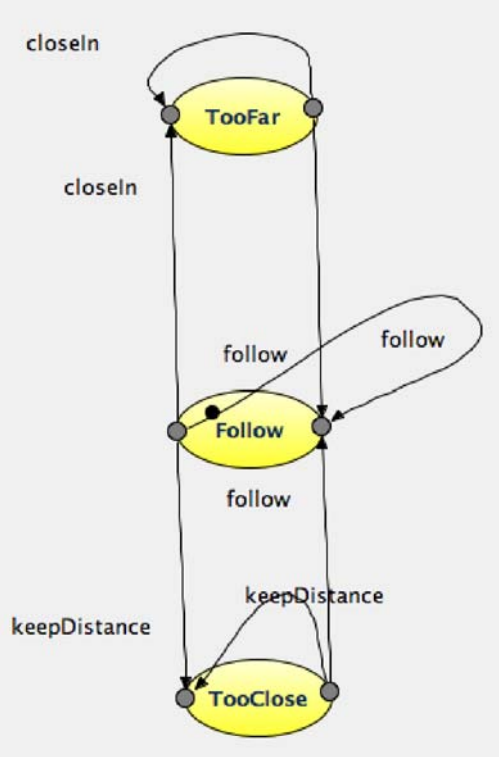


Figure 16: Automaton specification of the Distance Control component

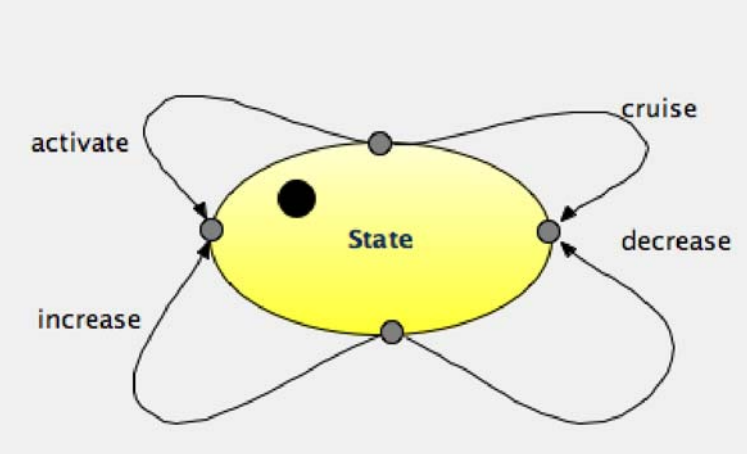


Figure 17: Automaton specification of the Desired Speed component

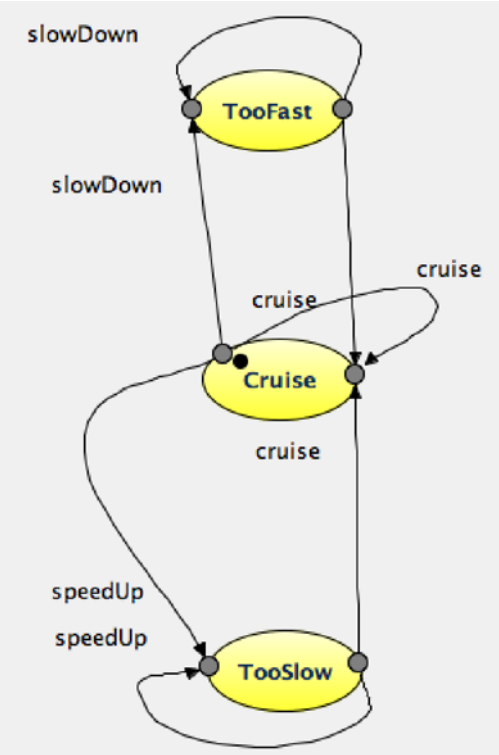


Figure 18: Automaton specification of the Speed Control component

4 Model Checking

4.1 Motivation

Quality assurance is an essential element of software development as its task is to ensure that the developed system actually meets the system requirements. A substantial difficulty is finding out, whether the system requirements specification has been interpreted in the way intended by the specifier – this problem arises mainly because requirements specifications in conventional software development are usually informal, i.e., they are given as text (and possibly diagrams), which have no precisely defined semantics. As a simple example consider a requirement saying *"Component C outputs a number n between 0 and 1"*: here the inaccuracy relates to the word *between*, because the requirements can mean $0 < n < 1$ or $0 \leq n < 1$ or $0 < n \leq 1$ or also $0 \leq n \leq 1$. When describing more complex requirements the lack of precise semantics may cause more misinterpretations and make them more intricate and their detection more difficult. This can be presumed as one of the reasons, why the quality of the implementation and of the quality assurance crucially depends on the expertise and domain knowledge of the participating developers and quality engineers.

The quality assurance in conventional software development is typically done by testing (as well as additional methods like reviews, posing modelling guidelines et al). An essential trait of testing is, that it can show presence of errors but not their absence, as a complete test is generally impossible (especially for arbitrarily long systems executions as they are needed, e.g., for safety requirements demanding that some system state is never reached). A remedy would be using methods allowing for proving certain properties. These methods are usually referred to as formal methods, because they use a formal precise specification to mathematically prove the regarded properties. They thus

require the creation of an unambiguous precise specification and enable a proof of the formulated specification.

4.2 Methodology

In this project we use model checking as formal verification method. The formal verification process comprises following steps:

- Select systems parts to be verified.
- Select requirements for the selected system parts to be verified.
- Formal specification: formalise selected requirements.
- Formal verification: run model checking.
  - If model checking succeeds then verification is finished.
  - If model checking for some requirements fails then analyse the reason (implementation error, requirement formalisation error, not enough resources and/or time for model checking), if possible correct it and repeat model checking.

We explain the verification process with a small example of a single requirement to the ACC mode controlling component OnOffArbiter.

4.2.1 Formal specification

The formalisation of requirements must be done using a formal specification notation. Model checking tools use various temporal logics for this purpose. A widespread notation, which will be used also here, is LTL (Linear Temporal Logic, e.g. [MP92]): it uses Boolean logical operators (Table 15) and temporal operators (Table 16) and allows for formalising properties of system states and their changes during system execution.

We consider the requirement 113 (Table 2.3.2 in Section 2.3.1, p. 27 ff.) stating, that ACC can be started only by the driver.

We begin with the informal textual requirement:

> ACC starts by driver interaction only

The first step is transforming this requirement in a more structured textual requirement:

> **if** acc is inactive **and** driver does not press on/off button
> **then** acc stays inactive

In the next step we map the informal requirements elements onto the formal model:

acc is inactive:	OnOffArbiter.@ = acc_not_active
driver does not press on/off button:	**Not** (Val(OnOff) = True)

The element "acc stays inactive" means, that ACC is inactive in the next step: it will be formalised using a temporal operator.

Now we can formalise the requirement by means of LTL:

$$\begin{aligned}&\textbf{Always } \big((\text{OnOffArbiter.@} = \text{acc\_not\_active } \textbf{And}\\ &\quad \textbf{Not } (\text{Val(onOff)} = \text{True})) \textbf{ Implies}\\ &\quad \textbf{Next } (\text{OnOffArbiter.@} = \text{acc\_not\_active})\big)\end{aligned} \tag{1}$$

The temporal **Always** operator indicates that the formula is an invariant, stating a property that must hold at all points of time in system execution. The Boolean **Implies** operator formulates the implication saying that if ACC is inactive and the driver does not press the on/off button, then in the next step the system is inactive, which is formulated using the temporal **Next** operator.

Operator	**Meaning**
P **And** Q	Both P and Q hold
P **Or** Q	P or Q or both hold
Not P	P is false
P **Implies** Q	if P holds then Q holds
P **Equiv** Q	P holds if and only if Q holds

Table 15: Boolean operators

Operator	**Meaning**
Next P	P holds in the next step
Always P	P holds henceforth
Eventually P	P eventually happens
P **Until** Q	P holds until Q happens

Table 16: Temporal operators

4.2.2 Formal Verification

Formal verification of requirements for a system is done using formal verification tools that check, whether the formalised requirements hold for the system. The choice of the verification technique depends essentially depends on the formal notation applied to formalise the requirements – e.g. efficient SAT solvers can be used for the Boolean logic possessing comparatively weak expressive power, while powerful HOL (Higher Order Logic) requires user interaction in order to check its formulas.

In this project we the model checking technique for temporal logic formulas (cf. Section 4.2.1), ranging between SAT solving and interactive verification with regard to its notational power and verification complexity.

Model checking is an automatic verification technique checking for a given temporal logic formula whether it holds for all possible executions of a given system. The general scheme of model checking procedure is shown in Figure 19:

- The finite state model of the system and the temporal property to be checked constitute the input of the model checker.

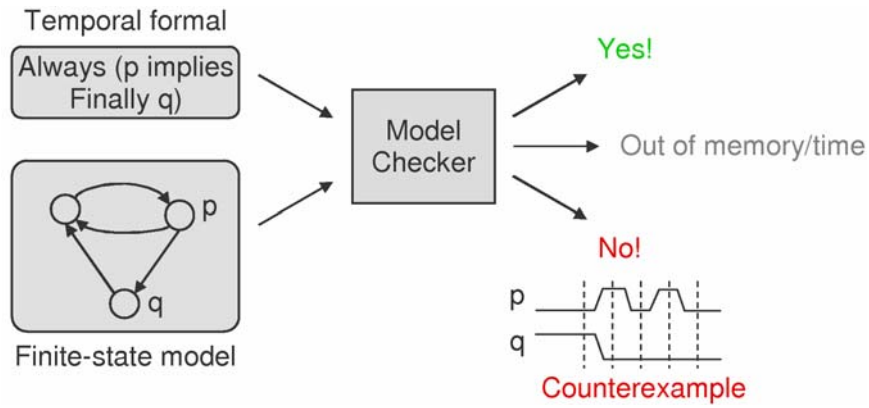


Figure 19: Model Checking – Verification Scheme

- The model checking tools checks whether the formula holds for all possible runs of the system. The possible results are:
  - **Yes**: The formula holds and is thus verified for this system.
  - **No**: The formula does not hold – a counter example is generated, i.e., a system execution trace violating the property. In this case the model (and/or property formulation) is analysed and corrected.
  - **?**: The model checking algorithm runs out of space or time (e.g. due to a preset time-out or because the user manually terminates verification process). In this case the model and/or property formulation is optimised to make verification possible. If completing the model checking cannot be achieved verifying smaller subsystems and/or properties should be taken into consideration.

In the first step of the verification procedure we have to export the system model into the modelling notation of the model checking tool SMV [McM99, McM93] used in this project (the model could also need auxiliary adjustments, e.g., auxiliary variables to capture system states not needed for computation but needed for verification).

The system model is automatically transformed into the SMV notation by the AutoFocus tool when starting the model checking (pressing the "Run" button in the model checking dialog, Figure 21).

