Analysis and Verification of Complex Robot Systems using Behaviour-Based Control

Lisa Kiekbusch

Vom Fachbereich Informatik der

Technischen Universität Kaiserslautern

zur Verleihung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation.

Datum der wissens. Aussprache:	25.11.2016
Vorsitzender:	Prof. Dr. Hans Hagen
Erster Berichterstatter:	Prof. Dr. Karsten Berns
Zweiter Berichterstatter:	Prof. Dr. Peter Liggesmeyer
Dekan:	Prof. Dr. Klaus Schneider
Zeichen der TU im Bibliotheksverkehr:	D 386

Acknowledgement

I want to thank everybody who accompanied and assisted me during my work on this thesis. First of all, I want to thank Prof. Karsten Berns who provided the basis for my research activities in robotics. Numerous discussions helped to specify the main topic of my thesis and to focus on the relevant aspects in the broad field of robotics. I also thank Martin Proetzsch who introduced me in the field of analysis of behaviour-based systems during my diploma thesis and encouraged me to start my PhD. The Carl-Zeiss foundation financed my research over several years.

The complex topic of robot control system verification cannot be discovered without the help of others. I thank Prof. Roland Meyer for his support during the first steps in the field of formal verification. Furthermore, I want to thank my colleagues Christopher Armbrust and Thorsten Ropertz for fruitful discussions, where new ideas were born and old approaches were checked for usefulness, and Max Reichardt, who was always willing to answer my questions regarding the framework and the tools. I especially thank Christopher Armbrust for always motivating me to go on, even in times of no progress. Several students supported my work. Thank goes to Patrick Vonwirth, Johannes Bayer, Yves Gänssinger and Marc Hauer.

I thank all my (former) colleagues, especially Jochen Hirth, Daniel Schmidt, Patrick Fleischmann and Thomas Pfister for interesting discussions about work-related topics as well as real-world stuff. It was a pleasure to work with you. For proof reading of some chapters or even the whole thesis, I thank Thorsten Ropertz and Isabel Stemler.

Last but not least, this work would not have been possible without my family. I thank my parents for their encouragement and assistance with our kids to offer me additional work time. My thank and love go to my husband Timo, who always helped me in times of (smaller and bigger) crises and gave me invaluable support.

Abstract

The development of autonomous mobile robots is a major topic of current research. As those robots must be able to react to changing environments and avoid collisions also with moving obstacles, the fulfilment of safety requirements is an important aspect. Behaviour-based systems (BBS) have proven to meet several of the properties required for these kinds of robots, such as reactivity, extensibility and re-usability of individual components. BBS consist of a number of behavioural components that individually realise simple tasks. Their interconnection allows to achieve complex robot behaviour, which implies that correct connections are crucial. The resulting networks can get very large making them difficult to verify. This dissertation presents a novel concept for the analysis and verification of complex autonomous robot systems controlled by behaviour-based software architectures with special focus on the integration of environmental aspects into the processes.

Several analysis techniques have been investigated and adapted to the special requirements of BBS. These include a structural analysis, which is used to find constraint violations and faults in the network layout. Fault tree analysis is applied to identify root causes of hazards and the relationship of system events. For this, a technique to map the behaviour-based control network to the structure of a fault tree has been developed. Testing and data analysis are used for the detection of failures and their root causes. Here, a new concept that identifies patterns in data recorded during test runs has been introduced.

All of these methods cannot guarantee failure-free and safe robot behaviour and can never prove the absence of failures. Therefore, model checking as formal verification technique that proves a property to be correct for the given system, has been chosen to complement the set of analysis techniques. A novel concept for the integration of environmental influences into the model checking process is proposed. Environmental situations and the sensor processing chain are represented as synchronised automata similar to the modelling of the behavioural network. Tools supporting the whole verification process including the creation of formal queries in its environment have been developed.

During the verification of large behavioural networks, the scalability of the model checking approach appears as a big problem. Several approaches that deal with this problem have been investigated and the selection of slicing and abstraction methods has been justified. A concept for the application of these methods is provided, that reduces the behavioural network to the relevant parts before the actual verification process.

All techniques have been applied to the behaviour-based control system of the autonomous outdoor robot RAVON. Its complex network with more than 400 components allows for demonstrating the soundness of the presented concepts. The set of different techniques provides a fundamental basis for a comprehensive analysis and verification of BBS acting in changing environments.

Contents

1. Introduction

Nowadays, robotic systems are an inherent part of industrial work and the market is still growing. According to the International Federation of Robotics, since 2010 the demand for industrial robots has grown rapidly (17% per year) and reached a new sales record in 2015 [IFR 15]. The most important customer in this context is the automotive industry, followed by the electrical/electronics industry. In the context of Industry 4.0, flexible and automated solutions are state-of-the art and the collaboration between humans and robots is a challenging topic.

Service robots are said to conquer the market in the near future. They are already established as household helpers such as floor cleaning robots or in the medical context, e.g. as mobile assistance systems or as assistance in surgery tasks in hospitals. Self-driving vehicles are also a big topic of current research where great successes have already been achieved recently [Lari 15]. Autonomous mobile robots are often deployed to solve dangerous tasks such as the search for humans in disaster areas or the identification of weapons of mass destruction.

1.1 Motivation

The growing number of robotic systems in our daily life, especially in such critical areas as medical engineering or car assistance systems, intensifies the need for reliable and safe systems. Several examples for failing robot behaviours are given in literature, among these are industrial robot arms that injured or killed workers or software errors in medical systems that lead to physical injuries or death [Dhillon 15, Alemzadeh 15, 0sha06 06, Backström 95]. Accidents like these substantiate the call for methods and standards that prove the safety and reliability of systems. This is especially needed for robotic systems that act autonomously in unstructured environments and/or interact with humans.

Autonomous mobile systems have special demands in contrast to stationary systems acting in a structured and well-defined environment. Such systems must be equipped with several sensors that capture the environmental conditions and a control software which is able to react in situations that were never defined explicitly. Behaviour-based architectures have been implemented successfully on robotic systems to meet those requirements [Arkin 98].

They are based on the *subsumption architecture* introduced by Rodney Brooks [Brooks 91]. Behaviour-based systems consist of a number of simple components where each is responsible for a basic task. In order to fulfil more complex tasks, the simple components are connected via their inputs and outputs to build a network. For example, a behaviour responsible for turning the robot to the left and a behaviour responsible for driving ahead are combined to enable the robot to turn a left curve. In contrast to the traditional "sense-plan-act" approach, these systems are able to react in unknown situations. Additionally, the distribution of functionality allows parallel software developing, early testing, re-usability of single components and good extensibility.

The downside of this modular approach is the complexity of the interconnections, which increases with the function volume. The lack of automated methods which link basic components and the absence of guidelines particularly for special tasks hold a high error potential. Due to the modular and distributed structure, the identification and elimination of faults can be difficult, as they might not only be rooted in components of the system, but based in the network structure. Furthermore, a misbehaviour of the robot is usually not based on a single point, which complicates the identification of its origin. The fact that not all failures are detected during testing, e.g. due to the usually implemented redundancy, presents an additional level of difficulty. Redundancy might be integrated using additional sensor systems or components with the same goal but different realisations, for example obstacle avoidance behaviours, that realise an evasive manoeuvre to the left or to the right, respectively.

The above mentioned problems focus on the software, but, for embedded systems as robots, the hardware component cannot be neglected. Hardware can also be the source of robot failure, e.g. broken sensors, which naturally have an effect on the software calculated robot control commands. Last but not least, special environmental conditions can cause the robot to miss its goals.

This raises the need for analysis techniques for the special demands of behaviour-based robotic systems, which allow to detect failures and their origins, as well as techniques that verify the correct execution.

1.2 Objectives

The scientific contribution of this work is the definition of a comprehensive set of techniques that allow to analyse and formally verify large-scale behaviour-based robot systems. Of special emphasis is the integration of hardware components and environmental conditions into the analysis processes.

Figure 1.1 illustrates the covered aspects. In the centre is the behaviour-based software control system together with its possible executable platforms embedded in the environment, which is surrounded by the different parts of the concept: Analysis methods and formal verification techniques.

The term *Analysis* handles a broad spectrum of purposes, and correspondingly a large number of techniques are available. In the context of this thesis, the goals of analysis are

- determining the characteristics of the behaviour-based network

- detecting faults in the network structure
- determining sources of robot failure
- determining violations of requirements
- identifying "key components" in the network structure

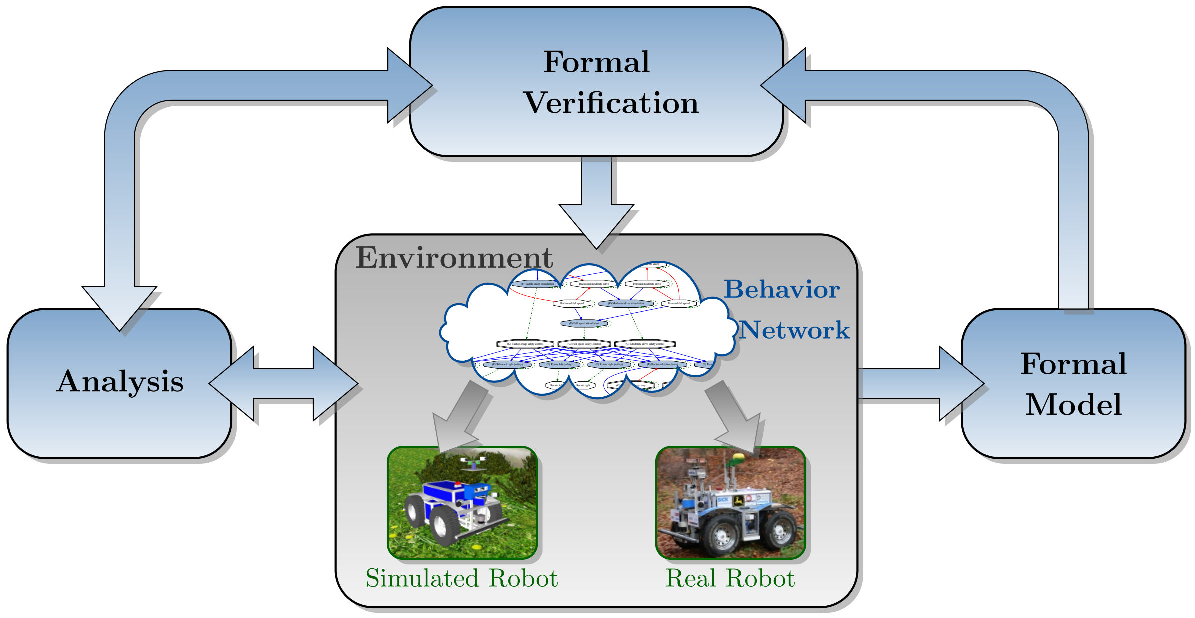


Figure 1.1: The concept of this thesis. The behaviour-based network together with its execution platforms (simulated or real robot) embedded in its environment is verified and examined using several analysis techniques. For formal verification a formal model is created. The results of the model checker can be used as reference point for analysis or vice versa analysis methods can be used to correct faults detected in the formal verification process.

The remaining two boxes in Fig. 1.1 handle the topic of formal verification. In contrast to testing, which can only show the presence of failures, formal verification can prove a system to be correct, and is thus an useful additional verification technique. Formal verification in the context of this thesis is based on a formal model of the system to be verified. Therefore, the modelling of the behaviour-based system is a central aspect with a focus on the integration of sensor hardware and environmental conditions. The verification process aims at checking a defined property to be correct for the model. Here again, the verification of properties that are related to the environment in which the robot acts are a major topic.

Large networks and the additional components naturally result in large formal models, which leads to long computation times and problems with the size of the available memory. The handling of these problems is also a major topic of this thesis. The basic idea is to utilise results of the analysis to focus the verification process on the relevant system parts. As the manual application of analysis techniques and the interpretation of the results is difficult and error-prone for large behaviour networks, automated methods and graphical support are also topics of this work.

The objectives of this thesis can be summarised in two aspects.

Analysis techniques: The common goal of the analysis techniques is to earn knowledge about the system in order to be able to identify faults. The methods shall support the developer to identify faults faster and with less effort. The investigation of the structure of the behaviour-based control network and the interfaces to hardware components, as well as the identification of special key components, shall serve as basis for further analysis and verification methods.

Formal verification: The purpose of developing a method to formally verify behaviour-based systems that take into account hardware components and environmental aspects is to assist the developer during the verification of system requirements that are dependent on these factors. Additionally, the technique shall be applicable to the large behaviour-based networks of robots realising complex tasks.

Both approaches shall support the developer of behaviour-based systems to create safe and reliable software.

1.3 Outline

This thesis is structured as follows. Chapter 2 starts with the definition of important terms, followed by an introduction to the fundamentals of the concepts presented later in this thesis. These include the behaviour-based architecture and the robot platform RAVON, which have been used to test the developed methods and techniques and to illustrate them throughout this work. As a special focus is set on the integration of the environment into the analysis process, the sensor processing of RAVON is also described. Furthermore, the visualisation tools which are the basis for the developed graphical user assistance system are introduced.

An overview of techniques targeting the analysis and formal verification of safety and reliability properties, which are widely used in the development of software and embedded systems is provided in Chap. 3. The general concepts of several hazard analysis techniques, testing, and formal verification methods are described. Example applications are presented which prove their applicability to robotic systems. The current state of the art of analysis methods in particular for behaviour-based systems is examined as well. The chapter concludes with a discussion of the applicability and expandability of the methods to behaviour-based systems.

Chapter 4 targets techniques to analyse behaviour-based systems. Starting from a structural analysis, providing automated methods to gather basic knowledge about the network structure and detecting faults in the interconnection of components, an approach to detected relevant system parts is presented. As a further statical analysis method, fault tree modelling is introduced, which provides a method to discover correlations in the system leading to a robot failure. The third section focuses on the analysis of data flowing in the network structure with the goal to determine the origins of faulty robot behaviour.

The formal modelling of behaviour-based systems as networks of timed automata is described in Chap. 5. In addition to a short introduction into model checking and the modelling of standard behavioural components, the integration of sensor hardware and environmental conditions is presented in detail. The generation of formal queries and the usage of the model checker are exemplified through the example of the autonomous robot RAVON using the developed assistance tools. An evaluation of time and memory consumption during the formal verification process and a discussion complete this chapter.

The results of the evaluation in Chap. 5 lead to the topic of Chap. 6 which focuses on possible methods to face the scalability problem of the model checking approach. Based on a brief overview of different approaches to this problem, the adaptation of the techniques *slicing* and *abstraction* to the needs of behaviour-based systems is explained in detail. The improvements achieved by using these techniques are shown on an application example. Additionally, the transferability of the methods to support other analysis processes is discussed.

Finally, Chap. 7 summarises the work of this thesis. The major results are specified and evaluated and options for future work are given.

2. Fundamentals

The work at hand deals with the analysis and verification of behaviour-based systems. To motivate and demonstrate the proposed methods several examples are given throughout this work. In this chapter, the fundamental structure of the used behaviour-based architecture, the *integrated Behaviour-Based Control* (Sec. 2.2), and its implementation for the control of the autonomous outdoor robot RAVON is described (Sec. 2.3). Besides, a short introduction into the software framework and the corresponding visualisation tool is given as the latter is consequently extended in the work at hand (Sec. 2.4). An introduction to the most relevant terms in the context of safety analysis and behaviour-based systems is given in Sec. 2.1.

2.1 Terms and Definitions

Throughout this work, several terms were used, either in the context of analysis and verification or in the context of behaviour-based systems, that have several meanings or no exact definitions. This section introduces the main terms as used in this thesis.

In the context of behaviour-based systems, talking about a ***behaviour*** is ambiguous, as it can mean the behaviour of the robot or the basic component inside the software structure. To distinguish both subjects, the latter are always called with an addition, e.g. ***behavioural component*** or ***fusion behaviour***. A network of behavioural components is called ***behavioural network*** or just ***network***.

During the analysis process, several special terms are used to distinguish between a visible misbehaviour (*failure*) of the system, the corresponding trigger (*fault, defect*) and its source (*error*). In literature, there exist slightly different definitions of these terms. In this thesis, they will be defined according to [Liggesmeyer 09].

Definition 2.1: Failure

A malfunction or failure is an inconsistent behaviour with respect to the specified (desired) behaviour (IEC61508 – 4). A failure occurs during the execution of the system. A failure is the result of a fault (in software).

Definition 2.2: Fault or Defect

A software fault or defect is the cause of a failure. A software fault means a fault in the program code, hardware faults can be e.g. construction faults.

Definition 2.3: Error

An error is the cause of a fault. This can be a mistake of the programmer or a simple typing error.

These terms are defined generally for (software) analysis methods. In the context of safety analysis, the term *safety* needs to be defined. According to the standard IEC 61508 *safety* and *functional safety* are defined as follows:

Definition 2.4: Safety and Functional Safety

A safe system is a system where unacceptable risks like injury or harm to the health of humans are absent. Safety describes a state where the danger of personal or property damage is reduced to an acceptable level.
Functional Safety is part of the overall safety that depends on a system or equipment operating correctly in response to its inputs.

Safety requirements are usually defined in the *Safety Requirements Specification* (SRS). An example for a safety requirement of an autonomous mobile robot would be that the robot must not drive into a person.

Safety requirements are derived from a so-called *hazard analysis*. The term *hazard* is defined as

Definition 2.5: Hazard

A hazard is any potential source of harm $(IEC61508 - 4)$.

Hazards can be classified according to their criticality using categories like *acceptable* or *unacceptable*. ***Hazard analysis*** deals with the identification of hazards.

2.2 Integrated Behaviour-Based Control

The behaviour-based architecture integrated Behaviour-Based Control (iB2C) is has been developed and is continuously improved at the RRLAB[1]. It has been implemented in the robotic frameworks MCA2-KL and FINROC (see Sec. 2.4).

The basic component of the iB2C is depicted in Fig. 2.1. Its standardised interface has three input possibilities:

Stimulation s**:** The stimulation input is used to gradually enable a behavioural component.

[1]Robotics Research Lab, University of Kaiserslautern

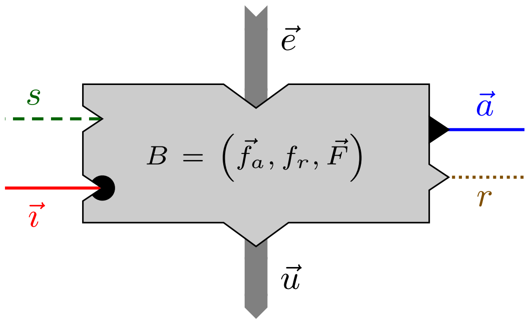
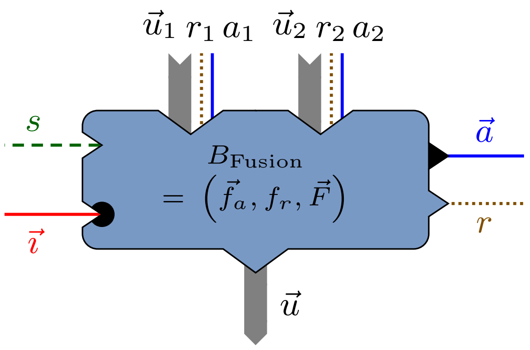


Figure 2.1: Basic iB2C component



$\vec{u}_1\ r_1\ a_1\ \ \vec{u}_2\ r_2\ a_2$
s
$\vec{a}$
B_{Fusion}
$= \left(\vec{f}_a, f_r, \vec{F}\right)$
$\vec{\imath}$
r
$\vec{u}$

Figure 2.2: iB2C fusion behaviour realising the maximum fusion for $n_c = 2$ competing behavioural components

Inhibition Vector $\vec{\imath}$: The inhibition input is used to gradually disable a behavioural component. A behavioural component can be inhibited by several other components, where i is calculated by $i = \|\vec{\imath}\|_\infty$.

Control Vector $\vec{e}$: The control vector $\vec{e} \in \mathbb{R}^m$ can transfer arbitrary data between the behavioural components.

The inputs are usually connected with the outputs of other components in the network. The three output possibilities are as follows:

Activity a: The activity describes the influence of a behavioural component in the network. A behavioural component can use only a part of its activity to influence other behavioural components by using so-called *derived activities* $\underline{a}_0, \underline{a}_1, \ldots, \underline{a}_{q-1}$ with $\underline{a}_i \leq a$, $\forall i \in \{0, 1, \ldots, q-1\}$. Together with a, they build the *activity vector* $\vec{a} = (a, \underline{\vec{a}})^T$.

Target Rating r: The target rating describes the satisfaction of a behavioural component, where 0 means maximal satisfaction. A component is satisfied when its goal is achieved.

Control Vector $\vec{u}$: The control vector $\vec{u} \in \mathbb{R}^n$ can transfer arbitrary data between behavioural components.

s, i, a and r together with the internal activation value $\iota = s \cdot (1 - i)$ are called *behavioural signals*. Their value range is limited to [0,1], whereas the control values of $\vec{e}$ and $\vec{u}$ are not limited. For activity and activation, $a \leq \iota$ must hold. A behavioural component is called *activated*, when its activation is greater than 0, it is called *active*, when additionally its activity is greater than 0. A behavioural component is formally defined as $B = \left(\vec{f}_a, f_r, \vec{F}\right)$ with $\vec{f}_a : \mathbb{R}^m \times [0,1] \to [0,1] \times [0,1]^q$ and $\vec{f}_a(\vec{e}, \iota) = \vec{a}$ being the *activity function*, f_r with $f_r : \mathbb{R}^m \to [0,1]$ and $f_r(\vec{e}) = r$ being the *target rating function* and the *transfer function* $\vec{F}$ with $\vec{F} : \mathbb{R}^m \times [0,1] \to \mathbb{R}^n$ and $\vec{F}(\vec{e}, \iota) = \vec{u}$, that calculates the *output vector* $\vec{u}$ from the *input vector* $\vec{e}$ and the *activation*. The transfer function can be arbitrarily complex, depending on the needed functionality.

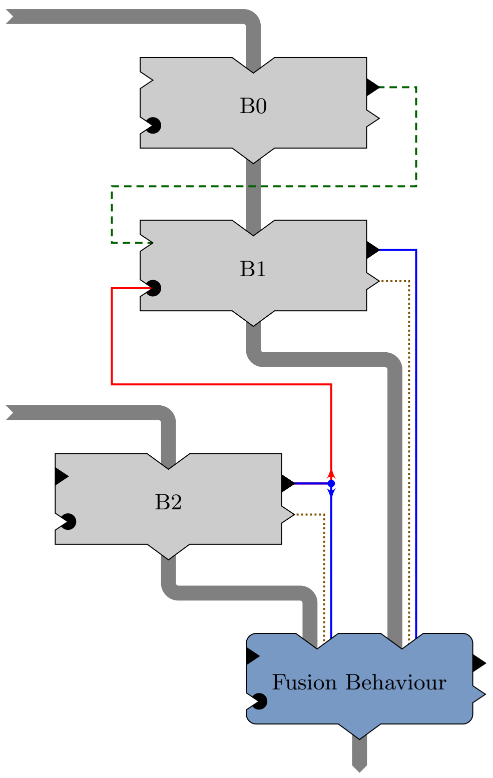


Figure 2.3: Example of an iB2C network. *B0* stimulates *B1* with its activity. *B2* inhibits *B1*. The outputs of *B1* and *B2* are combined by a maximum fusion behaviour. The filled triangle at the stimulation input of *B2* symbolises a constant full stimulation.

Behavioural components are combined with each other by connecting the activity output of one component with the stimulating or inhibiting input of another. Besides from that simple combination method, there exist several special components, that allow more sophisticated combinations. The most frequently used special behavioural component is the *fusion behaviour* (FB). It can coordinate the outputs of n_c competing behavioural components (B_{Input_d} with $d = 0, ..., n_c - 1$) to one output according to one of the three fusion methods: *maximum*, *weighted sum* and *weighted average*. For the work at hand, only the maximum fusion is of relevance and will be described in the following (see Fig. 2.2 for the graphical representation). Further fusion methods as well as the design and development of behaviour-based systems are described in [Proetzsch 10a, Armbrust 14a].

$$B_{Fmax} = (\vec{f}_{a\_Fmax}, f_{r\_Fmax}, \vec{F}_{Fmax}) \tag{2.1}$$

$$\vec{f}_{a\_Fmax}(\vec{e}_{Fmax}, \iota_{Fmax}) = \max_d(a_{\text{Input}_d}) \cdot \iota_{Fmax} \tag{2.2}$$

$$f_{r\_Fmax}(\vec{e}_{Fmax}) = r_{\text{Input}_g} \text{with } g = \arg\max_d(a_{\text{Input}_d}) \tag{2.3}$$

$$\vec{F}_{fmax}(\vec{e}_{fmax}) = \vec{u}_{\text{Input}_g} \text{with } g = \arg\max_d(a_{\text{Input}_d}) \tag{2.4}$$

Figure 2.3 shows an example of the mentioned connection possibilities. The fusion behaviour fuses the control values of *B1* and *B2*. As long as *B2* is not active and the stimulation of *B1* is greater than 0, the control values of *B1* have precedence over those of *B2*. With increasing activity of *B2* the activity of *B1* decreases as it is inhibited from *B2*.

As behaviour networks can be very large, the need to structure them leads to the introduction of *behavioural groups*. These groups have the same interface as a standard behavioural component. They can contain an unlimited number of behavioural components as well as other groups. The usage of groups leads to a hierarchical composition of behavioural components.

2.3 Application System: The Autonomous Off-Road Robot RAVON

The autonomous off-road robot RAVON (see Fig. 2.4) serves as test bed for localization, navigation and motion control in rough and vegetated terrain. RAVON shall fulfil high-level navigation tasks together with the requirement to operate safely and reliable in a complex environment.

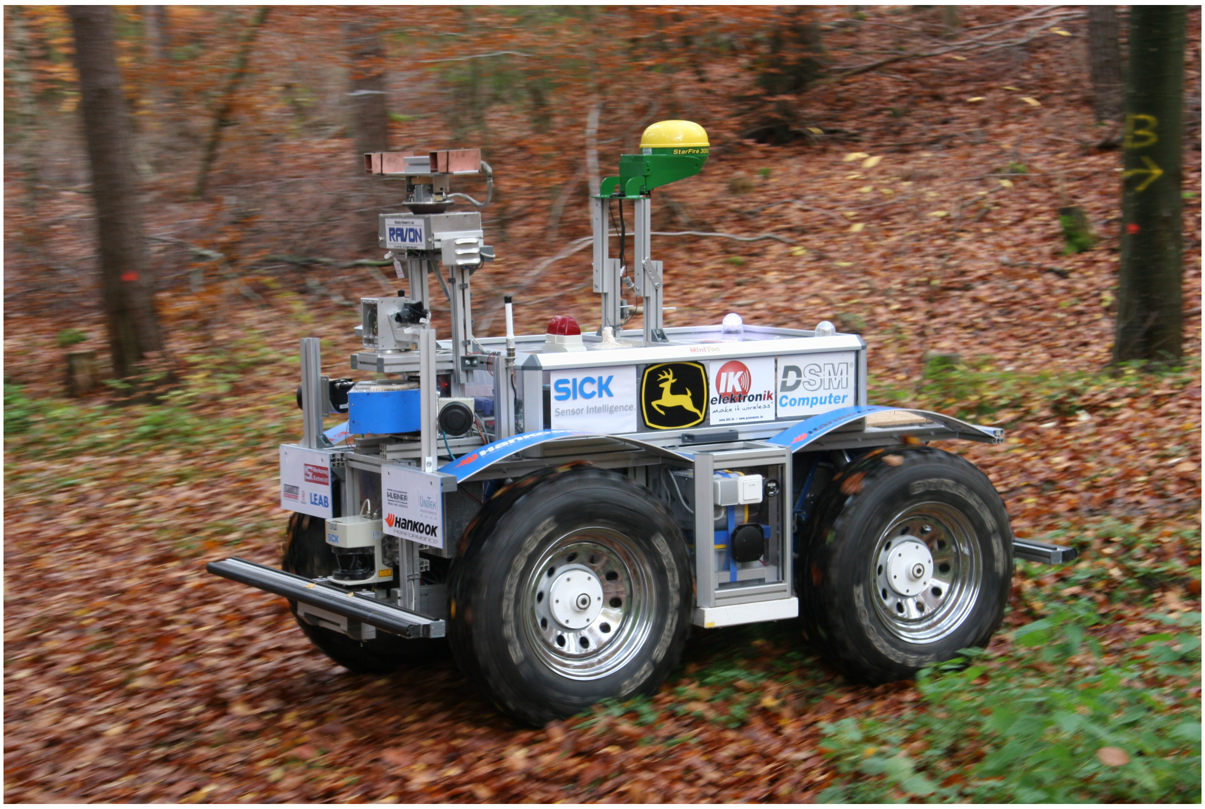


Figure 2.4: The autonomous off-road vehicle RAVON in the Palatinate Forest

Its basic platform is the robot platform robuCAR TT by Robosoft, which is extended by several components to meet the given requirements. Two independently steered axles allow the execution of advanced driving manoeuvres. It is equipped with a planar laser range finder, a panning laser scanner, a stereo camera and a spring-mounted bumper bar on its front as well as a planar laser scanner and a bumper bar on its rear. The bumper bars realise a double-stage detection of contacts. In a first level, two infrared sensors measure the stroke and draw conclusion to the force. These information is processed by the control software. The second level initiates an emergency stop in the case of high forces. For this, a safety shutdown board is connected to the safety chain of the robot. When the safety chain is interrupted, the robot stops. Four industrial Linux-PCs that run the control software complete the main components. The whole vehicle measures 2.4 by 1.4 by 1.8 m (length x width x height) and weights about 750 kg. Figure 2.5 depicts the hardware parts mounted on RAVON and their connections.

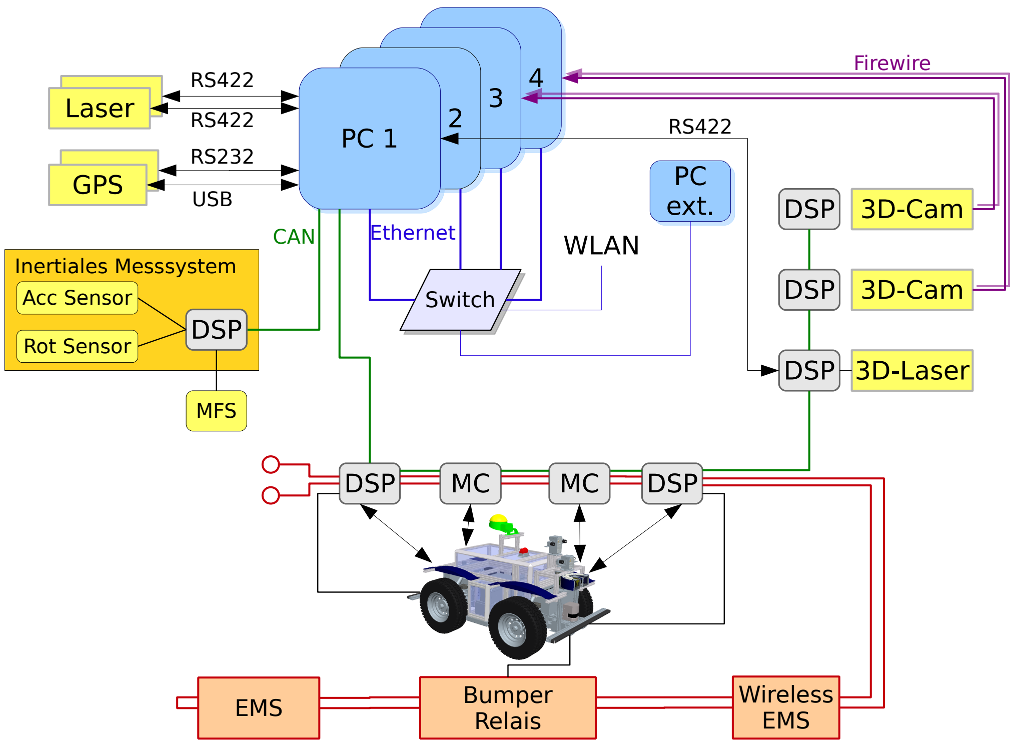


Figure 2.5: The hardware parts of the autonomous robot RAVON

2.3.1 Control Software

The control software of RAVON is divided into three navigation layers. The highest layer called *long-range navigation* or *navigator* plans the robots path to goals in a distance of 10 m up to several km. The navigator works on a topological map, which focuses on single waypoints that are relevant for the navigation. This map type reduces memory and computational demands by abstracting from local aspects, which are followed by the lower layers. The second navigation layer, *the mid-range navigation*, is needed to detect environmental situations like dead-ends, narrows or indentations.

The lowest level is called safety layer, responsible for *short-range navigation*, meaning collision-free operation in the local environment. Its tasks are e.g. to trigger an emergency stop or to avoid crashes with obstacles by driving around them. In order to do this, this layer can (partly) overwrite drive commands from higher layers by inhibiting them and providing corrective data. Its decisions are based on sensor data. In order to provide an abstract view for all types of sensor data and an arbitrary number of sensors, a virtual sensor layer is used [Schäfer 11].

The safety layer distinguishes three so-called drive-modi for *fast*, *moderate* and *slow* robot motion. Depending on the robots driving velocity different sensor data is needed. Fast driving requires long-range obstacle detection. When driving through vegetation, sensor data that allows to evaluate the traversability of the path is required (moderate drive). The slowest mode, called tactile creep, is needed in dense vegetation, where for example, the laser scanners can not evaluate if the obstacle can be passed or not. Here, the robot relies on data provided by the safety bumper. The behavioural groups implementing this concept are instances of the same class, each provided with the required sensor information. The switching between the modi is implemented in corresponding drive mode behaviours.

Figure 2.6 shows the network responsible for safety control during fast driving, called *Full Speed Safety Control*. It can be divided into three parts responsible for safe driving in respect to the three steering possibilities, sidewards motion (yellow background), rotation (light grey background), and straight-ahead motion (green background). All parts contain groups calculating the respective control data for the different sides of the robot and fusion behaviours to fuse the outputs with the pre-set motion command from a higher layer. Each group includes again several groups that encapsulate the calculation of the safe motion with respect to several sensor data evaluation methods. Figure 2.7 depicts exemplarily the group *(G) Fast Front Slow Down* from the network of components responsible for straight-ahead motion. The name of the group is a combination of the driving mode, *Fast*, the direction of the sensors, *Front*, and the action, *Slow Down*. The prefix *(G)* indicates the component as a group. The example group contains four subgroups that react to obstacles in the front of the robot. In the case of a front obstacle, the behavioural groups will get active and calculate a velocity value in dependence of the desired velocity and the distance to the obstacle. This value is fused and transferred to the groups output. In the surrounding group (Fig. 2.6) this value is fused and combined with other motion commands such as rotation and sidewards motion in order to initiate an evasion manoeuvre.

More detailed information about the concepts applied to this robot can be found in [Armbrust 10].

2.3.2 Sensor Processing

The task of sensor processing is to combine the data from different sensors and make them available for further use in the control software. The data needed for different navigation tasks shall be provided in an appropriate manner. For this, the data from the diverse sensors have to be composed and put into an abstract view for further access. For RAVON an approach using a *control-driven sensor processing* chain as presented in [Armbrust 10] is implemented. The approach uses abstract views called *sector maps* to provide information corresponding to special properties and a certain area around the robot.

The data from a sensor is firstly stored in a grid map which provides fast access to define areas and properties. Examples for properties are for example positive/negative obstacle, ground, and overhang. Properties are used to divide the covered region into several sectors. A sector map stores a property together with the distance from the sector's origin to the most relevant representative. Figure 2.8 shows a grid map monitored by the sensors of

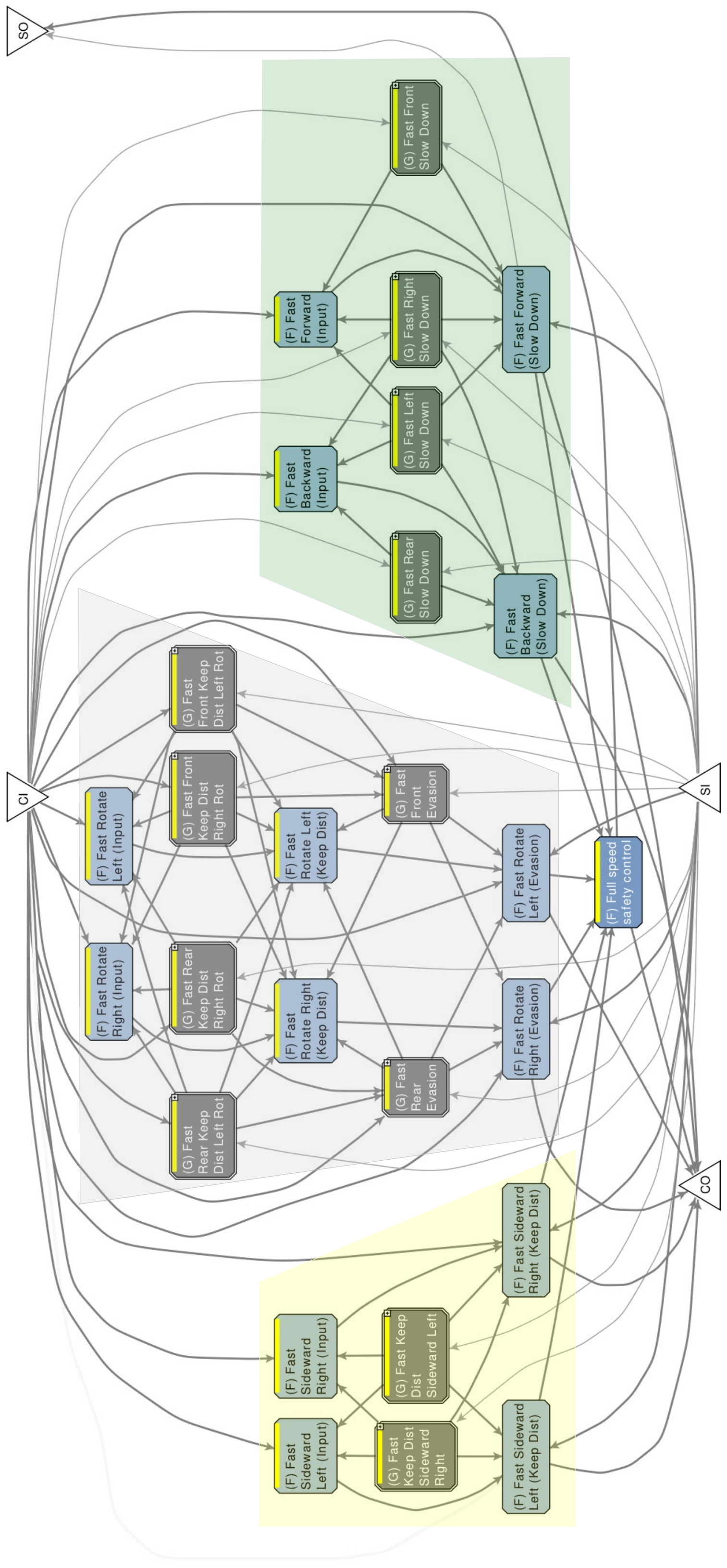

SO
CI
SI
CO
(G) Fast Front Slow Down
(F) Fast Forward (Input)
(G) Fast Right Slow Down
(F) Fast Forward (Slow Down)
(G) Fast Left Slow Down
(F) Fast Backward (Input)
(G) Fast Rear Slow Down
(F) Fast Backward (Slow Down)
(G) Fast Front Keep Dist Left Rot
(G) Fast Front Keep Dist Right Rot
(G) Fast Front Evasion
(F) Fast Rotate Left (Input)
(F) Fast Rotate Left (Keep Dist)
(F) Fast Rotate Left (Evasion)
(F) Full speed safety control
(F) Fast Rotate Right (Input)
(G) Fast Rear Keep Dist Right Rot
(F) Fast Rotate Right (Keep Dist)
(F) Fast Rotate Right (Evasion)
(G) Fast Rear Evasion
(G) Fast Rear Keep Dist Left Rot
(F) Fast Sideward Right (Keep Dist)
(F) Fast Sideward Right (Input)
(G) Fast Keep Dist Sideward Left
(F) Fast Sideward Left (Input)
(G) Fast Keep Dist Sideward Right
(F) Fast Sideward Left (Keep Dist)

Figure 2.6: Network of RAVON's safety control for the fast driving mode. Grey boxes represent behavioural groups, light blue boxes standard behavioural components and dark blue boxes fusion behaviours. The coloured backgrounds mark groups of components that belong to the same steering mode respectively. Triangles are symbols for the group borders. They bundle incoming and outgoing sensor values (SI and SO) and control values (CI and CO).

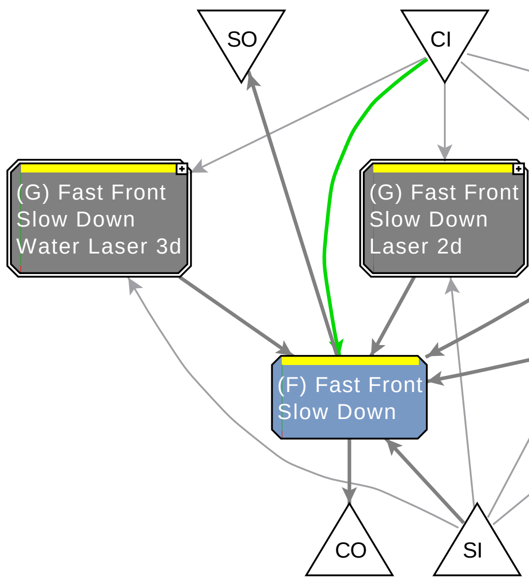
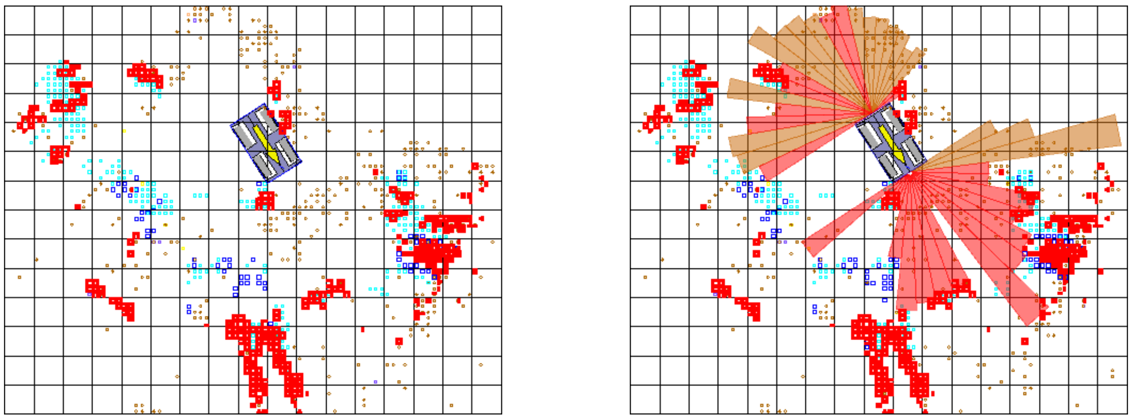

SO
CI
(G) Fast Front Slow Down Water Laser 3d
(G) Fast Front Slow Down Laser 2d
(G) Fast Front Slow Down Unfiltered Laser 3d
(G) Fast Front Slow Down Voxel Filter Laser 3d
(F) Fast Front Slow Down
CO
SI

Figure 2.7: RAVON network responsible for slowing down the robot in order to keep safety restrictions (*(G) Fast Front Slow Down*). The behavioural groups (grey boxes) encapsulate several behavioural components that calculate the robots motion in respect to the different sensors and the different evaluation methods.



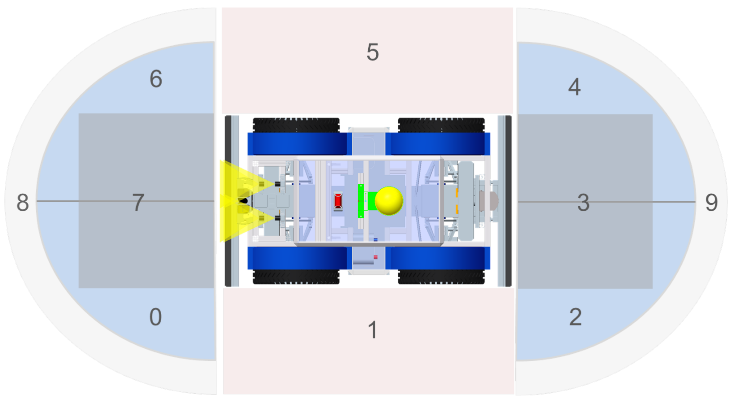
(a) Grid map **(b)** Sector maps

Figure 2.8: Grid map and sector maps monitoring the area around RAVON (source: [Armbrust 10])

RAVON and several sector maps covering the front and back of the robot. The information stored in sector maps can be combined in a *virtual sensor*, where each virtual sensor is responsible for a certain area and property related objects in this area.

With the help of these *virtual sensors*, it is possible to get task-relevant information from areas around the robot where no sensor could be mounted. For that, the area around the robot is divided into several subareas. Figure 2.9 depicts the subareas around RAVON. The information from a virtual sensor is provided to a corresponding behavioural component in the control network. Those react according to the given information.

Virtual sensors do not only correspond to a special area but also to different processing



(0) Front Left Polar
(1) Lateral Left Cartesian
(2) Rear Left Polar
(3) Rear Center Cartesian
(4) Rear Right Polar
(5) Lateral Right Cartesian
(6) Front Right Polar
(7) Front Center Cartesian
(8) Front Polar
(9) Rear Polar

Figure 2.9: Some of the virtual sensors around RAVON

algorithms like *raw laser scanner data evaluation*, *water detection*, and *voxel penetration methods*. This can lead to an arbitrary number of virtual sensors per real sensor.

A network that processes data from virtual sensors is given in Fig. 2.10. It shows the insights of three of the behavioural groups of the network given in Fig. 2.7. The components in the groups are arranged according to their real sensor type (3D laser or 2D laser) and the filter method (unfiltered or water detection). Every group contains three safety behaviours (grey octagons), one for each front sector of the robot (right, left, centre). Every safety behaviour processes its specific data exclusively and calculates its activity and a velocity control value. The fusion behaviours combine the results for further usage.

It can be seen that safety behaviours are a central point in behaviour networks as they include environmental information in the overall control system. Their correct integration into the complete network structure is important in order to achieve suitable robot behaviour.

2.4 Visualisation Tool

The application examples presented in this thesis have been implemented using the software framework MCA2-KL[2]. MCA2-KL is a branch of the C++-based software framework MCA2 originally developed at the Research Center for Information Technology (FZI) in Karlsruhe (see [Scholl 01] for an early version of MCA2). Since 2003, MCA2-KL has been developed at the Robotics Research Lab (RRLab) of the Departement of Computer Science at the University of Kaiserslautern.

FINROC, also developed at the RRLab, is the successor of MCA2-KL. It is downward compatible to MCA2. Details about this framework can be found in [Reichardt 13]. FINROC features two graphical tools, FINGUI and FINSTRUCT. The FINGUI is mainly used to control a robot during its application. It can be configured for the use of different types of robots and provides several screens to supervise the robot in its real/simulated environment, visualise sensor data and type in control data. FINSTRUCT provides a visualisation of the network, the modules, groups and their interconnections. It offers the possibility to follow

[2]MCA2-KL: Modular Controller Architecture Version 2 - Kaiserslautern Branch (see `http://rrlib.cs.uni-kl.de/mca-kl/`)

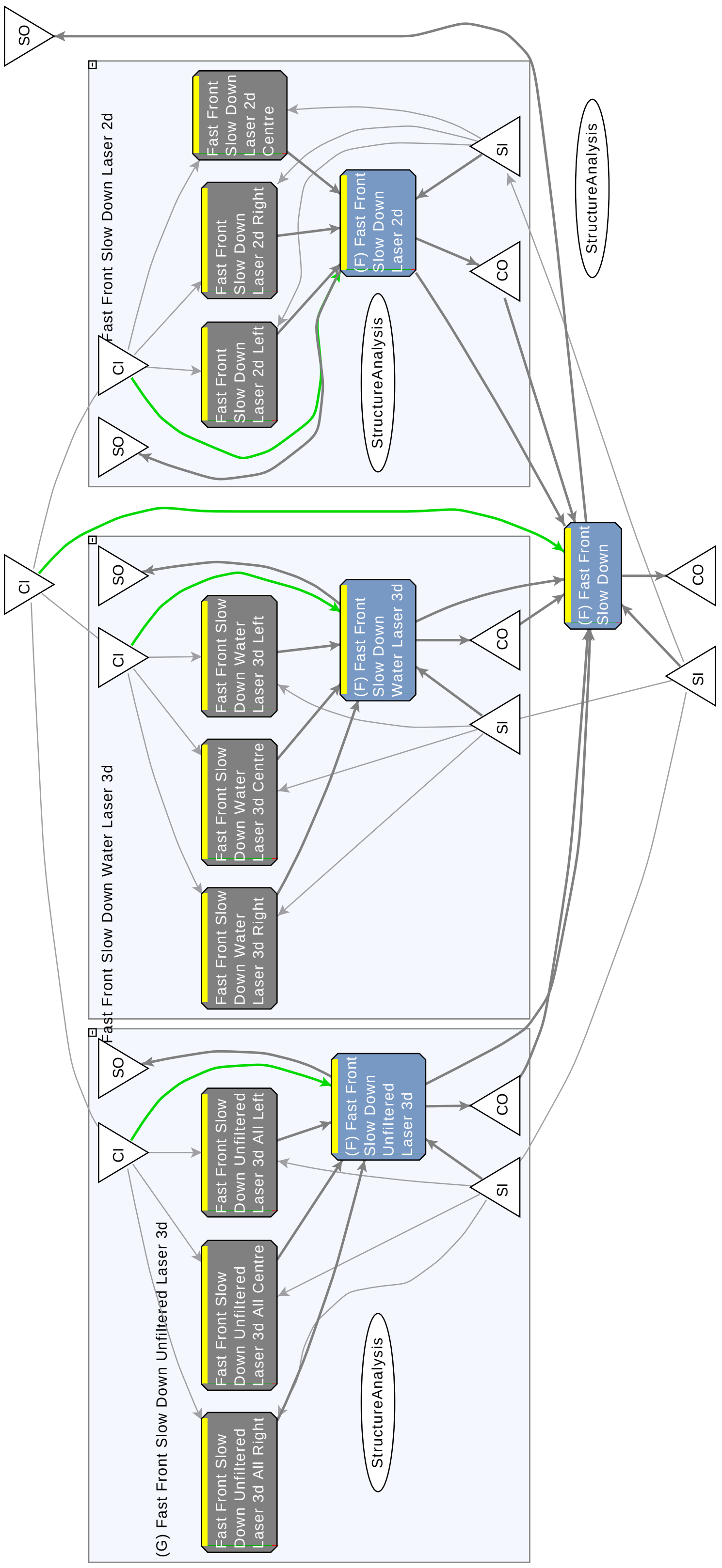
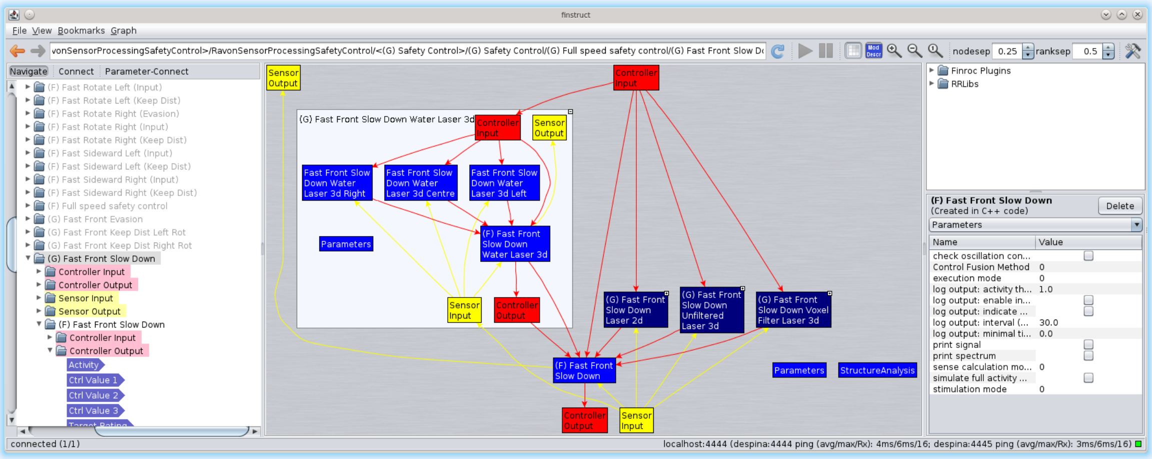

SO
Fast Front Slow Down Laser 2d
Fast Front Slow Down Laser 2d Centre
Fast Front Slow Down Laser 2d Right
Fast Front Slow Down Laser 2d Left
(F) Fast Front Slow Down Laser 2d
SI
CO
CI
SO
StructureAnalysis
StructureAnalysis
CI
Fast Front Slow Down Water Laser 3d
SO
CI
Fast Front Slow Down Water Laser 3d Left
Fast Front Slow Down Water Laser 3d Centre
Fast Front Slow Down Water Laser 3d Right
(F) Fast Front Slow Down Water Laser 3d
CO
SI
(F) Fast Front Slow Down
CO
SI
(G) Fast Front Slow Down Unfiltered Laser 3d
SO
CI
Fast Front Slow Down Unfiltered Laser 3d All Left
Fast Front Slow Down Unfiltered Laser 3d All Centre
Fast Front Slow Down Unfiltered Laser 3d All Right
(F) Fast Front Slow Down Unfiltered Laser 3d
CO
SI
StructureAnalysis

Figure 2.10: Part of the group *(G) Fast Front Slow Down* (Fig. 2.7) of the safety control of RAVON. Every group contains three safety behaviours (grey octagons), one for each sector that reacts to front obstacles (right, centre, left) and a fusion behaviour (blue octagons). Green arrows indicate a stimulation transfer, grey arrows depict miscellaneous data.

data flowing through the network during a robots action as well as to manipulate data. This tool is especially very useful for the manual inspection of the correct connections of behavioural components.

For the work at hand, the FINSTRUCT tool is used together with examples of behaviour-based control networks running under MCA2-KL. As the tool is extended in the scope of this work to support the proposed verification methods, it is described in more detail in the following.


finstruct
File View Bookmarks Graph
vonSensorProcessingSafetyControl>/RavonSensorProcessingSafetyControl/<(G) Safety Control>/(G) Safety Control/(G) Full speed safety control/(G) Fast Front Slow D
nodesep 0.25 ranksep 0.5
Navigate Connect Parameter-Connect
(F) Fast Rotate Left (Input)
(F) Fast Rotate Left (Keep Dist)
(F) Fast Rotate Right (Evasion)
(F) Fast Rotate Right (Input)
(F) Fast Rotate Right (Keep Dist)
(F) Fast Sideward Left (Input)
(F) Fast Sideward Left (Keep Dist)
(F) Fast Sideward Right (Input)
(F) Fast Sideward Right (Keep Dist)
(F) Full speed safety control
(G) Fast Front Evasion
(G) Fast Front Keep Dist Left Rot
(G) Fast Front Keep Dist Right Rot
(G) Fast Front Slow Down
Controller Input
Controller Output
Sensor Input
Sensor Output
(F) Fast Front Slow Down
Controller Input
Controller Output
Activity
Ctrl Value 1
Ctrl Value 2
Ctrl Value 3
Sensor Output
Controller Input
(G) Fast Front Slow Down Water Laser 3d
Controller Input
Sensor Output
Fast Front Slow Down Water Laser 3d Right
Fast Front Slow Down Water Laser 3d Centre
Fast Front Slow Down Water Laser 3d Left
Parameters
(F) Fast Front Slow Down Water Laser 3d
Sensor Input
Controller Output
(G) Fast Front Slow Down Laser 2d
(G) Fast Front Slow Down Unfiltered Laser 3d
(G) Fast Front Slow Down Voxel Filter Laser 3d
(F) Fast Front Slow Down
Parameters
StructureAnalysis
Controller Output
Sensor Input
Finroc Plugins
RRLibs
Delete
connected (1/1)
localhost:4444 (despina:4444 ping (avg/max/Rx): 4ms/6ms/16; despina:4445 ping (avg/max/Rx): 3ms/6ms/16)

(F) Fast Front Slow Down
(Created in C++ code)

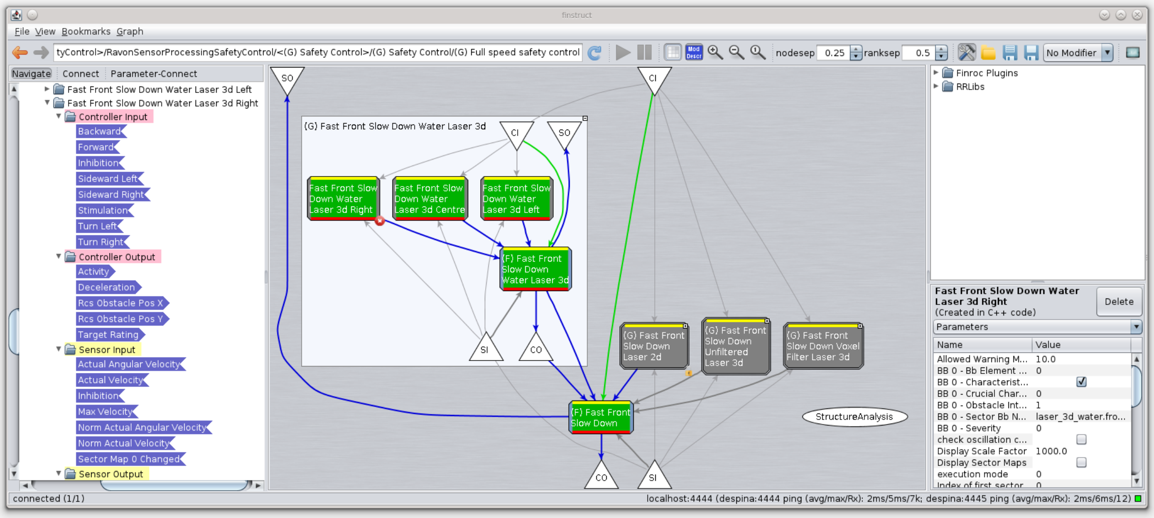
Parameters

Name	Value
check oscillation con...	☐
Control Fusion Method	0
execution mode	0
log output: activity th...	1.0
log output: enable in...	☐
log output: indicate ...	☐
log output: interval (...	30.0
log output: minimal ti...	0.0
print signal	☐
print spectrum	☐
sense calculation mo...	0
simulate full activity ...	☐
stimulation mode	0

Figure 2.11: The standard graph view in FINSTRUCT. The main window in the middle shows the graph with incoming and outgoing ports (yellow and red boxes) of the surrounding group. The network consists of four groups, coloured in dark blue, where the left most one is expanded to see the contained components. The left bar shows a navigation tree including the ports associated to the selected module/group. The right bar lists the parameters and input/output ports of the selected module/group and offers the possibility to change the values.

Figure 2.11 shows a screenshot of the standard view of the tool. The main window shows the network consisting of the components and their connections. The left column provides a navigation tree additionally displaying the signals a component sends or receives via its *ports*. The view also enables to graphically add further components and connections in the main window. The right column gives information about the data running through the ports and the parameters of a component. It is possible to manipulate the data to see the influence of changes inside the network.

The tool features several views which enable e.g. the representation of behaviour-based networks. The basic view for the visualisation is the *IB2CView*. The difference in contrast to standard networks is the visualisation of behavioural components, in particular the picturing of the behavioural signals as bars indicating the level of activation, activity and target rating. Basic behavioural components are depicted as grey octagons, fusion behaviours with octagons in three different types of blue (depending on the type of fusion) and non-behaviour modules as white ellipses. Behavioural groups are visualised as grey octagons with a double line as boundary. Edges between behavioural components are coloured according to the type of connection. Green edges visualise a stimulation transfer, red edges an inhibition transfer and blue edges the transfer of activity.


finstruct
File View Bookmarks Graph
tyControl>/RavonSensorProcessingSafetyControl/<(G) Safety Control>/(G) Safety Control/(G) Full speed safety control
nodesep 0.25 ranksep 0.5
No Modifier
Navigate Connect Parameter-Connect
Fast Front Slow Down Water Laser 3d Left
Fast Front Slow Down Water Laser 3d Right
Controller Input
Backward
Forward
Inhibition
Sideward Left
Sideward Right
Stimulation
Turn Left
Turn Right
Controller Output
Activity
Deceleration
Rcs Obstacle Pos X
Rcs Obstacle Pos Y
Target Rating
Sensor Input
Actual Angular Velocity
Actual Velocity
Inhibition
Max Velocity
Norm Actual Angular Velocity
Norm Actual Velocity
Sector Map 0 Changed
Sensor Output
SO
CI
(G) Fast Front Slow Down Water Laser 3d
CI
SO
Fast Front Slow Down Water Laser 3d Right
Fast Front Slow Down Water Laser 3d Centre
Fast Front Slow Down Water Laser 3d Left
(F) Fast Front Slow Down Water Laser 3d
SI
CO
(G) Fast Front Slow Down Laser 2d
(G) Fast Front Slow Down Unfiltered Laser 3d
(G) Fast Front Slow Down Voxel Filter Laser 3d
StructureAnalysis
(F) Fast Front Slow Down
CO
SI
Finroc Plugins
RRLibs
connected (1/1)
localhost:4444 (despina:4444 ping (avg/max/Rx): 2ms/5ms/7k; despina:4445 ping (avg/max/Rx): 2ms/6ms/12)

Fast Front Slow Down Water Laser 3d Right
(Created in C++ code)

Parameters

Name	Value
Allowed Warning M...	10.0
BB 0 - Bb Element ...	0
BB 0 - Characterist...	✓
BB 0 - Crucial Char...	0
BB 0 - Obstacle Int...	1
BB 0 - Sector Bb N...	laser_3d_water.fro...
BB 0 - Severity	0
check oscillation c...	☐
Display Scale Factor	1000.0
Display Sector Maps	☐
execution mode	0
Index of first sector	0

Figure 2.12: Screenshot of the *VerificationView* of FINSTRUCT. The network is the same as pictured in Fig. 2.11. Behavioural components are depicted with a special style, where the bars indicate the level of activation (yellow), activity (green), and target rating (red). Fusion behaviours are filled blue. Standard behavioural components are represented as ellipses.

Derived from the *IB2CView* is the *VerificationView* (see Fig. 2.12). It offers functionality for the analysis and verification of behaviour networks. On the one hand, automated processes that support the network verification can be initiated from within this view. As example, the creation of the formal model out of the network is supported as well as the graphical input of properties for the model checker (*Query Development User Interface* [Ropertz 12]). On the other hand, it allows for temporary changes to the network which lead to a clearer arrangement of the network. For example, the masking of behavioural components or the abstraction of the content of behavioural groups reduces the number of displayed components.

The component *StructureAnalysis*, which is found in almost all screenshots in this work is an auxiliary module, which enables the creation of different representations of the networks outside the tool, e.g. in the DOT language[3] or as a UPPAAL model (see Sec. 5.1.1) with the possibility to control the creation via several parameters. In the diploma thesis of the author of this work [Wilhelm 08], the basics of this functionality is explained.

Another view derived from the *VerificationView* is the *TraceView* developed by [Rohr 12] as part of his master's project work. It offers the possibility to visualise traces given by UPPAAL's verifier by showing the activation, activity and target rating of all behaviours in the displayed network for every step of the trace. This view is very helpful for the application of model checking, as it automatically maps a given trace, which is based on the formal model of the system back to the original network structure.

Both views have been developed in the beginning of the work on this thesis in collaboration with students and other group members. They have been consequently extended later on to enable the use of the newly presented concepts in a user-friendly environment.

[3]DOT-language: `http://www.graphviz.org/content/dot-language`

3. Analysis and Verification of Complex Systems

Robotic systems interact with their environment which implicates potential risks for the environment, humans or the robot itself. According to [Dhillon 15] a study reported 12% to 17% of the accidents in industries were concerned with automated production equipment. The need for methods that verify the systems to be safe and reliable is obvious. There exist a great number of methods to analyse either safety or reliability issues, or both. Examples for the latter are *Failure Mode and Effects Analysis* (FMEA) and *Fault Tree Analysis* (FTA). An additional classification can be made for methods analysing reliability and safety for hardware or for software, where it can be said, that several methods have been adapted from the field of hardware analysis to software analysis. As robotic systems, like all embedded systems in general, heavily depend on the combination of hardware and software, combinations of these methods are needed that are able to analyse the complete system. At the current point in time, there exist numerous approaches, that adapt existing techniques to special robotic problems. There is no mandatory direction which techniques should be used and how the result should look like, but in dependence on several standards and directives from other domains arises a general work flow and a number of useful analysis techniques to transfer to the autonomous robot domain. An example is the approach of [Zeller 14], who developed a concept for a cross-domain safety analysis process, which is based on the requirements of safety standards of different domains (e.g. automotive, nuclear power plants, railway). The process starts with a hazard analysis and ends with a certification about the safety state of the system.

Safety and *reliability* requirements heavily rely on each other. As example an autonomous driving robot is considered which shall avoid collisions with obstacles (safety). If the sensors of the robot do not work reliable, possibly, the safety requirement cannot be met. Another example can be taken from the area of elderly care. A non-reliable robot may cause serious harm to the aged people, e.g. if it does not remind the person to take his medicine. In the context of this thesis, the focus is on safety analysis, but, as can be seen from the examples, reliability issues are inevitable taken into account.

Safety analysis has two aspects. One is the definition of *safety requirements* in early development phases. Based on the consequences of a determined safety hazard, the potential danger is determined and the importance of the corresponding safety requirement is derived. This process is called *hazard analysis*. The second aspect is the verification whether the implemented system meets the previously defined requirements. Techniques like FMEA and FTA can be used for both, where in the latter case, the analysis can be based on the actual implementation details. Here, a further task is to detect the sources of requirement violations and eliminate them.

Several research is done for the identification and specification of safety requirements which can be used for the development of behaviour-based systems as well [Allenby 01, Bush 05, Firesmith 07, de Assis 16]. The work at hand concentrates on the analysis in late development phases, as the final behaviour-based system differs to traditional monolithic approaches and therefore needs special handling for analysis and verification tasks.

A well-known method to detect failures in an implemented system is *testing*. Testing is a dynamic analysis technique where the system is executed under certain conditions. Test cases can show violations of the previously defined requirements. There exist a big number of test techniques that aim at the generation of representative, little redundant and economical test cases. A commonly used distinction of testing techniques are *black box* tests and *white box* test. The latter is based on the program structure, while black box testing neglects the implementation details. Testing is an important verification method as dynamic aspects of the system are considered additionally. The data generated during the execution of test cases provide valuable information about the system. Hazard analysis methods and testing have in common, that they can show the existence of failures but not their absence. The benefit of these analysis techniques is highly dependable on the involved people and their expert knowledge.

In contrast to that *formal verification* proves a system to be correct for a given formal specification. To achieve this, formal mathematical methods are used. According to [Liggesmeyer 09], formal verification is not very popular in contrast to testing or reviewing techniques. This might be rooted in the needed expert knowledge and the effort to transform the system into a verifiable model. Formal verification is often based on an abstract system model and a formal system specification. The creation of the abstract model depends on the specific formal verification method and it can be difficult to meet the required properties while leaving all relevant system information in the model. The results refer to the abstract model, which points out the importance of a correct model. In the case of a negative outcome, meaning the property is not fulfilled for the model, another difficulty appears. The result must be interpreted on the real system in order to eliminate the problem.

It is obvious that every method has their own strengths and weaknesses and it appears to be useful to combine several different analysis methods in order to get a meaningful result. Summing up, the topic of this thesis is on the verification of safety requirements on robotic systems using a behaviour-based control, which is not extensively discussed in literature up to now. The special network structure of behaviour-based control systems allows the combination of function-based analysis techniques with software (code) analysis techniques, which make it a fascinating research area. Due to the different approaches of the mentioned techniques, several methods will be examined for their applicability to behaviour-based systems.

The following sections give an overview of the main techniques in the areas of hazard analysis, dynamic analysis techniques and formal verification. Section 3.1 presents three different hazard analysis methods. Dynamic analysis methods like testing are discussed in Sec. 3.2. Section 3.3 presents two formal verification techniques and their application to robotic systems. Most of the techniques are widely spread in the area of software and hardware analysis, respectively. Their application on robotic systems is not frequently published, but the examples show that the methods are definitely useful in this area. Section 3.4 investigates research activities in the special field of behaviour-based robotic systems dealing with analysis and verification tasks. The presented methods are summed up in Section 3.5 together with a discussion of their capabilities in the scope of this thesis.

3.1 Hazard Analysis

The investigation of possible hazards is necessary to learn about the situations the system could face and to determine the risk level of a system. Hazard analysis usually includes three steps, namely the identification of hazards, the determination of the criticality level and the resultant safety requirements. Hazard analysis is often applied in the early development stages, in order to avoid hazards from the beginning, for example by adding redundancy or surveillance modules. But also in later development phases, the analysis is useful to identify hazards on the basis of a more detailed functional description of the system. Before a hazard analysis can take place, the system requirements and functionality must be described. This includes the definition of intended use, the tasks that the system should fulfil and how they are fulfilled, and all physical and functional characteristics.

Several analysis techniques exist that are widely used in industry:

- Failure Modes and Effects Analysis FMEA
- Hazard and Operability studies HAZOP
- Fault Tree Analysis FTA
- Event Tree Analysis ETA

They have in common, that they are usually realised in several team meetings, where people with different backgrounds (analysis experts, system developers, hardware developers) try to identify preferably all hazards with the help of suitable guidelines. For each identified hazard its effect(s) on the system and the environment are determined and its risk level is established. The risk is first estimated in terms of severity and probability of occurrence and then evaluated whether it is acceptable or not. In the latter case, the risk must be reduced and the process starts again. Otherwise, a safety requirement is devised. Figure 3.1 depicts the process recommended in [ISO/IEC-Guide51 99] and adapted by [Hoang 12].

The above mentioned analysis techniques are standard procedures in the safety requirements specification procedure for several industrial processes and machines. They were also applied to industrial manipulator robot arms that work in a structured environment. A problem in this process is the transfer of the methods previously used in system engineering to the field of safety engineering. [Denger 08] state, that the techniques are often not customised to the special characteristics of software and therefore miss conceptual

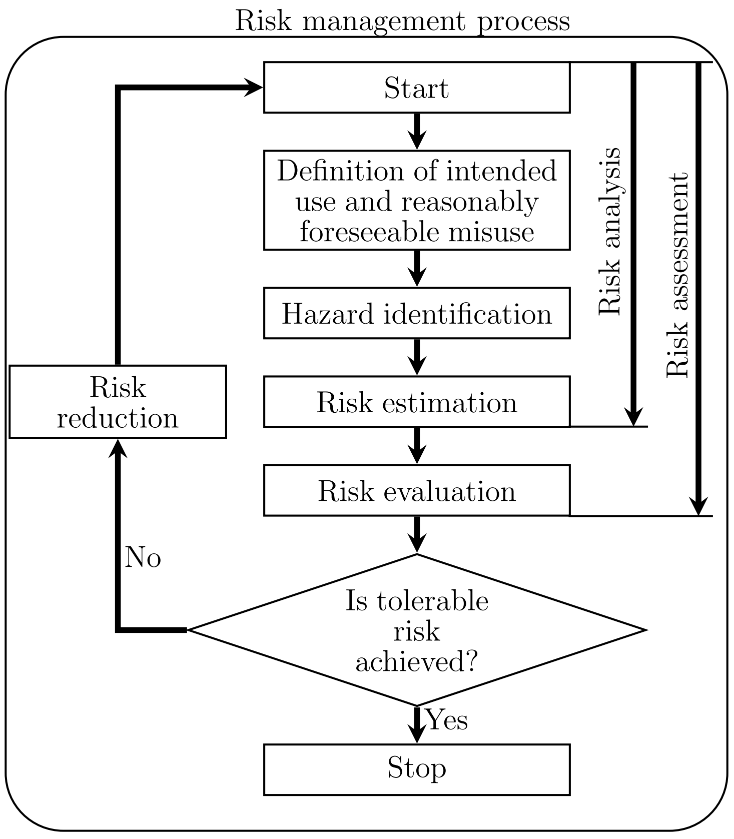


Figure 3.1: Safety analysis process

software faults. Another big problem is encountered considering the application of these techniques to autonomously working robots or robots with human/environment interaction. A complete description of the robots functionality and behaviour is not possible, as it is infeasible to describe all situations the robot can face. Current research deals with the adaptation of the method to the autonomous mobile robotic domain. The following subsections give a short introduction into the functionality of HAZOP, FMEA, and FTA and some examples for their successful application to robotic systems.

3.1.1 Hazard and Operability Study (HAZOP)

The *Hazard and operability* (HAZOP) study aims at identifying risk to personnel or equipment, or operability problems using a systematic and structured approach. The HAZOP study is executed by a team including a HAZOP specialist, system designers and users. This goes back to the idea that several experts with different background identify more problems when working together instead of working separately. As mentioned before, a precondition for successful hazard analysis is the proper definition of the system and the intention, how the system is expected to work. Then, possible deviations and their causes and consequences can be analysed. To structure this process a list of guide words is used which is adapted according to the specific system (see Tab. 3.1 for some example guide words).

Table 3.1: Example for guide words in HAZOP analysis (source: [hazop01 01], adapted)

Guide Word	**Meaning**
NO	No part of the intention is achieved
MORE	A quantitative increase
LESS	A quantitative decrease
AS WELL AS	Impurities present
...	...
BEFORE	Something happens too early in a sequence
AFTER	Something happens too late in a sequence

The HAZOP study was originally developed for chemical process systems and later extended to analyse complex operations in industry e.g. in nuclear power plants. In 1994, [McDermid 94] introduced a method for software safety analysis based on the HAZOP technique and discovered a wide-spread application including other HAZOP-based software analysis approaches, one year later [McDermid 95]. In recent years, several research is done using the method for the identification of hazards in robotic systems. Especially for autonomous robotic systems, the original approach needs to be extended to cover non-deterministic behaviour, e.g. robot interaction with humans or environment.

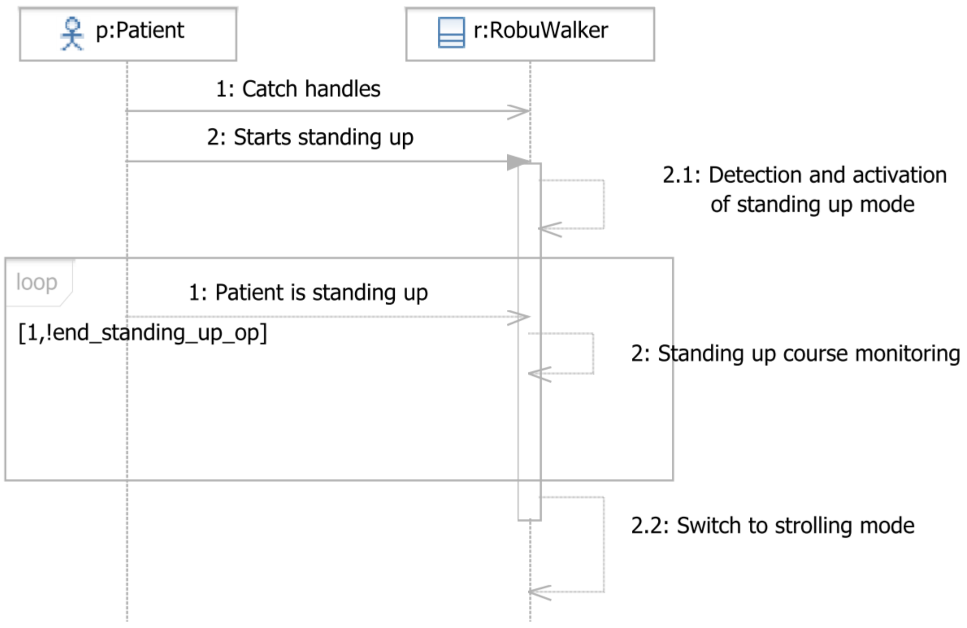

p:Patient
r:RobuWalker
1: Catch handles
2: Starts standing up
2.1: Detection and activation of standing up mode
loop
1: Patient is standing up
[1,!end_standing_up_op]
2: Standing up course monitoring
2.2: Switch to strolling mode

Figure 3.2: Example sequence diagram for the *standing up* operation of a patient using a robotic strolling assistant (source: [Hoang 12])

[Martin-Guillerez 10, Hoang 12, Guiochet 13] developed a method that uses UML diagrams to describe a system, its actions, and interactions with the environment. An example diagram is given in Fig. 3.2, which describes the use case where the robotic strolling assistant MIRAS should assist a patient in standing up by lifting its handgrips. The patient catches the handles of the robot and starts standing up. The robot detects this action and activates the standing up mode. The action is monitored until the patient stands, then the robot switches to strolling mode. In order to use the HAZOP analysis for the detection of possible problems and the determination of new safety requirements, the authors adapted and extended the list of standard keywords to express states in the diagrams.

An excerpt of an HAZOP table is shown in Fig. 3.3. Here, the necessary precondition for the standing up operation, *sufficient battery power*, is analysed. The authors found two deviations. One is a too low battery charge which can lead to a fall of the patient. They classify this event as serious, declare the possible causes and derive a new safety requirement, which must be integrated in the implementation process. The second deviation uses the key word *other than*, which implies that not the expected action is triggered. The authors classify this event as negligible as in the worst case the patient would be confused about the robots action. No new safety requirement is derived for this case.

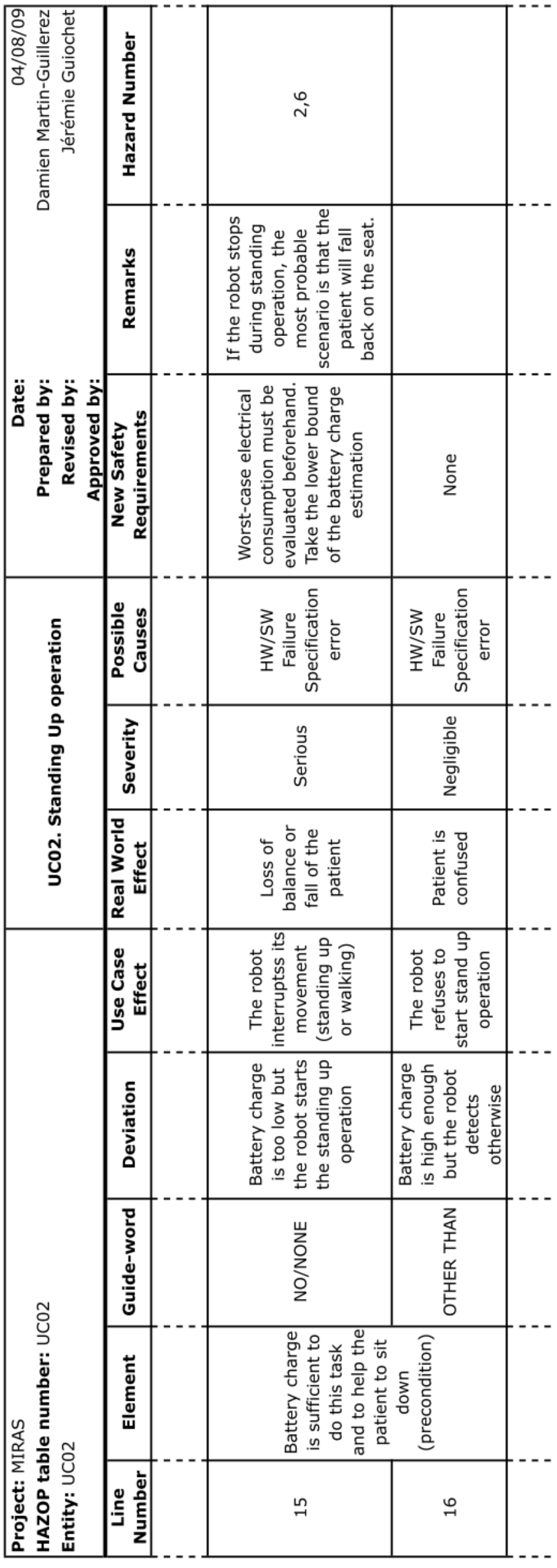
With their approach, the authors present a method to analyse the behaviour of a robot with the support of the graphical representation of diagrams and keywords corresponding to these diagrams. But, a common problem of HAZOP analysis that exists for all similar hazard analysis techniques is not addressed: The problem of a full coverage of robot actions. [Böhm 10] present a systematic approach for guaranteeing a good coverage of actions that occur during robots task execution. Their concept is based on a division of the HAZOP analysis into two parts, a *components view* and a *operations view*. The first refers to the block structure of system hardware, whereas the second is related with the task scenarios. The decomposition allows a more specific analysis of the different system parts, but it fails to integrate environmental conditions and interactions.

Both methods are suitable for the derivation of safety requirements for behaviour-based robotic systems as well. The concrete software structure of the control system is out of the scope of HAZOP analysis. It will therefore not be further treated in this thesis.

3.1.2 Fault Tree Analysis (FTA)

Fault Tree Analysis (FTA) was developed in the early 1960s at the Bell Telephone Laboratories for performing reliability analysis in aeronautics. It was then applied to the planning process of nuclear power plants and later used in the automotive industry. 1981, [Vesely 81] summarised a set of undocumented methods of fault tree creation and evaluation in a handbook. Over the years, sophisticated evaluation algorithms as well as tools for the construction of fault trees and their evaluation are developed. Today, fault tree analysis is widely used to perform reliability and safety analysis in mechanical systems.

Fault tree analysis is a top-down approach starting with an undesirable event, called the *top event*. In the analysis process, fault events that can lead to the top event are determined recursively and connected by logic operators such as OR and AND. The process stops when a fault event can not be developed any further. These elements are called *basic events*. The tree structure can be used for a qualitative analysis, determining the sets of basic events that together trigger the top event. This information can be used to improve the systems reliability by eliminating (at least) one of the causes. If it is possible to assign fault probabilities to the basic event, a quantitative analysis can be executed, calculating the probability of the occurrence of the top-event. This information allows the developer to focus on the most problematic parts of the system.



Project: MIRAS
HAZOP table number: UC02
Entity: UC02

UC02. Standing Up operation

Date: 04/08/09
Prepared by: Damien Martin-Guillerez
Revised by: Jérémie Guiochet
Approved by:

Line Number	Element	Guide-word	Deviation	Use Case Effect	Real World Effect	Severity	Possible Causes	New Safety Requirements	Remarks	Hazard Number
15	Battery charge is sufficient to do this task and to help the patient to sit down (precondition)	NO/NONE	Battery charge is too low but the robot starts the standing up operation	The robot interruptss its movement (standing up or walking)	Loss of balance or fall of the patient	Serious	HW/SW Failure Specification error	Worst-case electrical consumption must be evaluated beforehand. Take the lower bound of the battery charge estimation	If the robot stops during standing operation, the most probable scenario is that the patient will fall back on the seat.	2,6
16		OTHER THAN	Battery charge is high enough but the robot detects otherwise	The robot refuses to start stand up operation	Patient is confused	Negligible	HW/SW Failure Specification error	None		

Figure 3.3: HAZOP table example based on UML models for the robotic strolling assistant MIRAS (source: [Hoang 12])

20 years ago, FTA was mainly used for industrial robotic systems such as manipulators, which act in a structured and predefined environment [Walker 96, Khodabandehloo 96]. Software faults were not considered in the standard fault tree analysis, which is essential when talking about embedded systems as software faults can cause hardware to crash or cause accidents. Therefore, several research has been done that attempts to solve the open problems while benefiting from the advantages of FTA.