The property to be verified must be written using the LTL operator symbols as defined for AutoFocus, e.g. [] for **Always**, () for **Next**, && for **And** et al. For example Figure 20 shows an LTL formula in the AutoFocus notation corresponding to (1).

Then it is likewise automatically transformed into the SMV notation for LTL formulas when starting model checking in the model checking dialog (Figure 21).

Finally by pressing the "Run" button in the model checking dialog (Figure 21) the model and the specified LTL properties are exported into the SMV notation and the SMV model checker is invoked for verifying the properties: Figure 22 shows the SMV model checker after verification of several properties including, e.g., prop_no_start_no_button corresponding to (1).

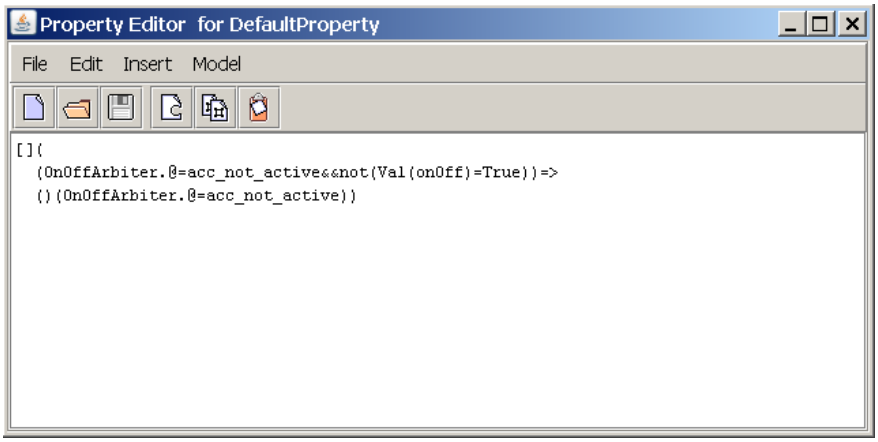


Figure 20: AutoFocus – Property Editor for Model Checking

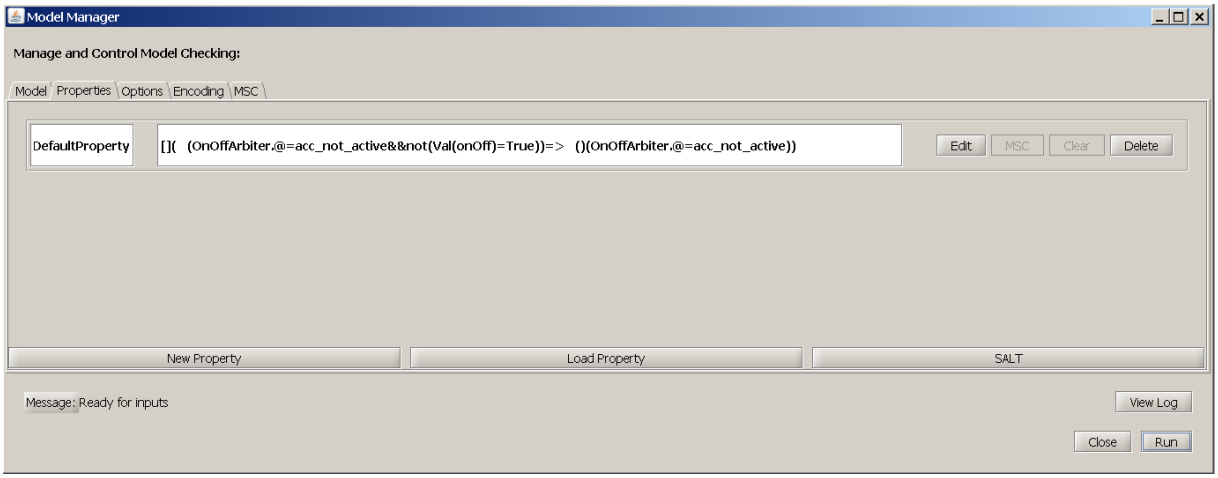


Figure 21: AutoFocus – Model Checking Dialog

4.3 Application to the ACC Case Study

The application of formal verification to the ACC case study comprises formal specification of selected requirements for the components OnOffArbiter and CoreArbiter as LTL properties and their formal verification using the SMV model checking tool.

4.3.1 Formal specification

Formalising requirements for OnOffArbiter.

The component OnOffArbiter (Figure 9 on p. 45) controls the ACC operating state (inactive, suspended, active) and its output with respect to the operating state. Its behaviour is defined by the automaton shown in Figure 23 (cf. also Figure 13 on p. 48)[3].

The component OnOffArbiter has three operating states:

- ACC not active: ACC is deactivated and outputs no acceleration values.

[3]Contrary to Figure 13 in Section 3 Figure 23 shows detailed preconditions and actions for the state transitions.

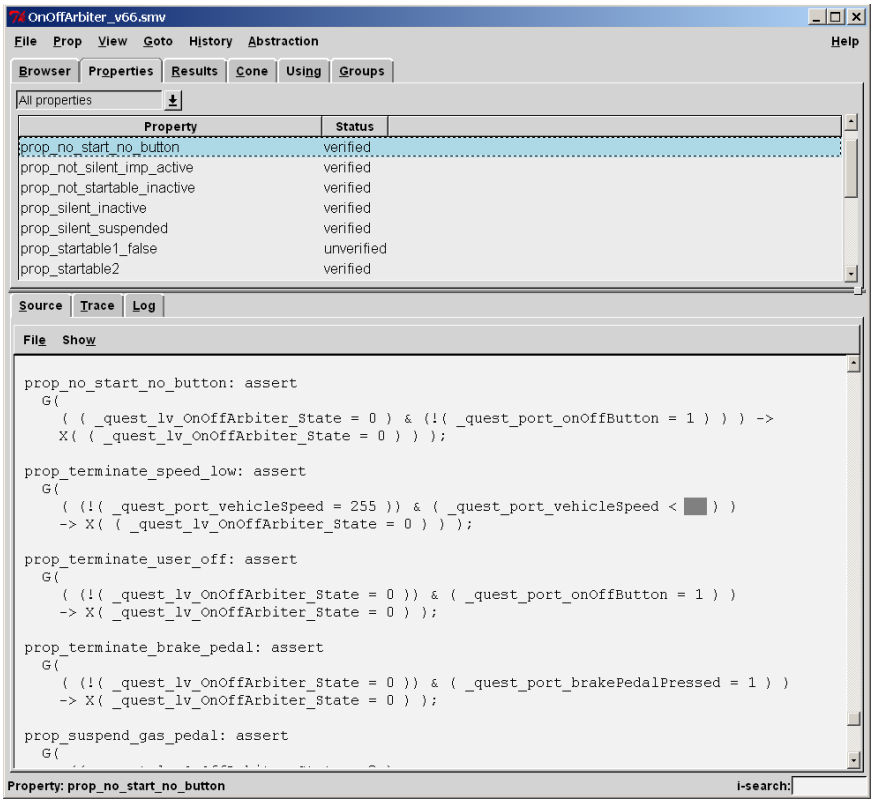


Figure 22: SMV Model Checker

- ACC active: ACC is active and outputs in every step the current ACC acceleration.
- ACC suspended: ACC is suspended, e.g., because the driver operated the gas pedal, and outputs not acceleration values.

These operating states correspond to control states of the automaton describing the behaviour of OnOffArbiter (Figure 23, Figure 13).

The input ports relevant for the properties formulated below, are:

- brake: An empty message (no signal) or the Boolean value False indicate that the brake pedal is not pressed, the value True indicates that the brake pedal is pressed.
- gas: The messages for the gas pedal are analogous to the brake pedal.
- onOff: This signal indicated whether the driver has pressed the on/off button. the value True means that the button has been pressed, the value False means that it has not been pressed.
- suspend: This input port receives suspension signals from the PCS. The the value True corresponds to a suspension signal and the value False to no suspension signal.
- vSpeed: The input port vSpeed received the current vehicle speed.
- accAccel: The input port accAccel receives the currently computed ACC acceleration from the Core component (cf. Figure 8).

The only output port relevant for requirements formulated below, is the following:

- acc_on_accel: This port outputs the ACC acceleration value selected by the OnOffArbiter respectively an empty message if ACC is not active, i.e., inactive or suspended.

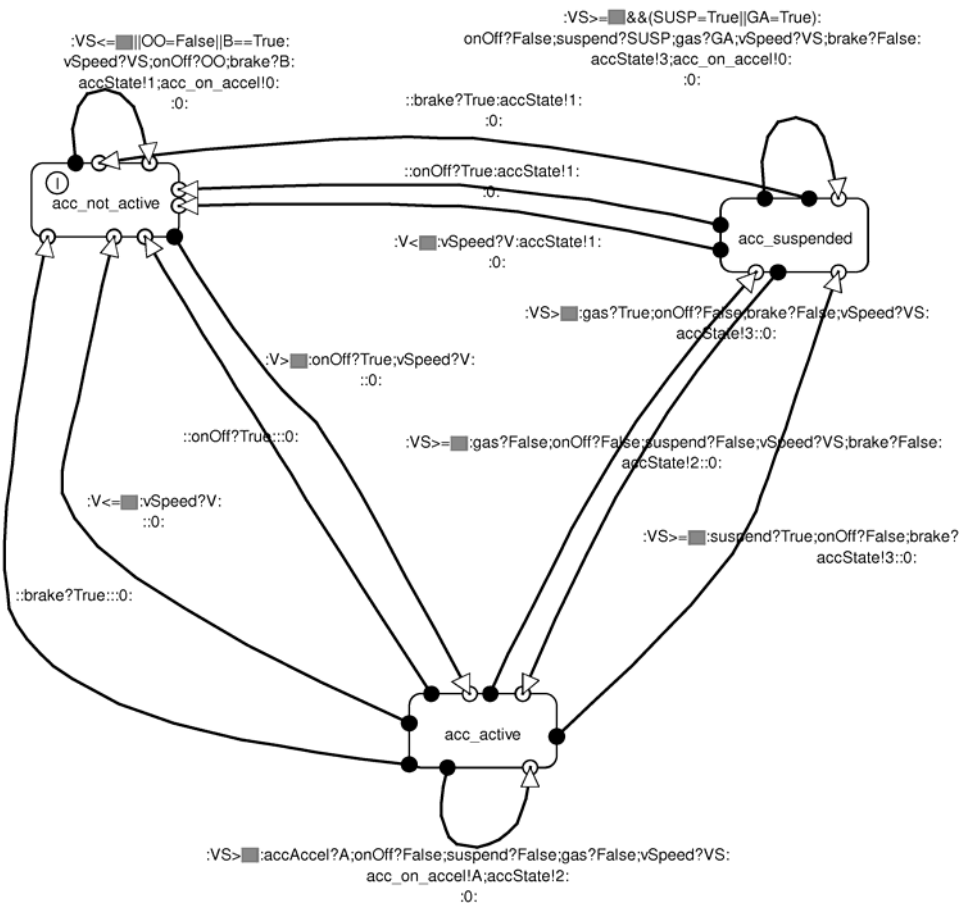


Figure 23: State transition diagram of the component OnOffArbiter

We now formalise selected requirements to the OnOffArbiter component as LTL formulas.