[Leveson 83] describe software fault tree analysis as a complement of the classic FTA method. The classic FTA is executed and failure events that are produced by software are marked as basic events. Those basic events are subsequently analysed with the so-called *software* FTA, which backtracks the program code from the output to the input. Hereby, the values of variables are analysed, in order to find out the program inputs that lead to the failure. In the scope of this thesis, software analysis is a central aspect. It has to be figured out how the concepts of software FTA work for the special software structure of behaviour-based systems.

[Thums 04] introduces a concept called *formal* FTA. This approach integrates the classical FTA into formal methods from software engineering. Formal methods are used to analyse the functional correctness of the software system whereas the FTA is used to analyse the failure rate of the technical system. With this approach, the strength of both analysis techniques is used to prove a whole system. The approach is based on a precise formal model of the complete system including hardware and software components. The derivation of such a model is not further discussed in the work.

As embedded systems are often built up as a combination of (technical) components, [Kaiser 03] introduced the concept of *Component Fault Trees* (CFT). Here, every architectural component is mapped as CFT component, which can encapsulate further CFT's. Failures may be transferred between CFT components using in- and output ports. The advantages of this method are, that several analysts can work on different parts of the same CFT at the same time. The division into CFT components is intuitively as they correspond to real-world components. Furthermore, due to the fact, that every component is modelled only once, *duplicate* (or *repeated*) events are modelled using an instance of the component. This prevents failures by copying parts of the tree and minimises the resulting CFT. This concept is also useful for the analysis of robotic systems. Especially hardware like sensor systems, power supply or the computers are components that would appear repeatedly in a fault tree. The concept of component based fault tree creation seems to be universal. The network structure of behaviour-based systems with its division into groups seems to provide a structure that can be mapped to CFT components.

[Seward 00] uses fault tree analysis for the determination of safety requirements for the autonomous excavator LUCIE. The excavator's task will be to autonomously dig foundation trenches on a building site. The unstructured environment was identified by the authors as a major problem when creating the fault tree, as the physical interaction properties could not be described as basic events. Therefore, they suggest a preliminary analysis of the physical dynamics and a definition of excavator and environment interaction in more formal terms before the actual analysis process can start. Unfortunately, the formalisation is not described.

While the authors of [Seward 00] focus on a special use case, [Denger 08] present a customizable framework concept. It enables the developer to define and implement a project-specific hazard identification approach using standard techniques like FMEA, FTA and HAZOP. They want to overcome the problem that analysis results are often not repeatable because of the manual and expert-dependent nature of these methods. The authors of [Guo 10] identified a further safety critical problem for embedded systems. Security attacks can have safety consequences and should therefore be considered during a safety analysis. The authors present an approach that provides one identical model derived by safety analysis techniques that can be used to derive security-safety requirements. The approach is applied to the autonomous outdoor robot RAVON, where the effects of safety failures and security attacks on its bumper system are determined. This approach uses a behaviour-based control system as application example, but the authors only describe the procedure to combine a safety fault tree and a security attack tree. They neglect the process of deriving the two trees.

Fault tree analysis is often combined with *Event Tree Analysis* (ETA). ETA is a bottom-up technique aiming at the opposite direction as FTA. It starts with an failure event and identifies all events that are caused by that initial event. The resulting tree can be used for assessing probabilities of the outcomes. ETA is used for example to analyse the correctness of safety monitoring systems.

3.1.3 Failure Modes, Effects and Criticality Analysis (FMECA)

The *Failure Mode, Effects and Criticality Analysis* (FMECA) is an extension of the *Failure Mode and Effects Analysis* (FMEA) including a criticality analysis. FMECA is used to identify *failure modes* that may seriously affect the product quality. Failure modes describe the way a component fails. The *effects* of failure modes are defined and classified with regard to their criticality, where criticality is a combination of the probability of a failure mode and the severity of the consequences. In contrast to FTA, FMECA is a forward analysis, starting from failures in basic components and determining the effects on the whole system.

The first step of FMECA is the definition of the system, which requires detailed expert knowledge. The system is usually represented as block diagrams for the functional structure and as automata for the description of system dynamics. In a further step, possible failure modes are identified. The effects of the failure modes are analysed and a description how the failure can be detected and (if possible) eliminated is added. Additionally, the criticality of the failure mode is calculated.

An example for an FMECA table is given in Tab. 3.2. It describes two failure modes of an artificial muscle. The *criticality* or *risk estimation* is divided into three parts. The *occurrence* of the failure mode is rated on a scale with five levels from I (improbable) to F (frequent). The *severity* value correlates with the fact that the failure effect can have a high level (1) or a low level (3), where a high level implies serious harm or even death, a low level implies no harm. The *risk* is noted as the product of the two values. Following actions include the ranking of the identified failure modes and the assignment of corrective actions in connection with a re-calculation of the criticality factor.

FMECA is a widely spread technique, which is comprehensible and applicable to many different systems and purposes. It was originally developed by the U.S. military and later modified by the U.S. National Aeronautics and Space Administration (NASA). Today, the method is commonly used in the automotive industry, for the analysis of medical devices

Table 3.2: FMECA table for a mechanical unit (source: [Guiochet 02], adapted)

Component / Function	**Failure mode**	**Failure cause**	**A. Local effect B. Effect on system**	**Risk estimation**			**A. Failure detection method B. Action required C.other**
				Occurrence	**Severity**	**Risk**	
Artificial muscle	pierced	- abrasion of inner tube - bad maintenance - bad use	A. decrease of air pressure B. Increase of length, decrease of pressure on the patient	F	3	F3	A. pressure sensor B. cut off the air pressure
	breaking	wear of muscle outer cover	A. no physical constraints B. patient struck by a moving part	I	1	I1	A. length sensor B. protection devise around muscles or emergency stop

and robotic systems, among others [Chen 12, Onofrio 15, Kazanzides 08, Guiochet 02]. Despite of the advantages of the method, there exist some limitations that are topic of current research. These are for example, the combination of failures, the inclusion of software parts and the integration of human interaction or, more generally, environmental factors. [Guiochet 02] cope with the problem of human interaction and present an approach to analyse the risk of human errors in the context of a medical robot. They propose UML-diagrams for the description of human-machine-interaction, which serve as basis for the failure mode analysis. They also discuss the problem of transferring the method to software. They claim that it is not possible to decide about the risk of a software error and identify two approaches to handle this problem, namely to either consider the software as a component and just analyse its inputs and outputs, or to identify failure modes based on class and sequence diagrams.

FMECA and FTA are used to determine hazards in a system, whereby they are based on completely different approaches. FMECA is applicable to systems with well-known behaviour like hardware components, where a certain risk level can be determined. FTA is able to handle the relation of failures and can also be applied to software. A combination of both methods is useful to overcome the shortcomings and maximise the output [Bluvband 05].

3.2 Dynamic Analysis of Safety Requirements

The term *dynamic analysis* is broadly used but not exactly defined in literature. In the scope of this work, dynamic analysis is defined, in contrast to static analysis, as an analysis process executed during run-time of the system. It is an important part of the system validation process as dynamic actions can not completely be covered by static analyses. Especially environmental aspects like unknown and changing environments or human-interaction can not be captured by system models in detail. They can only be considered during execution.

Two different approaches are presented in the following sections. A well-known technique to prove the systems correctness is testing. Here, the system is executed under different conditions and the systems behaviour is evaluated in correspondence to some expected behaviour. The special requirements on this method for the use with robotic systems is discussed on the example of current literature in Sec. 3.2.1.

Section 3.2.2 deals with several possibilities to implement safety requirements in the system, which shall enable the robot to act safely in all situations. This approach is different to the others presented in this chapter, as it is not applied to the system during the development process in order to find failures and eliminate them but rather is a part of the final control system. The system must be validated itself. Nevertheless, the approach is discussed here, as the iB2C architecture used in this work has a similar safety driven structure, and techniques to validate and verify it are proposed.

3.2.1 Testing

Testing is a commonly used technique to verify and validate a system, applied in different stages of the development. In dependence on the development phase and previous tests, various test methods are distinguished. *Unit testing* validates the single components of a system in respect to their functionality, whereas *system integration testing* validates their interconnection. The validation of system requirements is done in the *system validation test* [Woodman 13]. The special conditions of embedded systems (e.g. hardware-software combination, mixture of physical components, interaction with humans and environment) must be addressed with tailored test methods (e.g. [Saini 12, Tatar 14]). In early stages of development not all components may be available to test. For this, *component-in-the-loop* testing, where parts of the system are simulated, is a useful technique.

One major weakness of testing is the fact, that it is almost never possible to cover the whole set of possible scenarios the system could face. This would be either to time- and cost-expensive or just impossible, e.g. for robotic systems working in dynamic environments or interacting with humans. Several standards exist that formalise the testing approach in order to define guidelines for manufacturers which they have to follow before they are allowed to bring their product to the market. For example, the IEEE standard 829 describes how the documentation of software tests should be done. However, robotic applications have special requirements regarding sufficient testing. A suite of test methods for remotely controlled robots is standardised through the ASTM International Standards Committee on Homeland Security Applications; Operations Equipment; Robots ($E54.08.01$)[1]. Up to now, there are no special standards that define test criteria for autonomous mobile robots or human-robot-interaction.

[1]Guide for Evaluating, Purchasing, and Training with Response Robots using DHS-NIST-ASTM International Standard Test Methods `http://www.nist.gov/el/isd/ms/upload/DHS_NIST_ASTM_Robot_Test_Methods-2.pdf`

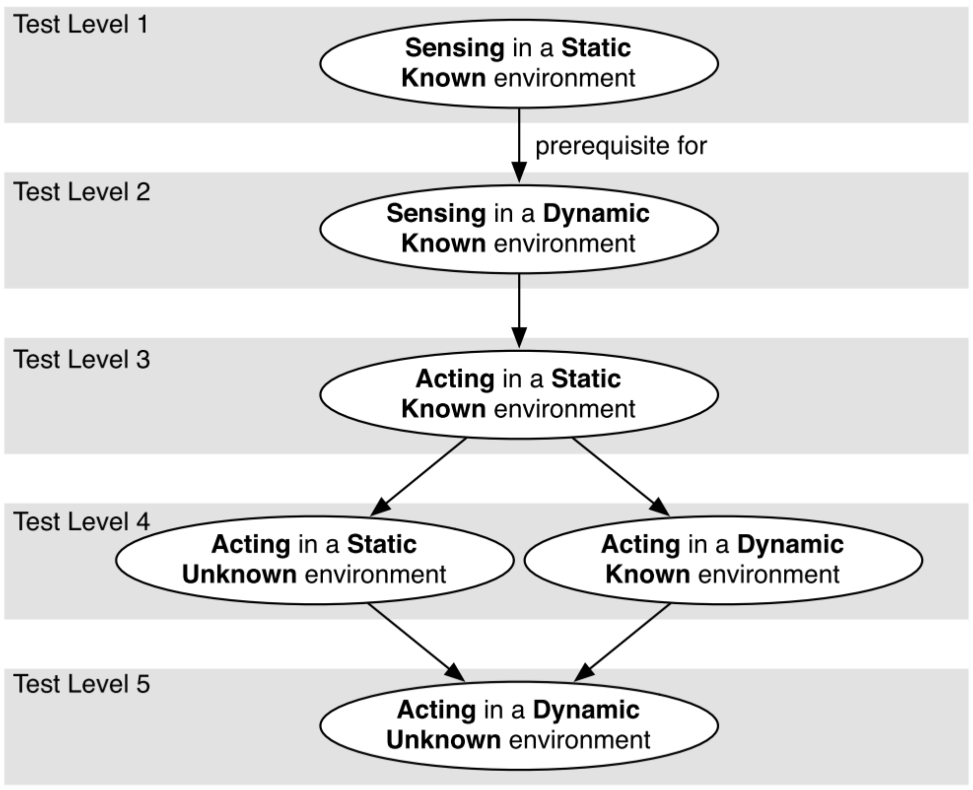

Test Level 1
Sensing in a **Static Known** environment
prerequisite for
Test Level 2
Sensing in a **Dynamic Known** environment
Test Level 3
Acting in a **Static Known** environment
Test Level 4
Acting in a **Static Unknown** environment
Acting in a **Dynamic Known** environment
Test Level 5
Acting in a **Dynamic Unknown** environment

Figure 3.4: Five test levels to test mobile autonomous robots (source: [Laval 13])

Several research is done in the last years to illuminate this topic under several aspects. All have in common that they see a great need for efficient robot testing methods as robotic systems enter the mass market and the topic is not sufficiently investigated at all. [Laval 13] argue that mobile autonomous robots built for the mass market need to be tested exhaustively, whereby tests should be repeatable and automated as much as possible. They present a methodology to test the robot control software on the final robot platform under several conditions. In their approach, they distinguish three test dimensions: operations (sensing vs. sensing and acting), human knowledge of the environment (precise vs. little) and the environment (static vs. dynamic). From these different dimensions, they derive a set of 5 test levels shown in Fig. 3.4. Test series should be done starting in level 1 and proceeding down to level 5 in order to assure the safety requirements. For example, the acting of a robot is usually based on perception data, therefore the correct functioning of sensing should be tested first. In the next steps, sensing and acting can be tested in combination whereby the complexity increases from static known environment to dynamic unknown environment. With their approach they introduce a structuring of the testing process of autonomous mobile robots, which helps to cover the most relevant test cases.

[Arnold 13] focus on the problem of generating concrete test scenarios to test control algorithms. They claim that test situations defined by the developers often neglect those scenarios that would reveal failures in the implementation. Therefore, they adapted an existing approach to automatically generate situations called *Procedural Content Generation* (PCG) to the needs of autonomous robot systems. Their resulting tool is able to generate situations, execute them, and prioritise the results to ease the evaluation process for the developer. In the paper, the scope of tested algorithms is limited to the question, how dangerous situations can the evoked by combinations of sensing, sensor processing, and driving algorithms, but is said to be expandable. The generated situations cover a set of different environments including rocky deserts and cave systems together with several types of (moving) obstacles. Additionally, the route the robot should take through the scenario is randomly generated. The approach offers great potential as the manual definition of test scenarios is very time-consuming and a coverage of unexpected situations is hard to achieve. The random automatic generation offers the possibility to explore also failures that would not be discovered using manually defined test scenarios. Nevertheless, the method cannot guarantee that the generated tests include all important scenarios that a human developer would have defined.

As mentioned by Arnold, the evaluation process is mostly manually done, expecting detailed system knowledge. Graphical support and (partly) automated processes can be identified as an important research topic in order to make the techniques available for many users.

3.2.2 Including Safety Requirements into the System

One approach to ensure the safety of a system is to implement mechanisms that detect safety problems during robot operation. Monitoring components determine the current situation and compare them with a set of given safety requirements. In the case of a deviation, counteractions are initiated.

Several researchers propose this approach using additional components that observe data flowing in the system during task execution and calculate the so-called *health status* of the system. [Forouher 14] provide statistics about the data flow between nodes inside the Robot Operating System (ROS) and check the data for anomalies. Detected problems are reported to the developer. Directly changing the robot control in accordance to the health status is utilised by [Schäfer 11] for the iB2C architecture. Here, the data flow is observed to check sensor data for their correctness. Monitoring components determine whether an obstacle detection facility is sane or not using thresholds. In case of a failure, they are able to reduce the maximal system velocity. A similar approach is implemented by [Maas 14] for the Organic Robot Control Architecture (ORCA). The observing modules are able to take over or change the robot control in a way that the best possible execution is achieved. Here, the authors focus on reliability but the concept seems to be adaptable to safety issues as well.

More complex approaches are often based on layered system architectures where safety requirements are proved by an additional intermediate layer. The LAAS-architecture [Ingrand 02] is build up with three layers, a decision layer which commands the next robot actions, a functional layer which controls the actuators and an execution layer in between. The execution layer compares the control commands from the decision layer with predefined rules and correct system states and decides on this basis if the command is passed to the functional layer. The correct system states are formally defined and verified using model checking techniques. In [Bensalem 09], the authors add a further validation techniques to the functional layer. This layer encapsulates each basic functionality of the robot in a module. More complex functions are achieved by a combination of several

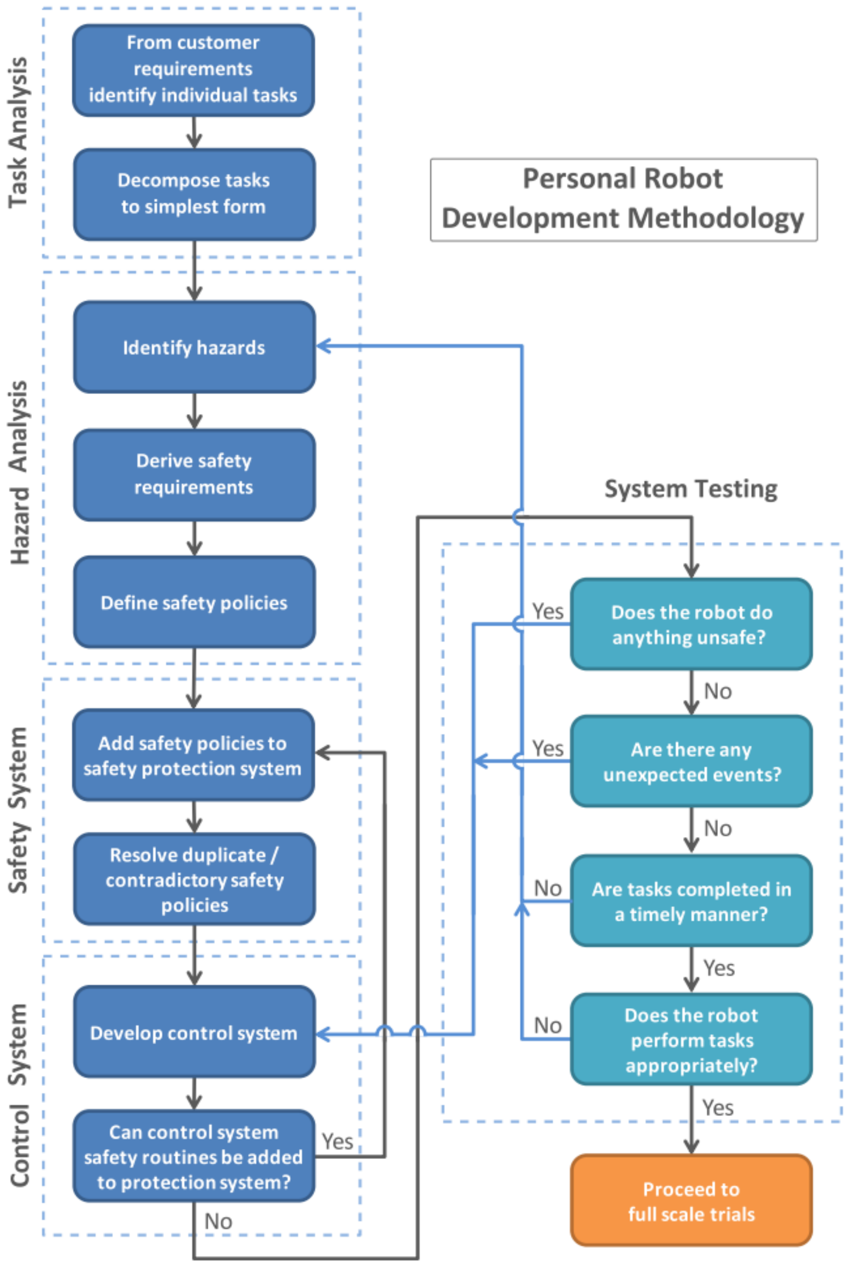

Personal Robot
Development Methodology
Task Analysis
From customer requirements identify individual tasks
Decompose tasks to simplest form
Hazard Analysis
Identify hazards
Derive safety requirements
Define safety policies
Safety System
Add safety policies to safety protection system
Resolve duplicate / contradictory safety policies
Control System
Develop control system
Can control system safety routines be added to protection system?
Yes
No
System Testing
Does the robot do anything unsafe?
Yes
No
Are there any unexpected events?
Yes
No
Are tasks completed in a timely manner?
No
Yes
Does the robot perform tasks appropriately?
No
Yes
Proceed to full scale trials

Figure 3.5: Development methodology for a safety-driven control system (source: [Woodman 12])

modules. The authors use the BIP[2] framework to model controllers on the functional level “correct by construction”, which enforce the modelled interactions during execution.

[Woodman 12] call this *layered system structure* where task execution is dependent on safety rules that are defined in the early development process and tightly integrated into the implementation process *safety-driven design.* Their goal is to integrate safety constraints into the implementation from the beginning instead of adding them afterwards. To achieve this, they consider a development methodology as pictured in Fig 3.5. The development process starts with a task analysis which is required for the following hazard analysis (see Sec. 3.1). For hazard analysis they use HAZOP in conjunction with a hazard check list which includes several aspects to unforeseen environments operations among others. Based on the hazard consequences which are classified with a risk, they define safety requirements, which again are used to form *safety policies.* Safety policies are independent rules which use perception data to decide about the system state an to impose restrictions on actuators. An example policy for the requirement, that the speed of the robots end effector is limited to $250 mm/s$ while operating at reduced speed is:

```
IF  robot_state.speed_mode = reduced_speed_control
AND end_effector_speed <= 250 mm/s
AND confidence_level >= 0.6
THEN allow actuators
ELSE restrict actuators
FINALLY return safety_rating based on confidence_level
```

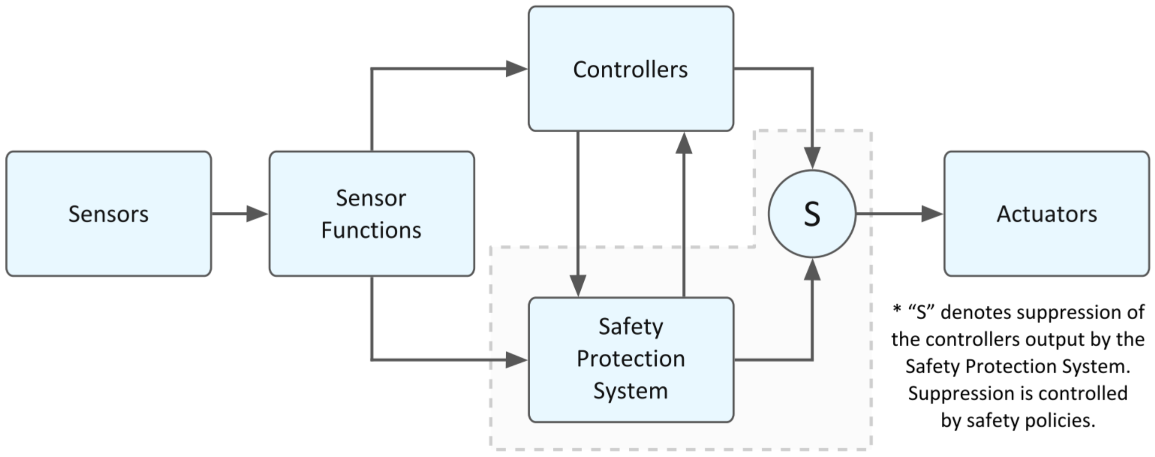

Controllers
Sensors
Sensor Functions
S
Actuators
Safety Protection System
* “S” denotes suppression of the controllers output by the Safety Protection System. Suppression is controlled by safety policies.

Figure 3.6: High-level diagram of the safety-driven control system (source: [Woodman 12])

The policies are directly implemented in the robot control software. Figure 3.6 shows the resulting safety-driven control system. This approach comprises the complete process from task analysis via system implementation up to *online* control. It benefits from the comprehensive and well-established methods used for hazard analysis and completes this technique with the derivation of easily understandable safety policies which are then implemented as a safety-protection system.

[2]BIP (Behaviour, Interaction, Priority) Component Framework

The behaviour-based architecture used in this thesis also contains a safety layer that is able to restrict or suppress commands coming from the control layer on the basis of sensor perception data. In contrast to the presented approaches, the implementation of the layer is not based on formal rules but on orally formulated requirements, which raises the need for verification and validation techniques that can be applied to the implementation.

3.3 Formal Verification

Formal verification uses mathematical methods to ensures that a required property holds on a system. In contrast to testing, which can only show the presence of failures, formal verification covers the complete system and is able to proof the absence of failures corresponding to a requirement. Formal verification is often said to be the technique that bridges the gap that testing leaves when verifying systems acting in unknown environmental conditions. The major reasons, why formal verification is not widely used in robotic research is that it requires expert knowledge for the formalization procedure and the fact that it is (currently) not possible to formally verify large complex systems. Nevertheless, several research is done in that topic as the need for having safe and correct acting robots increases. Here, the latter problem is often encountered by verifying only the safety-relevant parts of the system, while the former is solved by providing tools that automate parts of the verification procedure.

Two techniques exist for formal software verification.

Theorem proving takes a logical formula that represent the system and the required property. The formula is verified using mathematical proofs.

Model checking are algorithmic techniques used to verify the system modelled formally, e.g. as finite state automata (FSM) against its property, written in temporal logic, e.g. linear temporal logic (LTL) or computational tree logic (CTL). A formal definition of model checking is given in Sec. 5.1.

Within the context of this work model checking is chosen as formal verification technique, as it offers some advantages over other verification techniques. For example, no correctness proof has to be constructed. The checking of a given property is done automatically. Furthermore, model checking returns a counterexample in the case the specification is not fulfilled, which shows the reason for the problem. The following sections focus on related research using model checking methods.

When talking about formal verification in the development process of robotic systems, it can be distinguished between the use of formal techniques in the design process and after the development process. The first results in a “correct-by-construction” control program, which is synthesised from the previously verified model. This technique is often applied to the development of basic controllers. The advantage of this approach is that the verification process after the development phase can be shortened, because only the interaction of the components need to be verified. Examples, that successfully applied this approach to robotic systems are presented in Sec. 3.3.1.

As the behaviour-based system used in the context of the work at hand, serves as basis implementation for several different research problems, it is very important to have techniques to verify already implemented systems. Sec. 3.3.2 shows the most important examples of related work, that additionally illustrate the spectrum of different tasks and difficulties when verifying complex robotic systems.

3.3.1 Correct-By-Construction Synthesis

Synthesis of correct-by-construction controllers experiences growing interest in the robotics domain [Ames 15, Fu 15, Ulusoy 13, Espiau 95a]. At the moment, there seems to be no uniform method that defines how the system and its requirements are given and which form the result will have. The following illustrates this on two very different examples.

The authors of [Kim 05] present a case study on an home service robot which is developed and verified in an integrated framework. The controllers of the robot are implemented in *Esterel* [Berry 00] and verified using the *Esterel Toolset*. Once the program is correct against its requirements it can be automatically transferred to C Code, which can be executed on the robot.

The Esterel compiler automatically generates finite state machines from the code, which can be used by the model checker. Input for the model checker is a set of input variables and the desired output signal. For example, in order to prove the property

P1: If a user does not give a command to the robot, the robot must not move.

the developer has to feed the model checking tool with the input set {`Come_Command` = *always absent* and `STOP_Command` =*always absent*}, and the desired output that the `GO_Command` is *never emitted*. More complex properties can be verified by using observers, which can be implemented in the Esterel language as well. The approach faces some of the problems mentioned in the introduction. The toolset abstracts from the formal model and properties written in a formal language by allowing to formulate requirements as sets of variables and their desired states. However, to use this approach, developers have to learn the Esterel language. Furthermore, the synthesised program is in C code which might not be a problem for basic controllers, but may not be sufficient for higher level robot tasks.

The problem of synthesizing high-level robot control is addressed by [Kress-Gazit 11]. The authors take also the problem of dynamic environments into account and consider an approach that results in a robot control that satisfies a given specification for all valid environment behaviours. Robotic tasks are defined in linear temporal logic (LTL) as combination of propositions. The authors define three types of propositions: *Sensor propositions* abstract the environment (robot sensor perception), *location propositions* abstract the position of the robot in the workspace, and *action propositions* abstract robot actions. Examples for a sensor proposition are `RoadBlocked` or `RedLight`. `Stop` would be an action proposition which causes the robot to stop. A LTL formula representing the requirement that the robot should stop at red light would be:

$\square\ (\bigcirc\texttt{RedLight} \Rightarrow \bigcirc\texttt{Stop})$.

The synthesis process uses this formula to generate an automaton such that for every infinite run of the automaton, its behaviour satisfies the formula. As sensor propositions cannot be controlled by the robot, the synthesis process must take into account all possible values of a proposition. The synthesised automata are then implemented as a hybrid controller for the robot, which includes actual motion commands and feedback from sensor perception to initiate the next step. In order to simplify the application of their approach,

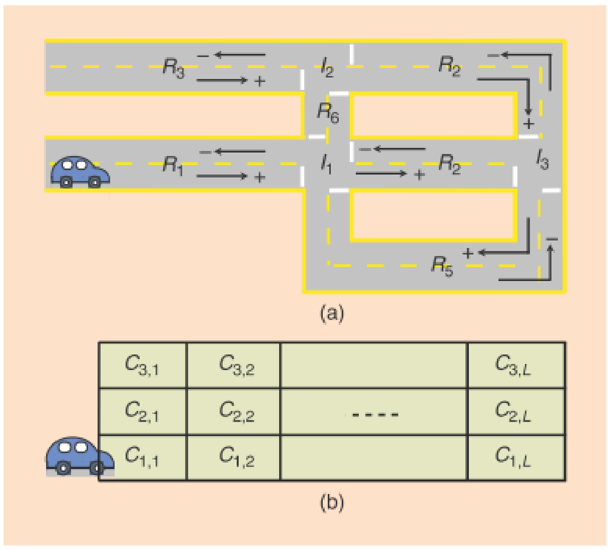

R_3
I_2
R_2
R_6
R_1
I_1
R_2
I_3
R_5
(a)
$C_{3,1}$ $C_{3,2}$ $C_{3,L}$
$C_{2,1}$ $C_{2,2}$ ---- $C_{2,L}$
$C_{1,1}$ $C_{1,2}$ $C_{1,L}$
(b)

Figure 3.7: Application example for the approach presented in (source: [Kress-Gazit 11])

the authors provide a toolkit, described in [Finucane 10], which allows the specification of robot tasks in structured English and the automated synthesis of LTL formulas, a graphical user interface for drawing the region/environment boundaries of interest and simulation window. The approach is applied to an autonomous vehicle acting in an urban-like environment (see Fig. 3.7). The picture shows different abstraction levels for the vehicle motion. (a) shows the overview of the roads and intersections, whereas (b) defines the possible robot movements on a two-lane road. Cells $C_{1,i}$ define the right lane, $C_{3,i}$ the left lane, and $C_{2,i}$ cover the center road ($i = [1, .., L]$). Already in this small scenario, the authors face the problem of state explosion, which they solve by reducing the overall plan for the robot mission to a more local plan of the nearer future in dependence on the current robot position, that will be recalculated iteratively for a new position. An open question is the scalability of more complex control systems.

3.3.2 Verification of Existing Robot Control Systems

The model checking of already developed systems is usually done by (manually) creating a formal model, formulating properties in a temporal logic and feed a model checker with model and properties. Several problems and questions can instantly be identified in this procedure:

1. Manually creating a formal model requires expert knowledge, is time-consuming, and needs to be done again for every change in the implementation.
2. A manually created model may contain errors.
3. The formulation of properties in temporal logic needs expert knowledge.
4. How can physical and environmental conditions be represented in the formal model?
5. How to proceed with a negative result of the model checker? How to get back from the model to the actual source code?

6. How to deal with large, complex systems?

The following approaches aim at one or more of the mentioned problems, trying to solve them for their respective application. The use of formal verification techniques can often be found in industrial robot systems, as their tasks can be completely defined and their work space is usually limited. Anyway, the need for a correct working robot is essential as errors can have extremely expensive consequences. [Weißmann 11] present a case study where they applied formal verification to a car-body welding station to verify the absence of collisions and deadlocks. In order to do this, they automatically transform the industrial robot program into PROMELA which is the input language for the model checker SPIN [Holzmann 97] (Problem 1, 2). Properties are written in linear temporal logic (LTL) where the authors provide patterns to simplify the formulation (Problem 3). Error-traces given by the model checker are mapped back to the original code to help the programmer understand the result (Problem 5). The authors focus on an automatic and easy to use verification process to enable system engineers without special knowledge to apply it to (new) systems.

Another industrial example is given by [Griesmayer 05] who verified a robotic handling system with Java Path Finder [Visser 03]. Their original control software is written in DACS (developed by FESTO), and properties are directly formulated in this source code. The DACS code is automatically translated to JAVA (Problem 1, 2). The application of model checking to the control software of a simple line-following robot is presented by [Scherer 05]. The software is written in JAVA which enables the direct input into Java Path Finder. The authors focus on the problem that developers of autonomous systems always face: the interaction of software with its physical system. To encounter this problem, they construct a simulation model from the physical system using differential equations, which is verified in conjunction with the software (Problem 4). As the special format of the physical system model has to be integrated in the checking algorithm, the authors needed to implement it on their own.

3.4 Evaluation of Behaviour-Based Systems

The concept of behaviour-based systems goes back to Brooks [Brooks 91] and has been developed since then by many research groups. Researching tasks comprise among others the design and modelling of the software structure, the realisation of low-level, real-time tasks as well as complex high level tasks, the analysis of the system, and the verification of single components as well as the complete system. Approaches for the design and analysis of behaviour-based robots are given in [Matarić 92, Zhang 09, Proetzsch 10a].

Analysis and verification considering behaviour-based robotics are not very frequently addressed in current research. But there exist quite a number of papers, published also in the last few years, that deal with various problems of this topic. This section discusses the most important examples of current research in the field of analysis and verification of behaviour-based systems, which goes beyond the evaluation by testing.

3.4.1 Formal Verification of BBS

The need to guarantee a certain robot behaviour leads to the investigation of formal verification techniques also in the area of behaviour-based robotics research. Several methods

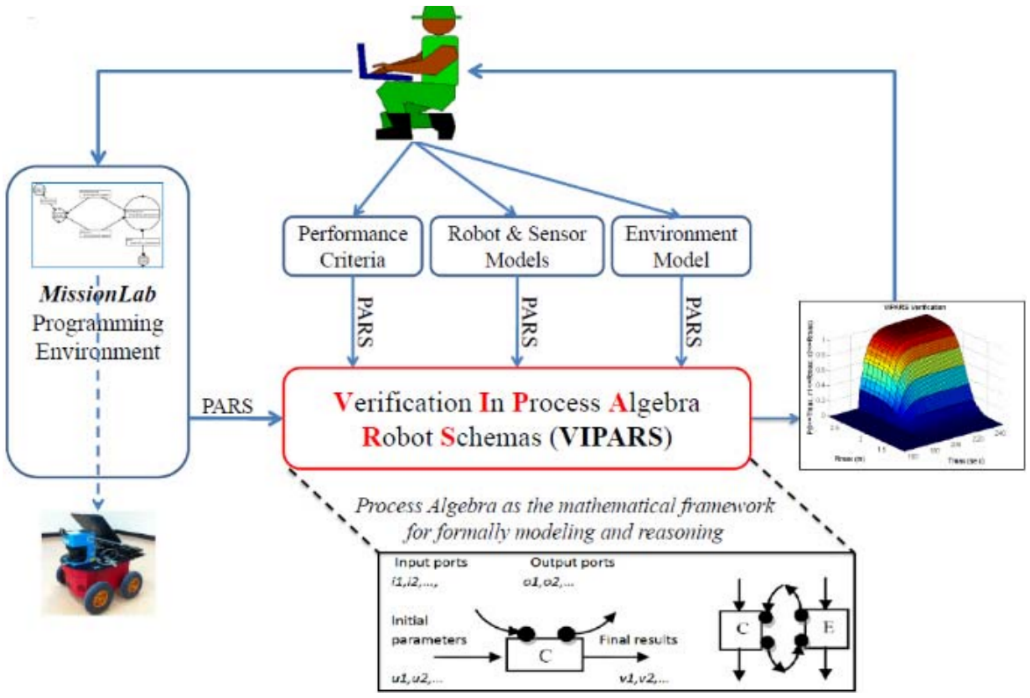


Figure 3.8: The *MissionLab*/VIPARS System Architecture. The mission program as well as models for robot, sensor systems, environment and mission performance criteria are implemented in *MissionLab*. The *mission* is translated to the formal input language PARS. VIPARS (*Verification in PARS*) returns a probabilistic distribution whether the mission will achieve its performance criteria under the given circumstances. (source: [Lyons 15a])

described in Sec. 3.3 have been adapted to the special properties of BBS. [Lamine 02] implement formal verification techniques to analyse the behaviour of robots using the SAPHIRA architecture. Their goal is to find errors in execution sequences, for example an oscillation between the goal-oriented behaviour and the obstacle-avoidance behaviour. The authors propose tools that prove the satisfaction of LTL statements to monitor and plan robot behaviour (Problem 3). [Proetzsch 07] propose an approach using model checking techniques to verify a part of the robots control system. They model every behavioural component in the synchronous language Quartz [Schneider 09] and verify them using the Averest framework [Schneider 05]. With their approach they were able to verify a number of specifications for the given part of the network. Nevertheless, the large number of states indicates that the approach does not scale well, and need therefore further investigation for the application to larger systems.

With this problem in mind, the authors of [Lyons 15b] base their verification on an approach that completely avoid state-space combinatorics. They use a process-algebra representation, where the program is mapped to a set of equations over the program variables. The specification (*performance criteria*) and environmental models can also be described using the process algebra. The result of the verification process is a probability landscape describing the systems performance. Figure 3.8 represents the developed system architecture. The authors state that their method should be transferable to other behaviour-based architectures with small effort but not to other robot control systems as it is strongly dependent on the modular and concurrent structure of BBS. In [Lyons 15a] the approach is extended to verify the behaviour of multiple robots acting in an environment which can include obstacles.

The described approaches show the complex demands on the developer when using formal verification techniques. Not only the software system, but also the robot equipment (e.g. sensors) and the environment (path, obstacles) as well as the requirements on the robots action have to be defined and modelled in a formal way. Nevertheless, the results show that it is possible and the topic is worth to be further investigated.

3.4.2 Requirements Analysis

The authors of [Zhang 09] propose an approach for the modelling and analysis of the obstacle avoidance behaviour of a robot using object oriented techniques. They lay special interest on the representation of robots interaction with its environment and propose to use Use-Case-Models (UML) for requirements analysis. Figure 3.9 shows an example use-case model for a behaviour-based robot. The three boxes (*objects*) in the bottom represent the three behavioural tasks of the robot (*detecting, finding path, avoiding obstacles*). The environment is modelled as *actor* which provides information to the robot and thereby influences the behaviours. Analysis in this concept targets only requirements analysis in order to have a sound requirements definition as basis for the following development steps. Unfortunately, the authors say nothing about the process to identify the requirements and the use-case-model in the example is not complete in a way. The authors just state that the environmental influences on the individual objects needs to be modelled as well, but give no example.

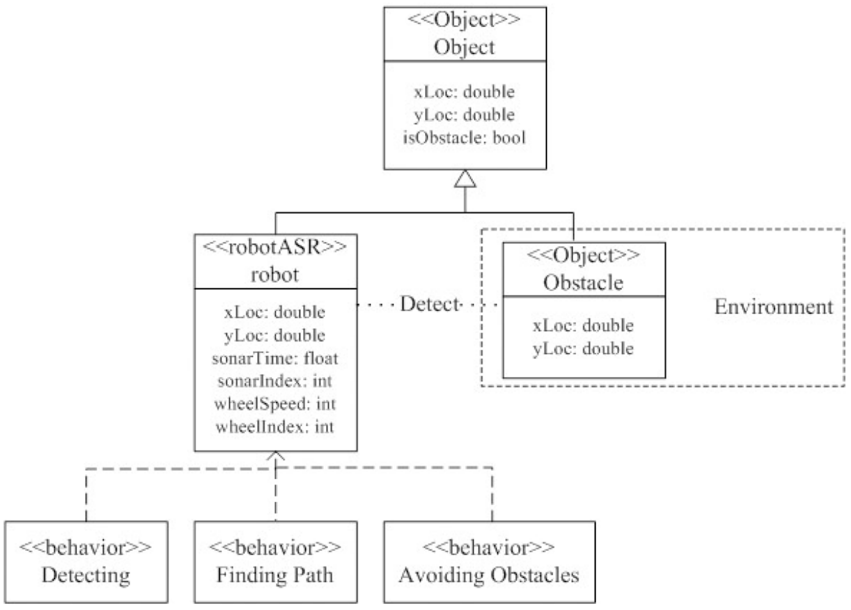


Figure 3.9: Use case model for requirements analysis (source: [Zhang 09])

[Guo 10] present an approach using fault trees to derive safety and security requirements for behaviour-based systems. Their main goal is to show how the effects of security hazards can influence the safety of a system. To reach that they use several standard safety analysis techniques like FMEA to define the hazards and integrate the events of the security fault tree into the safety fault tree. With their approach they show the applicability of standard techniques to behaviour-based systems, although the given example is quite small.

3.4.3 Safety/Security Analysis

[Steiner 12] deal with the topic of analysing the safety and reliability of complex embedded systems in dependence on their software. The main idea is to model the behaviour based network as *State Event Fault Trees* (SEFT), which are a combination of deterministic state machines, Markov chains and Fault Trees. In these state event fault trees, undesired events are selected and the SEFT is translated to a *Deterministic and Stochastic Petri Net* (DSPN) for a quantitative analysis. With the analysis results the reliability of the system can be measured for the chosen event. The authors provide translation rules for the functional description of the BBS to a state event fault tree. The application is shown on a small network example. They make no assumptions about the scalability of the approach and the application to different problems. The quantitative analysis of this approach is based on probability values assigned to basic events. The determination of these probabilities is not further discussed in the paper. Nevertheless, the approach to use a kind of fault tree to model behaviour-based networks in order to reason about safety and reliability issues seems to be a promising approach.

3.5 Discussion

The goal of this chapter was to investigate current analysis techniques in the field of safety analysis. Well-known and established methods were described and their possible application to robotic systems are documented with examples. The big number of examples from the autonomous mobile robot domain document the importance of safety aspects in that area. This could also be shown for the field of behaviour-based systems in Sec. 3.4. Although, the verification of BBS has not been exhaustively discussed till now, some of the well-known standard techniques have already been applied to such systems before, for example fault tree analysis and model checking.