- **111**: ACC startable / not startable.

  The requirement of ACC startability defines, under which precondition the ACC is able to start. However it does not directly define a formalisable action because ACC does not necessarily start when it is startable. Thus we consider this requirement together with requirement 112, which is the only requirement referring to ACC startability (cf. 2.3.1 and Tab. 10 on P. 28 ff.)

  We formalise the startability requirement together with requirement 112 by expressing two requirements to ACC start. The first property says that if the acc is not active, the brake pedal is not pressed, the speed is in the defined range and the user presses the on/off button then ACC starts:

  if acc is inactive **and** brake pedal not pressed
  and speed in defined range **and** driver presses on/off button
  then acc becomes active

  This property can be expressed by the following LTL formula, wherein the formula makes an even stronger statement saying that ACC must start if and only if the speed is in the defined

range:

$$\begin{aligned}&\textbf{Always } \big((\text{OnOffArbiter.@} = \text{acc\_not\_active } \textbf{And}\\ &\quad \textbf{Not } (\text{Val(brake)} = \text{True}) \textbf{ And } \text{Val(onOff)} = \text{True}) \textbf{ Implies}\\ &\quad ((\textbf{Next } (\text{OnOffArbiter.@} = \text{acc\_active})) \textbf{ Equiv}\\ &\quad (\text{Val(vSpeed)} > \mathit{MinAccSpeed}_{const,1} \textbf{ And } \text{Val(vSpeed)} < \mathit{MaxAccSpeed}_{const}))\big)\end{aligned} \tag{2}$$

The second property states that if ACC is inactive and not startable than it remains inactive:

if acc is inactive **and** (driver does not press on/off button **or**
speed below $\mathit{MinAccSpeed}_{const,1}$km/h **or** speed above $\mathit{MaxAccSpeed}_{const}$km/h)
then acc stays inactive

It is formalised by the LTL formula:

$$\begin{aligned}&\textbf{Always } \big((\text{OnOffArbiter.@} = \text{acc\_not\_active } \textbf{And}\\ &\quad (\textbf{Not } (\text{Val(onOff)} = \text{True}) \textbf{ Or}\\ &\quad \text{Val(vSpeed)} \leq \mathit{MinAccSpeed}_{const,1} \textbf{ Or } \text{Val(vSpeed)} \geq \mathit{MaxAccSpeed}_{const})) \textbf{ Implies}\\ &\quad \textbf{Next } (\text{OnOffArbiter.@} = \text{acc\_not\_active})\big)\end{aligned} \tag{3}$$

- **112**: ACC starts by driver interaction

This requirement has been formalised together with the requirement 111 by (2) and (3).

- **113**: ACC starts by driver interaction only

ACC can be started only by the driver:

if acc is inactive **and** driver does not press on/off button
then acc stays inactive

In the structured informal requirement above we see that this requirement is subsumed by (3), though it is not a problem to separately formulate (and model check) it, especially as it is a critical requirement. The corresponding LTL formula is:

$$\begin{aligned}&\textbf{Always } \big((\text{OnOffArbiter.@} = \text{acc\_not\_active } \textbf{And}\\ &\quad \textbf{Not } (\text{Val(onOff)} = \text{True})) \textbf{ Implies}\\ &\quad \textbf{Next } (\text{OnOffArbiter.@} = \text{acc\_not\_active})\big)\end{aligned} \tag{4}$$

- **114**: ACC terminates by low speed.

ACC terminates if the vehicle speed drops below the lower speed bound:

if acc is active **and** vehicle speed is lower than $\mathit{MinAccSpeed}_{const,2}$km/h
then acc becomes inactive

The requirement is expressed by the LTL formula:

$$\begin{aligned}&\textbf{Always } \big((\text{is\_Msg(vSpeed)} \textbf{ And } \text{Val(vSpeed)} < \mathit{MinAccSpeed}_{const,2}) \textbf{ Implies}\\ &\quad \textbf{Next } (\text{OnOffArbiter.@} = \text{acc\_not\_active})\big)\end{aligned} \tag{5}$$

Here the formula is even stronger than the informal requirement, because it drops the pre-condition "ACC is active" and thus requires ACC termination not only in the active state but independently from the ACC state: this means that ACC also must terminate in the suspended state and that it must stay inactive if it was inactive.

- **115**: ACC terminates by driver pressing button.

  ACC terminates the on/off button is pressed while ACC is active:

  if acc is active **and** driver presses on/off button
  then acc becomes inactive

  The corresponding LTL formula is:

  Always (**Not** (OnOffArbiter.@ = acc_not_active) **And**
  Val(onOff) = True) **Implies** (6)
  Next (OnOffArbiter.@ = acc_not_active))

- **116**: ACC terminates by driver brake pedal.

  ACC terminates if the brake pedal is pressed while ACC is active:

  if acc is active **and** driver operates brake pedal **then** acc becomes inactive

  The LTL formula is:

  Always (
  (**Not** (OnOffArbiter.@ = acc_not_active) **And**
  (Val(brake) = True)) **Implies** (7)
  Next (OnOffArbiter.@ = acc_not_active))

  We again strengthen the requirement by using the precondition saying that ACC must terminate if it was not inactive thus adding the suspended state to the precondition. This strengthening ensures that the informal requirement is fulfilled, because the informal precondition "acc is active" can be interpreted as "ACC is in active state" but also "ACC is not inactive, i.e. active or suspended": when using only the precondition in "ACC is in active state" the case could occur that ACC is suspended by pressing gas pedal and would not terminate when brake pedal is pressed. To avoid it, we interpret the informal requirement in a stronger way formalised in (7).

- **117**: ACC break overruled by driver acceleration.

  If ACC is active then it may not brake when the driver presses the gas pedal:

  if acc is active **and** driver operates gas pedal
  then acc may not brake

  We strengthen the requirements by dropping the precondition "acc is active" and generalising "acc may not brake" to "acc may not send acceleration messages", i.e., both acc acceleration and brake is overruled when driver pressed gas pedal[4]. The LTL formula is:

  Always (
  Val(gas) = True **Implies** (8)
  Next (is_NoVal(acc_on_accel) **Or** Val(acc_on_accel) = 0))

[4]The CASE tool version used in this case study did not support exporting models with possibly negative integers as message values for model checking, so that differentiating between positive and negative ACC acceleration was not directly possible – it would have required substantial remodelling in order to separate ACC acceleration output values for positive and negative acceleration. This option was put aside as the technical problem was not due possible limits of the model checking approach but to limits of the used modelling CASE tool prototype.

- **311**: ACC is suspended if PCS brakes.

  ACC is suspended if PCS brakes:

  if acc is active **and** PCS brakes **then** acc becomes suspended

  In the LTL formula we assume that PCS send a suspend signal to ACC when it brakes (otherwise ACC would not find out that PCS brakes). The LTL formula is:

  Always (
  (**Not** (OnOffArbiter.@ = acc_active) **And** Val(suspend) = True **And**
  Val(brake) = False **And** Val(onOff) = False **And** (9)
  is_Msg(vSpeed) **And** Val(vSpeed) > $MinAccSpeed_{const,1}$) **Implies**
  Next (OnOffArbiter.@ = acc_suspended))

  Here we would like to remark that adding a precondition that the vehicle speed must be lower than $MaxAccSpeed_{const}$km/h would not make this property possibly false for a model. Quite on the contrary strengthening the precondition weakens the whole property because it does not need to be verified when the precondition is not fulfilled and a stronger precondition is fulfilled for fewer models than a weaker one. So the property would still be valid for any model it was valid for without an additional precondition.

Formalising requirements for CoreArbiter.

The component CoreArbiter (Figure 9 on p. 45) determines, whether ACC is in follow-up mode or in constant speed mode, selects the corresponding acceleration value and outputs it. Its behaviour is defined by the automaton shown in Figure 24[5].

The component CoreArbiter has two operating states:

- Follow-up mode: ACC is in follow-up mode because a target vehicle is present, i.e., it is following a target vehicle with its speed, as long as its speed does not exceed the constant speed setting
- Constant speed mode: ACC is in constant speed mode because no target vehicle is present, i.e., it is cruising at the desired speed set for constant speed mode.

These operating states correspond to control states of the automaton describing the behaviour of CoreArbiter (Figure 24).

The input ports relevant for the properties formulated below, are:

- target: An empty message (no signal) or the Boolean value False indicate that no target vehicle is present, the value True indicates that a target vehicle is present.
- vSpeed: The input port vSpeed received the current vehicle speed.
- desiredSpeed: The desired speed set for constant speed mode.
- constantAccel: ACC acceleration calculated for constant speed mode.
- followUpAccel: ACC acceleration calculated for follow-up mode.

[5] The automaton on Figure 14 (p. 48) represents a newer version created after the model checking working package had been finished.

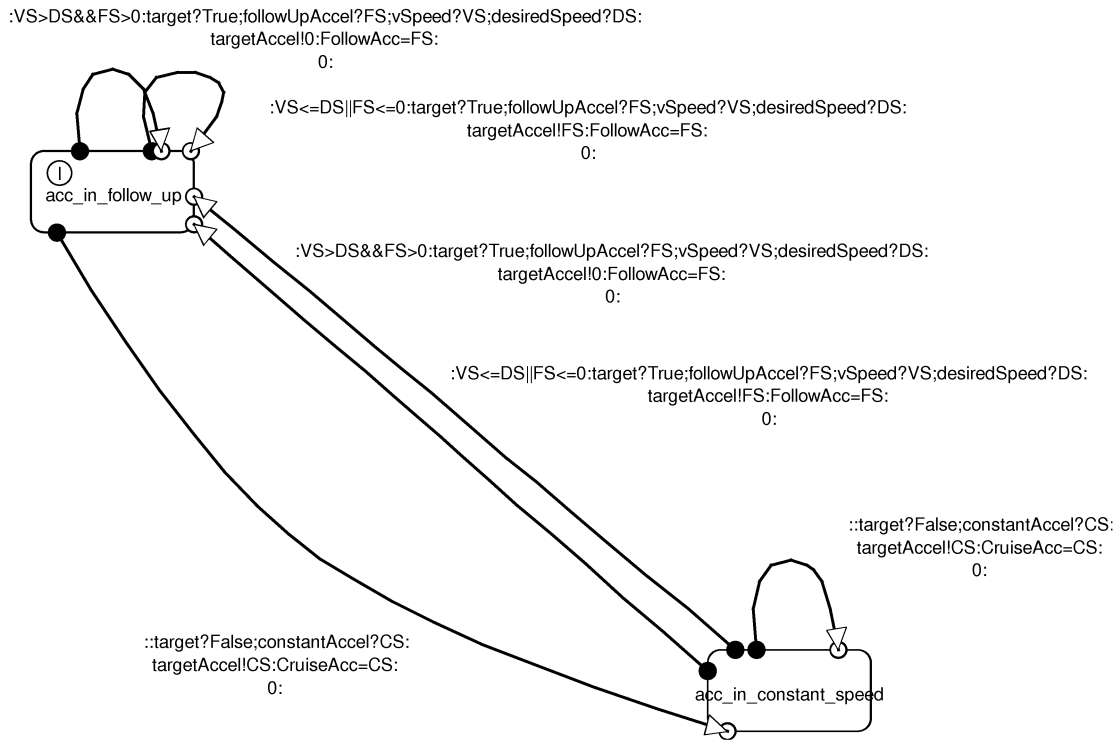


Figure 24: State transition diagram of the component CoreArbiter

The only output port relevant for requirements formulated below, is the following:

- targetAccel: This port outputs the ACC acceleration value selected by the CoreArbiter dependent on the mode – it equals the constantAccel input value if in constant speed mode and followUpAccel input value in the target speed model.

In order to formalise the requirements for this component using LTL we introduce auxiliary local variables, needed only for verification and having no influence on the computation:

- FollowAcc: This variable stores the last input acceleration value for follow-up mode.
- CruiseAcc: This variable stores the last input acceleration value for constant speed mode.

These variables will be used for specifying that the output acceleration value of the CoreArbiter equals on the two input acceleration values.

We now formalise selected requirements to the CoreArbiter component as LTL formulas. The requirements formalised below have the precondition that the ACC is active. Due to the internal design of the ACC component (cf. Figure 8) the check whether ACC is active is performed by the component OnOffArbiter so that the component CoreArbiter does not need to control ACC activity mode and deals only with the choice between follow-up mode and constant speed mode. Thus the precondition that the ACC is active does not need to checked in the following LTL formulas.

- **121**: Follow-up control condition.

  The ACC switches into respectively stays in the follow-up mode.

  if target vehicle present

  then acc switches into resp. stays in the follow-up mode

The LTL formula must additionally check that all input values required by CoreArbiter in follow-up mode are present:

$$\begin{aligned}&\textbf{Always } \big(\text{Val(target)} = \text{True } \textbf{And } \text{is\_Msg(desiredSpeed) } \textbf{And}\\ &\quad \text{is\_Msg(vSpeed) } \textbf{And } \text{is\_Msg(followUpAccel)) } \textbf{Implies}\\ &\quad \textbf{Next } (\text{CoreArbiter.@} = \text{acc\_in\_follow\_up})\big)\end{aligned} \tag{10}$$

- **122**: Constant speed control condition.

The ACC switches into respectively stays in the constant speed mode.

if target vehicle not present
then acc switches into resp. stays in the constant speed mode

The LTL formula additionally checks for presence of input values required in the constant speed mode:

$$\begin{aligned}&\textbf{Always } \big(\text{Val(target)} = \text{False } \textbf{And } \text{is\_Msg(desiredAccel)) } \textbf{Implies}\\ &\quad \textbf{Next } (\text{CoreArbiter.@} = \text{acc\_in\_constant\_speed})\big)\end{aligned} \tag{11}$$

- **131**: Follow-up control behaviour.

If ACC is in follow-up mode then is controls the following distance to target vehicle:

if acc is in follow-up mode
then acc controls following distance to target vehicle

This informal requirement formalised by stating that if CoreArbiter is in follow-up mode and outputs an acceleration value then it must equal the acceleration value received from the component FollowUpControl (Figure 8) responsible for computing the target acceleration in follow-up mode.

$$\begin{aligned}&\textbf{Always } \big(\text{CoreArbiter.@} = \text{acc\_in\_follow\_up } \textbf{And } \text{is\_Msg(targetAccel)) } \textbf{Implies}\\ &\quad \textbf{Next } (\text{Val(target)} = \text{FollowAcc } \textbf{Or } \text{Val(target)} = 0)\big)\end{aligned} \tag{12}$$

The target acceleration is 0 when the current vehicle speed exceeds the desired speed setting for constant speed so that no further acceleration is allowed.

- **141**: Constant speed control behaviour.

If ACC is in constant speed mode then is controlls the vehicle speed according to desired speed setting:

if acc is in constant speed mode
then acc controls vehicle speed according to desired speed setting

Analogously to the requirement (12) above, this requirement is formalised by stating that if CoreArbiter is in constant speed mode and outputs an acceleration value then it must equal the acceleration value received from the component ConstantSpeedControl (Figure 8) responsible for computing the target acceleration in constant speed mode.

$$\begin{aligned}&\textbf{Always } \big(\text{CoreArbiter.@} = \text{acc\_in\_constant\_speed } \textbf{And } \text{is\_Msg(targetAccel)) } \textbf{Implies}\\ &\quad \textbf{Next } (\text{Val(target)} = \text{CruiseAcc})\big)\end{aligned} \tag{13}$$

4.3.2 Formal verification

Formal verification of the requirements described in Section 4.3.2 has been performed using the SMV model checker.

Model checking requirements for OnOffArbiter.

Model checking the requirements 111–117, 311 for the component OnOffArbiter yielded following results:

- All requirements except of 111, 116 have been verified.
- The requirement 111 (cf. (2)) has been falsified. Analysing the cause revealed an inconsistency between the requirement specification and implementation in the model: while the requirement specified an admissible vehicle speed range $MinAccSpeed_{const,1}$ km/h $<$ vSpeed $<$ $MaxAccSpeed_{const}$ km/h the model used the condition $MinAccSpeed_{const,1} <$ vSpeed. Here the decision must be taken what specification is desired and then either the LTL property or the model must be corrected according to it.
- The requirement 116 (cf. (7)) has been falsified because the model implementation the transition from state acc_active to itself did not check whether brake pedal had been pressed (Figure 23). After adding the precondition brake?False to this transition the property has been verified.

Model checking requirements for CoreArbiter.

For the component CoreArbiter model checking the requirements 111, 121, 131, 141 verified all requirements (as well as some additional formulations). Figure 25 shows the SMV Model Checker window after the model checking process for CoreArbiter has been finished.

5 Model-based Test-Case generation

Model-based testing was part of the quality assurance activities applied during the development of the active cruise control and pre crash control system (abbr. ACC-PCC-System).

In Section 5.1, we will motivate the application of model-based testing in general and in the context of the present case study. In Section 5.2, we will describe our model-based testing methodology in general. Subsequently, in Section 5.3, we discuss its application in the context of the ACC case-study. There we describe the involved activities and artefacts in detail. Finally in Section 5.4 we summarize the chapter.

5.1 Motivation

Model-based testing is a quality assurance activity applied in the ACC case-study in addition to formal verification methods. In section 4 we applied model checking for the formal verification of specified properties. These properties reflected the requirements 111–117 and 311. Since is it a laborious task to define formal properties, formal verification will in practice only be done for the most important requirements. Furthermore, usually requirements documents do not specify the expected system behavior for every possible input sequence—it is the designers' task to implicitly

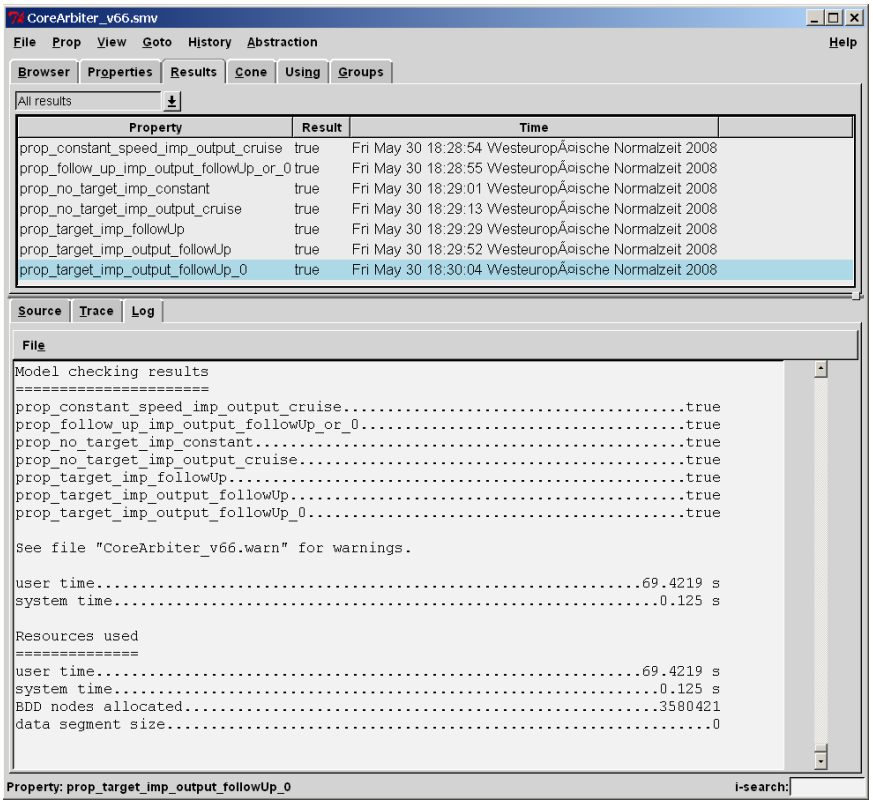


Figure 25: SMV Model Checker – results for CoreArbiter

add knowledge to the system model to finally get a complete (total regarding all input sequences) behavior specification. To sumarize this, following observations can be made in general:

- Requirements documents formally do not specify the intended behavior totally
- For requirements which are not explicitly stated in the document, a suitable verification property hardly will be defined
- Defining verification properties is too laborious if it would be done for every requirement

The result of these observations is that even by applying formal verification testing in in most cases still necessary. To reduce the cost of testing it should be automatically done in large parts. Model-based test generation is a way to automatically derive test cases from a test model. In general it yields large benefits in comparison to testing based on manually specified test cases. Some of the benefits are the following:

- Empirical studies like the one, which was published in [PPW+05], have shown that test cases which are generated automatically, detect errors which are not found by manually specified test cases. The use of models in testing especially pays off, when detecting failures in requirements.
- Automated test case generation has the potential to be turned into a push-button technology.
- Behavior in presence of failure, e.g. a sensor, or communication failure can explicitly be modeled in a test model.
- When regression testing is necessary, in consequence of a change in requirements, individual test cases do not need to be reviewed in order to decide, whether they need to be adapted or discarded. The complete test model is adapted and new test cases are generated.

5.2 Methodology

This section provides a generic description of our model-based testing methodology. It explains how model-based testing is understood in this context, it sketches how AUTOFOCUS models can be used as test models and how test cases can be generated from such models. The content was mostly adapted from [PPW+05] for more detailed information see [Pre03].