Section 3.1 has dealt with hazard analysis techniques, which were primarily used in system engineering and are well-established in software engineering by now. The ability to analyse a system in order to find the trigger points of failures is of great importance. FMECA and HAZOP are table-based methods using key-words to identify possible hazards in order to determine safety requirements. The methods need a representation of the possible actions (e.g. Use-Cases) the system can trigger. They can be applied to behaviour-based systems as well, as they usually abstract from the actual software implementation structure.

Fault tree analysis has the same target but uses a different representation of the analysis result. The tree structure allows to illustrate the relationship of system components, which supports the developer in the analysis process. Additionally, it is possible to determine the root causes of failures as well as the correlation of events that cause a failure, and to assign a probability to a hazardous event. The latter can be automated which is a great benefit of this method. The method allows the developer to improve his understanding of the software and the relation between software and hardware events, which is an important factor when developing safe mobile robots. The examples have shown the possibility to transfer FTA to robotic systems. The versatility of the method and the good tool support are reasons to examine the applicability to behaviour-based systems. Fault tree analysis will be used to gain understanding about the causes of hazards and the relationship between the corresponding components. Several problems have been mentioned that have to be investigated further in the context of this thesis: These are the adaptation of the method to the special software structure of BBS, the handling of large scale systems and their corresponding fault trees, the combination of hardware and software in the analysis process and the automation or at least tool support for the analysis process of BBS.

Dynamic analysis methods have been the topic of Sec. 3.2. Definitely, every system is tested in one way or the other, therefore, a lot of research is done in that field in order to reach the best possible test coverage, efficient test case generation, reliable results, and so on. The definition of test cases for behaviour-based robots does not differ from that of other robot implementations. However, the special component based software structure fosters the development of special analysis techniques to detect the causes of faulty robot behaviour. Therefore, the focus of this thesis will be on the identification of useful data that can be recorded *online* during test runs and the corresponding *offline* data analysis afterwards. Additionally, it must be proven how the resulting analysis methods can be applied to the system in order to improve the systems performance at run-time.

Formal verification methods have been presented in Sec. 3.3. Although they enable the complete verification of a system they are not as frequently used as the previously mentioned methods. Since testing can never cover the full scope of operation of complex robot systems, a lot of researchers currently deal with the topic of formally verifying robotic systems. As mentioned in Sec. 3.3, model checking will be chosen as formal verification technique and the methods shall be applicable to already existing control systems. In this context, several problems have been identified during literature research. The creation of a formal model is often done for a special system or a certain programming language (such as Java), the same serves for the mapping of the resulting trace to the original system. The formulation of properties usually needs expert knowledge to express them in terms of a temporal logic. Furthermore, the verification process is often limited to parts of the complete system in order to handle the scalability problem of the technique. The integration of the environment is a central point in the verification of robot systems as safety requirements usually include environmental interaction. However, the integration is complex. On the one hand, the robot can face an unlimited number of situations including among others hardware failures and human interaction. On the other hand, those situations need to be modelled formally in order to check them in combination with the software part of the system.

This leads to the following research topics of this thesis: Environmental conditions shall be integrated into the formal model of the robot system. It should be possible to check the resulting model without the need of adapting the model checker. The process of the creation of formal properties shall be tool supported and intuitively usable for robot system developers. Tool support shall also be given for the interpretation of counterexamples. As behaviour-based networks tend to get very huge for complex robot systems, the scalability problem of the approach shall be further investigated in the special context of BBS.

Formal verification techniques like model checking are often used in combination with other analysis techniques. For example, [Nazier 12] integrate the results of fault tree analysis into a formal system behaviour model. Test cases are generated out of this model by using model checking. The combination of the different analysis techniques shall also be discussed in the context of behaviour-based systems in this thesis.

All the presented application examples in this section have in common that they are based on a deep knowledge of the system to be analysed. It is therefore necessary to have automated techniques and visualisation methods that provide information about the inner system structure and support the developer during the analysis process. This is especially important for complex systems and non-system experts that shall execute verification tasks.

4. Analysis of Behaviour-Based Systems

Analysis, generally defined, is the process of breaking down a something into its parts to learn what they do and how they relate to each another. Several types of analysis for different specific tasks and numerous analysis methods are discussed in literature. Some of the most important techniques for safety analysis of complex systems has been presented in Chap. 3.

The special issue of behaviour-based robotic control systems is their modular software structure. This allows for new approaches in the analysis of the overall system, as it offers the possibility to investigate the software structure as well as their connections to hardware components while abstracting from the concrete implementation of the robots behaviour. The goal of this chapter is to determine techniques that are suitable for the analysis of behaviour-based systems. The following sub goals shall be achieved

- Analysis of the system structure to gain knowledge about it
- Analysis of the system structure to find failures and faults
- Analysis of the system to determine violations of requirements
- Clear representation of analysis results

The creation of behaviour-based networks to control robots is (or should be) a process guided by architectural principles and rules that help the developer to transfer the requirements from the requirements document into a working system. Nevertheless, implementation faults, unforeseen environmental conditions or subsequently integrated components can lead to failures. It is therefore necessary to validate the system after the implementation phase. The methods discussed in Sec. 3.1 are primarily developed to analyse systems in early development phases, but they can also be applied to the complete system.

The following sections deal with the adaptation of well-known analysis methods to behaviour-based robotic systems.

Section 4.1 examines static analysis methods considering the network structure of the BBS and fundamental checking methods. The results are useful for a deeper understanding of the system structure and the avoidance of errors. They serve as a basis for the more sophisticated methods described in subsequent sections. In Sec. 4.2 the application of fault tree analysis in order to learn about the relationship of system events and to identify potential sources of hazards is discussed. Sec. 4.3 deals with data analysis applied to data sets monitored during test runs. Here, techniques to structure and represent data with the goal to bring out critical system components and causes of failures are presented. Figure 4.1 provides an overview of the analysis problems and the corresponding analysis techniques. In the end of this chapter, the presented methods and their benefits to the analysis of behaviour-based networks are discussed (see Sec. 4.4).

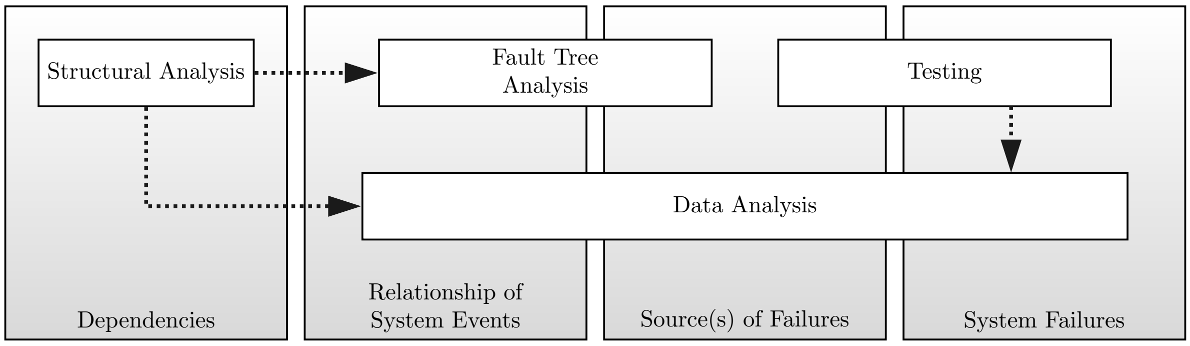


Figure 4.1: Overview of analysis tasks (grey boxes) and the corresponding techniques (white boxes). Some techniques need input/support from others, which is marked with dotted lines.

4.1 Structural Analysis

The fundamental structure of behaviour-based systems facilitates the application of automated basic checks. All components are built up with the same interface, where the input and output data is often restricted due to architectural conditions. For example, the limitation of data values to a certain range or the definition of values in dependence on other values. The violation of these requirements can be automatically checked with little programming effort (see [Proetzsch 10a] for the application of this method on the iB2C or [Forouher 14] for the application in ROS). The indication of a violation can be made visible to the developer and/or used to automatically correct the value in order to preserve system safety.

Another task is given by the network structure of a BBS. The combination of components follows rules, that must also be checked. For the iB2C for example, a component can only be stimulated or inhibited with the activity of one other component. This problem can be checked prior to program execution and thereby prevent system failures.

The analysis of the network structure serves two purposes, on the one hand it allows to gain a deeper understanding of the system and, on the other hand, detect faults. Two main subjects can be identified that shall be answered in this section. One is the definition of system properties that can be automatically analysed. The second is the representation of the network and the analysis results in a way to support the developer.

Behaviour-based networks can be represented by graph structures, where the components are the vertices and the data connections are the edges. The tool FINSTRUCT, described in Sec. 2.4 provides such a view. Furthermore, the graph visualization software *Graphviz* [Gansner 00, Ellson 03] is used in this thesis for special views of graphs, as it offers a great number of possibilities for design and style. Figure 4.2 depicts an example network. It serves no special purpose but shall be used to demonstrate some of the analysis methods presented in the following. Behaviour A stimulates behaviour B. Behaviours A and D are coordinated by a fusion behaviour. Its output is transferred to E and F. Several stimulation edges transfer the influence within the network.

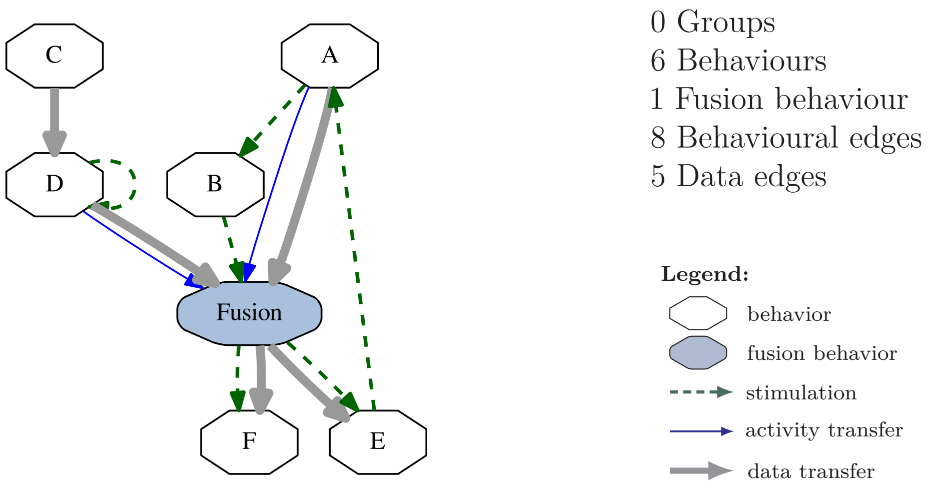


Figure 4.2: Example network for illustrating the presented analysis methods together with fundamental information like number and types of components

Fundamental information about the network is given by a collection of basic information about number and type of the components (e.g. standard behaviours, fusion behaviours, groups) and their dependencies. More sophisticated analysis techniques help to find problems in the structure. In the following a number of automated methods for the analysis of behaviour-based systems are presented. The major goal of these methods is to gain a deeper understanding of the system, but can also be used in combination with the methods introduced in Sec. 4.2 and 4.3 to find the root causes of failures of the overall system.

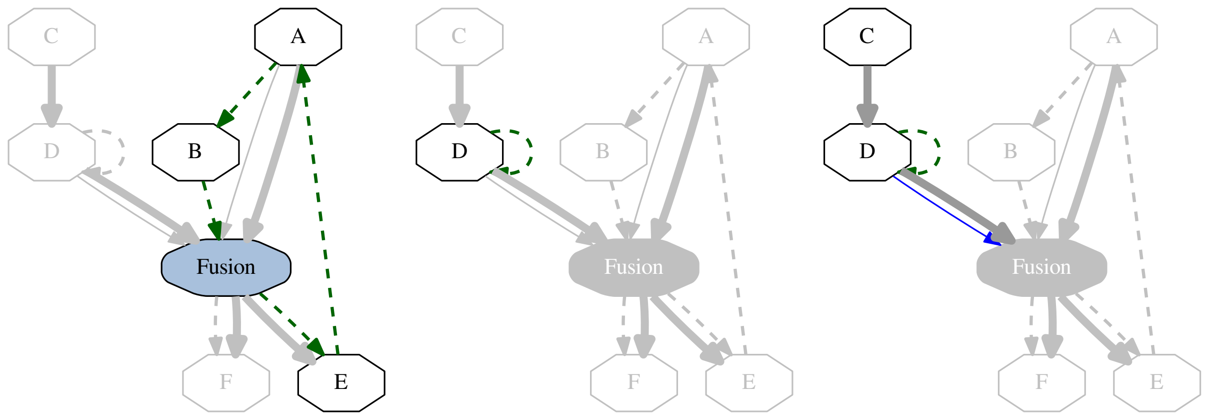
Number of components A very basic but nevertheless very important information. The number of standard behavioural components, fusion behaviours, behavioural groups and their connections (also called *edges*) are an indicator of the complexity of a system.

Pattern Recognition The implementation of behaviour-based networks follows rules in order to achieve a correct system. Similar to design patterns used in object-oriented programming that support the developer in structuring the code, behaviour-based networks can also be constructed according to structural patterns. For the iB2C these are called behaviour network patterns [Proetzsch 10a]. The recognition of patterns in the network allows the system analyst on the one hand to prove the correctness of an application. On the other hand, it enables to identify network parts that can not be assigned to a pattern which might indicate a wrong implementation.

Cycles Cycles can produce an undesired behaviour of the system. For the iB2C cycles of stimulation and inhibition are explicitly forbidden according to [Proetzsch 10a]. He proposes the use of strongly connected components to find cycles in the network structure.

Dependencies The analysis of the interdependencies of components provides valuable information about the system for the developer or analyst and is crucial for several analysis tasks. Here, not only direct connections are of interest, but also the flow of control data inside the network. This kind of information is known as *control flow analysis* and will be discussed in Sec. 4.1.1.

Critical Components With the analysis of the interdependencies it is also possible to find out components with special properties. These can e.g. be sources and sinks of activity or behaviours without stimulation. The components are called critical components, as they show a non-standardised behaviour and need further analysis in order to verify their correctness.



(a) Stimulation cycle **(b)** Source of activity **(c)** Critical components

Figure 4.3: Results of network structure analyses

Figure 4.3 shows the application of the approaches mentioned above on the example network. Cycle detection revealed a stimulation cycle between the components A, B, E and $Fusion$ (Fig. 4.3a). In Fig. 4.3b node D is highlighted as a source of activity. This representation offers the developer the possibility to easily check the correctness of the stimulation mode of behaviours. The result of the critical components analysis is presented in Fig. 4.3c where all components with non-standard behaviour are highlighted. In addition to the node D as activity source, node C is marked, because it is not stimulated and does not transfer activity. It therefore has no influence in the network and should be checked for correctness.

It becomes clear from the examples, the analysis of the interdependencies in the network is crucial. As networks can become very large and relevant information, for example the tracing of the data flow, cannot be done manually, the next section deals with the extraction and representation of those parts of the networks that include the relevant information. Furthermore, it presents a technique to identify a special kind of critical components.

4.1.1 Data Flow Analysis and Relevance of Components

The network structure of behaviour-based systems can be seen as equivalent to a program dependence graph (PDG). In a PDG, program statements and expressions represent the nodes of a graph, edges define the dependencies between them. For behaviour-based systems this definition is adapted so that nodes represent the state of a behaviour component and edges the connection between components. A state of a component is defined via its stimulation, activity, inhibition, target rating and activation (see Sec. 2.2). The new structure will be called behaviour dependence graph (BDG) in the following.

Definition 4.1: Behaviour Dependence Graph

A behaviour dependence graph $G_{BD} = (N, E)$ is a directed graph, where N is a set of behaviour nodes and E a set of edges. A behaviour node n_i is defined via its incoming and outgoing behavioural signals $s_b = (s, a, i, r)$, with s being its stimulation, a its activity, i its inhibition, and r its target rating. An edge $e = (n_j, n_k)$ describes the dependence from node n_k on node n_j.

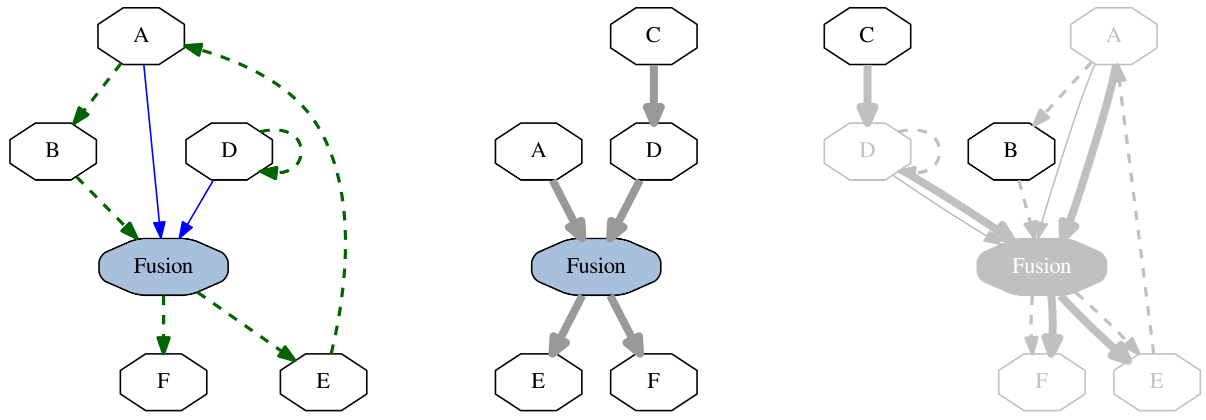
The understanding of how signals are propagated in the network supports the comprehension of behaviour functionality. Therefore, it is of interest to create different behaviour dependence graphs according to certain signals. For example, a *stimulation graph* shall only contain those components that are connected via a stimulation transfer. In order to do this, the definition of the BDG is extended to

Definition 4.2: Typed Behaviour Dependence Graph

A typed behaviour dependence graph (TBDG) is a BDG with an additional type property $t \in \{s, i, d\}$, with s indicating stimulation, i inhibition and d data transfer. The TBDG $G_{TBD}(t) = (V, E(t))$ contains those nodes $n \in V \subseteq N$ that are connected via edges that transfer signals of type t.

Figure 4.4 shows two typed behaviour dependence graphs. The comparison with the example graph given in Fig. 4.2 reveals that behaviour C is not contained in $G_{TBD}(s)$ (Fig. 4.4a) as it does not stimulate another behaviour. The activity transfer edges from behaviours A and D to the fusion behaviour are included as they are relevant for the stimulation output of the fusion behaviour. For the data transfer graph (Fig. 4.4b), behaviour B is not contained, as it does not transfer its data to any other behaviour.

The comparison of BDG's of different types can also reveal failures in the network structure of the graph. Figure 4.4c highlights the differences in the graphs $G_{TBD}(s)$ and $G_{TBD}(d)$. Node B transfers only its activity and not its data vector to the fusion node. This transition is correct, as B stimulates the fusion node with its activity. In turn, node C transfers its data vector without a corresponding activity vector. This is a fault and must be corrected. A corrected version of the example graph is given in Fig. 4.5a, where all previously detected faults are eliminated. Here, the missing stimulation of behaviour



(a) Graph representing the stimulation graph $G_{TBD}(s)$

(b) Graph representing the data transfer graph $G_{TBD}(d)$

(c) Graph representing the difference of $G_{TBD}(s)$ and $G_{TBD}(d)$ in the set of nodes

Figure 4.4: Examples for typed behaviour dependence graphs

C is fixed by adding a self-stimulating edge. Additionally, the component transfers its activity to stimulate behaviour D. The stimulation cycle $A \to B \to Fusion \to E \to A$ is disrupted by removing the edge $E \to A$.

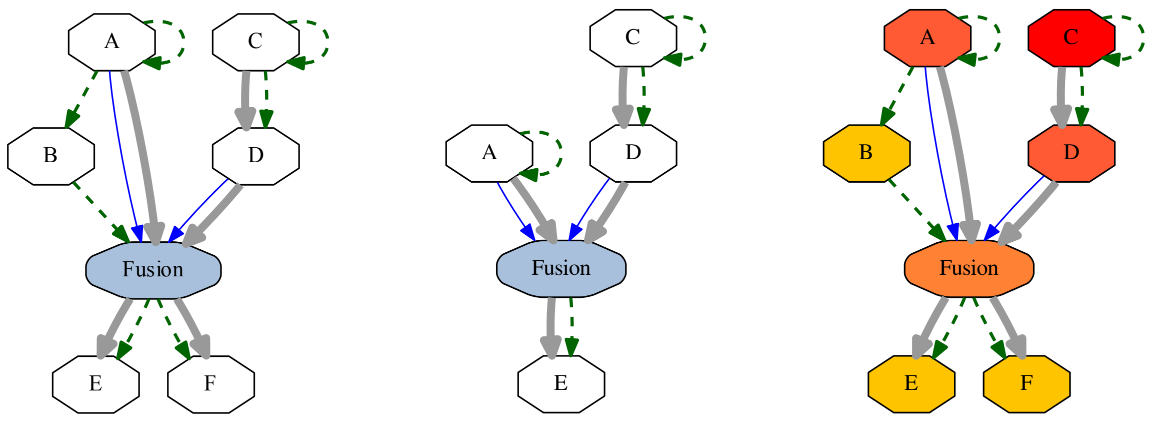
Another usage of behaviour dependence graphs is the *influence graph*. This is a sub-graph of the BDG that contains only those nodes that influence a specific one. This is especially useful in large networks to reduce the size and focus on the relevant parts. An influence graph is defined as follows

Definition 4.3: Typed Behaviour Influence Graph

A typed behaviour influence graph (TBIG) $G_{TBI}(t, n_{initial})$ is a TBDG containing the maximal set of nodes $V \subseteq N$ such that for all $n \in V$ there is a path from n to $n_{initial}$ using edges $e \in E(t)$.

Figure 4.5b depicts such a graph for the start node E tracing the control data input. These kind of influence graphs can also be used to make statements about the relevance of components. The assumption is, that a component that influences many others is more relevant than a component that influences only a few. Certainly, all components in a behaviour-based control network should be relevant to make the system work as required, but, when analysing a system in order to find the causes of failures it might save time to start analysing those behaviours, that influence many others. Alg. 4.1 shows the algorithm to determine the relevance of nodes. For the determination of the number of nodes that are influenced by a special one, firstly, the TBIGs for every node are created. Afterwards, the occurrence of the special node is counted over all influence graphs.

Figure 4.5c shows graphically the results of the algorithm for every node in the example graph according to the flow of control data. The colour gradient from red to yellow indicates the frequency of the individual nodes. The control data from node C influences four other nodes, namely D, $Fusion$, E, and F. Nodes D and A influence three nodes, respectively and the fusion node the two nodes E and F. Finally, the nodes B, E and F do not influence any other node with their control data.



(a) Corrected example graph **(b)** Graph depicting the influence of control data on behaviour E **(c)** Relevance of behaviours in respect to the data flow (red: most relevant, yellow: less relevant)

Figure 4.5: Examples for an influence graph and a graph depicting the relevance of behaviours using different fill colours

Algorithm 4.1: Determination of frequency of occurrence of nodes

```
Input: graph, type, node
forall the $n \in N$ do
    $g_n$ := CreateInfluenceGraph($type$, $n$);
    Push($g_n$, $influence\_graphs$);
end
count:= 0;
forall the $g_n$ in influence_graphs do
    if $node \in g_n$ then
        count++;
    end
end
Result: count
```

The automated analysis together with the multiple visualisation possibilities allow the developer a quick overview over the system. The different views facilitate the detection of faults and the verification of structural requirements.

4.2 Fault Tree Analysis

Fault tree analysis is a hazard identification technique widely used in industry as shown in Sec. 3.1.2. A fault tree can be described as a graphical model of the chain representing the relation between an undesired event (system level hazard) and the failures that cause it. (Compare Fig. 4.1 for the classification of this method in the concepts of this chapter.) The generation of fault trees is a deductive method. The undesired event, the so-called *top event*, is broken down to the failures of components that lead to it, connected via logical AND and OR gates, until the lowest component that cannot be developed further is reached. These components are called *basic events*. Basic events can occur individually or

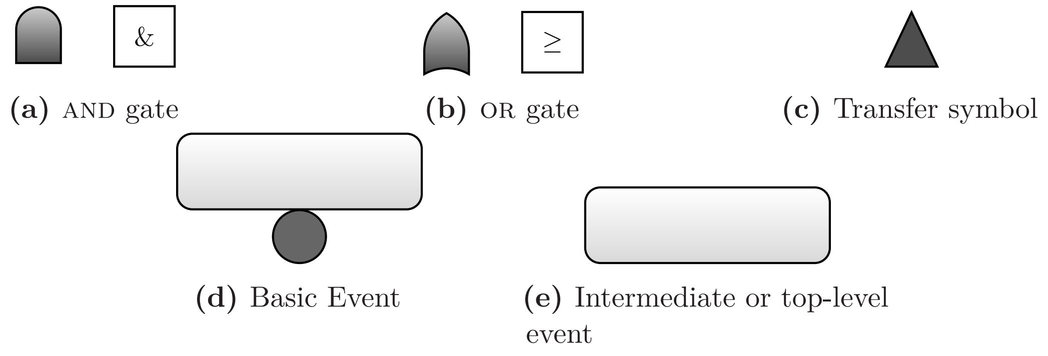


Figure 4.6: Basic symbols used in fault tree analysis. a and b show two variants of the gate symbols, respectively.

in combination with others. The basic symbols used in the construction of fault trees are shown in Fig. 4.6. The fault tree in Fig. 4.7 provides an example of a washing machine that overflows [Lyu 96]. The overflow is the top event, which can be caused by either the shut-off valve being stuck open or the machine staying in fill mode for too long. The first is a basic event as it can not be evaluated further, the latter event can occur when the time-out control failed and the sensor that controls the filling level failed at the same time.

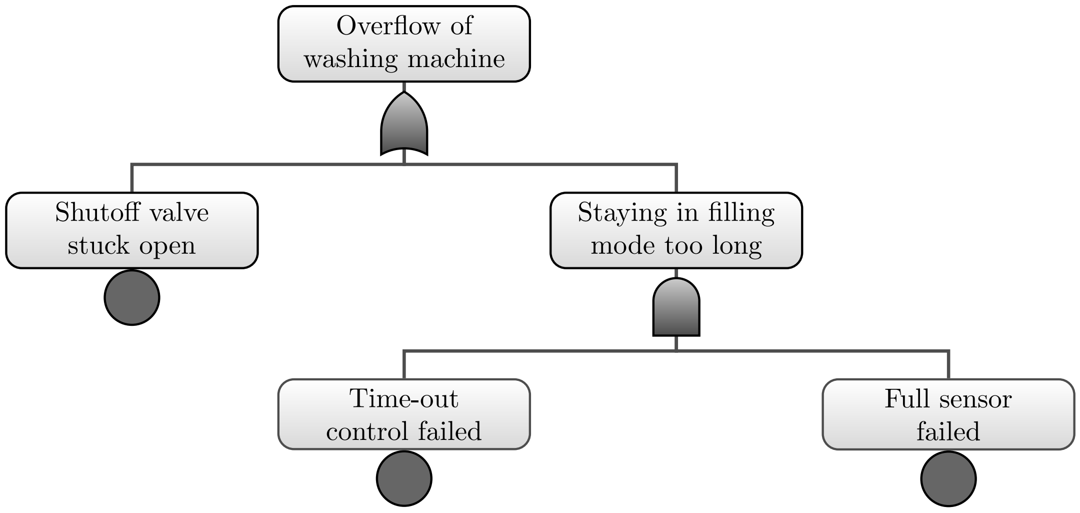


Figure 4.7: Example for a simple fault tree (redraw from [Lyu 96])

Fault tree analysis is used to gain understanding about the types of failures and their causes, the functional relationships of system failures and the identification of critical areas with potential for improvements [Vesely 81, Dhillon 15]. It can be distinguished between two types of analysis. The *qualitative analysis* identifies weak points in the system. The *quantitative analysis* calculates the probability of occurrence of the hazard.

For a qualitative evaluation the fault tree must be complete. A statement about the reliability of the system can be made via the graphical structure. Possible evaluation approaches are, among others, the analysis of *minimal cut sets* and *critical paths*. A minimal cut set contains all basic events that together trigger the top event. For the example given in Fig. 4.6 two minimal cut sets can be generated, namely {*shut-off valve* *stuck open*} and {*time-out control failed, full sensor failed*}. Cut sets with only one element imply a single point of failure for the system.

Information about the relation of events is provided by the critical path, which leads from the top event to a causative event. The qualitative FTA allows to gain a deep understanding of the system and to discover yet unknown causes of failures. However, this method is time expensive and needs expert knowledge to identify all components of the tree.

The goal of the quantitative FTA is to give a prediction of the reliability of complex systems. The calculation is based on the probability of occurrence of the basic events. These can be failure rates or the reliability of electrical components which are known from the manufacturer information, gained in tests or estimated via probability calculations. The probability of failure of the whole system can be calculated iteratively starting from the basic events. For the simplest case of two independent events A and B, the AND gate connects its inputs $P(A)$ and $P(B)$ to

$$P(A) \cap P(B) = P(A) \cdot P(B) \tag{4.1}$$

The OR gate correlates with

$$P(A) \cup P(B) = P(A) + P(B) - P(A) \cdot P(B) \tag{4.2}$$

This method is useful to verify reliability requirements and to compare different systems. But, in many cases it can be problematic to determine the failure rates of all possible basic events. The failure rates of hardware are not always known, and the probabilities of software faults cannot be measured accurately. Nevertheless, there exist several approaches that use the lines of code or the number of faults corrected in the past as probability measure.

The general process of fault tree analysis comprises four steps.

1. Preparation including the determination of the topic, goal and scope of the analysis as well the needed system information.
2. Fault tree modelling
3. Analysis
4. Interpretation of the results

A major advantage of fault tree analysis is its wide-spread application and the high degree of popularity. This makes it also interesting for the analysis of behaviour-based systems. The possibility to easily connect software and hardware components to learn about their relationship is a very useful property for robotic systems. The main effort using FTA is in the construction of the fault tree. The goal of this section is to provide guidelines on how to built fault trees of behaviour-based systems.

For the analysis of fault trees, several tools exist that calculate the probabilities of top-events, the number and composition of (minimal) cut sets, and so on. Examples are among others *FaultTree+*[1], *BlockSim*[2], and *RiskSpectrum*[3]. For the work at hand the *Embedded Systems Safety & Reliability Analyser* (ESSaRel) [Kaiser 03][4] is used.

The following section introduces a concept to model the behaviour-based system structure as fault tree. In Sec. 4.2.2 the method is applied to the example of the autonomous RAVON robot, where the focus lies on the creation of the fault tree and not on the analysis procedure.

4.2.1 Fault Tree Modelling for Behaviour-Based Systems

The use of fault tree analysis to robotic systems offers some advantages. A failure event can be caused from many different components, e.g. sensors, actuators, data connections, hardware that executes software and certainly the software itself. Fault tree analysis helps to understand the potential errors and the relationship of failure events and thereby facilitates system improvements.

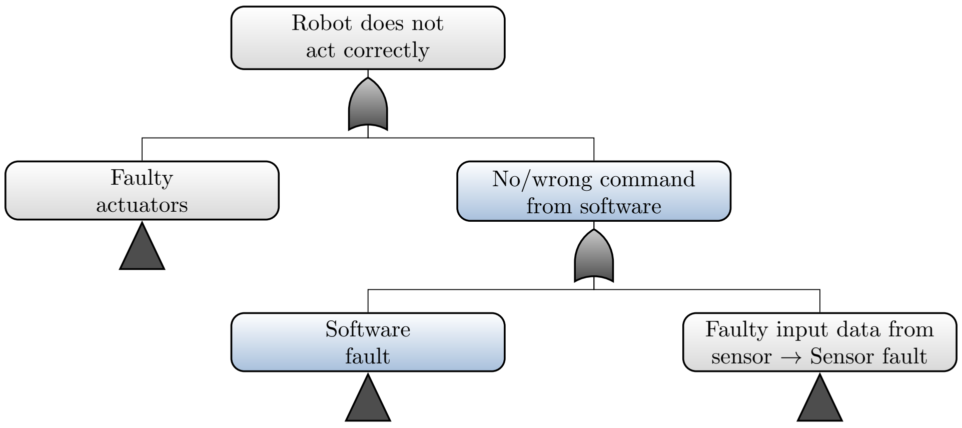

Robot does not act correctly
Faulty actuators
No/wrong command from software
Software fault
Faulty input data from sensor → Sensor fault

Figure 4.8: Abstract example of a fault tree including hardware, here, actuators and sensors (grey boxes) and software (blue boxes) for a robotic problem

In literature, often hardware and software parts of a system are analysed individually. But exactly this correlation is interesting. Figure 4.8 depicts exemplarily the different faulty components that can cause a failure of the whole system.

Software fault tree analysis has been topic to research since the number of embedded systems grows. A problem here is the complexity of the resulting trees when several variables in the software need to be respected. The special structure of behaviour-based systems offers the possibility to analyse the software while abstracting from the actual implementation of functionality. Therewith, fault tree creation can be based on the network structure where events reflects faults in the structure. The inner implementation of the behavioural components can be verified in a separate step.

Since all behavioural components have the same interface, this approach proposes the application of a standard fault tree modelling process for a single component together

[1]FaultTree+: `www.isograph.com/software/reliability-workbench/fault-tree-analysis/`
[2]BlockSim: `www.reliasoft.com/BlockSim/`
[3]RiskSpectrum: `www.riskspectrum.com/en/risk/Meny_2/RiskSpectrum_FTA/`
[4]ESSaRel: `http://essarel.de/`

with guidelines to integrate single trees into the main tree. This procedure also facilitates the automation of fault tree creation which is essential for such complex systems as robot systems are.

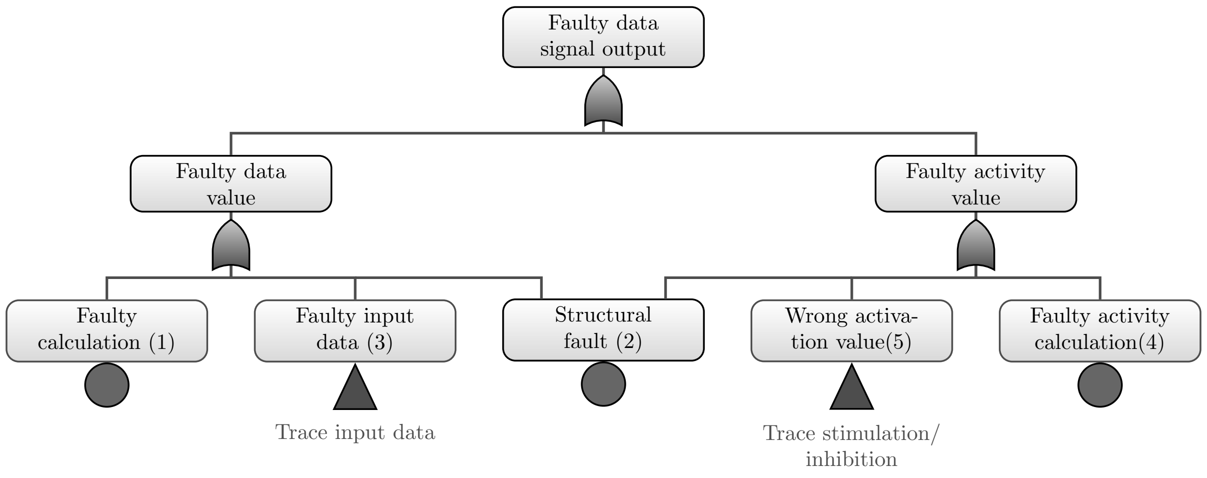


Figure 4.9: Fault tree of a standard iB2C component which outputs faulty control data

The following paragraph introduces the modelling of a behavioural component. The fundamental assumption is that a wrong software command coming from one (or several) components caused a problem. Therefore, the top event for the software sub fault tree is defined as *faulty data signal output*.

For standard behavioural components, there are two possible reasons for a faulty data signal output, namely, *faulty data value* and *faulty activity value*. The first relates to all possibilities that lead to faulty data. These are (1) a *faulty calculation* inside the behavioural component, (2) a *structural fault* in the network or (3) *faulty input data*. The latter implies a fault in another component of the behaviour-based network. This is the link to map the network structure in the fault tree. The needed information about the components that provide date to the component in question can be taken from the TBIG (see Sec. 4.1).

A *faulty calculation* corresponds with a component internal implementation problem. It is handled as basic event in this modelling approach and can be further analysed using software analysis techniques. The aspect *structural fault*, targets faults in the network structure related to the behavioural component in question. These include wrong or missing connections to other components. It is also handled as basic event.

Similar causes are found for the event of a *faulty activity value*. The activity value is significantly relevant for the transfer of control data as it decides about the importance of the control data. In a maximum fusion for example, only the control data of the behaviour with the maximum activity value are passed. Therefore, a faulty calculation of an activity value can lead to problems in the overall robot behaviour. The activity-related part of the fault tree is build with a basic event considering a *faulty calculation* of the activity output value (4) and an event considering a *wrong activation value* (5) (Remember, the activation value limits the activity see Chap. 2.2). The latter is based on stimulation and inhibition inputs to the behavioural component and needs therefore further processing. Additionally,

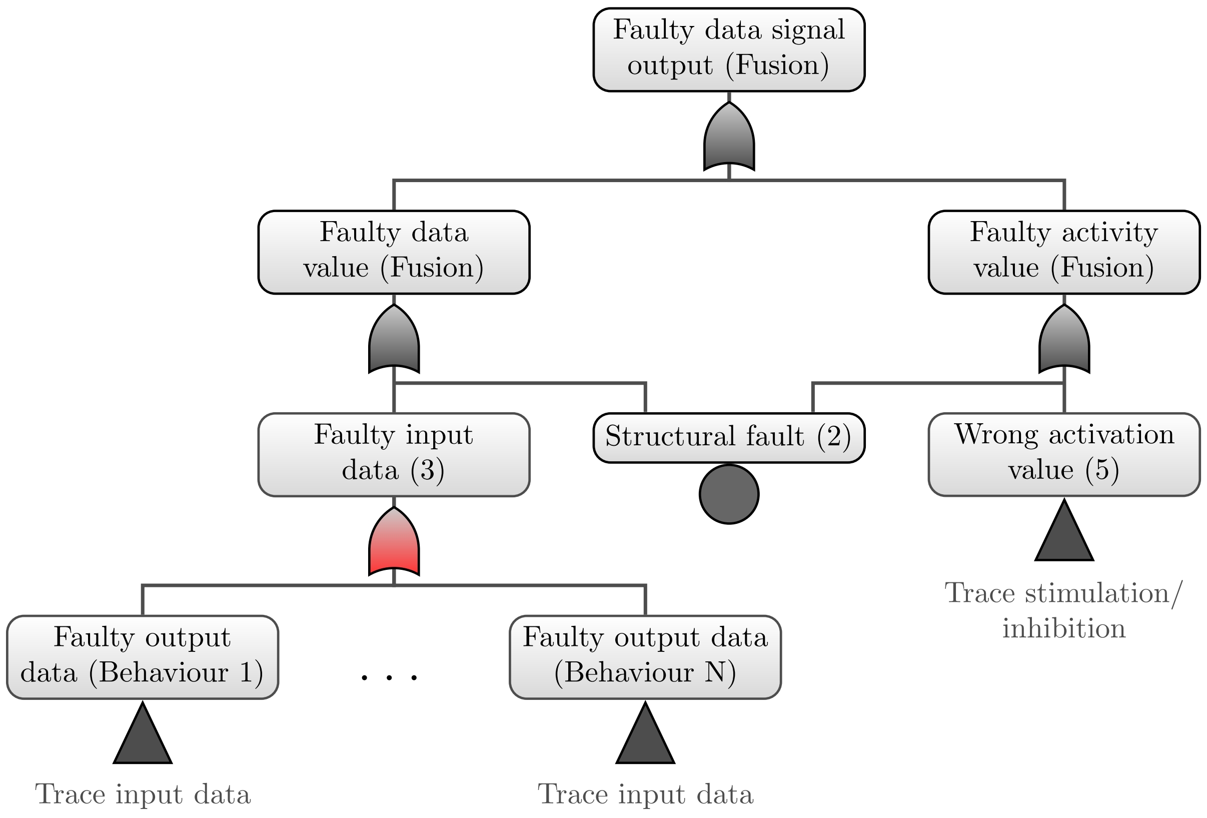

Faulty data signal output (Fusion)
Faulty data value (Fusion)
Faulty activity value (Fusion)
Faulty input data (3)
Structural fault (2)
Wrong activation value (5)
Trace stimulation/ inhibition
Faulty output data (Behaviour 1)
. . .
Faulty output data (Behaviour N)
Trace input data
Trace input data

Figure 4.10: Fault tree of an iB2C fusion behaviour which outputs faulty control data. The red marked OR gate may be replaced by an AND gate in some cases

a structural fault as mentioned for the faulty data signal can also be a problem for the activity calculation (2). Both structural faults are combined in one basic event, as their verification is done simultaneously. Figure 4.9 shows the fault tree of a standard iB2C behaviour.

iB2C fusion behaviours are modelled in a similar way (see Fig. 4.10). In contrast to standard behavioural components the calculation of the outputs (data and activity) is a standard calculation that should be verified to be correct. These events are therefore neglected in the fault tree. The event *Faulty input data* (3) is split into several branches according to the number of input behaviours. They are connected via an OR gate in the figure. This is only a very general model of the fusion behaviour component. It suffices to give information about the interconnections of the behavioural components, but for an exact modelling of the fusion function special system knowledge is needed in order to connect the data outputs correctly. The gate must therefore be chosen according to the special task. In order to reduce the size of the fault tree, behavioural groups can be handled as normal standard behaviours. Their internal structure can be modelled in an individual fault tree, which is only analysed if necessary.

Using these standardised sub fault trees, a complete fault tree for the control software of the system can be modelled. It can be integrated in a combined fault tree containing also hardware parts as shown in Fig. 4.8. The application of fault-tree analysis to a robotic system with a behaviour-based control will be presented in the following section using the robot RAVON as example.

4.2.2 Application Example

This section will describe the application of fault-tree analysis to the RAVON robot. Due to readability reasons, several steps in the procedure are shortened or described only in general. In the introduction of this chapter, the four steps of fault tree analysis have been presented: preparation, fault tree modelling, analysis of the fault tree and interpretation of the results. The preparation phase includes several aspects that will be further explained and answered specific to the RAVON example in the following:

1. Determine the subject of analysis
2. Understand the system
3. Define the top-level event(s)
4. Define the scope of the analysis
5. Define the depth and resolution of the analysis
6. Determine basic rules for e.g. the naming of events, the modelling of recurring structures, ...

In the example presented here, the **subject of the analysis** are safety aspects of the autonomous driving robot RAVON. In more detail, the evasion and slow-down behaviour during forward motion in the case of obstacles to the front of the robot shall be proved. In that case, the hardware parts of the robot that execute a motion are considered as just as the sensor systems that monitor the environment. Furthermore, the software parts that process the sensor data and control the robots motion are to be analysed.

The **understanding of the system** is needed in order to build the fault-tree. It includes an understanding of the systems structure and its components, the functions of the individual components, environmental conditions,.... Information about the software parts of the system is gained using the methods presented in Sec. 4.1. For information about the hardware components the failure mode and effects analysis (FMEA) [Carlson 12] can be used.

For the determination of the **top-level event** a FMEA or HAZOP analysis can be applied. In this example, the event is defined as RAVON *collides with an obstacle*. The **scope of analysis** defines the components that are included and the ones that are not included. Here, among others the main hardware components as motors and sensors and the control software will be considered, but not cable connections, power supply, manual interventions by the operator and environmental conditions such as bad visibility. With respect to the control software only the slowing down behaviour of the robot shall be considered, the evasion behaviour will be neglected in this case.

The **depth and resolution of the analysis** describe the level of detail in which the fault tree will be split. For hardware, a basic event can describe the state of a complete component bought by the manufacturer, for example a sensor system including the sensor

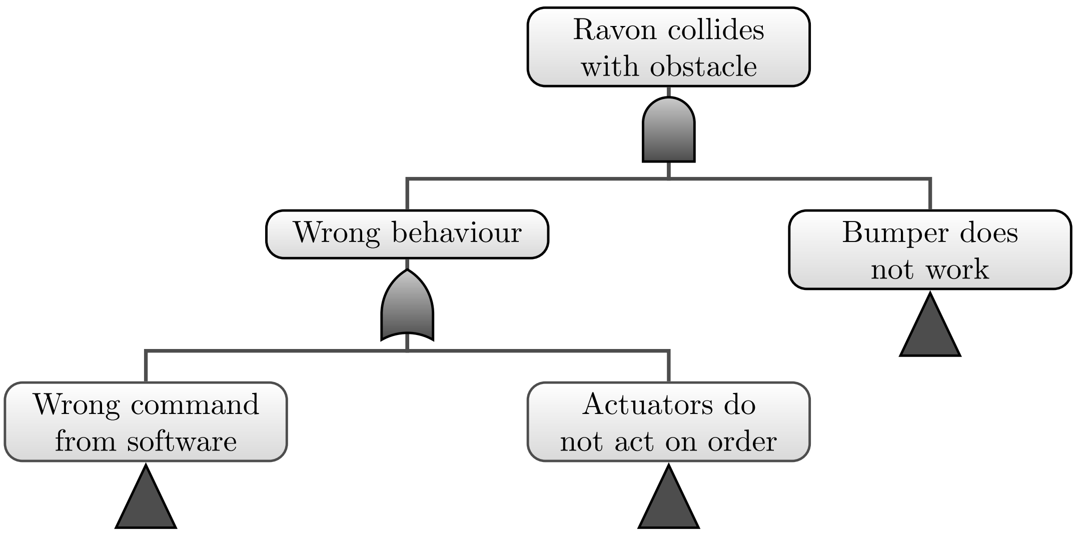

Ravon collides with obstacle
Wrong behaviour
Bumper does not work
Wrong command from software
Actuators do not act on order

Figure 4.11: First level of the fault tree analysing the top-event RAVON *collides with an obstacle*

and a preprocessing unit. For a more detailed analysis of handmade components, they can be divided down to their cable joints or small electrical components. As many of the hardware components of RAVON are handmade, they are analysed in detail down to their electrical components. For the software parts, the lowest level of detail is a behavioural component which is modelled in a fault tree as presented in the previous section. The inner implementation of the individual components is represented as basic events. The sensor data preprocessing steps that are fulfilled before the data is used in a behavioural software component are neglected.

The **determination of basic rules** is important for reusing the results and to work jointly with others. Here, only two rules used in the following application are described exemplarily.

Naming of behavioural components A behavioural component is named by the name of its surrounding group and an addition that describes its function. The groups name is shorted with the initial letters in this example. For example *Fast Front Slow Down* is shortened to *FFSD*.

Structuring of the software fault tree Data signals are traced up/down to group borders. The fault tree for the group is build up as a separate tree. This facilitates the re-use.

Subsequent to the preparation phase the modelling of the fault tree can start. Figure 4.11 shows the top-event RAVON *collides with an obstacle* and the first level of the fault tree. The top-event can occur when the robot is not stopped by software control **and** the safety chain of the robot is not interrupted by the bumper (see Sec. 2.3 for a detailed description of the bumper system). For the former, two possible reasons exist: There can either be a wrong motion (velocity, angle) command from software or the actuators do not act on order. All three events need further analysis. Exemplarily for hardware fault trees, the

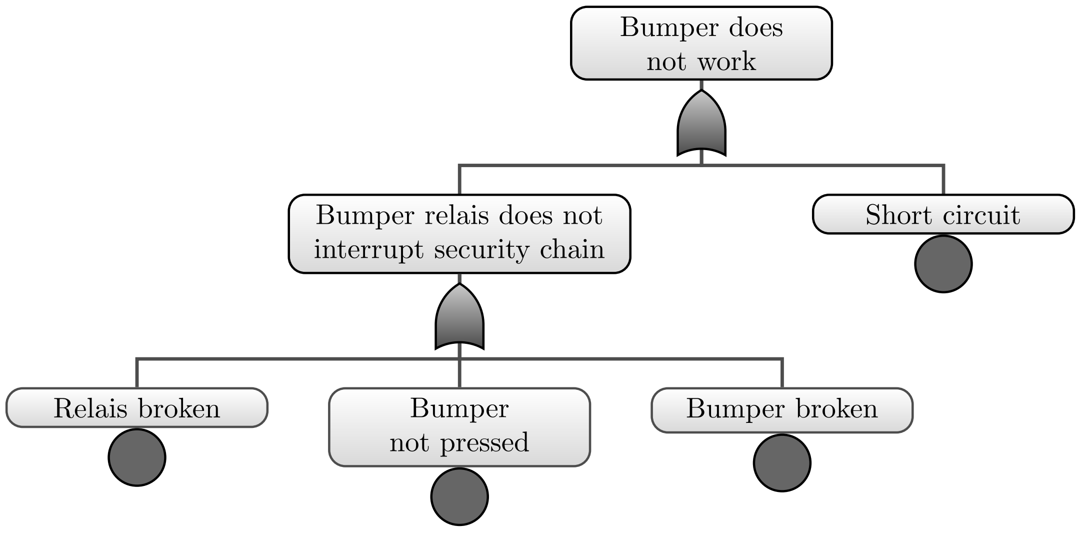


Figure 4.12: Fault tree examining the possible reasons for a not-correct working bumper

fault tree for the event *Bumper does not work* is depicted in Fig. 4.12. The fault tree for the event *Actuators do not act on order* can be build up in a similar way.

For the third event *Wrong command from software*, the source of the velocity command need to be analysed. This is done as described above, using the influence graph starting from that component that generates an actuator command. As the complete fault tree cannot be depicted due to the limitations of the format of this thesis, the path of the control commands is shortly described and fault tree creation is exemplary described for a small part of the network.

Starting from that point where the velocity control command from software is transferred to hardware control, the command can be followed back to a component that fuses the commands of two groups. One is the safety control group that is responsible for safe robot navigation, the other is the drive control group responsible for reaching a goal (compare Sec. 2.3). The fusion behaviour marks the first fork, where the fault tree need to be build in two directions. In the example here, only the path of the velocity command coming from the safety control is followed further. Figure 4.13 sketches the main components of the path used in this example.

(F) Velocity Control gets a velocity command as well as the corresponding activity value from the safety control group. Both values need to be traced. The creation of the data transfer graphs reveal that the groups velocity and activity output stem from different fusion behaviours inside the group. See Fig. 4.14 and Fig. 4.15 for the data and activity transfer graphs. Here, the path of the velocity command coming from *Velocity control output* is further analysed. With the help of the graphs showing the interdependencies of the behaviours in respect to the given data signal, the software fault tree can be build up including every connection that transfers velocity as well as the corresponding activity. For the example here, the command coming from the group *(G) Fast Front Slow Down Laser 2d* ((G) FFSD Laser 2d), which is an inner group of *(G) Full Speed Safety Control* and *(G) Fast Front Slow Down* (see Fig. 2.6 for the containing network) is followed. The complete fault tree cannot be pictured here, therefore, only the fault tree of the behavioural group

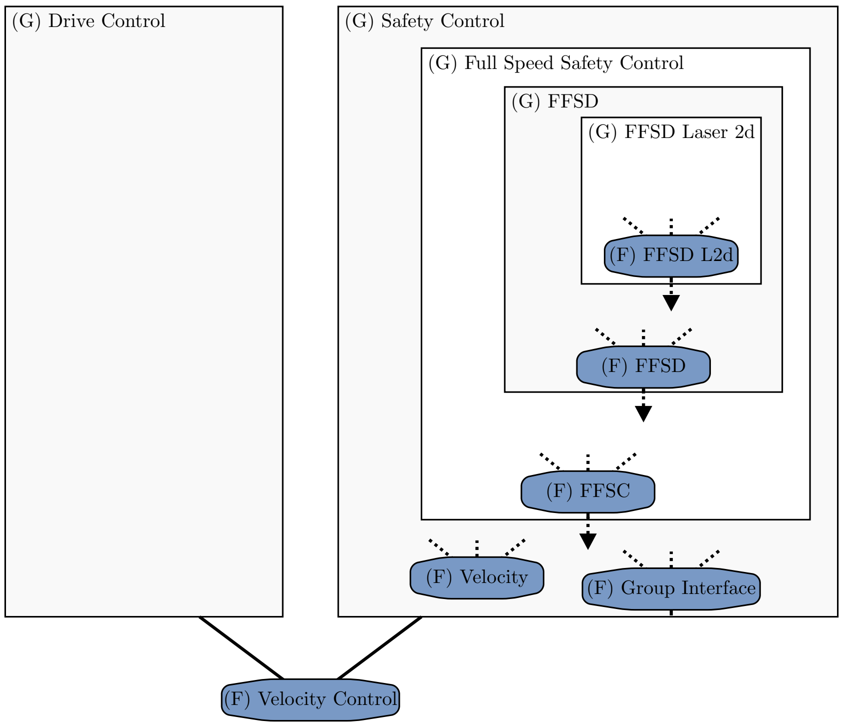


Figure 4.13: Overview of the main components (groups and fusion behaviours) that transfer the velocity on the example path used in the application example. The transfer starts in group *(G) FFSD Laser 2D* and ends in the fusion behaviour *(F) Velocity Control*.

(G) FFSD Laser 2d is modelled. The behavioural network of the group *(G) FFSD Laser 2d* is depicted in Fig. 4.16. As data dependence graph and activity dependence graph include the same components in this group, they are depicted in the same figure.

The network consists of three behavioural components that individually calculate a safe velocity value in dependence on the data from the 2D laser scanner for the three sides of the robot that influence the robots forward motion, whereby left and right mean the left and right part of the robots front respectively. The three components are continuously stimulated. Their activity value and velocity control commands are merged by a maximum fusion behaviour. The result is transferred to the group's output.

The fault tree for this group starts with the event *Faulty data signal output*. As the signal is directly transferred from the fusion behaviour to the group's output the fault tree modelling begins with the analysis of the fusion behaviour. Figure 4.17 depicts the fault tree. As explained earlier in this chapter, a wrong data output can be caused either by a faulty data value or a faulty activation value. The faulty activation value correlates with stimulating (and inhibiting) inputs of the fusion behaviour, which are not further analysed in this example. The faulty data value is analysed further proving all behavioural components connected to the fusion behaviour. As the three input behaviours all observe the same area of the robot (its front), all of them must compute wrong data in order to let the robot crash into an obstacle. The three components are therefore connected via an AND gate. Additionally, a *structural fault* event that includes all missing or redundant connections to the fusion behaviour is pictured. It belongs to both parts of the fault tree, as both stimulating/inhibiting connections and data inputs can be faulty.

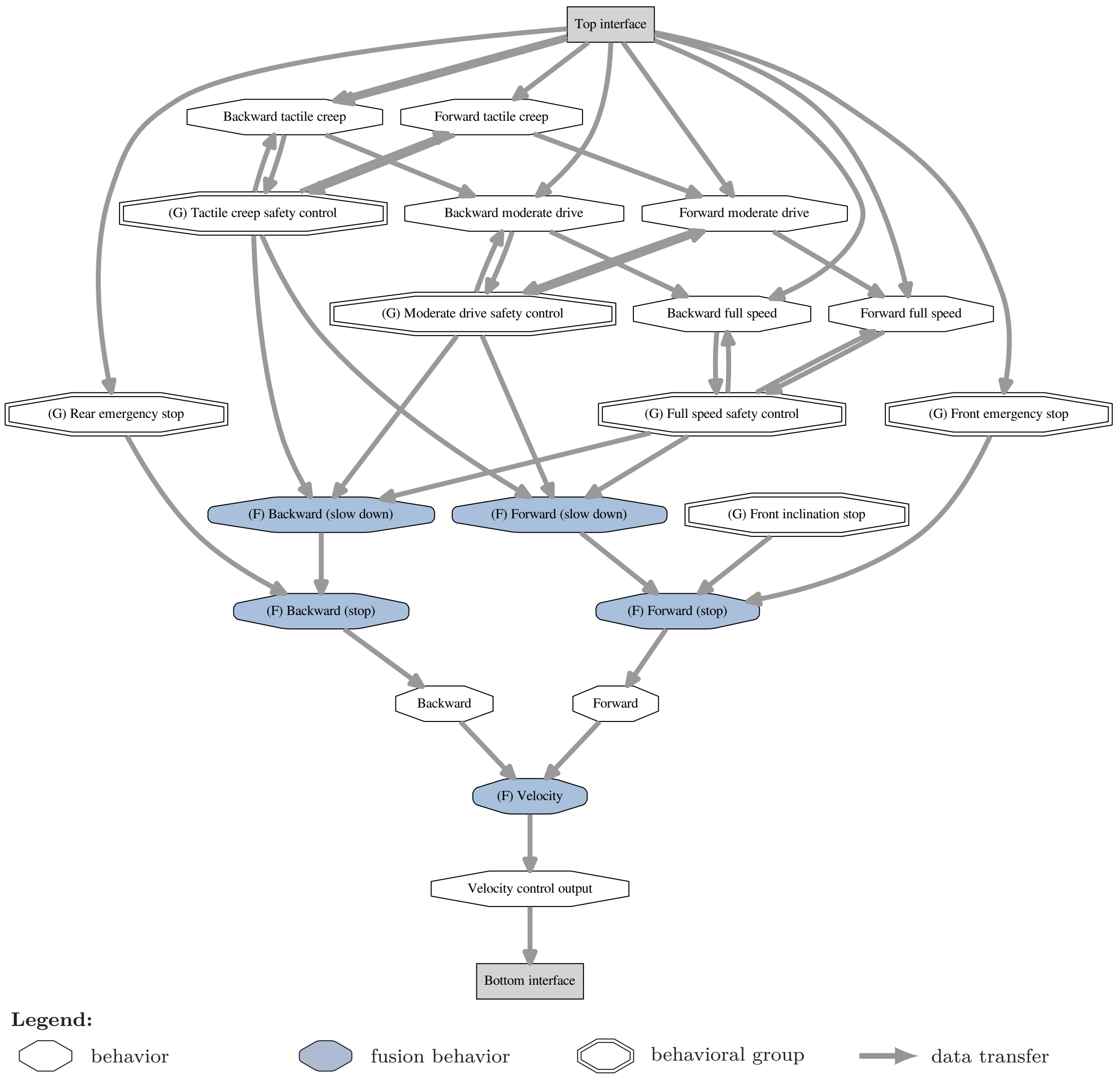


Figure 4.14: Data transfer graph of RAVON's safety control group to follow the flow of velocity control data from the groups output to its origin

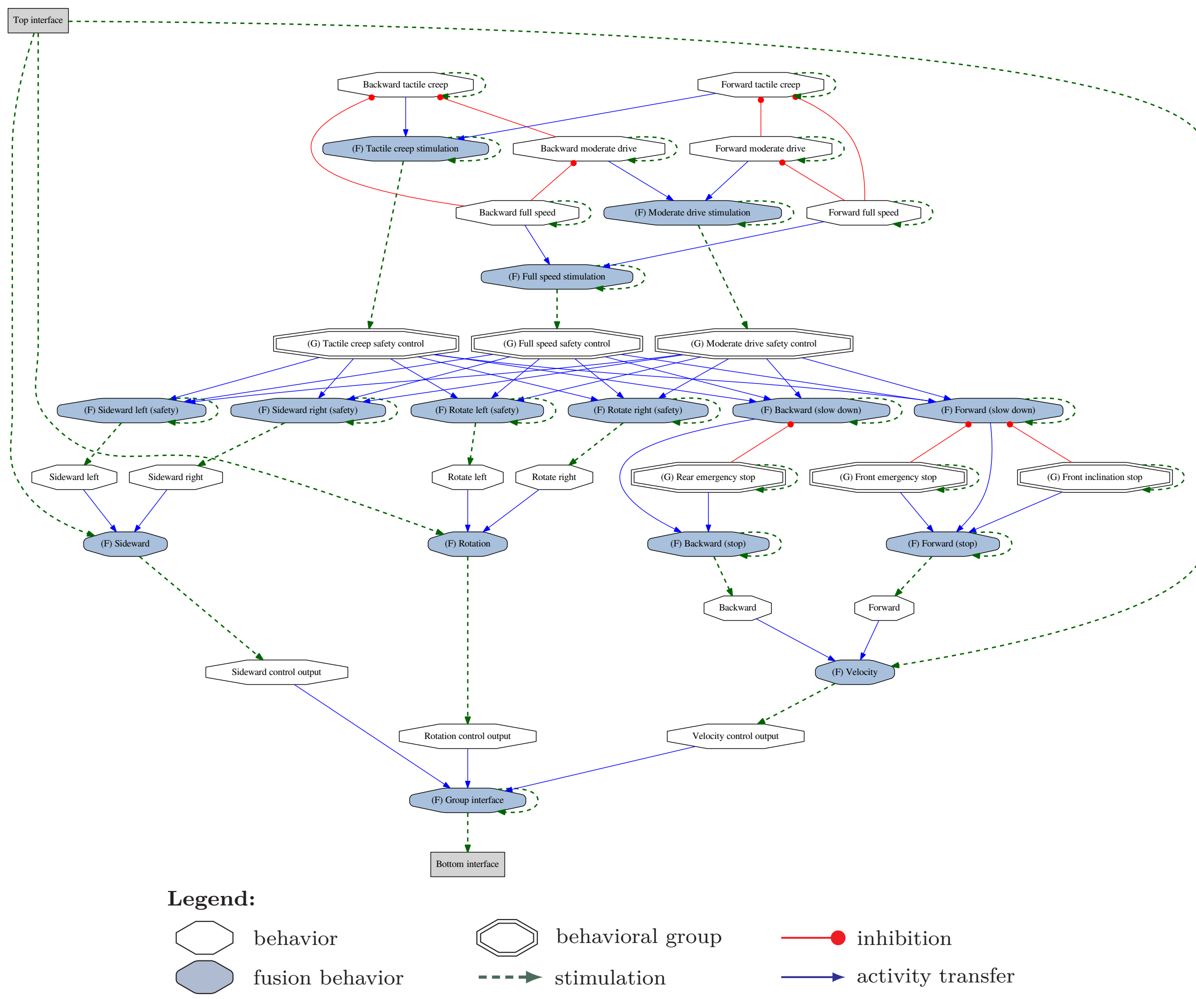


Figure 4.15: Activity transfer graph of RAVON's safety control group to follow the flow of activity from the groups output to its origin

Figure 4.18 shows the fault tree for the event of a faulty data signal coming from one of the three connected behaviours, in this case *FFSD Laser 2D* ***Centre***. It is mainly the same as the standard fault tree (see Fig. 4.9) with the difference that the event *wrong activation value* is marked as basic event. This is due to the fact, that the component is continuously stimulated and has no inhibition inputs. Therefore, the only event that needs further analysis is *faulty input data*. In this special case, the data used for calculating the output stems not only from the connected components but also from sensors, that provide their data for safety requirements. Thus, the next step is to further analyse the software on the one hand and analyse the sensor hardware on the other hand. Due to space constraints, the software fault tree is not depicted. For the sensor fault tree, the data processing chain and hardware internal problems are neglected, so that the causes of wrong sensor data are either *wrong data* or *no data*. Equal fault trees are created for the other two behavioural components *FFSD Laser 2D* ***Left*** and *FFSD Laser 2D* ***Right***

After the creation of the complete fault tree, the analysis procedure is applied. Here, a qualitative analysis is chosen as the main interest is on the relationship of system events. As the analysis procedure and the representation of the results is subject to many other

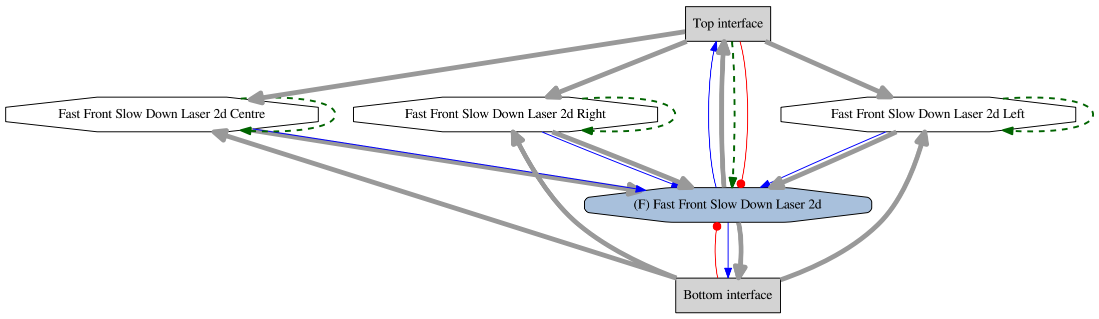

Top interface
Fast Front Slow Down Laser 2d Centre
Fast Front Slow Down Laser 2d Right
Fast Front Slow Down Laser 2d Left
(F) Fast Front Slow Down Laser 2d
Bottom interface

Figure 4.16: Network of the group *FFSD Laser 2D* realising a slowing down action in dependence on input data of the 2D laser scanner

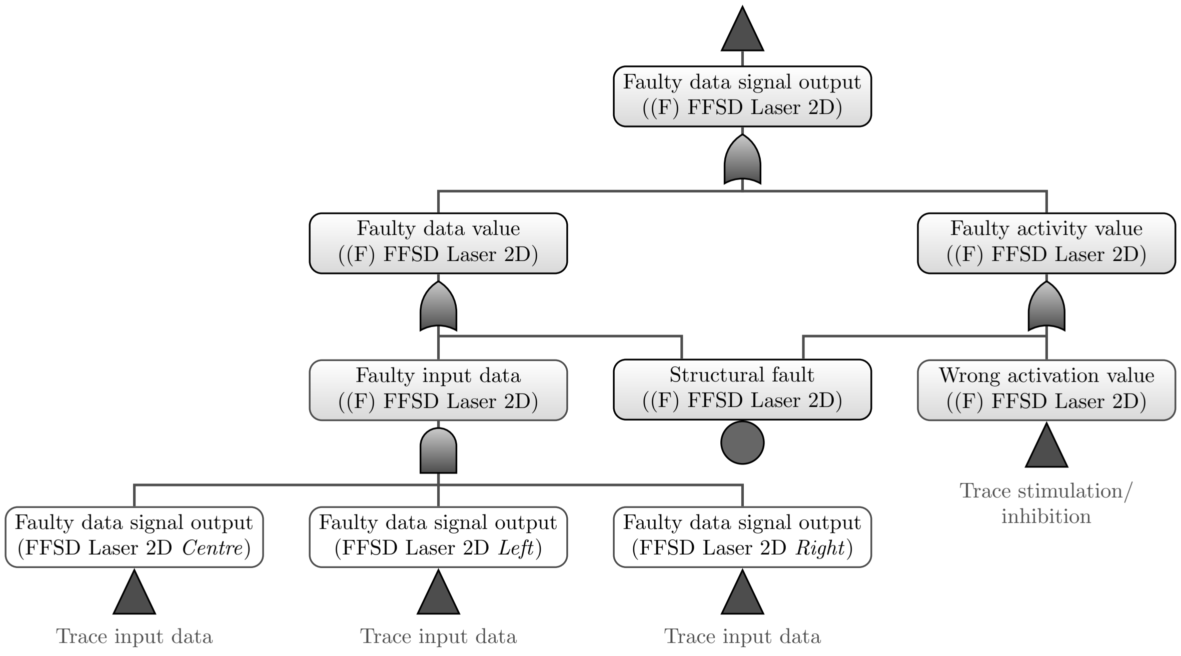

Faulty data signal output ((F) FFSD Laser 2D)
Faulty data value ((F) FFSD Laser 2D)
Faulty activity value ((F) FFSD Laser 2D)
Faulty input data ((F) FFSD Laser 2D)
Structural fault ((F) FFSD Laser 2D)
Wrong activation value ((F) FFSD Laser 2D)
Trace stimulation/ inhibition
Faulty data signal output (FFSD Laser 2D Centre)
Faulty data signal output (FFSD Laser 2D Left)
Faulty data signal output (FFSD Laser 2D Right)
Trace input data
Trace input data
Trace input data

Figure 4.17: Fault tree for the fusion behaviour of the behavioural group *FFSD Laser 2D*

research projects, here only the aspects belonging to the presented approach are discussed.

The example illustrates the combination of hardware related parts and software related parts in the same fault tree based on the embedded system RAVON. The influence of hardware parts (sensors) on certain software components and vice versa can be seen very well in the tree. It therefore provides the developer a detailed insight into their relationship.

Cut set analysis reveals those basic events that together trigger the top event. As mentioned before, events that are frequently contained in cut sets or even in several minimal cut sets should get special attention. In this example, one of the events that causes a failing bumper bar is contained in every cut set. Additionally, the different sensor systems also frequently appear in the cut sets. This is as expected and reflects the fact, that the perception of the environment is an essential part when building safe autonomous driving robots.

Another fact that can be noticed when analysing the cut sets is that some behavioural components from the control software appear more frequently than others. This is due to

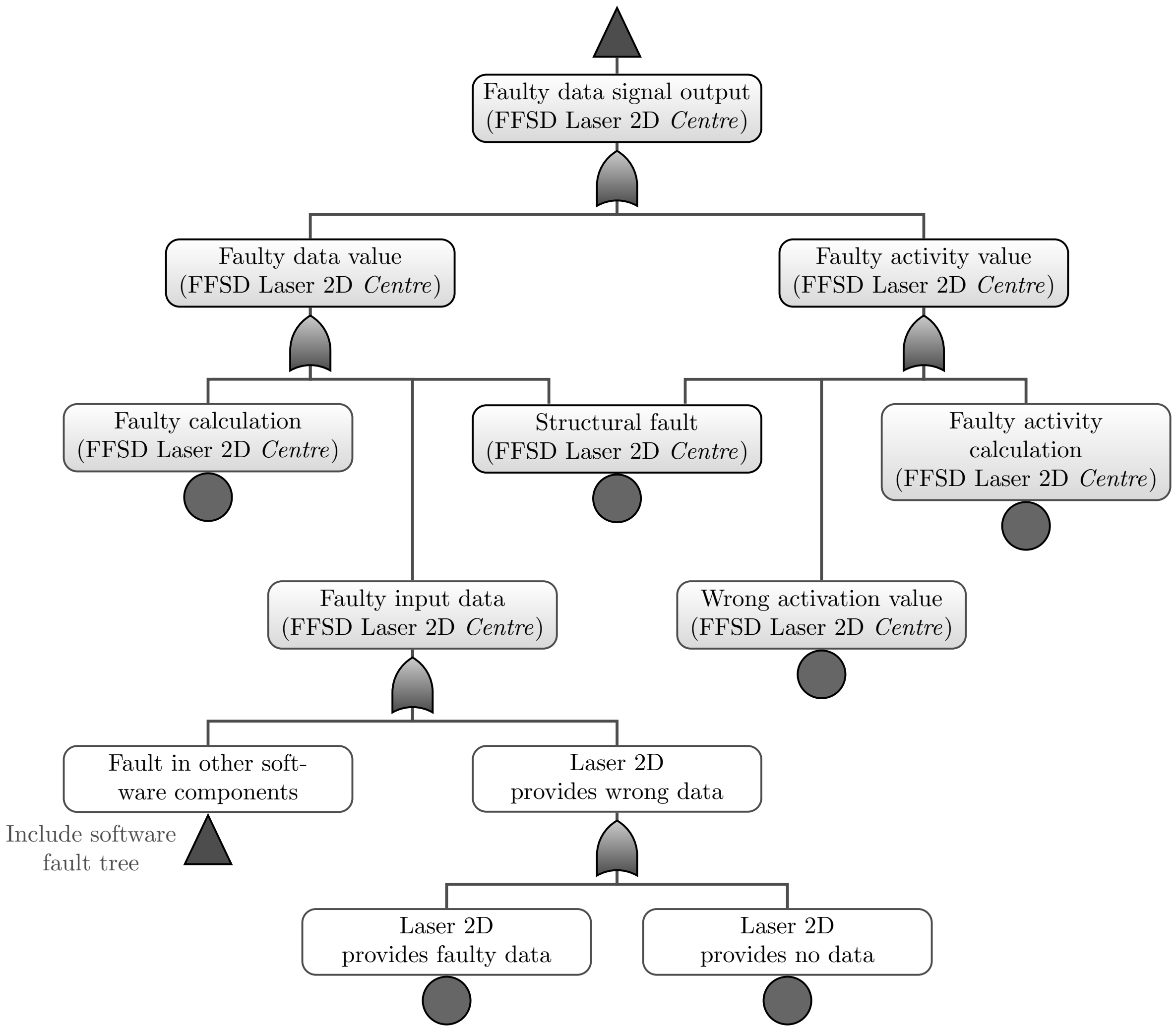

Faulty data signal output (FFSD Laser 2D *Centre*)
Faulty data value (FFSD Laser 2D *Centre*)
Faulty activity value (FFSD Laser 2D *Centre*)
Faulty calculation (FFSD Laser 2D *Centre*)
Structural fault (FFSD Laser 2D *Centre*)
Faulty activity calculation (FFSD Laser 2D *Centre*)
Faulty input data (FFSD Laser 2D *Centre*)
Wrong activation value (FFSD Laser 2D *Centre*)
Fault in other software components
Laser 2D provides wrong data
Include software fault tree
Laser 2D provides faulty data
Laser 2D provides no data

Figure 4.18: Fault tree for the behavioural component *Fast Front Slow Down* ***Centre***

the fact, that the data output of a behavioural component can be transferred to several others. In reverse, that implies that those components providing their data to several others are more important to further analyse than others. Besides, the event *structural fault* is often contained in the cut sets. This shows the relevance of correct connections in the behaviour-based control network.

The last step of fault tree analysis is to control the identified hazards. That means, the events that lead to the occurrence of the top event need to be improved in order to decrease the probability of their occurrence. In the case of hardware, this can be realised using more reliable hardware components or using additionally redundant systems. Similar possibilities can be applied to software components, where a higher reliability in that case, means to verify them in order to minimise the risk of programming errors. This must be done for the implementation of the single behaviour components as well as for the overall network structure. The latter will be addressed in Chap. 5.

The example has shown the application of the previously presented approach, applying fault tree analysis to embedded systems using a behaviour-based control. The resulting fault tree provides the developer or analyst a good insight into the relationships between hardware and software in respect to a special top event. For RAVON, the sensor systems have been identified as important elements in the fulfilment of safety requirements, as expected. As already several redundant sensor systems exist as well as redundancy in the control software, a further quantitative analysis using additional probability values of basic events would provide interesting information about the effective compliance of safety requirements. As many hardware components of the robot are individually designed, their failure rates can only be estimated using similar components as reference.

4.3 Data Analysis

"Robot failure data provides invaluable information to concerned professionals [...] as it indicates the results of the reliability-related effort put in during the design and manufacture phases of robots." [Dhillon 15] In order to prove the systems performance it must be tested under the compliance of the defined requirements. The data monitored during system execution give useful hints about the internal dynamic processes and help to identify failures. Especially for behaviour-based systems where the data calculation is not straightforward, but distributed to concurrent components, the analysis of dynamic data provides very valuable information. As illustrated in Fig. 4.1, data analysis can be assigned to the following analysis tasks: Gaining information about the relationship of system events, the detection of failure sources and the detection of system faults.

This section deals with data analysis methods applied to data derived from behaviour-based robot control systems. It is distinguished between static and dynamic analysis methods which relate to the analysis after program execution or in the meantime, respectively. The analysis only refers to the software parts of the robotic system. The hardware parts are considered in the description of the robot conditions that define the context of a test case.

It is well-known that "program testing can be used to show the presence of bugs, but never show their absence!" E.W. Dijkstra (1970), because it is impossible to cover the whole set of possible scenarios in test cases. This holds especially for robotic systems, that act in a complex environment which cannot be completely modelled. Nevertheless, testing is an essential part of software development as it allows to test a system under real conditions where also dynamic aspects can be proved. Numerous testing methods exist which differ according to the subject to test (software, hardware, embedded system, ...), the goal (functionality, reliability, ...), and the available time. What all have in common is to achieve the most possible coverage of scenarios in order to find as many bugs as possible. In this thesis, a test case will be defined by the following parameters

1. Property
2. Robot Action
3. Context
4. Monitored aspects

Property describes the requirement(s) that are verified with the test case. This can be for example the safety property, that the robot shall not hit an obstacle. The actual subject of the test case is described in the **robot action**. This can be a goal position a driving robot shall reach. **Context** defines the environmental conditions in which the robot shall act. These are obstacles, ground conditions, weather conditions, and also robot conditions like broken sensor systems, and so on. In the **monitoring aspect**, the system properties and additional data that shall be surveyed during the test run are defined. Here, for example the compliance of certain distances to obstacles or time constraints are mentioned. The definition of data to be monitored correlates to the property aspect. For robotic systems, software tests can be executed on the real system, as well as on a simulated system. The latter provides advantages in early development stages as hardware cannot break due to wrong software commands and several hardware conditions, like broken sensors can be simulated with less effort.

The goal of data collection is to gain as much information about the inner systems state as possible, in order to be able to find the causes of a failed test run. The question is which information should be collected and how. For behaviour-based systems, single components that realise basic robot actions, can be verified individually using common software testing methods. The speciality of behaviour-based systems is the interconnection of components that lead to the execution of sophisticated manoeuvres. Therefore, the focus of this thesis lies on the data communicated between the single components. These are *activity*, *target rating* and control data in the case of an iB2C network (see Sec. 2.2). The first two provide information about a components state, whereas the latter transfers the data to actually control the robot. In order to analyse failure cause, all this data need to be monitored at every point in time during a test case. Special techniques are necessary to sort and structure the data afterwards and to assist the developer in finding the problematic areas. The remainder of this section presents methods for analysis subsequent to test run execution (Sec. 4.3.1) and in the meantime (Sec. 4.3.2). Sec. 4.3.3 shows the application the methods on the example of the autonomous driving robot RAVON.

4.3.1 Static Analysis

In the context of this thesis, static analysis means that data is analysed after its retrieval during test runs. This is also called *offline analysis*. A positive result of test case execution is the achievement of the defined goals under the compliance of all constraints. Naturally, in the negative case, the cause of the violation of requirements (1a) or the missing of the goal (1b) has to be discovered. Accordingly, the two tasks need to be handled separately. For the first, the point in time where the violation occurs can be identified, which limits the data set to be considered. The latter is more complex to analyse as there is no particular time when the problem occurs.

But even for the positive case, that the goal is reached and all constraints are met (2), it might be useful to analyse the data set. For the system expert, this analysis can reveal irregularities in the system behaviour, that have no direct influence on the overall robot behaviour, but might lead to problems in other situations. Additionally, it can be used from non-system experts to gain understanding of the processes inside the behaviour based network during run-time. Regardless the special analysis task, for all applies the need of clearly structuring and representing the data. Furthermore, the handling of large data sets is time expensive or impossible and therefore demands for automation.

The remainder of this section introduces a concept to structure and represent data with a focus to support the developer in finding failures and their sources. Figure 4.19 gives an overview over the methods and their application areas.

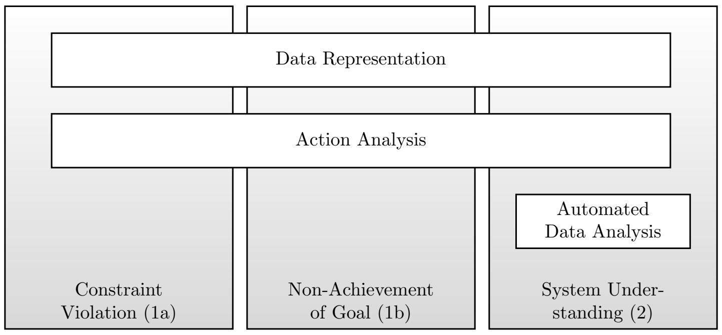


Figure 4.19: Analysis techniques (white boxes) and their application areas (grey boxes)

Data structuring

As mentioned before, the data collected in a behaviour-based control system is activity, target rating and control data. At every point in time during system execution, all components are in a certain state dependent on the values of the respective input and output values. Derived from that single datasets, a state for the complete system can be defined. In a first step the data collected at a certain point in time can be summarised as belonging to a system state.

Definition 4.4: System State

The **state of a behaviour-based system** is defined as $S_{System}(t) = \{BS_{B_0}(t), BS_{B_1}(t), \ldots, BS_{B_n}(t)\}$ with n being the number of behavioural components in the system.
$BS_{B_i}(t) = \{a_{B_i}(t), r_{B_i}(t), \iota_{B_i}(t)\}$ defines the **state of a behavioural component** B_i, with $a_{B_i}, r_{B_i}, \iota_{B_i}$ being the activity, target rating and activation of the component.

The stimulation and inhibition values of the component are implicitly included through the activation (see Sec. 2.2) and are therefore not necessary for the definition of the state of a behavioural component.

The system state is the basic structuring level. The system state at the time a problem occurs is called *critical system state*. In the simplest case, this is the time where a constraint is violated. Starting from critical system states a detailed analysis of the components and the relation between the components belonging to this state takes place. In the following, three techniques to analyse the data set are discussed.

Data Representation

A structured representation of the data is essential for all further analysis. The components and their corresponding logged data belonging to a system state are arranged in a table according to several appropriate aspects, e.g. components are grouped according to their activity level and only the data in a certain time interval (e.g. before the crash) will be displayed (see Fig. 4.20). With this representation, the developer has a structured view of the data, which helps to detect the irregularities, that indicate the failure source. Then, a closer look to the implementation of the respective component is possible to find the cause of the wrong data signal.

State	Component	Activity	Control Data	Data Value
1	B	1		
	C	1	v	-1.0
			av	0.0
	A	0.8	v	1.0
			av	0.0
	Fusion	0.8	v	1.0
			av	0.0
	D	0.4	v	-1.0
			av	0.0
	E			
	F			

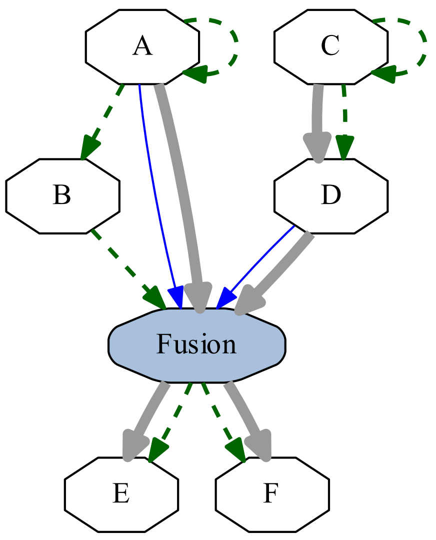


Figure 4.20: Table and graph representation of the behaviour-based network. The table provides information about the states of each behavioural component in a network belonging to one system state. The graph view simplifies the understanding. The abbreviations *v* and *av* stand for the control data *velocity* and *angular velocity*.

A problem with this basic method is, that more than one problematic spot could be identified, as data is distributed in the behaviour-based network. In order to find the root cause of the problem, it is helpful to discover the path of the data inside the network. For that task the data can be arranged according to the sequence of the components which can be done with the help of a typed behaviour dependence graph as presented in Sec. 4.1.

Action Analysis

The manual analysis of the structured data set is still a time-consuming task, and expert knowledge is needed in order to find anomalies. To mitigate the effort, a second technique is introduced. The basic idea is to automatically distinguish between known (and proved) and unknown system states.

A simple robot action like drive forward/backward or turn left/right, can be described as a set of system states. The overall movement of a robot on the other hand, can be described as sequence of several simple movements. As example, Fig. 4.21 shows a scene where several way points are marked. Consider a robot task, that is to reach the goal defined at point 6. The way points 0 and 2 must be included in the robots trajectory when driving there. To do this, the robot has to drive straight forward, turn left and drive forward again. These simple movements, so-called *basic actions*, can also be identified in more complex scenarios. Deviations from the basic actions can again point to problems in the network or the component implementation.