5.2.1 Basics

The general idea of model-based testing (of deterministic systems) is as follows: An explicit behavior model (test model) encodes the intended behavior of an implementation called system under test, or SUT. Modeling languages include statecharts, Petri nets, the UML RT, or ordinary code. Traces of the model are selected, and these traces constitute test cases for the SUT: input and expected output.

The *test model* must be more abstract than the SUT. In general, abstraction can be achieved in two different ways: (1) by means of encapsulation: macro-like structures as found in compilers, library calls, the MDA, or J2EE, or (2) by deliberately omitting details and losing information such as timing behavior. Now, if the test model was not more abstract than the SUT in the second sense, then the efforts of validating the test model would exactly match the efforts of validating the SUT. (We use the term validation when an artefact is compared to often implicit, informal requirements.) While the use of abstraction in model-based testing appears methodically indispensable, and, for the sake of intellectual mastery, desirable, it incurs a cost: details that are not encoded in the model obviously cannot be tested on the grounds of this model. In addition, it entails the obligation of bridging the different levels of abstraction between test model and SUT: input to the test model, as given by a test case, is concretized before it is fed to the SUT. The output of the latter is abstracted before it is compared to the output of the test model as defined by the test case. The hope is that one can split the inherent complexity of a system into an abstract model, and *test driver* components that concretize and abstract in and outputs respectively. The granularity of the comparison between the system's and the model's output depends on the desired precision of the test process: as an extreme case, each output can be abstracted into whether or not an exception was thrown. In some situations, this may be meaningful enough to initiate further actions.

In most cases, one needs selection criteria on the set of all traces of the model. We call them *test case specifications*. These are intentional: rather than specifying each test case on its own, one specifies characteristics and has some manual or automatic generator derived test cases that exhibit the characteristics. Examples include coverage criteria like state, branch or MC/DC coverage, probability distributions, or the definition of a state of the model one considers interesting. They can also be given by functional requirements in the form of restricted environment models that make sure the model of the SUT can only perform certain steps. This also includes fault models. In this sense, test case specifications can be structural, stochastic, or functional.

5.2.2 Using AUTOFOCUS for test model specification

For the case study, we use the CASE tool AUTOFOCUS 2 [Aut] for modeling the test model[6] for the ACC-PCC system. We only give a brief overview here, in order to keep the chapter self contained, for more details refer to Section 3.

[6] We didn't use AutoFOCUS 3 like in section 3 since the test generation tool is only available for AutoFOCUS 2 in the moment.

The core items of AUTOFOCUS specifications are *components*. A component is an independent computational unit that communicates with its environment via so called ports. Ports are typed. Two or more components can be linked by connecting their ports with directed channels. This way, component networks are described by system structure diagrams (SSDs). As an example, Figure 26 shows the structure of the ACC-PCC system. It is explained in more detail in Section 5.3.1. SSDs are hierarchical, so every component can be described recursively as a set of communicating subcomponents.

Atomic components are components which are not further refined. For these components a behavior must be defined. This is achieved by means of extended finite state machines (EFSMs). Figure 27, depicts the EFSM of a component of the ACC and pre crash control system, it will also be explained in more detail in Section 5.3. An EFSM consists of a set of control states (bubbles), transitions (arrows), and is associated with local variables. The local variables form the component's data state. Each transition is defined by its source and destination control states, a guard with conditions on input and the current data state, as well as an assignment for local variables and output ports. Transitions can fire if the condition on the current data state holds and the actual input matches the input conditions. Assignments modify local variables. After execution of the transition, the local variables are set accordingly, and the output ports are bound to the values computed in the output statements. These values are then copied to the input ports that are connected by channels. Guards and assignments are defined in a Gofer-like functional language that allows for the definition of possibly recursive data types and functions.

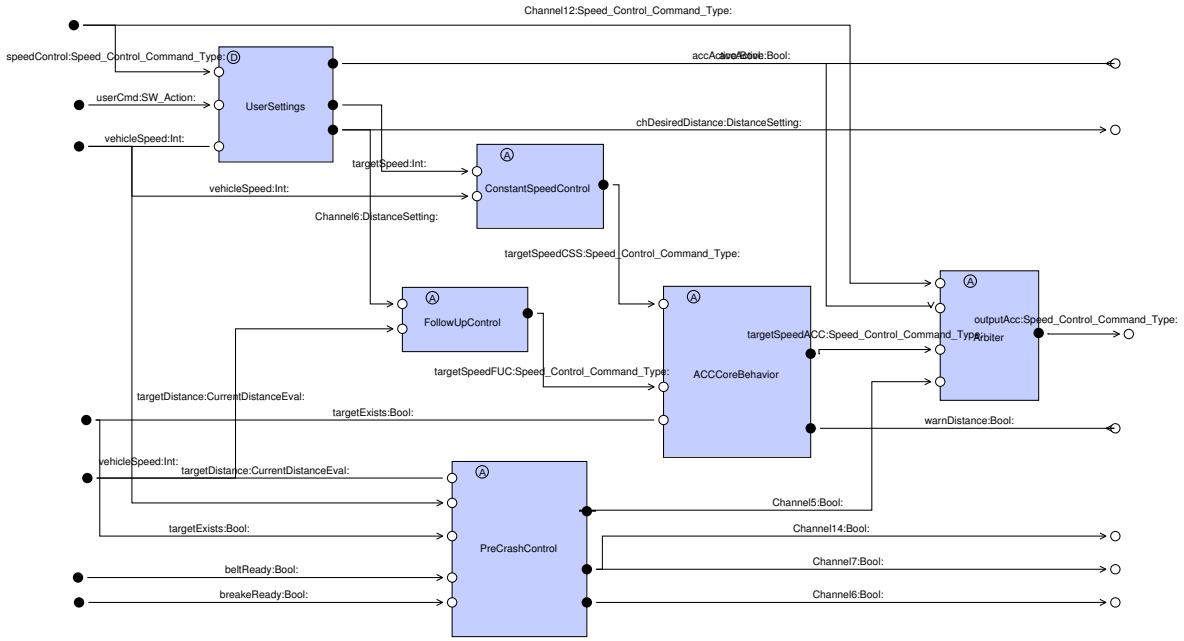


Figure 26: System structure diagram (SSD) of the ACC-PCC system. Boxes denote subcomponents, e.g. 'UserSettings' or 'ConstantSpeedControl'. Non-filled circles on subcomponent borders represent their input ports, filled circles represent their output ports and lines between ports represent channels, which are assigned a name and their type. Ports which are not associated with a subcomponent are input and output ports of the top level ACC-PCC system.

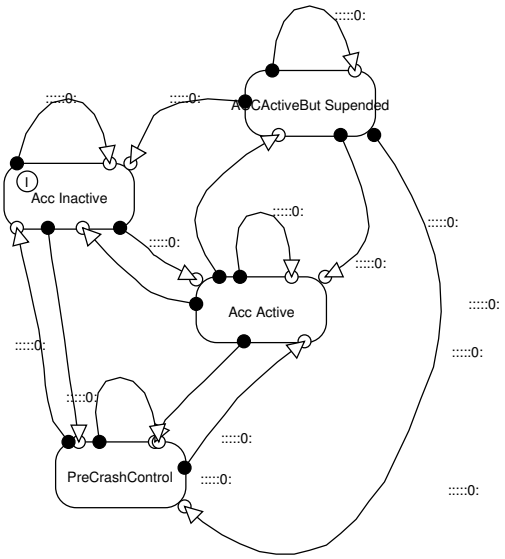


Figure 27: Behavioral specification of the Arbiter Component in form of an extended finite state machine (EFSM). Bubbles represent control states. Initial states are labeled with an 'I' (in this example, 'Acc Inactive' is the initial state), arrows between components represent transitions. Transitions are usually labeled with their guards and assignments to local variables and output states. As in this model guards are very long, and there make the diagram rather difficult to read, we replaced them by '::::0:'. For their exact specification refer to the model.

5.2.3 Test case generation

Test case generation is done using the AUTOFOCUS 2 Test Case Generator [Pfa08]. The test generator provides different techniques. Within the DENTUM project we applied two techniques or types of test case specifications respectively:

- Randomly, equally distributed test case generation
- Randomized test case generation using a usage profile

For both techniques a sequence of inputs is randomly, or according to the usage profile respectively, generated. The expected outputs are calculated automatically by the AutoFOCUS simulation engine. This ensures that the underlying semantics of the test cases is equal to the semantics which is shown by sumulating the model.

When a usage profile was used, the expected usage of the ACC-PCC was stated as well. This was done by stating a probability distribution over the input values and the likelihood of change for every input. Wheres the (simple) random test case generation does not distinguish different probability distributions for different input ports, the usage profile based test case generation allows to state a probability distribution for every port. The main benefits of applying usage profile based test case generation are:

- Test cases ressemble the expected behavior of the users
- Since usage profiles and random generation are combined exceptional cases are tested es well, not only standard cases.

- Test case generation can be controlled and applied to the testers' needs

As a further method AutoFOCUS offers test case generation by translating the test model into a Constraint Logic Programming (CLP) language, and adding the test case specification – a full-fledged environment model or sets of constraints. Execution of this CLP program then successively enumerates all traces of the model (and 'guesses' all possible input values). In fact, the model is executed symbolically: rather than enumerating single traces-input, output, local data of all components- of the model, we work with sets of values in each step instead. States are not visited multiple times which is why in each step, the currently visited set of states is only taken into consideration if it is not a specialization of a previously visited set of states. We omit details of the translation and state storage here and refer to earlier work [PPS+03].

Even with test case specifications and state storage, the number of computed test cases that satisfy a test case specification may be too large. In this case, one has to add further constraints, i.e., test case specifications, or pick some tests at random.

5.2.4 Process

In this section, we describe the process context of model-based testing as well as activities and artefacts involved. The model-based testing process needs the following four artefacts as input:

Input:

- Service graph, refer to Section 2 for detailed description
- Service architecture, refer to Section 2 for detailed description
- Test specification, which specifies the testing goals, e.g., model-based coverage criteria, which should be achieved, for example, state coverage, which demands, that any state within the test model is reached in at least one test case or the definition of a state of the test model one considers interesting
- System implementation, the system that is to be tested

The output of the process are test results, like a list of identified faults or an estimation of the current quality level of the implementation.

Output:

- Test results, i.e. a list of identified faults

Figure 28 gives and overview on the activities and artefacts involved in model-based testing. In the following we describe the activities of the process briefly. In Section 5.3, we will discuss the activities and resulting artefacts for the ACC case study in detail.