The determination of basic action as sequence of system states turns out to be unfeasible. Environmental influences lead to changes in system states that make it nearly impossible to identify one and the same state in different situations. Therefore, a more rough definition of a basic action is chosen

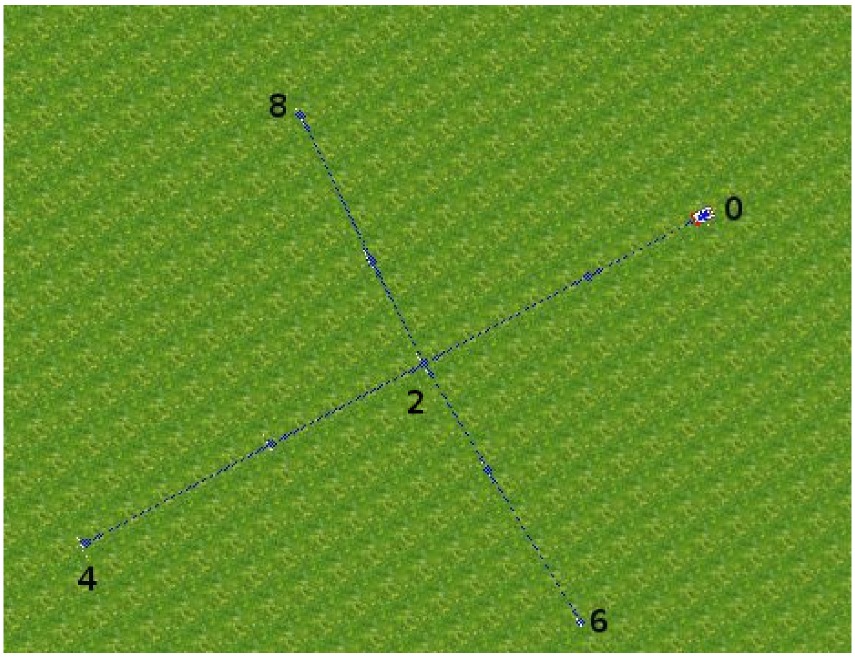


Figure 4.21: Simple scenario with four way points for the testing of basic drive manoeuvres

Definition 4.5: Basic Action

A **basic action** is defined by those components, that are active during a robots action.

At best the collected data of a test run can be reduced to a sequence of basic actions. States that cannot be assigned to a basic action point to possible failure sources. The state can for example include additional or less active components. Both give hints to the developer where to find the cause of the faulty robot behaviour.

This technique is also useful in finding actions that lead to the missing of the goal as mentioned in the beginning of this section and in Fig. 4.19. For example, assume the example above (Fig. 4.21) and the case that the robot never reaches point 6. The analysis of the sequence of basic actions reveals that the movements sticks in the turn left action. Further inspection of the responsible components may show that the activity and target rating values are not set correctly.

Another application of this method is to find deviations in the specified system behaviour without having the robot behave obviously wrong. In the example above, the sequence of basic actions does not only consist of drive forward, turn left, drive forward. Instead of the simple turn left manoeuvre, an oscillation between turn right and turn left actions can be identified. This happens because the robot cannot turn in place and tries to find the best trajectory under the given constraints. Further interesting insights in the control network gives the analysis of the components that are active in addition to the behaviours belonging to a basic action. Both usages give the system developer and also a non-expert the possibility to gain understanding about the actions inside the complex network.

Action analysis can be automated. Basic actions can be learned by monitoring the data during execution of the corresponding simple movements. Data from complex test runs can then be automatically compared with the data from basic actions. Section 4.3.3 shows the application of the action analysis on the autonomous robot RAVON and introduces a tool that facilitate the analysis.

Automated Data Analysis

The presented techniques to analyse the big data set collected during a robots test run can be supported by several automatic data checks. A very useful information for the developer is the **influence of a component** inside the network. The component's influence is equal to its activity. The higher the activity, the more influence the component has in the network. In that context an analysis that outputs components sorted according to their activity value in *always active*, *never active* and *sporadically active* is very useful. The representation can be done either in the form of a table or with different colours or markers in the graph structure. Expert knowledge is needed to prove if the classification of the single components satisfies the requirements. In combination with the action analysis, this method provides information about the relevance of components for simple robot actions.

Another check refers to **signal variability**. Constant signals or heavily changing signals can refer to faulty implemented components. A further, very important analysis task in the context of behaviour-based systems is to search for oscillations. For the iB2C it was developed in the diploma thesis of the author [Wilhelm 08] and published in [Wilhelm 09, Proetzsch 10a]. Because of the competing components, it is possible to generate an unintended oscillating robot motion. For an autonomous driving robot, this can for example occur, when the robot faces an obstacle when driving to a goal. The component responsible for evading the obstacle forces the robot to turn left, where the one responsible for driving to the goal wants the robot to turn back in order to keep the shortest path. This results in the robot getting stuck in front of the obstacle and just turning left and right. Such a situation can be identified in the activity data of a monitored robot manoeuvre. The automatic check for oscillations in a data set determines the associated components, the type of data and the point in time where the oscillation occurs. With this information, the developer can improve the systems functionality.

This section presented several techniques to analyse the data collected in a behaviour-based control system after the test run execution.

4.3.2 Dynamic Analysis and Representation

Usually, a series of test runs is executed to prove the robots behaviour for the same subject under different conditions. When using a simulation environment that simulates the hardware parts of the embedded system, test runs can be performed automatically. The success or failure is reported and the data can be analysed afterwards as described in the previous section. Sometimes, it might be useful to survey the execution of a test, for example, in the case the developer wants to test different implementations and see the robots reaction. For this purpose, it is useful to not only observe the acting robot, but to gain insights into the data flow inside the control network.

The FINSTRUCT tool (see Sec. 2.4) provides the possibility to survey the state of individual components by displaying their current activation, activity and target rating value with coloured bars. Additional information provides the integration of the previously mentioned oscillation detection method, which allows to mark oscillating components in the graph representation of the network. This representation is very useful when wanting to observe the states of only few components, but can be very challenging for many components, which might be additionally distributed in different parts of the control network. For that cases it is useful to chose only those information that is necessary for the situation to test and display it in an adequate manner.

Consider the previously introduced example that RAVON shall reach a goal while avoiding obstacles lying on its way. As already described, the driving commands stem from the drive control part of the control network, which can be influenced by commands coming from the safety control part (see Sec. 4.2.2, Fig. 4.13). An interesting question for the developer or analyst is, which part of the network is responsible for the robots action at the current point in time. Both software parts output their activity and control data which is merged and further processed until it is finally executed by the actuators. The data can be surveyed using FINSTRUCT in an uncomfortable manner, as several parts of the network need to be observed at the same time.

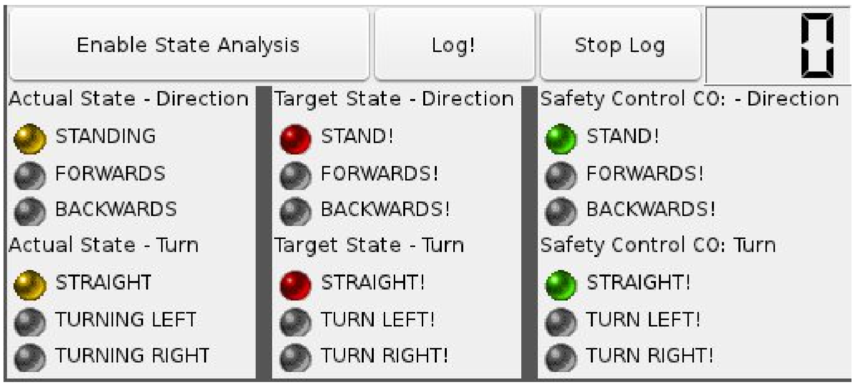


Figure 4.22: Example for dynamic data representation during test runs of complex systems in a frame of the MCAGUI [Gänßinger 11]

The approach used in this thesis utilises the MCAGUI to solve this task as this tool allows to visualise the robot in its environment during a (simulated) test run as well as several user-defined information e.g. data flowing in the system. For the example above it is preferable to visualise the outputs of the two different parts of the network as well as the command that actually controls the robot at the current time. Figure 4.22 depicts the view for the example. The view displays three columns for the three parts of network: *actual state* for the final driving command output, *target state* for the output of the control part and *safety control* for the output of the safety control part. Each column is divided into two aspects (*direction* and *turn*) with three states, respectively. During a test run execution, the coloured diodes indicate the actual output of the groups. This abstract view provides a good overview of the current state of the system in respect to the velocity and turning commands. It can be easily adapted to show other information according to a certain problem. Three steps serve as guideline for the adaptation to new problems

1. Identify the relevant information

2. Monitor data (and process them if necessary)

3. Display data in GUI

This representation of dynamic data supports the developer in understanding system behaviour and can speed up fault detection.

4.3.3 Tooling and Application Examples

For the execution of test cases in a simulation environment the *Test Automation Runtime System* (TAURUS) is used [Proetzsch 10b, Schäfer 11]. It allows for the automatic execution of several different test runs, checking the violation of given constraints, monitoring the progress and journalise the state of the control system. TAURUS is itself implemented as a behaviour network which is docked in between the robot control system and the user interface (see Fig. 4.23). From this position TAURUS can control and monitor the system by setting properties according to a special test run and in turn observing data in the control system.

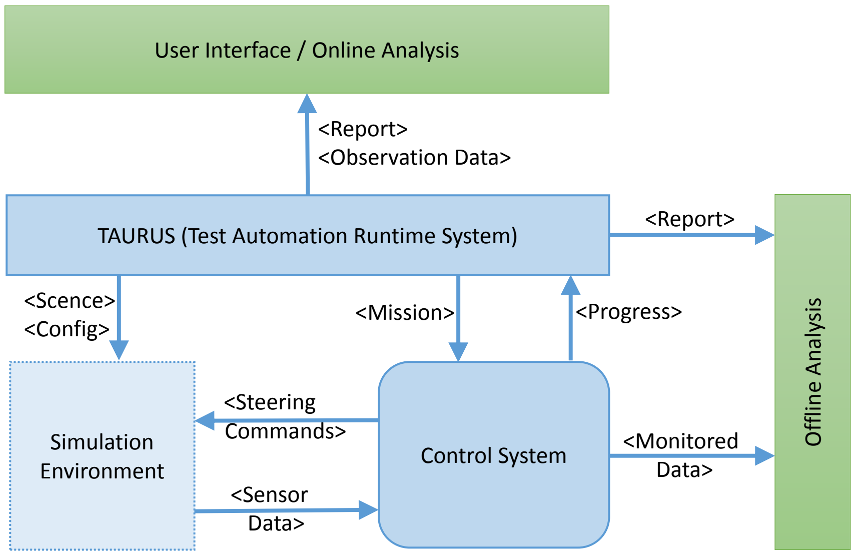


Figure 4.23: Test case execution and evaluation using the TAURUS tool and different analysis methods (adapted from [Schäfer 11])

Test cases consist of a *scenario description* containing information about the environment and a *mission* the robot shall accomplish. In the case of driving robots, one or several *paths* can be defined for each mission containing a sequence of way points. Additionally, system properties that shall be supervised during a test run can be defined as so-called *invariants*. They can be defined for all recorded data. TAURUS monitors the violation of invariants, logs the respective data and uses them for a test report, created after each test run. Examples for invariants are time-limits, distances to obstacles, that must be kept, etc. The test report summarises the results of a test run. Among others, it holds information of the elapsed time, and the time stamp of invariant violations.

In the context of this thesis, the robot control system is additionally equipped with several logging units to collect the data for the static analysis as described in Sec. 4.3.1 and the report of TAURUS is extended to provide additional information. Furthermore, the

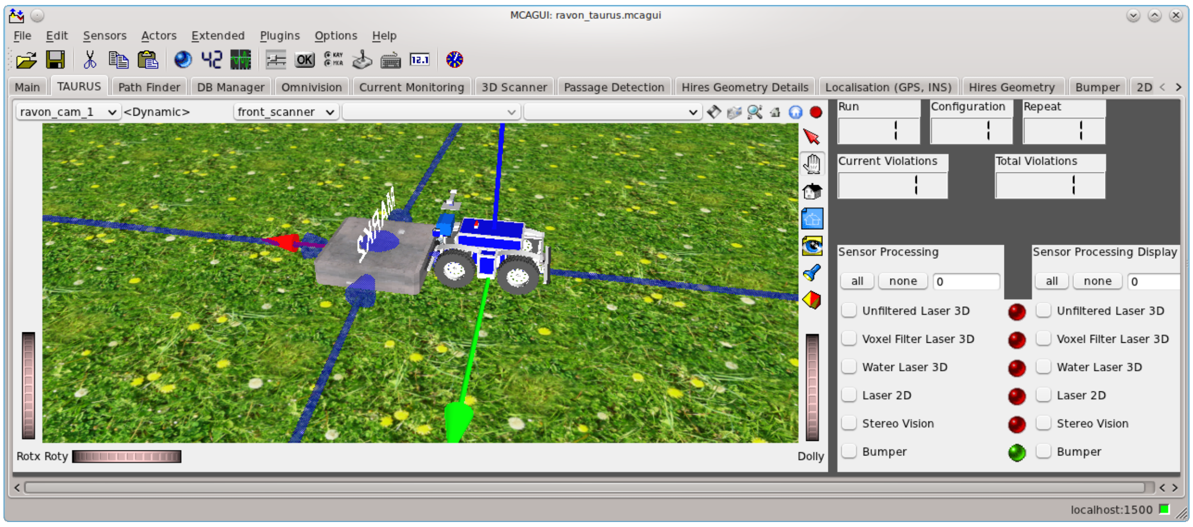

MCAGUI: ravon_taurus.mcagui
File Edit Sensors Actors Extended Plugins Options Help
Main TAURUS Path Finder DB Manager Omnivision Current Monitoring 3D Scanner Passage Detection Hires Geometry Details Localisation (GPS, INS) Hires Geometry Bumper 2D
ravon_cam_1 <Dynamic> front_scanner
Run
Configuration
Repeat
Current Violations
Total Violations
Sensor Processing
Sensor Processing Display
all none 0
Unfiltered Laser 3D
Voxel Filter Laser 3D
Water Laser 3D
Laser 2D
Stereo Vision
Bumper
Rotx Roty
Dolly
localhost:1500

Figure 4.24: TAURUS view in the MCAGUI. The relevant test parameters are displayed (run number, configuration number and repeat number, as well as the number of violations in the current test run and the number of all violations in the already executed test runs). Additionally the configuration of the robot is displayed.

evaluation of the test runs is modified to be performed in real-time instead of once at the end of the test run. This enables the dynamic analysis and the visualisation of the current state of a test run for the user.

To exemplify the application of tools in combination with the previously described action analysis method, a safety property of the off-road robot RAVON shall be checked. The requirement can be writen as *The robot shall not hit obstacles* or rather in more detail *The robot shall hold a distance of at least 50 cm to obstacles*. For the complete definition of test cases, different paths (including different drive manoeuvres) and different robot configurations are pretended, additionally.

In this section, the following test case (including three test runs) has been chosen:

1. The robot shall drive to point 4, which lies straight ahead in relation to the robots position and orientation (see Fig. 4.21).
2. An overhanging obstacle is positioned in the middle of the direct path
3. Invariant: distance to obstacles greater then 50 cm
4. Different configurations
   (a) All sensors work
   (b) 3D Laser Scanner fails
   (c) Only bumper system works

The data to be monitored have been chosen according to this test case. As the fulfilment of the requirement is related to the drive control of the robot, control data regarding the velocity and turning movements have been monitored. Furthermore, behavioural data of the components responsible for changes in the mentioned control data have been recorded.

After the definition of the test case and the data to be monitored, the test cases have been executed by TAURUS. Additionally, FINSTRUCT was used to supervise the state of the behaviour-based network and to visualise oscillations.

The TAURUS report revealed an invariant violation in test runs 2 and 3. For this case, a comparison between the three test runs based on the sequence of basic actions is an appropriate procedure to find differences in the robot manoeuvres and possibly hints to the failure source.

As mentioned before, the system states belonging to a basic action can be determined automatically. For that purpose, the ANALYSER tool has been developed in the scope of this work which determines the system states during simple test runs in an empty scenario. The tool searches for components that are active during the whole test run. These components are the relevant components for the respective basic action and are stored together with a user given action name in the so-called action database. The basic actions to be identified for the presented test case are *straight forward* (SF), *turn left* (TL) and *turn right* (TR). They has been determined for all three configurations as RAVON uses different drive modi (see 2.3), which are implemented in different behavioural networks and therefore using different components.

Once the database was filled with all basic actions, the monitored data from the test run with the obstacle could be compared with them. This can also be done automatically with the ANALYSER tool. It compares every system state with the actions stored in the action library and outputs the name of the most suitable one. Additional active components are also displayed.

For configuration 4a) of the test case, the result of the action analysis was a sequence of SF, TL, TR and SF actions. For configuration 4b) and 4c), only the SF action could be determined. A more detailed analysis of the system state at the point of time of the collision with the obstacle revealed that no components responsible for evasion were active. Further analysis of the internal implementation of the corresponding components discovered that either the sensor (planar laser scanner) could not detect the obstacle, or the faulty implementation of the component that evaluates the sensor data can cause this fault.

The example showed the applicability of the presented analysis concept on a real-world task. The identification of critical parts of the system that are responsible for wrong robot behaviour is supported by the developed methods and tools facilitate their usage. The automation of the action analysis process allows for analysing test cases in less time, which enables the developer to execute more tests. This contributes to the detection of more failures and additionally, leads to a better understanding of the system's internal behaviour.

4.4 Discussion

In this chapter several techniques to analyse behaviour-based robotic systems with respect to their reliability and safety requirements were introduced and applied to the autonomous robot RAVON to demonstrate their use. A fundamental difference to monolithic control programs is the graph-like structure of behaviour-based control software. This offers the possibility to use standard graph operations like the strongly connected components algorithm for cycle detection. In Sec. 4.1, several methods were presented which are used to gain knowledge about the system or to represent the system in a way that the developer or analyst can easily distinguish between relevant and irrelevant information. One of these is the typed behaviour dependence graph, which allows to represent only those parts of the network that transfer the respective information. This representation is the basis for several other analysis methods. As the behavioural components of the behaviour-based network have a standardised interface, behavioural data like activity and target rating can be automated traced accurately. For control data arises the problem, that only the full control vector can be traced through the network which can contain an arbitrary number of data. The tracing of only one control factor like velocity cannot be automated easily, as long as there exist no naming rules for the in- and outputs connection points of the behavioural components. However, for correct designed behaviour-based systems, this should not be a problem as different aspects of control are encapsulated in different parts of the network where the control vectors only include information relevant to the respective aspect.

In Sec. 4.2, fault tree analysis as method to analyse safety of a system is considered. The application of fault tree analysis to engineering systems is a well-known and widely used technique. It provides deep insight into the system behaviour, shows the relationships of components and allows to concentrate on single specific failures. Several tools exist that support the user in the creation and analysis process. Research is also done for applying fault tree analysis to software systems. The structured composition of behaviour-based control networks facilitates the generation of fault trees. In Sec. 4.2.1 fault tree models for standard behaviours and fusion behaviours are introduced. Together with the behaviour dependence graphs presented in Sec. 4.1 it could be shown that it is very easy to create a fault tree for a behaviour-based network. The interfaces between hardware and software are automatically integrated as data exchange is only done via dedicated input and output ports. This allows for gaining good insight into the relationships of hardware and software events. But, as could be noticed in the example in Sec. 4.2.2, there are also problematic aspects of fault tree analysis. One is the modelling of fusion behaviours. Here, system expert knowledge is needed in order to find out the correct combination between the connected behavioural components. This makes an automatic fault tree creation difficult to realise.

A further, well-known problem is that a fault tree grows very fast and thereby cannot be visualised in a suitable manner. This can be solved partially by displaying only smaller parts of the tree and marking the connections to the overall tree, or, in the special case of behaviour-based software systems, omitting the inner structure of a group and just integrating the groups interface as standard behavioural component in the tree. This in turn leads to problems when trying to understand the overall relationship of events. The visualisation of fault trees and analysis results for embedded systems in a manner that fulfils the needs of system engineers and safety analysts is an open research task. This problem was also subject of a research project utilizing RAVON as demonstrator. The results are published in [AlTarawneh 14, AlTarawneh 15]. The intelligent visualization of cut sets is subject to the work of [Al-Zokari 10, Al-Zokari 12]. Both visualisation techniques aim at the combined representation of hardware and fault tree components, respectively.

Software is mainly unattended. Due to the standardised modelling approach of behavioural components into fault tree elements presented in the thesis at hand, the mapping can be done vice-versa to visualise the analysis results in the network representation.

Fault tree analysis is a good strategy to reveal dependencies between components. The presented method allows to execute the analysis for hardware and software in one procedure. For large systems as the RAVON example is, it might not be the analysis technique of choice to make a statement about the systems reliability or safety property as the fault tree creation is time-expensive and the risk of missing relevant parts is high. However, fault tree analysis can be established in combination with other analysis methods, which require detailed knowledge about the interdependencies in specific parts of the whole system.

The evaluation of data monitored during the execution of a test case was the topic of Sec. 4.3. Testing is, despite of its deficiencies, the method of choice to evaluate the robots performance. Though, the analysis of the monitored data is system dependent and needs expert knowledge in order to find the relevant information that help to improve the overall system. The techniques introduced in this section demonstrate a first step in automating the analysis for the subject of behaviour-based systems. The presented methods and tools assist the developer or analyst to find the root causes of failures and, furthermore, they provide the possibility to analyse the system also for non-experts. They help to gain understanding about the system interdependencies and to improve the systems functionality and allow to make statements about the reliability of a system by using (quantitative) fault tree analysis or testing (calculating the mean-time-to-problems or mean-time-to-failures) and to find the causes of failures using (qualitative) fault tree analysis and data-analysis methods.

5. Formal Verification of Behaviour-Based Systems

In contrast to testing, which tries to find errors in the control system in a special test case, formal verification assures that a system fully satisfies a given specification in all situations. Especially for robotic systems, that act in a natural environment, it is a major task to ensure the fulfilment of several safety requirements. Already in 1995, the authors of [Espiau 95b] discussed extensively the need for specification and verification and former approaches evolving from the computer science area to the robotics area. In [van de Molengraft 11] published in [Corke 11], this topic is also discussed with a focus on the special challenges that come with robotic systems, e.g. the interaction with their environment.

Several formal verification techniques exist. Among these are deductive methods using theorem provers and model checking methods. Model checking techniques have already been applied in the development process of robot control systems to gain correct robot behaviour, e.g. in [Webster 11, Lowry 97] (see also Sec. 3.3). They offer two significant advantages over other techniques: First, model checking is a highly automated process, which reduces the verification effort. Second, model checking yields counterexamples and witnesses, respectively, which help finding the cause of errors.

The reason why formal verification is not commonly used for the validation of robotic systems is the high complexity. Software and the corresponding requirements must be formalised in a first step. This can usually only be done by verification experts and can take a lot of time. For large and complex systems, it is often not possible to fully formalise them. A further open problem in this context is the formal modelling of the combination of hardware and software parts and their interaction with the environment. In a second step, properties are proven to be correct for the given system model by a chosen model checker. The last step includes the interpretation of the results which needs an experienced user as well.

This leads to the following requirements that shall be addressed in this thesis:

- Automation of the formal model creation process

- Combination of hardware, software and environmental interactions into the formal model

- Tool support for the interpretation of the model checker's results

In the work at hand, behaviour-based systems are in the focus, which are used to realise complex robot behaviour. The central aspect in these systems is the network structure, where simple behaviours are interconnected to achieve a more sophisticated one. This leads to the approach used in this thesis and described in [Armbrust 12, Armbrust 14a] to focus on the formal verification of the interconnections of behaviours.

As connections between components must be programmed manually by the software developer, failures (missing connections, connections to wrong components/ports) are highly possible. A formal representation of the network structure is achieved using the input language of the Uppaal toolbox. The representation comprises the components, described via their common interface (see Sec. 2.2), and the connections between them. The Uppaal model checking toolbox has been chosen because of its powerful graphical user interface, a stand-alone verifier that can also be called from the command line and on remote machines and the possibility of saving automata in an XML-based format. The latter two features are important for the automation of the verification process and the integration in the finstruct tool.

The work described in [Armbrust 12, Armbrust 13b, Armbrust 14b] focused on the inner structure of the (software) control system. Environmental interactions as well as the hardware of the robotic system is disregarded, although they have a great influence of the fulfilment of safety requirements. Intuitive questions like "How many/Which sensors must fail to lead to wrong robot behaviour?" or "Does the evasion behaviour get active in case of obstacles?" represent important safety problems, but they cannot be answered by verifying only the software system. A robot can fail for example when sensor data is not correctly delivered to corresponding safety behaviours because of missing connections. A further interesting task is the verification of robot behaviour under the assumption of failing hardware. A possibility to integrate major hardware parts and their connections to the control software as well as environmental situations is therefore a crucial task.

The work at hand presents a novel concept that allows to integrate the sensor processing chain and environmental situations into the model checking process. Sensors and environmental situations are formally defined and integrated into the system model in order to be checked together. To simplify the verification process as much as possible, automated processes and graphical support is additionally developed.

This chapter is structured as follows: Section 5.1 gives a short introduction into model checking and the Uppaal toolbox. The basic approach of formally modelling a behaviour-based system is described in Sec. 5.2. In Sec. 5.3 the novel concept integrating sensors and the environment of the robot into the formal model is presented. Section 5.4, gives several examples on the usage of model checking on the behaviour model. Here, the extension to the finstruct tool to support the verification process graphically is described as well. Quantitative aspects like computation time and memory consumption are discussed in Sec. 5.5. Sec. 5.6 discusses the application of model checking to behaviour-based systems.

5.1 Model Checking

Model checking [Clarke 99] is a formal verification technique that proves a given property of a system by systematically exploring the state space. Figure 5.1 depicts the process of model checking. A model of the system and a corresponding property are given to the model checker. If the property is fulfilled, the model checker returns a notification and the next property can be checked, if not, a counterexample is provided and the system must be revised.

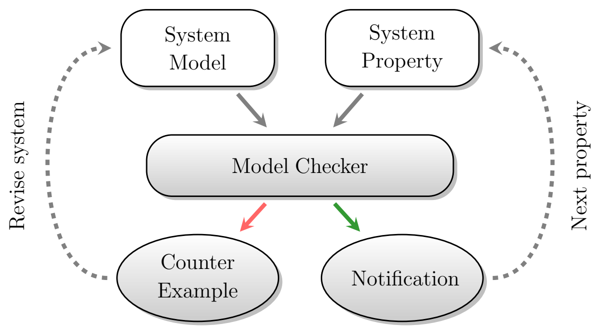


Figure 5.1: The general model checking process

A formal definition of the model checking problem based on the definition given in [Clarke 08] is

Definition 5.1: Model Checking Problem

Let M be a Kripke structure (i.e. a state-transition graph). Let f be a formula of temporal logic (i.e. the property). Find all states s of M such that $M, s \models f$.

Naturally spoken, model checking determines whether M is a model for the formula f.

The system model is given as a *Kripke structure* which can be seen as kind of automaton. The following is a definition of a simple Kripke structure based on [Clarke 01].

Definition 5.2: Kripke Structure

A Kripke structure over a set of atomic propositions A is a tuple $K = (S, R, L, I)$ where
S is the set of states
$R \subseteq S^2$ is the set of transitions
$I \subseteq S$ is the non-empty set of initial states
$L : S \rightarrow 2^A$ labels each state by a set of atomic propositions

Temporal logics are introduced for the specification of system properties. They use atomic statements and boolean operators, such as conjunction and disjunction to formulate a property. Temporal logics differ in the number and meaning of their operators. The widespread used logic for model checking tasks is the CTL*. It is a superset of the *computation tree logic* (CTL) and the *linear temporal logic* (LTL). Its formulae are based on computation trees that can be extracted from a Kripke structure. For CTL* there exist two *path quantifiers*. The *universal quantifier* is used to express that a formula holds for all computation paths, whereas the *existential quantifier* expresses that there exist at least one path where the formula holds.

1. **A**: for every path
2. **E**: there exist a path

Path quantifiers can prefix a combination the five basic temporal operators (p, q: formulae):

1. **X**p: p holds in the next state of the path
2. **F**p: p holds in some state of the path
3. **G**p: p holds in all states of the path
4. p**U**q: p holds until q holds
5. p**R**q: q holds up to the state and including the first state where p holds.

The combination of path quantifiers (**A**,**E**) and temporal operators (**F**,**G**) yields the following cases:

1. **AG**p: p holds in all states of all paths
2. **EG**p: p holds in all states of some path
3. **AF**p: p holds in some state of all paths
4. **EF**p: p holds in some state of some paths

In other logics the path quantifiers **A** and **E** are written as $\forall$ and $\exists$, while the temporal operators **F** and **G** are written as $\Diamond$ and $\Box$.

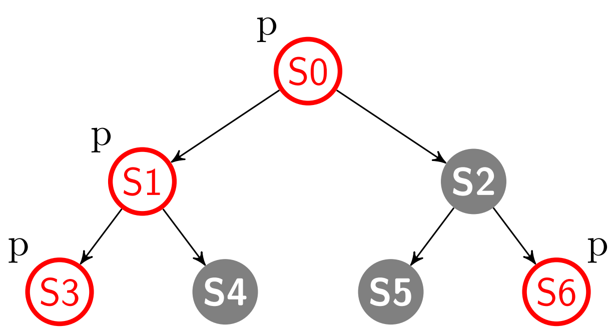


Figure 5.2: Computation tree example

An important advantage of model checking is that it provides traces for a witness or a counterexample, which help to find problems and failures in the system. Witnesses are given in the case an existential property is evaluated to true, counterexamples are shown when a universal property is evaluated to false. Traces represent a path in the tree and therefore consist of states and transitions. Figure 5.2 gives an example of a computation tree. A witness for the fulfilment of the formula **EG**p is given by the path $S0 \rightarrow S1 \rightarrow S3$. In contrast, the formula **AG**p does not hold. A counterexample is the path $S0 \rightarrow S2$.

There exist a great number of model checking tools that differ among other things in their input language and their graphical interface. For the work at hand, the UPPAAL toolbox is used. Its basic features are described in the following.

5.1.1 The UPPAAL Toolbox

UPPAAL is an integrated tool for modelling, validation and verification of real-time systems modelled as networks of timed automata. It is developed at the Department of Information Technology at Uppsala University in Sweden and the Department of Computer Science at Aalborg University in Denmark.

A UPPAAL system is a set of interacting timed automata. The basic definition is given in the following, it is based on the one given in [Behrmann 06].

Definition 5.3: Timed Automaton

A *timed automaton* is a tuple (L, l_0, C, A, E, I), where L is a set of locations, $l_0 \in L$ is the initial location, and C is a set of clocks. $B(C)$ is a set of conjunctions over simple conditions of the form $x \otimes y$ or $x - y \otimes c$, where $x, y \in C$, $c \in \mathbb{N}$, and $\otimes \in \{<, \leq, =, \geq, >\}$. Furthermore, A is a set of actions, co-actions, and the internal τ-action, $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges between locations with an action, a guard, and a set of clocks to be reset, and $I : L \rightarrow B(C)$ assigns invariants to locations.

A UPPAAL *system* is build up of a set of automata, which are parametrised instantiations of *templates*. Additionally, a system can have a set of global declarations of constants, variables and channels. Automata consist of a number of *locations* (one of them must be marked as *initial location*), which are connected via *edges*. *Locations* can have the special property *committed* or *urgent*. The latter implies that time does not pass when the system is in such a location. *Committed locations* are even more restrictive. If one automaton is in a committed location, this state must be left with the next transition.

Edges can be labelled with *guards* (side-effect free Boolean expressions to determine whether an edge is enabled), *updates* (assignments), and *synchronisations*. Detailed information about the UPPAAL toolbox can be found in [Behrmann 06] and on the official website[1]. The modelling language of UPPAAL extends the above given definition of timed automata with several, additional features. In the following a list of those features relevant to this thesis is described.

Templates: Templates are defined by a set of parameters of any type. They are instantiated in the process declaration.

[1]UPPAAL website: `http://www.uppaal.org/`

Constants: Constants are of type integer and cannot be modified

Bounded integer variables: Guards, invariants, and assignments may contain expressions ranging over bounded integer variables. They are declared as integers with a minimal and maximal bound (`int[min,max] name`)

Binary synchronisation channels: Channels are used to synchronise different automata. They are declared as `chan c`. Sending edges are labelled with `c!`, receiving edges with `c?`.

Broadcast channels: Automata sending over a broadcast channel must synchronise with any receiver that can synchronise in its current state. Broadcast sending is never blocking.

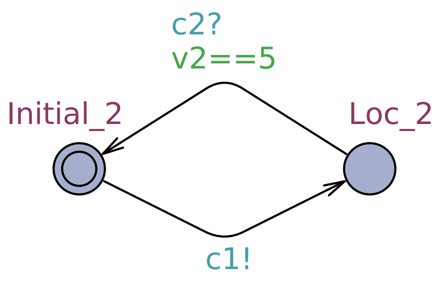
Urgent locations: Time does not pass in an urgent location

Committed locations: More restrictive than urgent. At least one of the outgoing edges of a committed location must be taken.

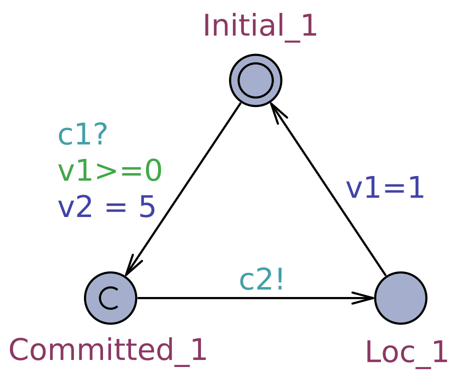
Arrays: Arrays can be defined for clocks, channels, constants and integer variables.

Initialisers: Integer variables and arrays of integer variables are initialised with initialisers.

User functions: User functions allow to define functions either globally or locally to templates.



(a) `ExampleAutomaton1`



(b) `ExampleAutomaton2`

Figure 5.3: Two communicating automata. Initial locations are marked with a double border, committed locations with a C. Channel names are coloured light blue ($c1$, $c2$), guards are green ($v1 \geq 0$, $v2 == 5$) and updates are blue ($v2 = 5$, $v1 = 1$)

In Fig. 5.3 an example of two communicating UPPAAL automata is given to illustrate the main elements used in the work at hand. The locations of all automata, clock values, and the values of variables define the state of a system.

The properties to check on the system's model need to be formulated in a formal language. UPPAAL uses a simplified version of the *Timed Computation Tree Logic* (TCTL) introduced in [Alur 93]. It allows to specify *path formulae* and *state formulae*, but does not support the nesting of path formulae. State formulae are used to describe individual states of the system using side-effect free expressions, e.g. `v2 == 5` to evaluate the value of the variable $v2$, or `ExampleAutomaton.Loc_2` to evaluate if a particular process (`ExampleAutomaton1`) is in a given location (`Loc_2`).

Path formulae can be classified into *reachability*, *safety*, and *liveness* properties.

Reachability Properties: A reachability property checks if there exists a path starting from the initial state that eventually satisfies the state formula φ along this path. This is written as $E\Diamond\varphi$ or $E <> \varphi$ in UPPAAL.

Safety Properties: Safety properties are used to express properties of the form "something bad will never happen". They are written in the form $A\Box\varphi$ or $A[\,]\varphi$ meaning that a state formula φ will always be fulfilled in all reachable states. Another possibility is to ask whether there is a maximal path on which φ is always fullfilled ($E\Box\phi$). A maximal path is an infinite path or a path with no outgoing transition in the last state.

Liveness Properties: To evaluate whether something will eventually happen, liveness properties are used. They can be written as $A\Diamond\phi$, meaning that ϕ is eventually fulfilled. Another possibility is to the check whether φ *leads to* (or *response*) ψ, written as $\varphi \leadsto \psi$ or φ `-->` ψ in UPPAAL.

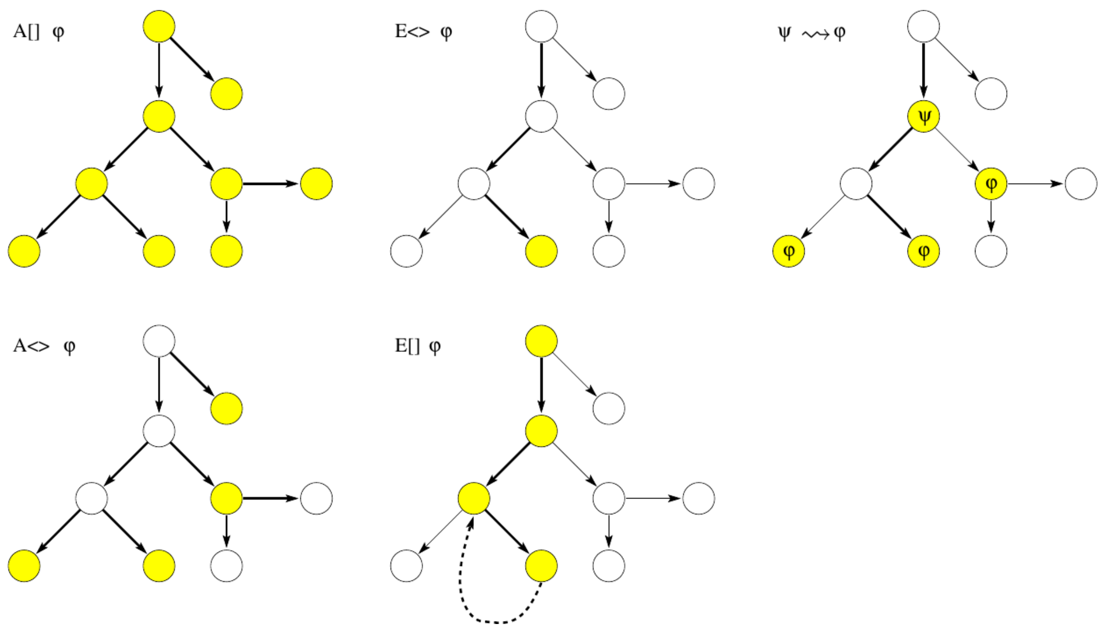


Figure 5.4: The path formulae available in UPPAAL. The yellow marked locations are those in which the formula φ holds. Bold edges depict the paths the formulae evaluate on. [Behrmann 06]

The available path formulae are depicted in Fig. 5.4.

The UPPAAL toolbox features a graphical user interface with three views, the *editor*, the *simulator*, and the *verifier* view. In the *editor* it is possible to define automata and declare variables and functions. The *simulator* offers the possibility to simulate the resulting system, see the values of variables and states of automata. The user has the choice to select the transitions to be taken or use a random simulation. The simulator view also enables the read-in of files, containing traces given by the verifier. The *verifier* serves the purpose to type in queries, verify them on the model and show the results. The tool is extended about two new views, named *concrete simulator* (added in version 4.1.15 in 2103) which offers the possibility to simulate a system with concrete values of clocks and *yggdrasil* (added in version 4.1.19 in 2015) for test case generation. Both views are not relevant to the work at hand and are therefore not further explained. Furthermore, UPPAAL offers a stand-alone command line verifier, *verifyta*, which is more suitable to use with larger applications. The verification can be done on a remote UNIX machine. The model files and query files produced by the UPPAAL toolbox can be used with the command line tool and the resulting trace can be read back.

5.2 Formal Modelling of iB2C Networks

In the approach used in this thesis, the formal modelling of behaviour-based systems is broken down to the modelling of single behavioural components and the connections between those models. This allows to automate the modelling process for different (parts of) robot systems. As the interaction of behavioural components is the central point in the creation of complex robot behaviour, the focus of the verification described here, lies in the interconnections as well. The transfer function $\vec{F}$ expressing the inner functionality of a behavioural component is not integrated in the model checking process.

Single components are described via their interface of incoming, outgoing and internal behavioural signals as described in Chap. 2.2. The influence of the incoming data, stimulation s and inhibition i, on the outputs activity a and target rating r and the internal activation value ι is modelled via a combination of five interconnected automata, one for each behavioural signal.

The automata are created according to the definition of the automata used in the UPPAAL toolbox (see Sec. 5.1.1). As UPPAAL allows only discrete values, the range [0, 1] of the signals is abstracted to the set $\{0, 1\}$. These and several other design decisions are extensively discussed in [Armbrust 14a].

The following sections give an overview of the modelling of standard behaviours and their interconnections and a detailed description of the extensions that were made in the context of this work related to the integration of hardware like sensors into the formal model.

5.2.1 Standard Behaviour Modelling

A standard iB2C behaviour is modelled via a set of five automata named: `StimulationInterface`, `InhibitionInterface`, `ActivationCalculation`, `ActivityCalculation` and `TargetRatingCalculation` (see also [Armbrust 12, Armbrust 14a]).

The `StimulationInterface` as shown in Fig. 5.5 models the stimulation of a behavioural component. It contains two main states representing a *stimulated* behaviour or an *unstimulated* behaviour. The distinction between only those two locations is due to the fact that the decision to use discrete values is made. This means that a behavioural component can only be stimulated or not, but it cannot be partially stimulated.