1. **Build and validate test model**
   The test engineer builds a testing model according to the specification, i.e., the service graph and the service architecture, which are results of a preceding development phase. While doing so, he decides on abstractions, which meet the testing goals best. Possible types of abstractions are:
   - *Functional abstraction*: Functionality, which is not within the scope of the testing goals is omitted

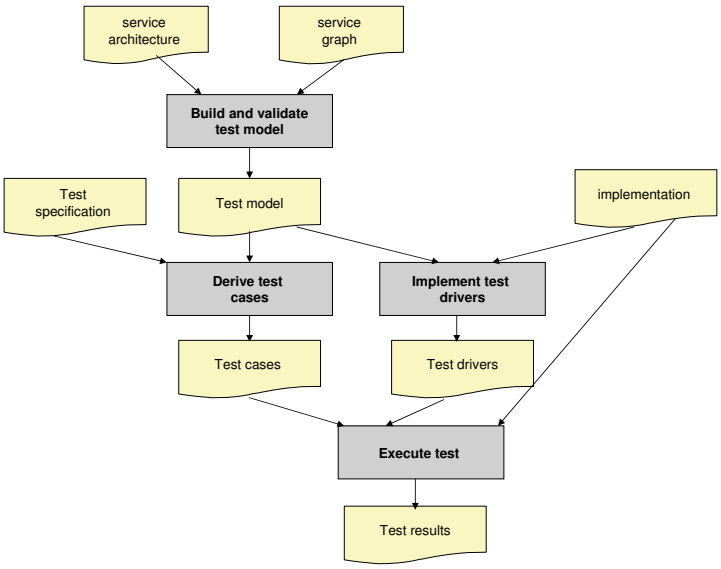


Figure 28: Overview on the model-based testing process. 'Service architecture' and 'service graph' are, for example, artefacts necessary for the activity 'Build and validate test model'. The artefact 'test model' is then used as input by the activities 'Derive test cases' and 'Implement test drivers'.

- *Data abstraction*: data complexity is reduced, e.g., by narrowing the set of signals to those which are relevant for the scope of testing and by building equivalence classes for certain input and or output signals
- *Communication abstraction*: For example, merge consecutive signals which belong to the same transaction
- *Temporal abstraction*: For example, discretize continuous time to time intervals of fixed length
- *Structural abstraction*: The test model may have less subcomponents than the implementation.

Subsequently, the test model is validated, i.e., it is inspected or verified, whether or not the behavior of the test model is conform to the specification. If it is not, it is adapted respectively.

2. **Derive test cases**
   Test cases are derived from the test model and according to the test case specification, which specifies for example coverage criteria like state or branch coverage. Test cases are traces of inputs to the implementation and expected outputs.

3. **Implement test drivers**
   In order to execute test cases and as a consequence of the abstractions introduced in the test model, test drivers need to be developed, which concretize the inputs specified in test cases to a form, that is an acceptable input to the implementation and which interpret the more concrete outputs of the SUT, to make them comparable to the expected outputs specified in the test cases.

4. **Execute test**
   The implementation is tested using the test cases and test drivers. Results are evaluated.

5.3 Application to the ACC Case Study

In our case study, we built a test model for the ACC-PCC system. The model was used to derive a set of test traces which were applied to the implementation of the system using specific test drivers, which bridge the abstraction level between test traces and the in- and outputs accepted and produced by the implementation.

In this section, we describe activities and artefacts involved in the testing process in the case study in detail.

5.3.1 Building and validating test model

The test model for the ACC case study was built in AUTOFOCUS 2. The service graph and the service architecture as described in Section 2 were used as input. The test model was validated manually using the AUTOFOCUS simulator. While performing a stepwise or continuous simulation, input parameters to the system can be provided manually and the state of atomic components as well as values on channels can be observed.

Description of the test model. Figure 26 shows the top level of the test model for the ACC-PCC system. The test model has six input ports: the current speed, the user's input to the ACC, the users command to the drive-train (the user's wish to keep speed, accelerate or decelerate), an indicator for the presence of a preceding vehicle, the evaluation of the distance to a preceding vehicle, and two input ports which signal the states of the brake and seat belt in case of an emergency. The model has five output ports: The resulting command to the drive-train, two ports which signal warnings to the user, either because the pre crash control is active and will shortly initiate an emergency stop, or because the ACC needs to brake hard in order to restore the minimal distance to a preceding vehicle and finally a port which is used to signal the brake and seat belt to get ready for an emergency stop, if the distance of objects in front is critically short. The test model has six subcomponents, which communicate via channels:

- **'UserSettings':** This subcomponent computes the user input and provides information about whether or not the ACC is active and in case it is active, what target speed and the distance preference settings are.
- **'ConstantSpeedControl':** This subcomponent computes a control signal to the vehicles drive-train, in case the ACC is active and assuming that there is *no* obstructing vehicle in front. The control signal denotes whether the vehicle is to keep the speed, to brake or to accelerate.
- **'FollowUpControl':** This subcomponent computes a control signal to the vehicles drive-train, in case ACC is active and assuming that there is an obstructing vehicle in front. It warns the user in case it has to brake hard in order to restore the desired distance to the preceding vehicle.
- **'PC' (Pre crash control):** This subcomponent observes the distance to objects in front. If this distance gets critically short it warns the driver and signals to the seat belt and brake to get ready for an emergency stop. It then operates the brake.

- **ACCCoreBehavior:** Decides, whether constant speed control or follow up control determine the control signal to the vehicle's drive-train if ACC is active.
- **'Arbiter':** Decides on priorities between the control signal to the vehicle's drive-train from the user, from the pre crash control and the ACC.

The component 'UserSettings', is substructured in three subcomponents. Figure 29 shows its internal structure. 'ActivateACC' is responsible for the activation of the ACC. It controls all constraints on the activation of the ACC, i.e., it checks, if speed is not too slow or too fast, it awaits the activation demand by the user before it activates the ACC. Finally it deactivates the ACC on the occurrence of a deactivation command or in case the user operates the brake. 'DistanceSettings' stores the distance settings, according to the user's adjustments as long as the ACC is active. Distance settings are toggled between three states, short distance, middle distance, and long distance. 'SpeedSettings' stores the desired target speed and adjusts it, if it is incremented or decremented by the user, as long as ACC is active.

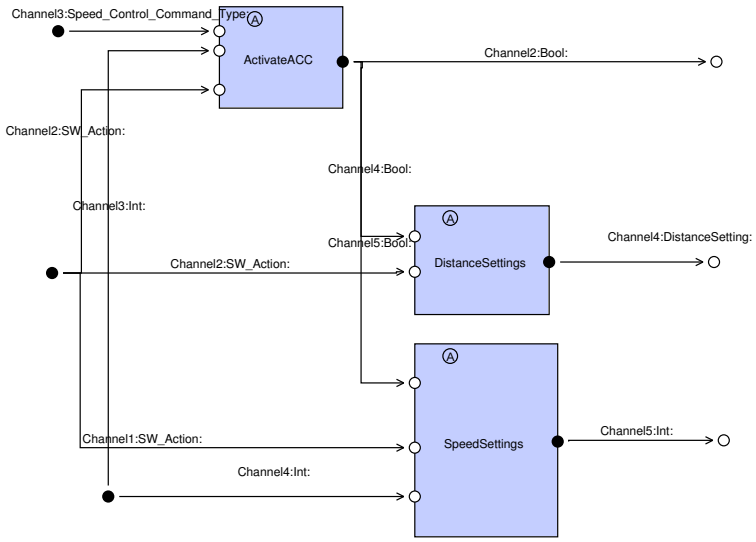


Figure 29: Internal structure of the ACC-PCC system subcomponent 'UserSettings'. The component has three input ports. The user input to the ACC, the user's command to the vehicles drive-train in order to detect, whether the user is operating the brake and the current vehicles speed in order to check for restrictions, when activating the ACC and in order to set the initial target speed, when the ACC is activated. Three subcomponents compute the input. 'ActivateACC' decieds, if the ACC is active, 'DistanceSettings' stores distance preferences, while the ACC is active, and 'SpeedSettings' stores the desired target speed settings. The 'UserSettings' component has three output ports. One indicates wether the ACC is active and the two others signal target speed and distance preference settings.

Apart from the 'UserSettings' component, all mentioned subcomponents are atomic. In consequence, EFSM specified, which describes the component's behavior. Figure 27 shows the EFSM of the 'Arbiter' component an an example. The 'Arbiter' component has four states: 'ACC Inactive', in which the command to the vehicle's drive-train is determined by the driver's input only. 'ACC Active' in which the command to the vehicle's drive-train is determined by the input from the ACC. In case the driver wishes to accelerate and indicates this by pressing the gas pedal, the ACC is suspended and the command to the drive-train is determined by the user input. Then the arbiter

is in state 'ACCActiveBut Suspended'. In the 'PreCrashControl' state, the pre crash control takes over and determines the command to the drive-train. We omit the details of the transition guards and assignments to output ports and local variables. For the exact specification refer to the model itself.

Abstractions in the test model. One aspect which is essential to discuss at this point, are the abstractions applied to the test model. Those abstractions essentially differentiate the test model from the model used for code generation in this case study. All five general abstraction principles mentioned above were applied to this test model:

1. In terms of *data abstraction*, (1) we reduced data complexity in the model by building equivalence classes for the distance to preceding vehicles. In the implementation, the distance is given in a metric scale and it needs to be evaluated by the ACC-PCC system. In contrast, the test model receives an evaluation of this distance relative to the vehicle's current speed. The evaluated input can have six different values: critically short, too short, ok for short distance settings, ok for middle distance settings, ok for long distance settings, too long and infinite.
   (2) In addition, we only differentiate five different drive-train command types: Brake strongly, brake gently, keep speed, accelerate gently and accelerate strongly. In the implementation, more fine grained commands need to be given to the drive-train.
2. In consequence of the data abstraction, the model is also a *functional abstraction* of the implementation: (1) The test model does not evaluate the distance, but expects the test driver to provide already evaluated distances.
   (2) The test model does not calculate the exact output drive-train command, but leaves the evaluation of the vehicle's reaction to certain inputs to the test drivers. The test drivers are responsible, to evaluate the vehicle's reaction to the command, e.g. whether it accelerates correctly, and typical controller characteristics, like latency or whether speed is held at a sufficiently stable.
3. In terms of *communication abstraction*, we merged the signals from pre crash control to brake and seat belt, which were specified in the service architecture as two individual signals into a single signal, as they concern the same transaction and only occur together.
4. In terms of *temporal abstraction*, we abstracted from physical time. Only the order of events is relevant to the test cases, not their exact timing. This is because there were no timing requirements specified. The model could however be extended.
5. Finally, in terms of *structural abstraction*, the subcomponents of the ACC-PCC system do not correspond to subcomponents in the implementation.

5.3.2 Deriving test cases

Finally we did not only generate test cases from the test model of the ACC-PCC system, but also used the system model for test case generation. The actual test cases to test the implementation were derived from the test model. We will describe this process first.

Test cases from the Test Model Test cases were derived from the test model using two different test specifications:

1. randomly generated, with equal probability distribution of values
2. defining an usage profile to control random test case generation

Figure 30 shows an example test case. The test case consists of ten consecutive steps, in which values for input parameters, as well as expected outputs are specified. In Step 0, 'currentSpeed' is set to 60 kmH, 'currentDistance' is evaluated to 'tooLong', the user does not send any command to the ACC ('userCange' == NoVal), a target vehicle exists ('TargetVehicleExists' == True), but operates the gas pedal, i.e., UserCommandToControlTrain == 'GentlyAccelerate', etc. There are no values expected on output ports, i.e., all parameters are specified as 'NoVal'.

The simple randomly generated test cases we mainly used for building and refining the test model and the test driver component. But these test cases were not very useful to test the normal use of the ACCPCS system. For example the input of the vehicle speed switched permanently to completely different values (like form 34 km/h to 98 to 2). More interesting test case we retrieved by applying the usage profile based test case generation. Figure 31 shows the setting of the usage profile we applied in the end.

Test Cases from the System (Implementation) Model Additionally we used the system model to derive test cases. Here we just applied the simple random test generation techniques. The main use of this approach was during the implementation of the test driver. However, we also encountered a fault in the code generator, which could be fixed in the new version of the AutoFOCUS 3 code generator. The faulty code generator would have lead to faulty code of the implementation. Using these test cases also the general correctness of the operating system and the manually written glue code for the CAN communication were tested. This ensured that the overall test setup worked correctly before the test cases generated from the test model were applied to test the real system functionality.

5.3.3 Implementing test drivers

The test driver component we implemented basically fulfilled three tasks:

- Translates abstract test cases to concrete (executable) instances
- Controls test execution
- Judges of test pass / fail (builds verdict)

When implementing test drivers, interfaces of the test model and the implementation need to be matched. The abstractions of the test model need to be explicitly bridged. Test drivers can be reused in consecutive development steps. As mentioned in the section before we used test cases from the system model to initially set up the test driver component. That approach especially simplified the implementation of the test execution platform.

For bridging the abstraction between test model and implementation we further analyzed the interface of the test model. Table 17 summarizes the interface of the test model. For most ports the matching form test model and implementation was straight forward. However, there were three ports where the abstraction in the test model was not straight forward and had to be bridged by the test driver:

- **userCmdToCtrlTrain** (In): The difficulty with that port was that it had to be matched to two ports in the implementation, namely *gasPedalPressed* and *brakePedalPressed*. However the solution of this difference was still quite easy: The acceleration value of *userCmdToCtrl-Train* was mapped to *gasPedalPressed = true, brakePadelPressed = false*, the brake value to *gasPedalPressed = false, brakePadelPressed = true* and in the case of *keep* both ports of the implementation are set to *false*.
- **userChange** (In): This was more difficult to solve since on the implementation model three ports were used: *changedDistance*, *onOffButton* and *incDecDesiredSpeed*. In case of *None* no of the three ports was set, in case of *SpeedInc* or *SpeedDec incDecDesiredSpeed* was set to *true* or *false*, resp. The difficulty with that port stem from the on/off-button command. Whereas in the test model the value has a distinct semantics for *Activate* or *Deactivate* (data based semantics), in the implementation only one signal was used, the interpretation of that signal depended on the current state of the system: A first occurence of *onOffButton* lead to activate, a second to deactivate and so on. The solution for this port was to keep track of the system state (activated or deactivated) even in the test driver. But we did not do this to the full extend: Keeping track of the exact internal state would have required too much logic calculation within the test driver itself. To avoid such problems the interface specification and abstraction level of the test model must be defined in more detail before starting modeling the implementation and test model.
- **outputAcc** (Out): We assumed different control algorithms for the distance control of ACC in test and implementation model. Therefore a mapping could not be defined for in every case. To avoid such situations more precise requirements would be necessary.

Figure 30: Example test case. The test case specifies provided inputs and expected outputs for ten consecutive steps.

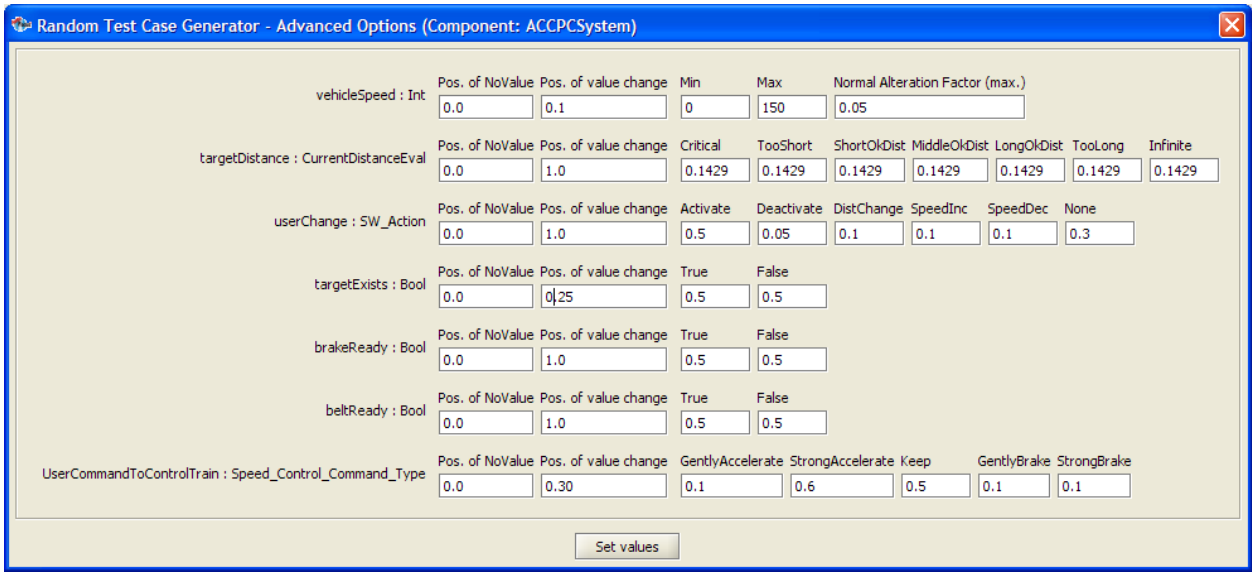


Figure 31: Applied usage profile for test case generation.

Table 17: Interface of test model. The last column indicates if the interface at that port is already matching

	Port	**Component(s)**	**Type**	**matches IM**
In	userCmdToCtrlTrain	UserSettings, Arbiter	{GentlyAccel, StrongAccel, Keep, GentlyBrake, StrongBrake}	no
In	userChange	UserSettings	{Activate, Deactive, DistChange, SpeedInc, SpeedDec, None}	no
In	vehicleSpeed	UserSettings, CSCtrl, FUCtrl	Int	yes
In	targetExists	AccCore, PCCtrl	Bool	yes
In	targetDistance	PCCtrl, FUCtrl	{Critical, TooShort, ShortOk, MiddleOK, LongOK, TooLong, Infinite}	yes
In	beltReady	PCCtrl	Bool	yes
In	breakeReady	PCCtrl	Bool	yes
Out	accState	UserSettings	Bool	yes
Out	desiredDistance	UserSettings	{ShortDist, MiddleDist, LongDist}	yes
Out	outputAcc	Arbiter	{GentlyAccel, StrongAccel, GentlyBrake, StrongBrake}	no
Out	executeBrake	AccCore	Bool	yes
Out	prepareBelt	PCCtrl	Bool	yes
Out	prepareBrake	PCCtrl	Bool	yes
Out	criticalCollisionTimeWarn	PCCtrl	Bool	yes

Figure 32 shows the user interface of the developed test driver.

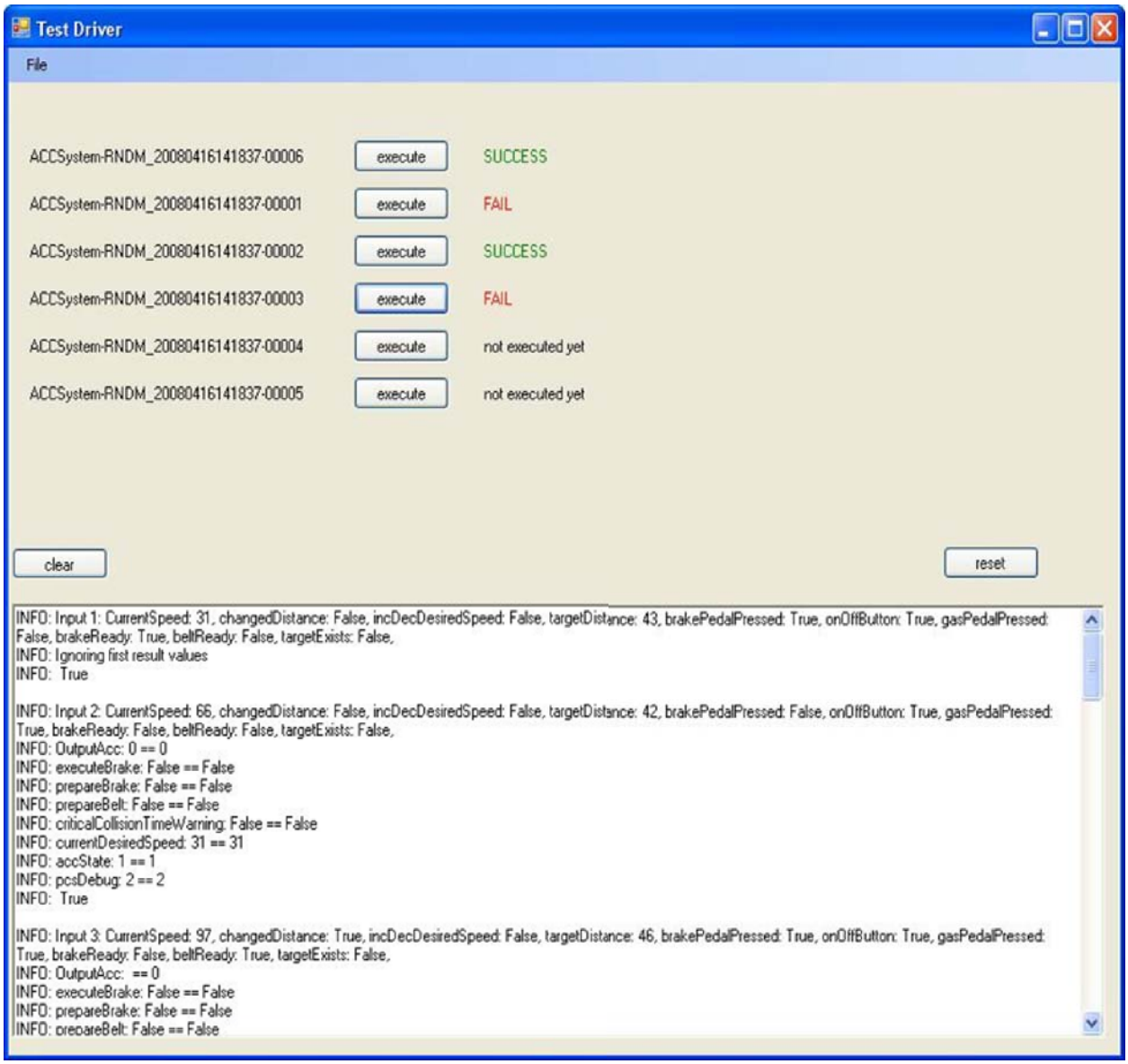


Figure 32: The user interface of the implemented test driver.