When the automaton receives a stimulation changed signal (`s_changed?`) it sets the global stimulation value variable (`s_value`) to 1 and sends another signal (`s_changed_internal!`)

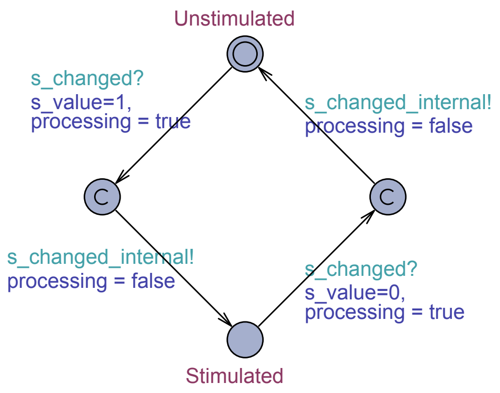


Figure 5.5: `StimulationInterface` of a behaviour-based component

to signalise its current state to the activation automaton. The same procedure is done when the automaton receives another stimulation changed signal, setting the stimulation value variable back to 0. As UPPAAL allows only one synchronisation per transition, one committed location is added on both transitions, respectively. The use of committed locations is a common workaround and adds no further value to the model. As mentioned in Sec. 5.1.1 transitions from committed locations have precedence over transitions from normal locations, implying the automaton will transition to the next location before any other "normal" transition is taken. To exclude those intermediate locations from the results of queries a `processing` flag is introduced, which is set to true whenever a committed location is entered and back to false when the committed location is left. This technique is used in most of the other automata, too.

The basic functionality of the `InhibitionInterface` automaton is similar to the one of the `StimulationInterface`. It receives an incoming inhibition signal, updates the variable indicating the inhibition status of the behavioural component and sends out a signal to the `ActivationCalculation`. Besides these basic similarities, the `InhibitionInterface` must be treated specially, because of the possibility that a component can be inhibited by more than one other. Therefore, the automaton for the inhibition interface is generated in dependence of the number of connected behavioural components. Figure 5.6 shows the automaton for two connected components. Because of the abstraction to the discrete values $\{0, 1\}$ the automaton can be in location `Uninhibited` where the global variable `i_value` is 0 or in a location indicating an inhibited behavioural component (locations `Inhibited_by_First`, `Inhibited_by_Second`, and `Inhibited_by_Both` for the example automaton). In the latter case, the set of locations indicating an inhibited component is needed to remember the correct number of inhibiting components. In all of these locations, the variable `i_value` is 1. It is set only once when the first inhibiting signal is received. At the same time a signal is send to the corresponding activation automaton. When the last connected behavioural component changes its inhibition value back to *no inhibition*, the variable is set back to 0, the signal is sent again and the automaton comes back to the start location.

The signals from the stimulation automaton and inhibition automaton are received by the `ActivationCalculation` automaton of the behavioural component, shown in Fig. 5.7. In dependence of the global variables indicating the stimulation value and the inhibition value the automaton is set back to the start location `Inactivated` or transfers to location

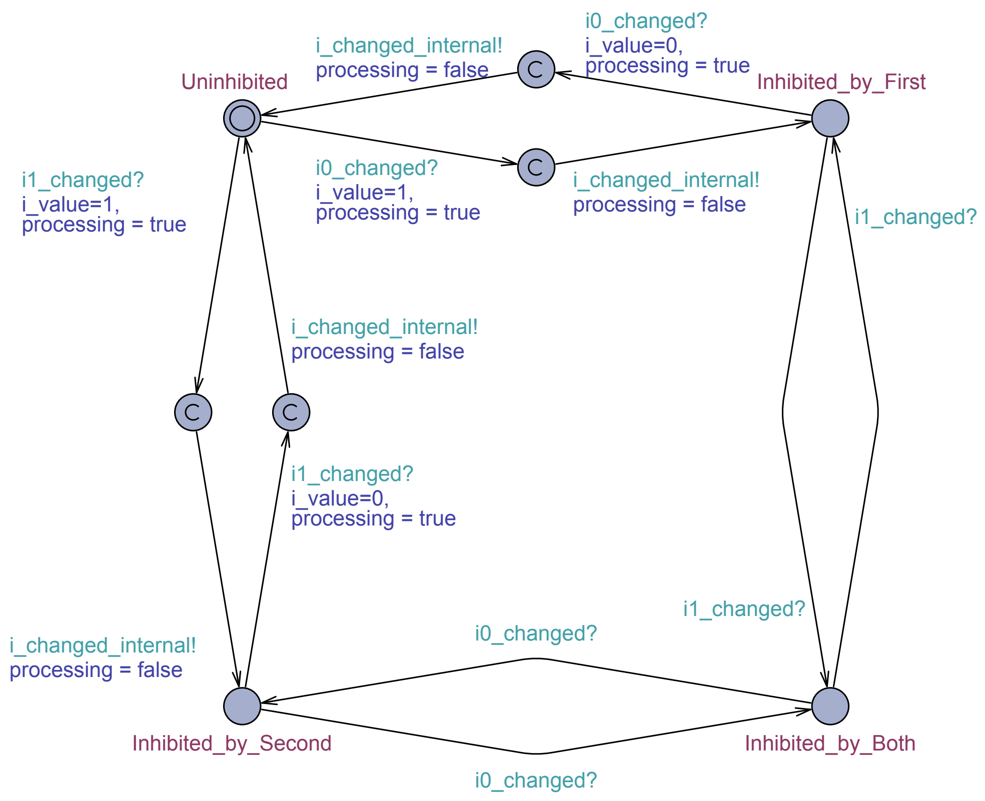


Figure 5.6: `InhibitionInterface` of a behaviour-based component that is inhibited by two others

`Activated`. In the latter case, where the stimulation value is 1 and the inhibition value is 0, the change of activation is signalled via `iota_changed` to its corresponding activity automaton. The automaton transfers back to the start location `Inactivated` if either the stimulation or the inhibition change their respective values again.

While the before mentioned automata trigger automata corresponding to the same behaviour, the signal coming from the `ActivityCalculation` automaton is sent to the stimulation or inhibition automata belonging to other behaviours. Figure 5.8 shows the `ActivityCalculation` automaton, which has three main states: `Inactive`, in the case the behavioural component is not activated, `Inactive_but_Activated` for the case the automaton received an activation changed signal from the activation automaton and `Active`, for the case the component is both, activated and active. The latter is reached non-deterministically as the activity of a behavioural component is calculated internally dependent on different input conditions e.g. sensor data or other control values, which are not modelled for basic behavioural components. This abstraction is sufficient in most cases, as the focus lies on the verification of the network and the flow of the behavioural data. In this context it is only interesting to verify the possible impact of an active component inside the network. Although, for some cases it is useful to refine this automaton. An example is described in Sec. 5.2.2.

The last behavioural variable is modelled by a very simple automaton, called `Target-RatingCalculation`, containing only two states indicating satisfaction or dissatisfaction of a behaviour (see Fig. 5.9). The states are chosen non-deterministically, where a `r_changed`

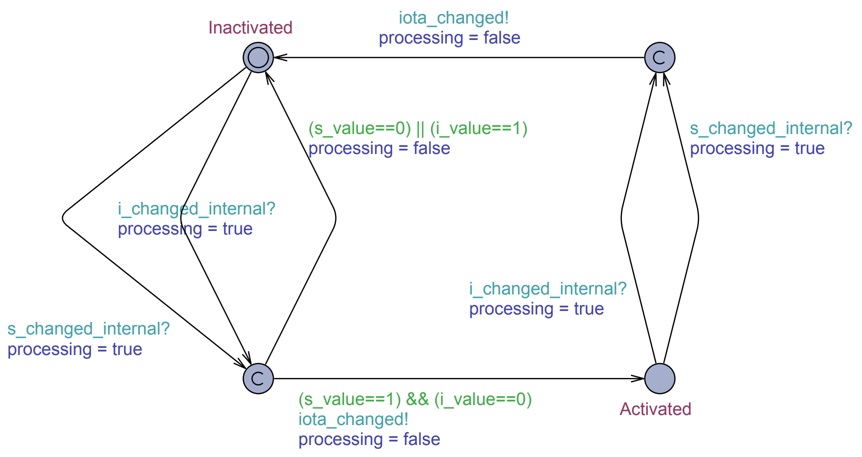


Figure 5.7: Template for the `ActivationCalculation` of a behaviour-based component

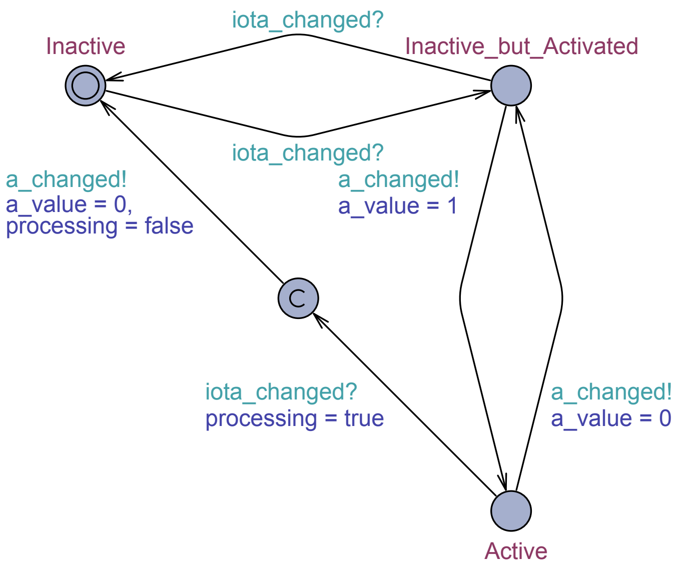


Figure 5.8: Template for the `ActivityCalculation` of a behaviour-based component

signal is sent to the connected automata of other components and a global target rating value variable is set to either 1 or 0 indicating the current status. This abstraction is based on the same background as for the activity automaton. The target rating value is calculated internally in the behavioural component and can therefore not be modelled.

Formal models of iB2C behaviour networks are generated automatically, where every behavioural component is modelled as a set of the five above described automata. The interconnection between the models of the components is achieved by renaming the channels. Figure 5.10 shows an example. Here the previously used dummy names, `a_changed!` in Fig. 5.8 and `s_changed?` in Fig. 5.5 are replaced by `behaviour-name_a_changed` which at the same time improves the readability as the user directly knows which behavioural component is connected to the stimulating input.

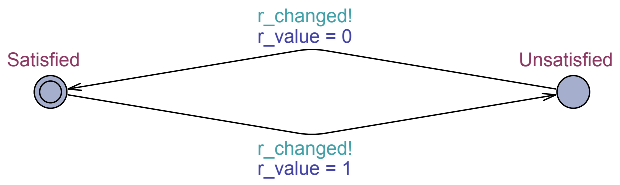


Figure 5.9: Template for the `TargetRatingCalculation` of a behaviour-based component

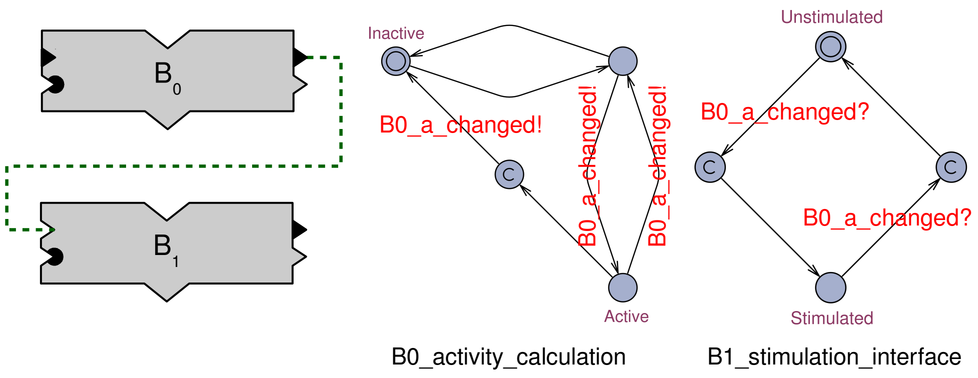


Figure 5.10: Example for the modelling of a connection between two behavioural components, where component B_0 stimulates B_1 with it activity. The names of the synchronisation are adapted to the component names.

5.2.2 Special Behaviour Modelling

The iB2C networks contain not only standard behavioural components but also several special ones that are e.g. fusion behaviours and behavioural groups. See Chap. 2.2 for details about them. The modelling of a fusion behaviour is a bit tricky as an arbitrary number of behaviours can be connected to it and the calculation of its activity depends on all inputs and the implemented fusion function. The details of the modelling of fusion behaviours can be found in [Armbrust 15]. The modelling of behavioural groups is explained in the following paragraph.

For the modelling of groups exist two possibilities. One is to model the group as one single element in the network and hide the information inside. As a behavioural group has the same interface as a standard behavioural component, the model of a group is the same as the model of a single component. The second is to include the behavioural components inside a group into the surrounding network. In this case the group information is skipped, which means, the network of components and inner groups is modelled as a flat structure. For the work at hand, both possibilities are used according to the given task. The representation as a single behavioural component reduces the model size, which can be very important for large networks. On the other hand, the inner structure of the group cannot be verified in this case, which could be necessary for some queries.

Another special component, important for the work at hand are so-called *safety behaviours* (SB). These components process sensor data and calculate their activity and target rating in dependence of outer incidents. As an example, a behavioural component in the safety layer of a robots control system responsible for slowing down the robot will get active in cases where a front sensor detects an obstacle. It will calculate the desired drive velocity

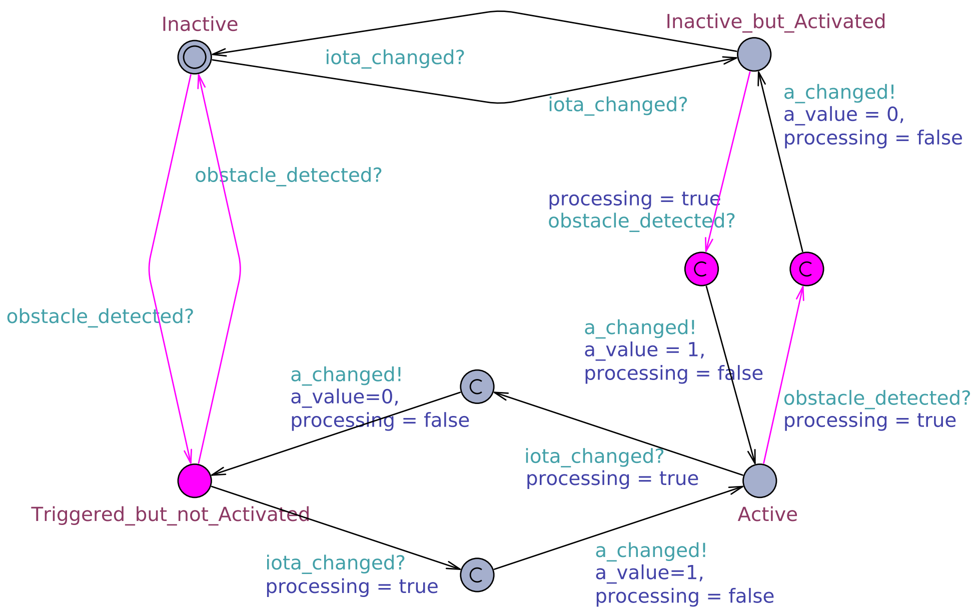


Figure 5.11: Template for the `SBActivityCalculation` of a safety behaviour. The pink coloured locations and transitions mark the differences to the `ActivityCalculation` template of a standard behavioural component.

of the robot and inhibit other components that send other driving commands, for example a “Drive to Goal” behaviour. The target rating will be close to 1 as long as the obstacle is close to the robot and will be set back to 0 when the obstacle is passed.

In this example, it becomes clear, that the modelling of the activity and the target rating need to be adapted to the special properties of a safety behaviour. That means, the automaton must react to outer signals. Figure 5.11 shows the extended `SBActivity-Calculation` template. The edge from location `Inactive_but_Activated` to location `Active` is refined, so that the `a_changed` signal is not send non-deterministically anymore but only after receiving an `obstacle_detected` signal. Additionally, a new location is added which represents the state that an obstacle is detected but the safety behaviour is not activated. From that location, the automaton traverses to location `Active` when the activation of the behaviour changes. The newly added location `Triggered_but_not_-Activated` should never be reached as safety behaviours should always be activated to ensure a safe robot behaviour.

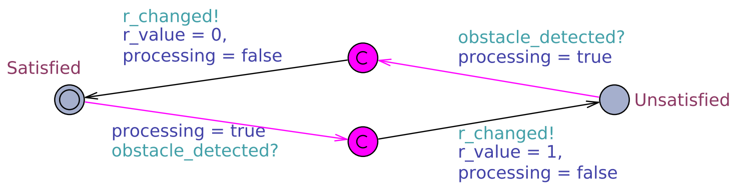


Figure 5.12: Template for the `SBTargetRatingCalculation` of a safety behaviour. The pink coloured locations and transitions mark the differences to the `TargetRatingCalculation` template of a standard behavioural component.

The modelling of a safety behaviours target rating is done in a similar way (see Fig. 5.12).

The previously non-deterministically taken transitions are now triggered by an `obstacle_detected` signal, representing the fact, that a safety behaviour is unsatisfied as long as an obstacle exist and satisfied in the other case.

The generation of the `obstacle_detected` signal is achieved by including the whole sensor chain the robot is working with, into the formal model. This will be explained in detail in the following section.

5.3 Including Environmental Influences in the Model Checking Process

Model checking is most often applied to hardware models or software models. A combination is usually neglected, which is, in the case of embedded systems, done by assuming perfect sensing and acting when verifying the software system. As especially robotic systems permanently interact with their (changing) environment, the need to check the overall system, also in the presence of hardware failures, is obvious.

Several papers address this topic using different modelling approaches as basis. For example, [Johnson 11, Johnson 13, Johnson 15, Lahijanian 10] use robot controllers, that are automatically synthesised out of a set of specified formal properties. These synthesised systems are guaranteed to meet their specified properties under the assumption of perfect sensors and actuators. Johnson et al. relax this assumption in order to analyse the probability that the synthesised robot controller satisfies a set of high-level specifications in the presence of sensor and actuator errors. They define models for the robot controller, the environment behaviour and the sensor error, combine them and analyse the resulting model with a parametric and a probabilistic model checker. [Lahijanian 10] consider disturbances in sensors and actuators in their analysis to find a synthesis algorithm that satisfies the specification with the highest probability.

In [Webster 14], an approach similar to the one used in the work at hand serves as basis for the verification process. A specified system is modelled in the Process Meta-Language PROMELA, which is the input language for model checker SPIN [Holzmann 97]. The authors additionally model the robots environment to be able to verify specifications with respect to the robots situation. In contrast to Johnsons work and the approach presented in the following, the authors do not consider the actual hardware components and their possible errors.

The approach presented in this thesis is published in [Kiekbusch 14] and [Kiekbusch 15]. The focus lies on the interface between the hardware (sensors) and the software (safety behaviours, that react on sensor data). This enables to verify the correct connection of sensor data into the system. Additionally, the possibility to include an arbitrary number of failure sources in the sensor chain is modelled. To simplify the definition of queries, a so-called *scenario automaton* is introduced which allows to specify the environment conditions for a special scenario in an easy manner. In Sec. 5.4 tool support for the generation of corresponding queries is introduced.

The following subsections give a detailed description of the integration of sensors into the formal model of the behaviour-based software system.

5.3.1 Sensor Integration

For the integration of sensor information into the formal model of the control system, detailed knowledge about the sensor processing chain is needed. In the work at hand, the *control-driven sensor processing* chain as described in Sec. 2.3.2 is used as basis.

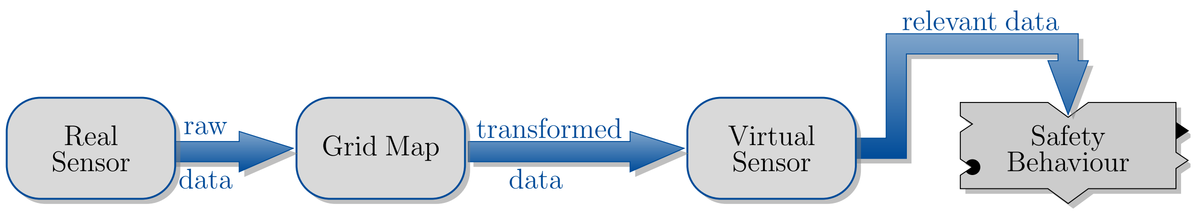


Figure 5.13: Flow of sensor data from the real sensor to the controlling behavioural component inside the network

The overall goal of sensor processing is to provide the data needed for different navigation tasks in an appropriate manner. For this, the data from the diverse sensors have to be composed and put into an abstract view for further access. Figure 5.13 depicts the flow of data. Starting from the raw data sensed by the real sensor and saved in a grid map, selected and processed data are combined in a so-called *virtual sensor* which provides the data to a safety behaviour.

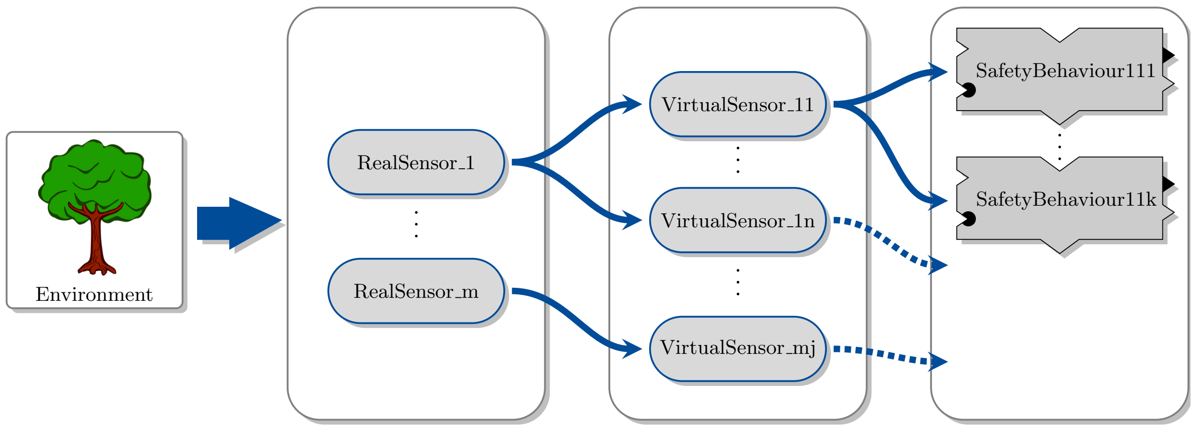


Figure 5.14: Abstract view of the sensor processing chain depicting the relation between environment, real sensors, virtual sensors, and safety behaviours

Since data from a real sensor can be processed in several different ways according to special properties or processing algorithms, there may exist multiple virtual sensors per real sensor. In turn, virtual sensors can provide their data to several safety behaviours. This leads to the abstract view given in Fig. 5.14. In this representation, grid maps are neglected as they belong to the real sensor from a software view. Instead, the environment is added as trigger for sensor events.

For the formal verification real sensors and virtual sensors are modelled as finite state automata to get integrated into the formal model of the control system. Additional *failure* *automata* are included to model failure states. The modelling procedure for the two links in the sensor processing chain will be explained in detail in the following sections. Furthermore, a so-called *ScenarioAutomaton* is introduced, which is used to define the scenario in which a system property will be checked.

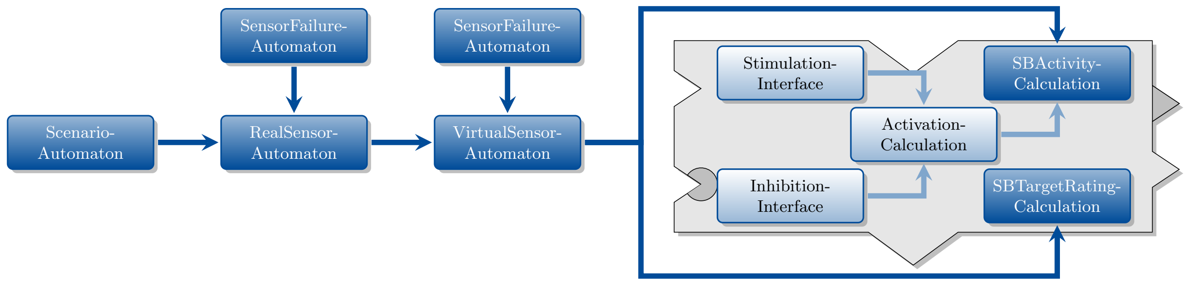


Figure 5.15: The dark blue boxes mark the components that must be formally modelled as automata in order to integrate sensor information into the system model. For the behavioural component, the models for the behavioural signals *activity* and *target rating* must be adapted.

Figure 5.15 depicts the different automata that are needed to model the sensor chain. For the modelling of a safety behaviour, the automata modelling the activity and target rating have to be adapted as described in the previous section. As mentioned before, the modelling of grid maps is neglected in this approach. This decision was made, as they would be represented only as another simple failure automaton indicating whether the data is recorded correctly or not. They add no further connection to the model that need to be verified for the whole system.

5.3.2 The Virtual Sensor Automaton

In the concept of sensor processing used as basis for the modelling process described in this work (see 5.3.1) virtual sensors serve as abstraction layer between the map including all data coming from the real sensors and the software layer operating on the sensor information. A virtual sensor covers an area around the robot with the data coming from this special part of the grid map. As there can be several processing algorithms for the data, there may exist more than one virtual sensor for a certain area (e.g. obstacle detection algorithms, water detection algorithm, unfiltered data).

5.3.2.1 Preliminary Modelling Ideas

In the modelling process every virtual sensor is mapped to an instance of a `Virtual-SensorTemplate`. On the one hand, this assures that the data flows correctly (from real sensors to safety behaviours) as every safety behaviour acts on the data of one virtual sensor. On the other hand possible programming errors can be modelled individually as every virtual sensor stands for a special algorithm, which could possibly be incorrect.

There exist several possibilities to model failures and failure states of a virtual sensor (implying the modelling of failures in the implemented algorithm). The concept proposed in this work follows the idea, that the failing of a virtual sensor means that its output is the opposite of the correct output. A correctly working sensor would detect the occurrence of obstacles in the area observed by this sensor correctly. A failing sensor would output "no obstacle" in cases there is one and "obstacle detected" in cases where no obstacle exist, accordingly.

Other possibilities to model failures in the implementation of a virtual sensor could be the output of the same value independent of the actual state, or no output at all. The advantage of the used variant is that a special failure state can be queried using one state of the automaton. For the others, a combination of the failure trigger and the actual state must be used for verification tasks.

5.3.2.2 The VirtualSensorTemplate

The `VirtualSensorTemplate` (Fig. 5.16) consists of one location modelling the correctly working of the sensor (`Correct`) and two locations modelling a failure state (false negative: `Failure_FN` and false positive: `Failure_FP`). Initially, the automaton is in location `Correct` and the variable `obstacle` which holds the information whether an obstacle is currently detected or not, is false. If the automaton receives the signal `exist_obstacle` from its corresponding `RealSensorAutomaton` (see Sec. 5.3.3.1), `obstacle` is negated and `detect_obstacle` is send to the corresponding safety behaviour.

For triggering failure events, an additional *failure automaton* is used[2], see Sec. 5.3.4 and Fig. 5.18. It sends out a `failure` signal accidentally. Dependent on the current state of the `obstacle` variable, the automaton changes to one of the failure locations sending out `detect_obstacle`. If the sensor recovers from the failure sending out `failure` again, the automaton comes back to location `Correct` with sending the respective signal to the safety behaviour.

5.3.3 The Real Sensor Automaton

The real sensor is the first link in the chain of processing sensor data (see Fig. 5.13). Its task is to collect environmental data and process them with some rudimentary preprocessing techniques. As the intermediate step of creating grid maps is not modelled, the model of the real sensor includes that step. The model reacts to changes in the environment and sends out the relevant information to the virtual sensors. A formal model is created for every real sensor. This allows to introduce failures in every single sensor.

5.3.3.1 The RealSensorTemplate

The `RealSensorTemplate` is proposed to model arbitrary sensors. It contains three main locations representing a correctly working sensor (`Correct`) and a defective sensor (`False_Positive` and `False_Negative`). Starting from the `Correct` location, the automaton can receive a signal indicating a change in the environment (`covered_area_changed`). It then sends out `area_changed` signals for all the areas where an environment change occurred. This signal is received by the corresponding `VirtualSensorAutomata`. The number of areas is defined in dependence on the characteristics of the robot. In the case of RAVON the whole area around the robot is divided into ten sectors (see Fig. 2.9).

[2]This version of the modelling of virtual sensors is described in [Kiekbusch 15], whereas the previous version described in [Kiekbusch 14] used an automaton with an integrated failure automaton.

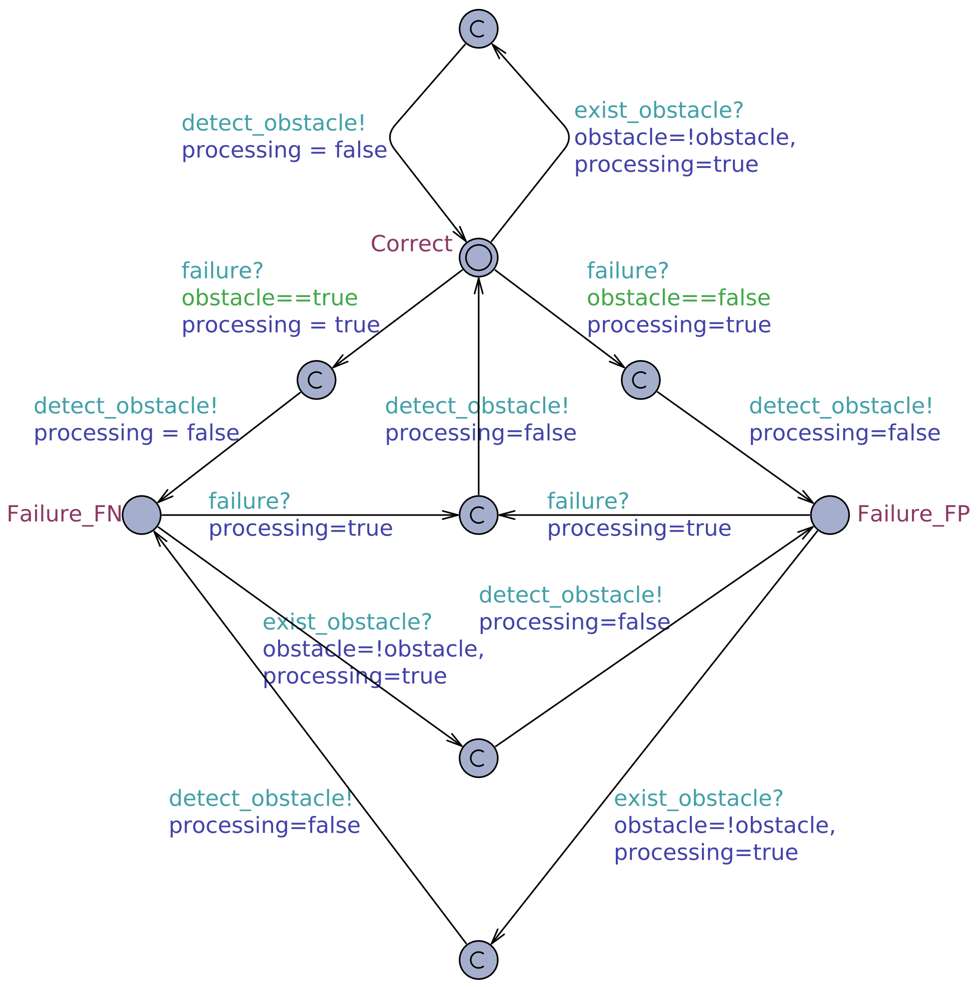


Figure 5.16: Template for the `VirtualSensorAutomaton`

The modelling of the single sectors is done via a boolean array where 1 means that an obstacle exists in the corresponding sector. The array is updated whenever a change in the environment occurs (see Sec. 5.3.5). As example, an obstacle in the front of ravon would lead to the assignment of the array $[1, 0, 0, 0, 0, 0, 1, 1, 1, 0]$, as the front area of the robot is divided into 4 sectors.

The `RealSensorAutomata` are notified about changes via the `covered_area_changed` signal. To determine which areas are affected, a loop using two committed locations is included. Here, the internal function `HasAreaChanged()` compares the values of the current array with the values of the array given by the scenario automaton (Sec. 5.3.5). In the case a deviation is found, a signal is sent to the corresponding virtual sensor.

A sensor failure is modelled via an external automaton in the same way as for the virtual sensor. A difference is, that for the real sensor two possible failures are distinguished, *false positive* and *false negative* failures, which are modelled by two different automata. This gives the analyst more freedom to model different types of sensor failure. A failure state can only be reached from the `Correct` location when a respective signal is received from one of the failure automata. The path to the failure locations and back includes the same steps as for a normal environment change signal, with the addition, that the actual

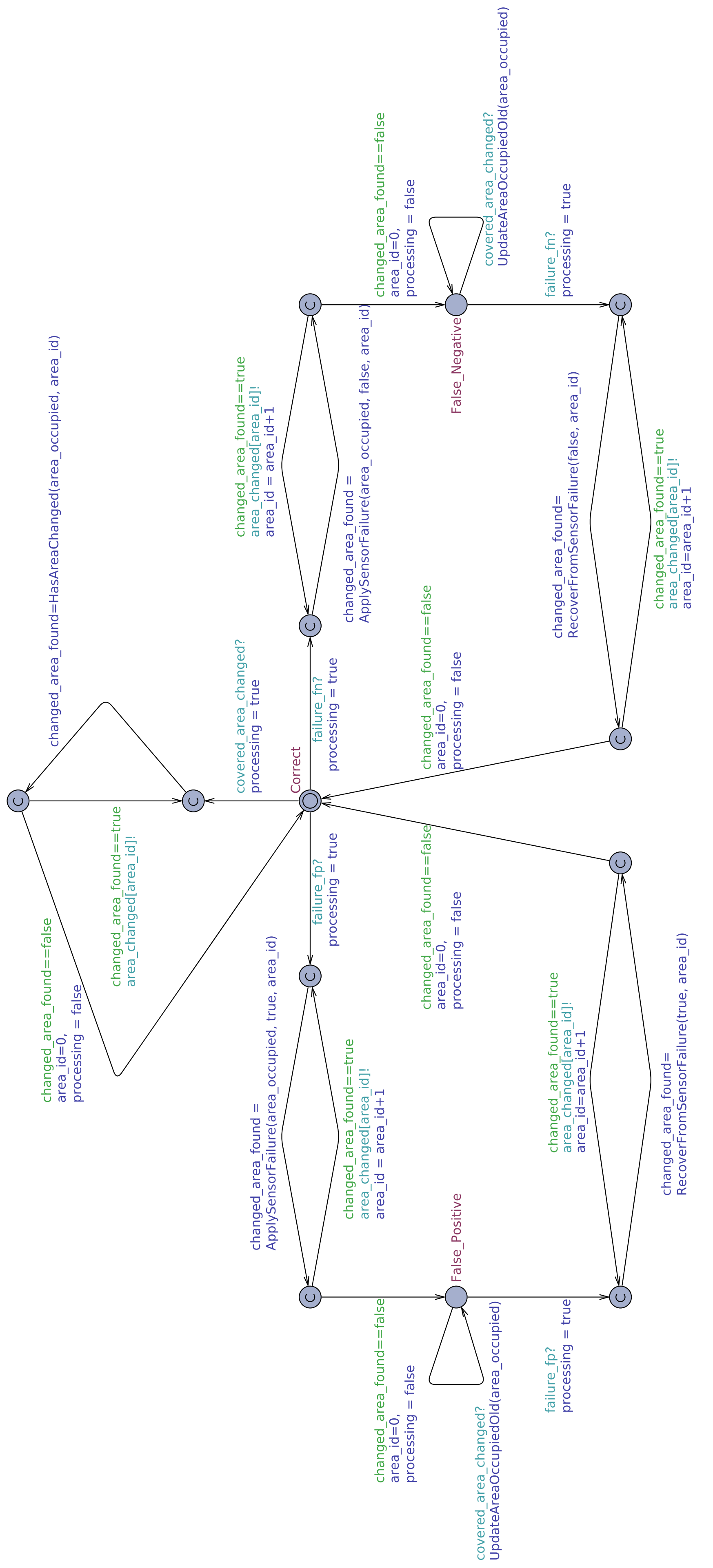


Figure 5.17: Template for the `RealSensorAutomaton`

environment state is saved to facilitate the way back to the correct state.

5.3.4 The Sensor Failure Automaton

Sensor failures are modelled as simple automata consisting of two locations `Correct` and `SensorFailure`. The transitions between them are taken non-deterministically to simulate a sudden occurrence of a failure. The `SensorFailureAutomaton` is pictured in Fig. 5.18. The recovery function of the sensor (transition from location `SensorFailure` back to location `Correct`) is a theoretical possibility to model a sensor with the ability to recover from a failure. Dependent on the real hardware and software, this edge can be omitted.

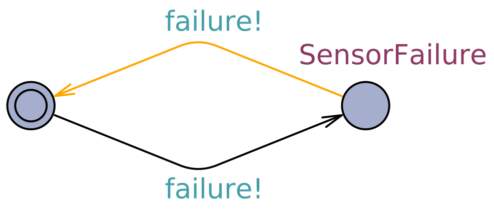


Figure 5.18: Template for the `SensorFailureAutomaton`. The recovery edge (orange) is optional.

The advantage of having an externally modelled failure instead of modelling failures directly in the automaton of the real or virtual sensor is the option to easily add several failure causes. This enables the definition of more exact queries. This concept seems to be necessary for real sensors, but not essentially for virtual sensors as software might fail because of wrong code, without the need of defining different causes. Therefore, the first version of the virtual sensor model as introduced in [Kiekbusch 14] used integrated failure locations. The second version as described in Sec. 5.3.2.2 is adapted to unify the modelling of the two sensor aspects. This simplifies the understanding of the automata.

5.3.5 The Scenario Automaton

Requirements of a robot control system often correspond to a certain context. For example, "the robot must slow down in case of *obstacles to its front*" or "the robot should drive in full speed mode if *the respective control signal is given* and *no obstacles stand in its way"*.

To verify such requirements, environmental properties must be included into the model and the formal query. This can be done using the previously described real sensor automaton. The values of the areas must be set manually. A query formulating the part of the property that a front obstacle exist could be written as

```
E<> ...  && RealSensor_1.Correct && area_occupied[0]==1 && area_occupied[1]==0
&& area_occupied[2]==0 && area_occupied[3]==0 && area_occupied[4]==0
&& area_occupied[5]==0 && area_occupied[6]==1 && area_occupied[7]==1
&& area_occupied[8]==1 && ...,
```

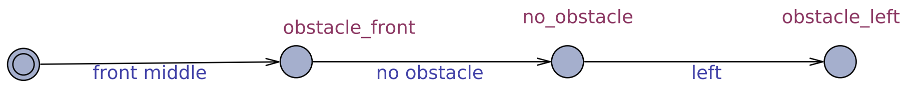
with the boolean array `area_occupied` containing the states of all sector maps around the robot. 1 indicates an obstacle in an area. According to Fig. 2.9 and the scenario *obstacle in the front*, the areas corresponding to the sector maps $0, 6, 7, 8$ are set to 1. With increasing number of sensors, such a query would get long and confusing. The possibility of failures during the query definition process increases.

To avoid these problems and to simplify the generation of queries for unexperienced users a so-called *scenario automaton* is developed. It is a simple automaton describing the environment of the robot in terms of existent obstacles using a set of predefined keywords (e.g. front, left, rear middle, …). Figure 5.19a shows an example of a scenario automaton which includes three different scenes. Edges define the position of the obstacle with the keywords. The names of the locations are user-defined and needed for the query formulation. As can be seen in this example, several environmental states can be defined in one scenario automaton.

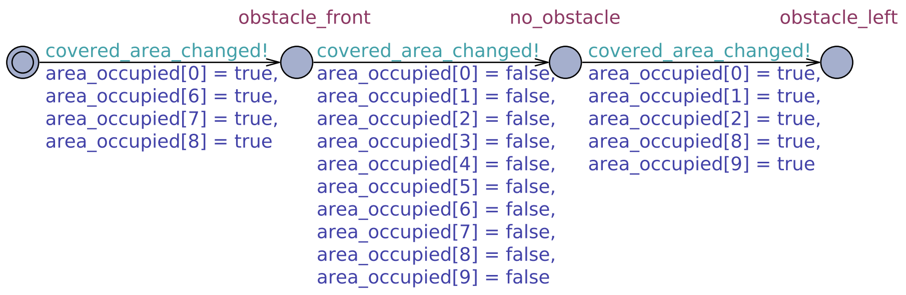
During the automatic modelling process, this user-defined automaton is automatically adapted to the extended formal model of the specific robot control system. Figure 5.19b illustrates the generated automaton on the example of the robot RAVON, where the keywords are replaced by variable assignments and the signal `covered_area_changed`. The broadcast signal `covered_area_changed` is received by all of the real sensor automata. With the help of this scenario automaton, the formulation of the query becomes very easy as only one location in the scenario automaton needs to be specified instead of a location in several real sensor automata. The above described query can now be written as

```
E<> ...  && Scenario.obstacle_front && ...
```

with the help of the scenario automaton.



(a) User defined scenario automaton. Edge labels contain keywords.



(b) Generated scenario automaton for the robot RAVON (see Fig. 2.9 for the assignment of array positions to virtual sensors)

Figure 5.19: Definition of a scenario automaton and adaptation to the respective formal model. The keywords from Fig. 5.19a are transformed to a signal and variable assignments according to the formal system model of a specific robots control system in Fig. 5.19b.

5.4 Tool-Assisted Query Generation

As mentioned before, the formulation of formal properties is a tedious task and knowledge about the formal language and the procedure to translate a property is needed. In Sec. 2.4 as well as in [Armbrust 14a] and [Ropertz 12], an extension of the FINSTRUCT tool is described, which enables the user to graphically define queries. The underlying concept is to select behavioural components in the *VerificationView* of FINSTRUCT, transfer them to the *Query Development User Interface* (QDUI) and connect them via edges representing their relation. The presented concept allows to define rather complex properties, which cannot be formulated in one query. They are automatically transferred to observer automata, which can then be checked to verify the property on the system model.

The extension about the environmental influences described in the previous sections can not be represented in that standard *VerificationView* as the components modelling the sensor chain are not represented there. That means, properties including special conditions of sensors need to be written by hand. This can get a really tedious and error-prone task. The procedure of checking safety related properties shall be presented on the example introduced in Sec. 5.2.2, where a robot faces an obstacle to its front and shall slow down. The example is restricted to the part of the safety layer of RAVON given in Fig. 2.10 for readability reasons. The state of this sub-network depends on the output of the fusion behaviour *(F) Fast Front Slow Down*. The property to be checked in this example can be written as follows:

R1) *(F) Fast Front Slow Down* (*(F) Fast Front SD*) shall be active if an obstacle is present in front of the robot.