5.3.4 Executing test cases

All two sets of test cases were executed on the implementation. In total we generated and applied approx. 200 test cases to the implementation. Most test cases helped to set up the test execution environment and eliminate different requirements interpretation by the test model designer than the system model designer (that was mainly due to the fact that for modeling the case study none of the model designers were domain experts).

In the end we analyzed a set of 20 test cases which were generated according the usage profile showed in figure 31.

Table 18 summarizes the results of the test execution. In average a test case comprised approx. 20 execution steps. The resulting observations were the following:

Table 18: Results of test execution of sample test case set.

Test case	Length	Result	Issue
1	24	failed	$v > 110$
2	16	success	
3	15	failed	$v > 110$
4	23	failed	activateIgnored
5	16	failed	outputAcc
6	15	failed	$v > 110$
7	25	failed	$accState = 0$
8	20	failed	activateIgnored
9	24	failed	$v > 110$
10	21	failed	$v > 110$
11	24	failed	$v > 110$
12	18	failed	$v > 110$
13	19	failed	activateIgnored
14	22	failed	outputAcc
15	21	success	
16	22	failed	$v > 110$
17	22	failed	outputAcc
18	25	success	
19	17	failed	activateIgnored
20	22	success	

- 4 test cases **succeeded** in all execution steps.
- In average approx. 80% of the executed steps where correct.
- 7 test cases failed in at least one step due to the $v > 110$ issue: That was a fault in the system model where the ACC was not correctly deactivated if speed is exceeds 110 km/h.
- 1 test case failed in one step due the the $accState = 0$ issue: That was a fault in the system model where an illegal value for *accState* (0) was observed.
- 3 test cases failed in at least on step due to the **outputAcc** issue: That was because different requirements for the control algorithm were assumed by the test and system modeller.
- 5 test cases failed due to the **activateIgnored** issue: That was due to the unsolved abstraction of event and data sementics in the test driver. That issue could have been avoided by more precise interface specification.

It must be noted that three faults only where uncovered, in large parts the system behavior was indeed correct. The reason for 16 failing test cases is mainly due to the fact that nearly all test cases reach one of this three fault situation at some time.

Of course, in reality that faults should be fixed in the system. For demonstration issues of the test generation capabilities we omit that subsequent step.

5.4 Summary of the model-based testing activities

In this section, we described the methodology used for model-based testing in the ACC case study. We build up a test model, generated tests from the test model and executed this test cases on platform level of the implementation. We also used tests from the system model. For test case generation we used equal distribution for random tests and usage profiles as test case specification.

We could detect differences between test and implementation model which could be identified as actual faults or issues which need further requirements clarification. The independent development of test and system model here showed different views on requirements, in that way these redundant design proved to be very useful in showing requirements which are not precise enough. Out new enhancement for AutoFOCUS test case generator, the test case specification by usage profiles combined with random input generation turned out to be very useful: It provided suitable adjustment of test cases to realistic usage scenarios but still offered also a wide variation of test cases. In addition this method is very scalable, even for large models, or complex control functions. Test cases from the system model were useful for setting up the test execution environment and we could detect a fault in the code generator which lead to a faulty implementation first.

It was encountered that the interface definition must be more precise before splitting up in implementation and testing activities semantics of in and outputs especially for interfaces the question of event or data semantics must be defined. Important would be also a definition of the timing, e. g. how many cycles are allowed until the response to an input is expected (that problem did not arise within the ACCPCS case study). We also identified that the abstraction of the test model should have been clearly defined in the beginning, ideally one should be able to build the test driver before the test model and implementation, it is important to have a clear understanding which parts of the system logic should be in the test model and which parts will be bridged by the test driver. Fur further studies it is to consider of only one test model for the whole functionality should be used or several test models which describe partial behaviors of the system should better be used.

6 Deployment

6.1 Operating system and bus configuration

The ACC system was deployed onto one MPC5554 boards with 2 MB RAM. A simple OSEK operating system was used that contained only one task which is executed every 100 Milliseconds. The input parameters are encoded in a single CAN message and are sent every 100 milliseconds as well. The calculated output values are also encoded into a CAN message and sent at the end of every invocation of the task.

Many of the input signals have an event-like nature. The environment of the ACC should not have to be aware of the state the ACC system is in. So the user inputs that allow to switch the ACC on and off as well as the modification of the distance mode is encoded into a single bit which is always zero but becomes one when the button is pressed. Every reaction on this events occurs due to an edge in the signal gradient. Increasing and decreasing the target speed is done using the $ChangeSpeed_{const}$signal. This signal has a ternary encoding using two bits: 00 means ‘no modification’, 11 means ‘increase speed’, 10 indicates ‘decrease speed’, 01 is not used. All the other event signals are simply encoded binary as a boolean value. The current speed of the car and the distance to the car driving in front is encoded as a 16-bit integer. Also the output message uses such a ternary encoding for the distance mode.

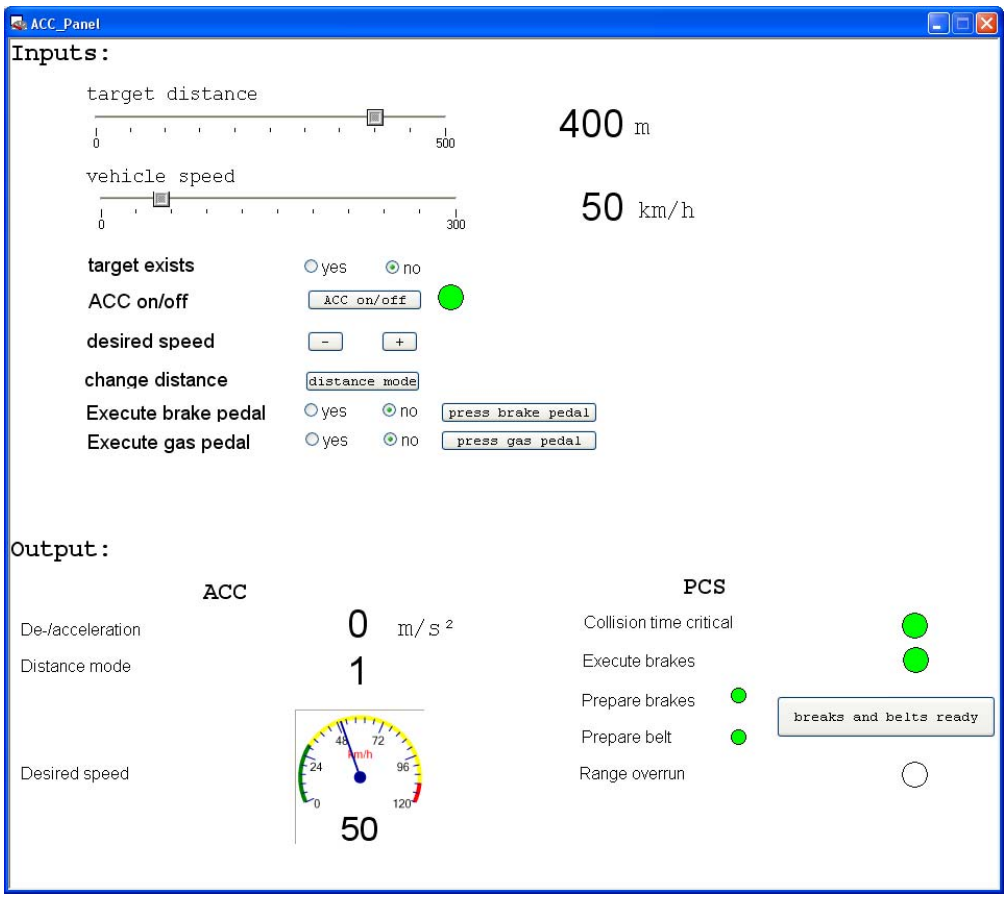


Figure 33: The panel used for manually testing the ACC system

6.2 Simulation of the environment

A simulation environment has been implemented to be able to monitor the system and test it manually. This has been achieved using the tool CANoe from Vector. Figure 33 shows this panel.

7 Conclusions

In this report we have presented the methodology for the model-based development of automotive systems. The starting point of the approach are the requirements engineering activities. The informally given requirements are analyzed, categorized, formulated, and structured according to the methodology presented in [Fle08]. After that the requirements will now be formalized step by step according the methodology presented in [Rit08]. The result of the requirements engineering phase is a formal model of the usage functionality.

Based on this results a logical model of the system that can be simulated in AutoFocus to do a first step in quality assurance is developed. This model is the central artifact for the subsequent development. It is also the basis for generating executable code (C code). After that we have used model checking to verify the AutoFocus model formally.

We also have described the methodology for model-based testing and its evaluation on the ACC case study. We build up a test model, generated tests from the test model and executed this test

cases on platform level of the implementation. We also used tests from the system model. For test case generation we used equal distribution for random tests and usage profiles as test case specification.

Combining all these steps together we get the process to achieve system development in a top-down manner.

The feasibility of the proposed approach was evaluated by developing an Adaptive Cruise Control (ACC) system with Pre-Crash Safety (PCS) functionality.

References

[Aut] Autofocus 2.

[Fle08] Andreas Fleischmann. *Model-based formalization of requirements of embedded automotive systens*. PhD thesis, Technische Universität München, 2008.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Norwell, MA, USA, 1993.

[McM99] Ken McMillan. Getting started with SMV. Tutorial, Cadence Berkeley Labs, 1999.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[PC85] D. L. Parnas and P. C. Clements. A rational design process: how and why to fake it. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT) on Formal Methods and Software, Vol.2: Colloquium on Software Engineering (CSE)*, pages 80–100, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[Pfa08] Christian Pfaller. Autofocus 2 test case generator. internal report (unpublished), Technische Universität München, Garching, Germany, May 2008.

[PPS+03] J. Philipps, A. Pretschner, 0. Slotosch, E. Aiglstorfer, S. Kriebel, and K. Scholl. Model based test case generation for smart cards. In *8th Intl. Workshop on Formal Methods for Industrial Critical Syst.*, pages 168–192, 2003.

[PPW+05] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *Proceedings of the 27th international conference on Software engineering*, 2005.

[Pre03] Walter Alexander Pretschner. *Zum modellbasierten funktionalen Test reaktiver Systeme*. PhD thesis, Technische Universität München, 2003.

[Rit08] Sabine Rittmann. *A methodology for modeling usage behavior of multi-functional systems*. PhD thesis, Technische Universität München, 2008.