The corresponding query

```
E<> F_Fast_Front_Slow_Down_a_value==0 && scenario_now.obstacle_front && !Processing()
```
[3]

uses the negation of the property "*(F) Fast Front SD* shall be active".

The result of the model checker states that it is possible that the behavioural component will not get active even if there are obstacles. The back trace reveals that failures in all sensor chain links can lead to that result. As consequence, failures must be excluded step-by-step to find the conditions under which the property is fulfilled. For example,

```
E<> F_Fast_Front_Slow_Down_a_value==0 && scenario_now.obstacle_front
&& laser_2d_real_sensor.Correct && front_cartesian_laser_2d_virtual_sensor.Correct
```

can be used to check whether the property is fulfilled in the case of one correct working real sensor and a corresponding correct virtual sensor.

To simplify the creation of such requests, an extension to the *VerificationView* is developed. The new *SafetyQueryView* (SQV) shall fulfil the following requirements:

SQVR 1) Components of the sensor chain and scenario information shall be integrated into the *VerificationView* of RRLABfinstruct

SQVR 2) Graphical support during query generation

SQVR 3) Integrated possibility to check the query on the formal model from within FINSTRUCT

SQVR 4) Display the trace of the witness or counterexample in the *TraceView* of FINSTRUCT using the integrated representation from the first requirement

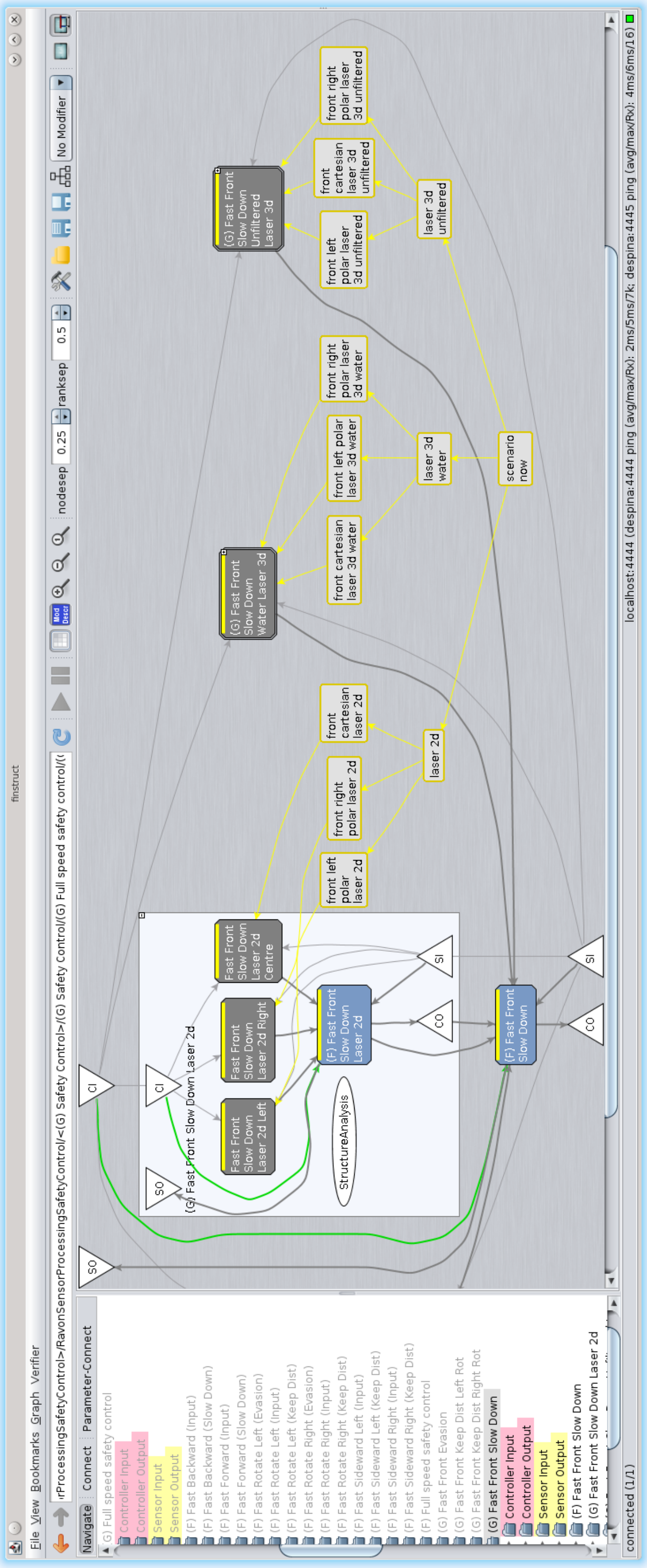
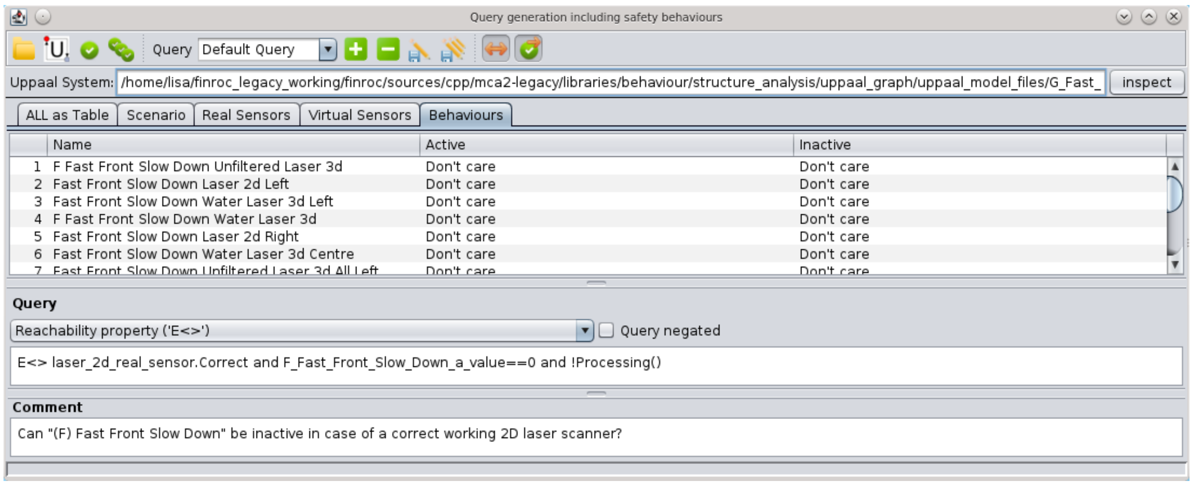

File View Bookmarks Graph Verifier
nodesep 0.25 ranksep 0.5
No Modifier
Navigate Connect Parameter-Connect
(G) Full speed safety control
Controller Input
Controller Output
Sensor Input
Sensor Output
(G) Fast Front Slow Down Unfiltered Laser 3d
(G) Fast Front Slow Down Water Laser 3d
(G) Fast Front Slow Down Laser 2d
(F) Fast Front Slow Down Laser 2d
(F) Fast Front Slow Down
Fast Front Slow Down Laser 2d Left
Fast Front Slow Down Laser 2d Right
Fast Front Slow Down Laser 2d Centre
StructureAnalysis
front left polar laser 3d unfiltered
front cartesian laser 3d unfiltered
front right polar laser 3d unfiltered
laser 3d unfiltered
front cartesian laser 3d water
front left polar laser 3d water
front right polar laser 3d water
laser 3d water
scenario now
front left polar laser 2d
front right polar laser 2d
front cartesian laser 2d
laser 2d
CI
CO
SI
SO
connected (1/1)
localhost:4444 (despina:4444 ping (avg/max/Rx): 2ms/5ms/7k; despina:4445 ping (avg/max/Rx): 4ms/6ms/16)

Figure 5.20: Main window of FINSTRUCT's *Verification View* showing a small part of RAVON's safety layer with integrated sensor chain (yellow boxes) (cp. Figure 2.10 showing the same network without the integration)

Figure 5.20 shows the integration of the sensor chain and the scenario into the *VerificationView* (requirement SQVR 1). The information of the connections between sensors and safety behaviours is extracted from the formal model that underlies the following model checking procedure. The model serves as basis for all parts of the *SafetyQueryView*.



Uppaal System: /home/lisa/finroc_legacy_working/finroc/sources/cpp/mca2-legacy/libraries/behaviour/structure_analysis/uppaal_graph/uppaal_model_files/G_Fast_

	Name	Active	Inactive
1	F Fast Front Slow Down Unfiltered Laser 3d	Don't care	Don't care
2	Fast Front Slow Down Laser 2d Left	Don't care	Don't care
3	Fast Front Slow Down Water Laser 3d Left	Don't care	Don't care
4	F Fast Front Slow Down Water Laser 3d	Don't care	Don't care
5	Fast Front Slow Down Laser 2d Right	Don't care	Don't care
6	Fast Front Slow Down Water Laser 3d Centre	Don't care	Don't care
7	Fast Front Slow Down Unfiltered Laser 3d All Left	Don't care	Don't care

Query

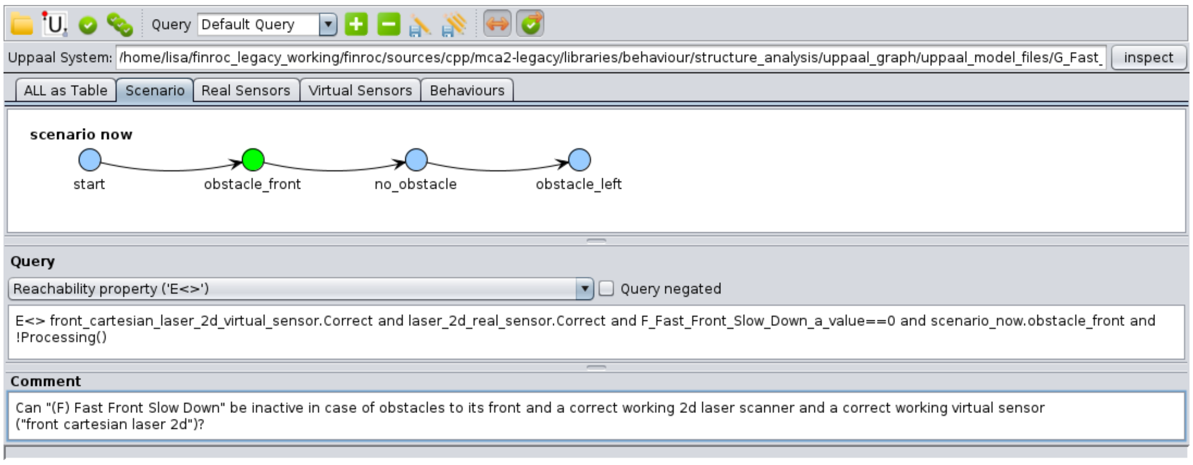
Reachability property ('E<>')

```
E<> laser_2d_real_sensor.Correct and F_Fast_Front_Slow_Down_a_value==0 and !Processing()
```

Comment

Can "(F) Fast Front Slow Down" be inactive in case of a correct working 2D laser scanner?

(a) The tab of the SQCW showing the behavioural components included in the model and some of their properties



Uppaal System: /home/lisa/finroc_legacy_working/finroc/sources/cpp/mca2-legacy/libraries/behaviour/structure_analysis/uppaal_graph/uppaal_model_files/G_Fast_

Query

Reachability property ('E<>')

```
E<> front_cartesian_laser_2d_virtual_sensor.Correct and laser_2d_real_sensor.Correct and F_Fast_Front_Slow_Down_a_value==0 and scenario_now.obstacle_front and
!Processing()
```

Comment

Can "(F) Fast Front Slow Down" be inactive in case of obstacles to its front and a correct working 2d laser scanner and a correct working virtual sensor ("front cartesian laser 2d")?

(b) The tab of the SQCW showing the scenario automaton

Figure 5.21: Two views of the *Safety Query Construction Window* (SQCW)

For the graphically supported creation of queries (requirement SQVR 2) a new window, called *Safety Query Construction Window* (see Fig. 5.21), is created. It displays the different components of the formal model in different tabs (*All as table*,*Scenario*,*Real Sensor*, *Virtual Sensors*, *Behaviours*). These tabs contain tables with one line for every behaviour/sensor automaton and columns with the name and properties of the component. As example, for behaviours the properties *Active* and *Inactive* are displayed, as those are the most frequently asked properties of a safety behaviour. For the definition of other properties, the table can be easily extended. The cells corresponding to a special

[3]As mentioned in Sec 5.2.1, the `Processing` flag needed because of UPPAAL's special restrictions. For the sake of clarity it is left out in the following queries.

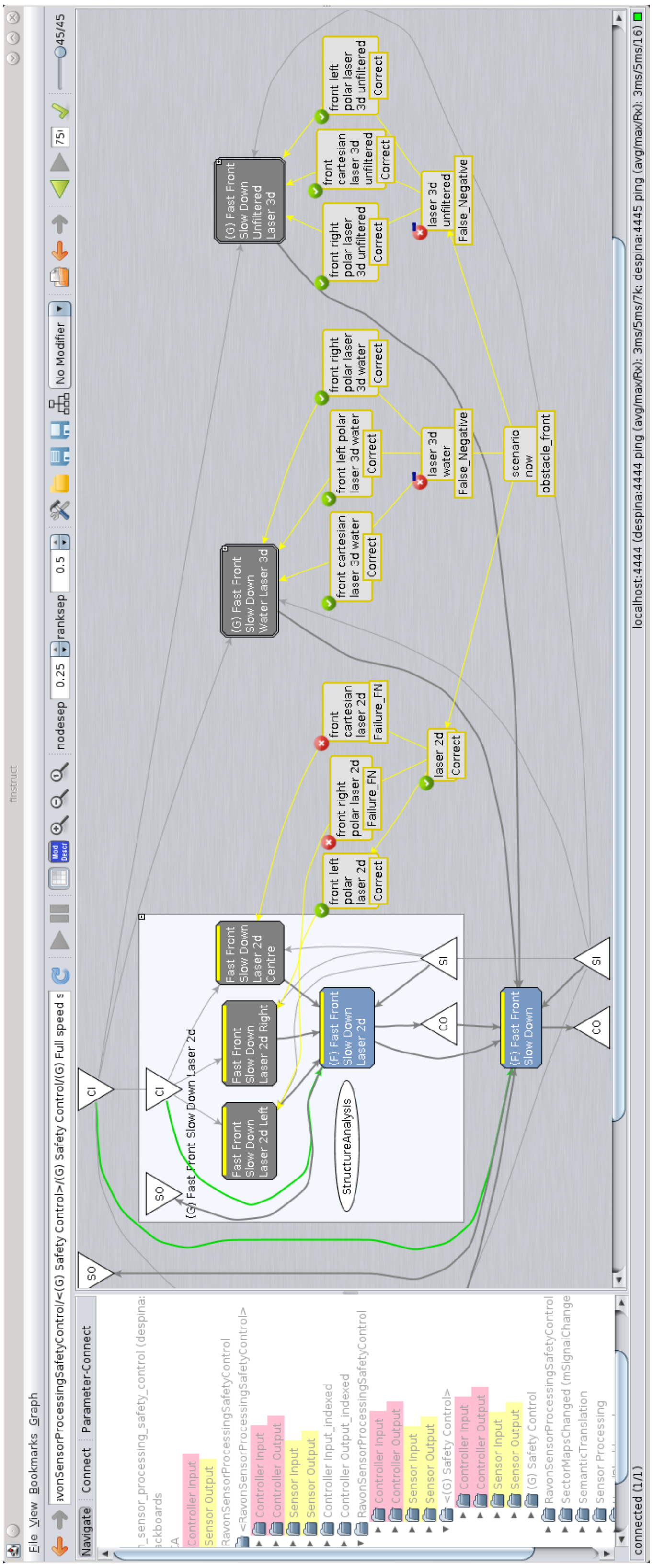

File View Bookmarks Graph
nodesep 0.25 ranksep 0.5
No Modifier
45/45
(G) Fast Front Slow Down Unfiltered Laser 3d
front left polar laser 3d unfiltered Correct
front cartesian laser 3d unfiltered Correct
front right polar laser 3d unfiltered Correct
laser 3d unfiltered False_Negative
(G) Fast Front Slow Down Water Laser 3d
front right polar laser 3d water Correct
front left polar laser 3d water Correct
front cartesian laser 3d water Correct
laser 3d water False_Negative
scenario now obstacle_front
front cartesian laser 2d Failure_FN
front right polar laser 2d Failure_FN
front left polar laser 2d Correct
laser 2d Correct
Fast Front Slow Down Laser 2d Centre
Fast Front Slow Down Laser 2d Right
Fast Front Slow Down Laser 2d Left
(F) Fast Front Slow Down Laser 2d
(F) Fast Front Slow Down
(G) Fast Front Slow Down Laser 2d
StructureAnalysis
CI
CO
SI
SO
Navigate Connect Parameter-Connect
Controller Input
Controller Output
Sensor Input
Sensor Output
(G) Safety Control
RavonSensorProcessingSafetyControl
SectorMapsChanged (mSignalChange
SemanticTranslation
Sensor Processing
connected (1/1)
localhost:4444 (despina:4444 ping (avg/max/Rx): 3ms/5ms/7k; despina:4445 ping (avg/max/Rx): 3ms/5ms/16)

Figure 5.22: Extended *TraceView* including the sensor chain and the scenario. The states of the sensor components that represent states of the backtrace of the model checker are displayed in the yellow box below the component's box. The green and red balls on the upper left edges of the boxes illustrate the state of the component additionally.

component and property contain a drop-down menu to allow the user to define the state (*Forced*, *Forbidden* or *Don't care*) of the property for this component. Once, the user chooses a special state, the respective part of the query is written in the query text field in the lower half of the window. As the component tables can become very large for big networks, it is possible to select a component in the *VerificationView*, which will highlight the corresponding component in the table to simplify the identification. One exception of the table representation of the components is the *Scenario* tab. As the scenario is normally a very simple automaton, a more intuitive representation which shows the original automaton with the possibility to right-click on a location to select a certain state is chosen (see Fig. 5.21b). The query field shows the final query and offers the possibility to select the type of the query and to negate the whole query. Additionally, a comment can be added.

The window offers the functionality to create several different queries for the model and save them into a file. Furthermore, according to requirement SQVR 3, it is possible to automatically check the query/queries on the formal model. Here, it is also possible to choose a verifier on an external machine, which is useful in case of computationally intensive queries.

The result of the model checking process is displayed in a pop-up window and the trace to the witness or counter-example can be shown in the *TraceView* as required in SQVR 4. The trace is a sequence of states of the formal model. This means the state of every automaton is given for every step in the trace. An example that depicts one of these steps is presented in Fig. 5.22. It shows the same components as Fig. 5.20 with the difference that additionally the states of the automata corresponding to the sensor processing components are illustrated. This is done using a textual information with the state name as well as a symbol indicating if this is a failure state. This visual support allows the developer a quick and easy insight into the model checker's result. More information about the implementation details of the *SafetyQueryView* and the underlying concepts can be found in the bachelor's thesis of [Vonwirth 15].

5.5 Quantitative Aspects

One major challenge of model checking techniques is the state-space explosion. To illustrate the meaning of this problem for the approach presented in this thesis, this section gives insights in the size of formal models of behaviour-based systems calculated in terms of numbers of locations and edges and the corresponding size of the state-space needed for UPPAAL's calculations. In [Armbrust 13a] and [Armbrust 15] this topic is already discussed with a focus on fusion behaviours and the central behaviour stimulator. In this section, the effects of including the sensor chain into the formal model will be discussed. Table 5.1 gives an overview of the numbers of locations and edges for each model presented before. As can be seen there, the templates for stimulation, inhibition and activation are equal for all behaviour types, whereas the activity and target rating are modelled individually for fusion behaviours and safety behaviours. The numbers of locations and edges for the `InhibitionInterface` cannot be directly extracted from the template, as the size of the automaton depends on the number n_i of inhibiting behaviours. The template has the structure of a hypercube of dimension n_i with bidirectional edges (Locations: 2^{n_i}, Edges: $2 \cdot (n_i \cdot 2^{n_i - 1})$). Additionally, whenever ι_B changes a committed location and an additional edge are needed. This must be applied for all additional inhibiting behaviours. In summary, the template contains $2^{n_i}+2\cdot n_i$ locations and $n_i\cdot 2^{(n_i-1)}\cdot 2+2\cdot n_i = n_i\cdot(2^{n_i}+2)$ edges.

Table 5.1: The numbers of locations and edges of each template (n_i: number of inhibiting components; n_c: number of competing components; n_s: number of scenario locations). The * indicates a template that is used for all types of behavioural components.

Template	#Locations	#Edges
`StimulationInterface*`	4	4
`InhibitionInterface*`	$2^{n_i}+2\cdot n_i$	$n_i\cdot(2^{n_i}+2)$
`ActivationCalculation*`	4	7
`ActivityCalculation`	4	6
`TargetRatingCalculation`	2	2
`FBIBActivityChanged`	8	12
`FBActivityCalculation` (V. 2)	$2^{n_c}+2\cdot n_c$	$n_c\cdot(2^{n_c}+2)$
`FBTargetRatingCalculation`	$2^{n_c}+2\cdot n_c$	$n_c\cdot(2^{n_c}+2)$
`SBActivityCalculation`	8	12
`SBTargetRatingCalculation`	4	4
`VirtualSensorAutomaton`	9	13
`RealSensorAutomaton`	13	22
`SensorFailureAutomaton`	2	1-2
`Scenario`	n_s	n_s-1

The same calculations must be done for the activity and target rating automata of the fusion behaviours, as they depend on the number of connected behaviours. As described in [Armbrust 13a], there exist two versions for the modelling of the activity of fusion behaviours. The first version uses only one automaton, whereas the second version uses two (`FBIBActivityChanged` and `FBActivityCalculation`). As the implementation is based on the latter, it will also be used for the following calculations. The numbers of locations and edges of the sensor automata (`VirtualSensorAutomaton`, `RealSensorAutomaton` and `SensorFailureAutomaton`) can be directly seen in the templates. For the sake of completeness Tab. 5.2 gives an overview of the number of variables and channels needed in the set of templates of the different components.

When calculating the total number of locations and edges of a behavioural network, the number of instances of each template has to be considered. As example, the group pictured in Fig. 2.10 will be taken. Each of the inner groups consists of one fusion behaviour (FB) and three safety behaviours (SB). Each safety behaviour corresponds to one virtual sensor (VS). Every group relates to one real sensor (RS). This leads to a set of automata containing 1 FB, 3 SB, 3 VS and 3 SF, 1 RS and 2 SF, where SF stands for `SensorFailureAutomaton`. This implies in total 154 locations and 230 edges for each inner group. As none of the behaviours is inhibited by another, the `InhibitionInterface` is not instantiated in the formal model and therefore provides no locations and edges to the model. Due to some improvements during the generation process of formal models, the number of locations and edges can be slightly smaller than the above mentioned. Those improvements are

Table 5.2: The numbers of variables and channels used in the set of templates of the different components (n_c: number of competing behavioural components; n_a: number of areas around the robot).

Component	**#bool**	**#int[0,1]**	**#chan**	**#broadcast chan**	**Others**
Behaviour	5	4	3	2	
Fusion Behaviour	$5 + n_c$	4	$2 + n_c$	3	
Safety Behaviour	5	4	3	3	bool $[n_a]$
Virtual Sensor	1	-	1	1	
Real Sensor	1	-	2	-	broadcast chan $[n_a]$

technical details which reduce the amount of locations and edges in some special cases. They will not be discussed here. The numbers presented in this section can be seen as worst case estimation.

The type of groups used in the example above, consisting of a fusion behaviour and about three safety behaviours occur frequently in the safety layer of the behaviour-based systems used in this work as the several parts of this layer can be generated automatically with the same design pattern. For the verification of tasks corresponding to environmental situations it is useful to verify such a group entirely as all contained behavioural components are based on the same real sensor. Because of that, Fig. 5.23 shows the trend of the number of states that need to be explored when traversing the whole state space in dependence on the number of locations or the number of groups consisting of three safety behaviours and one fusion behaviour (called SG for safety group in the following), respectively. The third and fifth markers correspond to the number of groups of the preceding markers with an additional fusion behaviour that fuses the outputs of the groups. The graph illustrates the state explosion problem when verifying a behaviour-based system with UPPAAL's verifier very well. For each additional SG the state space grows rapidly. For the exploration of the whole state space for three SG's already about 200 GB of memory are needed. This points out the need for improvements on the model size and complexity.

The tests are executed on an Intel XEON processor with 2.7 GHz and 1024 GB memory. UPPAAL's verifier is called with `verifyta -u -S 2 model.xml query.q`, where `-S 2` indicates that the optimisation of the memory consumption is set to "most" and `-u` shows a summary after the verification. The summary includes the number of explored states as well as time and memory consumption. The query used to determine the size of the complete state space of a model looks like

```
E<> !(FB_a_value==1) and RS.Correct and VS_1.Correct and VS_2.Correct and VS_3.Correct
and scenario.Pos_1 and !Processing()
```

where `FB` is any of the included fusion behaviours and `scenario.Pos_1` is the first location after the start location in the scenario automaton. For one of the inner groups (SG) of the network in Fig. 2.10 and the scenario automaton in Fig. 5.19 this will result in the query

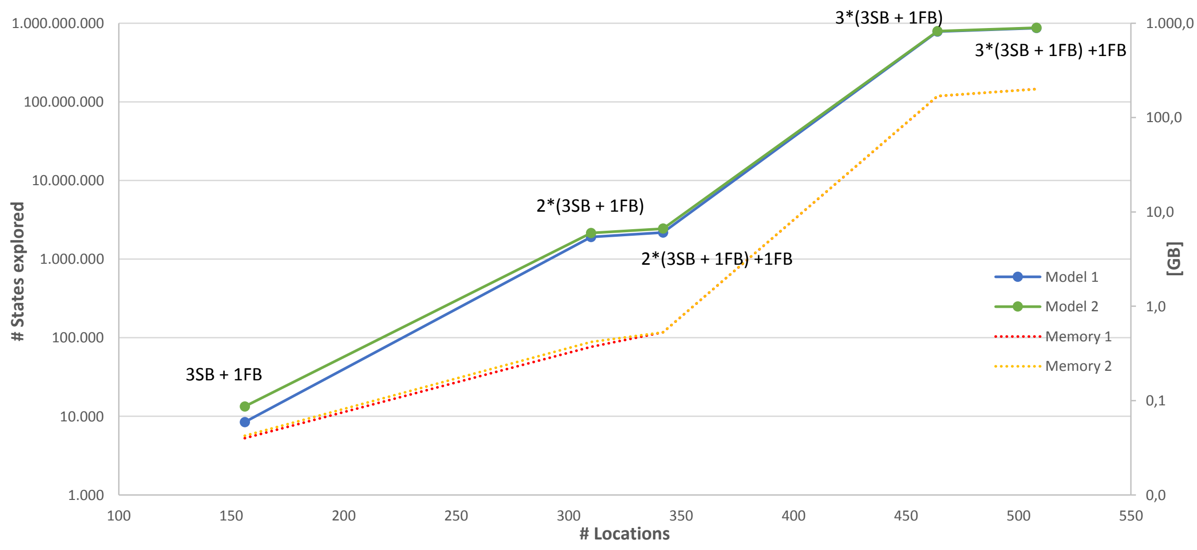


Figure 5.23: The chart depicts the number of explored states (blue, green) and the memory consumption (red, orange) in dependence on the number of locations calculated for one, two or three SG's containing three SB's and one FB. Model 1 is created with a sensor failure automaton without reset edge. The ones integrated in model 2 have a reset edge. Both models are created together with a scenario automaton with two locations.

```
E<> !(F_Fast_Front_Slow_Down_Laser_2d_0_a_value==1) and laser_2d_real_sensor.Correct
and front_right_polar_laser_2d_virtual_sensor.Correct and front_cartesian_laser_2d_
virtual_sensor.Correct and front_left_polar_laser_2d_virtual_sensor.Correct
and scenario.obstacle_front and !Processing()
```

which asks if it is possible that the fusion behaviour does not get active in case of correct working sensors and an obstacle to the robots front. As the example network is responsible to react to front obstacles, this query is evaluated to false after the whole state space is traversed in order to find a counter-example.

Sometimes, properties do not correspond to the whole network of the safety layer, but only on a part of it. The query above, for example, refers only to the functionality of the group responsible for sensor data coming from the 2d laser scanner. If this query is verified on a model of this group its state space contains about 13 371 states and can be evaluated in seconds using about 40 MB of memory. However, if it is evaluated using a model of the surrounding group (Fig. 2.10) again the whole state space is explored which contains about 876 428 843 states and uses more than 200 GB of memory in 6.6 h. This result shows the need to find a mechanism to model only those parts of the network, that are relevant for the property to check. An approach referring this topic is presented in Chap. 6.

In the following some basic ideas, to reduce the size of the model are discussed. One possibility mentioned in Sec. 5.3.4 to reduce the state space is to omit the reset edge of the `SensorFailureAutomaton`. As the automaton is instantiated twice per real sensor and once for each virtual sensor and can change its state non-deterministically, this seems to have great potential. The results show indeed a difference in the size of the state space, but not enough to have great influence on memory consumption. Figure 5.23 show the slight difference in the size of the state space and the memory requirements for models with sensor failure automata with and without reset edge.

Another question related to the integration of environmental conditions is the influence of the scenario automaton on the size of the formal model. To answer this, the models of the test cases written above are build with different scenario automata containing two, three or four locations, respectively. The used scenario automaton is based on the one presented in Fig. 5.19, where the last one or two locations are removed. The results show a big step in the number of explored states for models with two and three locations. In contrast, there is only a small difference between models with three and four locations. When changing the position of the location used in the query (e.g. for the examples above, position 2 *obstacle_front* is used, for the test here, the ordering of the scenario automaton is changed so that *obstacle_front* is on position 4), the number of explored states is smaller. The reason for that lie in the implementation of the verifier. In the context of this thesis, it is sufficient to know, that scenarios with more than two states expand the state space a lot.

Another aspect when trying to reduce the model size is to skip the instantiation of automata that are not needed. One example is the `InhibitionInterface` which is only included in case of inhibiting connections. Another one is the `TargetRatingCalculation`. For standard behaviours and fusion behaviours, the target is an independent automaton, which is not triggered from the outside. In the case, the output is not connected to another behaviour, there is no need to include its automaton in the model. For safety behaviours, the automaton is triggered by the virtual sensor, but if the output is not received by another behaviour it is in most cases unnecessary to model it. Tests showed that this step reduces the memory consumption a lot, from previously 200 GB to only 60 GB, approximately.

5.6 Discussion

In this chapter a concept for the formal verification of behaviour-based systems is presented. The need for a formal definition of a system and its properties is discussed and an approach to model behaviour-based components is introduced. With the focus on verifying autonomous robotic systems, a further step is taken, that integrates environmental conditions into the formal model. In Sec. 5.3, related work is discussed and the actual modelling of a robots sensor chain is presented. The robot RAVON serves as example application.

Several problems, that are often discussed in the context of formal verification, are also discovered during the application of the presented approach. These are, for example, the complexity of the model creation process and the formal definition of properties, the problem of interpreting the model checker results and the state-explosion problem. The first problem is faced with the automation of the modelling process. Together with the graphical user interface, this allows to model arbitrary behaviour-based networks in very short time. The integration of the sensor chain and the environmental conditions into the GUI is also presented. With that, all components that are formally modelled with finite-state machines can be seen in a simple component-network view in FINSTRUCT (see Sec. 5.4). The possibility to graphically define queries and to directly start the model checking process simplifies the verification procedure for the developer enormously. The problem of interpreting the model checkers result is faced with a special view, that shows the state of the behaviour network in every step of a trace. That *TraceView* provides the state of the elements of the sensor chain as well as of the scenario automaton. The scenario automaton is a facilitation for the creation of queries which allows to define the state of the environment in an easy manner.

In Sec. 5.5 quantitative aspects in relation to the integration of the different components of the sensor chain into the formal model are discussed. Here, it becomes apparent, that state-explosion is a big problem in the verification of models of behaviour-based systems already for relatively small networks. Some basic ideas to counteract this problem, which are also partly integrated into the implementation of the approach are discussed. Chapter 6 will deal with a more sophisticated approach and discusses some further ideas that support the developer in analysing and verifying large behaviour-based networks.

6. Reduction Techniques Supporting Large Network Verification

The formal verification techniques presented in this thesis are laid out to prove the structure of the behavioural network. Highly relevant are queries, that address the activity status of a certain component under special circumstances. The activity of a component indicates its relevance in the overall network, this implies, components that never get active, have no impact on the overall robot behaviour. For the verification of safety properties, the query if a certain component gets active at all or under certain environmental conditions is a standard question. For example, one of the queries verified in Sec. 5.4 is, if the slow down component of the robot is active during forward motion in the case of obstacles in the front of the robot. The proved component is the fusion behaviour of the group *(G) Fast Front Slow Down*, which is one of the innermost subgroups of the safety control network. But, this behavioural component does not actually indicate the state of the overall slow down behaviour of the robot. This can be seen in Fig. 2.6, where the outermost group for the safety control in full speed mode is displayed. The state of the fusion behaviour *(F) Fast Forward (Slow Down)* is responsible for the relevance of the transferred control data to slow down the robot. However, the proposed technique is not suitable to check the whole network related to that behaviour as the state space gets to large. Reduction techniques are needed.

The addressed problem is not specific to the verification of behaviour-based systems, but is well-known under the term *state-space explosion*. It means the number of states of the formal model exceeds the amount of available memory. Lots of research has been done considering this state-space explosion problem. According to [Pelánek 08] there were more than 100 paper addressing this topic from the year 1994 till 2009.

Many researchers aim at improving the actual model checking process by changing the formal model before or during the state exploration process in order to reduce the amount of needed memory. Since the freely available verification tool Uppaal is used in this work, improving the model checker goes beyond the scope. For the sake of completeness, Sec. 6.1 gives an overview of the major approaches and mentions the techniques used by Uppaal's model checker.

Another possibility for solving the state-space explosion problem is to reduce the system before the model checking process or even before formally modelling it. To achieve the best possible result a combination of the strategies can be applied which uses a minimised model and an "intelligent" model checker. Model reduction techniques can be divided into two subcategories: One summarises techniques that dynamically change the model with respect to the results of the model checker, the other category includes methods to statically reduce a model.

Dynamic reduction techniques are known under the term *Abstract-Check-Refine* or *counter-example-guided abstraction* [Clarke 00, Henzinger 02, Bouajjani 04]. Here, a minimal abstract model of the system is created, which is refined in dependence on the result of the model checker. The difficulty with this technique is to map the results of the abstract model to the original one. Section 6.3 gives an example of this technique and drives a method to reduce the model of behaviour-based systems.

The second category of model reduction techniques reduces the model statically according to properties or environment definitions. *Context-aware verification* is introduced by the authors of [Dhaussy 11], who assume that a special context belongs to a certain subset of the complete system. They form the model correspondent to the subset. Another static reduction technique is *slicing*. It aims at modelling only those parts of the software, that are relevant for the given problem. The first approach, known as *program slicing* was introduced by [Weiser 81, Weiser 84]. Since then lots of research has been done in this field and slicing was transferred to many different systems. Several papers summarise the approaches under different aspects, see [Tip 95, Xu 05, Silva 12]. Section 6.2 introduces the technique in more detail and describes the transfer to use it with behaviour-based systems. The presented methods to reduce the formal model of the behaviour-based system before the actual model checking process are applied to the example of the RAVON robot in Sec. 6.4.

Large and complex systems cause not only difficulties for formal verification, but also for other analysis methods. The complexity makes the analysis not necessarily impossible as with formal verification, but it makes the analysis process and the results confusing and difficult to evaluate. In Chap. 4.2 this problem was already mentioned in the context of fault tree analysis. The above mentioned techniques, slicing and abstraction, can also be used to reduce the behaviour network before analysis. But furthermore, current research deals with the combination of formal methods and fault tree creation, where one of the research topics is how to integrate one method into the other in order to reduce the analysis effort. Sec. 6.5 gives an overview of current research in this area and discusses the possible transfer to behaviour-based systems.

6.1 Techniques to Optimize the Model Checking Process

The methods to fight the state-space explosion problem during the model checking process can be divided into two categories based on how the state space is explored: enumerative algorithms and the symbolic algorithms. The authors of [Rafe 13] identify four major methods, namely *symbolic model checking*, *partial order reduction model checking*, *symmetry reduction*, and *on-the-fly model checking*, which will be shortly introduced in the following.

Table 6.1: Overview of reduction techniques used in UPPAAL (corresponding to [Slomp 10]). (*) In order to use symmetry reduction, some manual changes have to be done before model checking. (**) Implemented, but actually not integrated into current UPPAAL version.

Technique	**In Uppaal**	**Automatically enabled?**
Symmetry reduction	+	+/- (*)
Partial order reduction	- (**)	-
Slicing	- (**)	-
Dead variable analysis	- (**)	-

Symbolic model checking techniques are based on binary decision diagrams. The state space is represented by boolean functions and formulae in propositional logic. With this method, large systems can be checked as this representation needs much less memory than the explicit storage of the transition system as done using enumerative methods.

For concurrent systems the **partial order reduction** technique can be applied. Here the system is modelled as a set of interleaving sequences. Some of theses sequences might be similar in a way that they produce the same output but have a different internal arrangement of the concurrent components. Partial order reduction techniques find these and reduce the system by these sequences. [Pelánek 08] gives further examples of techniques that reduce the model by analysing the paths in the model.

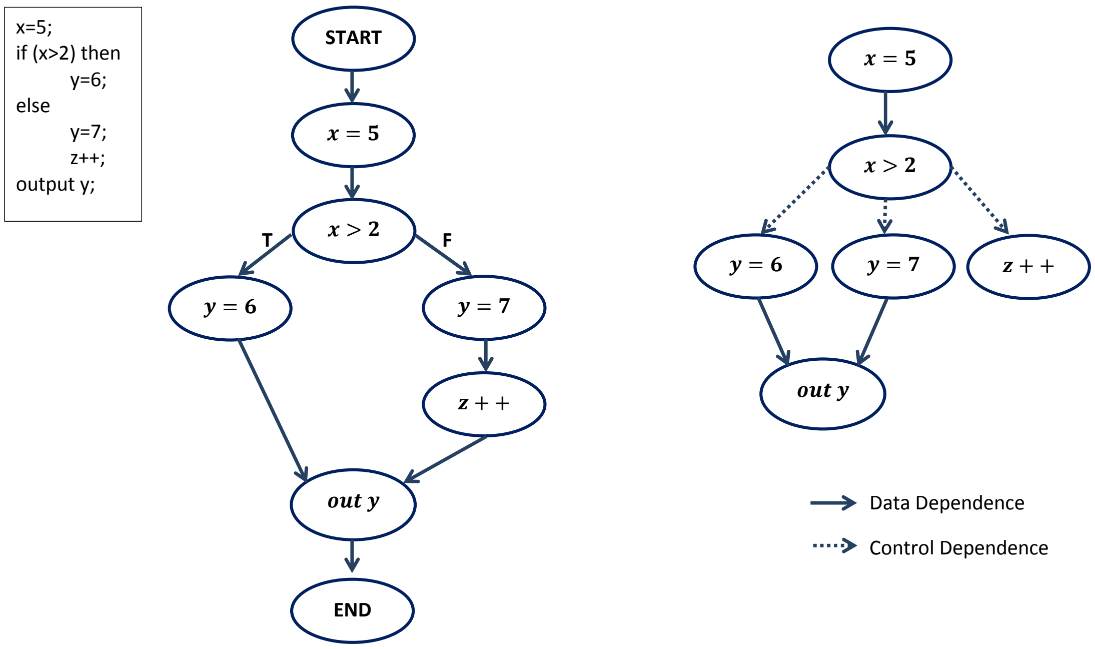
In contrast to that path based reduction techniques, **symmetry reduction** is based on the system states. The technique identifies *bisimilar* states. These are states that behave in the same way, meaning one system simulates the other and vice-versa. The traversal is done for only one of them. This method can be used **on-the-fly** during the exploration of the state space or before the exploration as a static modification of the system. On-the-fly model checking techniques construct the state space incrementally instead of building the whole state space first. This method reduces the state space in case where a counter-example is found early in the model checking process.

More detailed information about these techniques can be found in [Pelánek 08]. The paper shall be used as a decision guidance for practitioners who want to use model checking for the verification of their industrial projects. They focus on enumerative methods. Other overview papers which focus on different aspects are written by [Clarke 01, Clarke 12].

For the execution of model checking tasks the model checker of the UPPAAL toolbox is used in the context of this thesis. Table 6.1 gives an overview of the reduction techniques implemented in UPPAAL's model checker according to [Slomp 10]. Actually, only symmetry reduction is implemented in the available UPPAAL version (see [Hendriks 04] for details). The other three reduction techniques are implemented as tools or prototypes but not (yet) integrated into UPPAAL. The implementations of the techniques all aim at modifying the formal model before the actual model checking process in order to reduce the resulting state space. The authors of [Thrane 08] present a promising approach, that uses *slicing* techniques to reduce the model with respect to a given property. In contrast to the approach presented in Sec. 6.2, they construct a slice from the formal model including C code and automata components. According to the UPPAAL tutorial [Behrmann 06], partial order reduction [Bengtsson 98] is not supported by the tool, but some unnecessary interleavings can be avoided using committed locations. A further reduction possibility mentioned in the tutorial is the resetting of variables to their initial value when they are not used. [Slomp 10] discusses this problem and presents a tool that automatically identifies *dead variables* and includes the resets into the model.

6.2 Model Reduction Using Slicing Techniques

In contrast to the previously described methods, that are based on the formal model, the slicing technique introduced by [Weiser 81] reduces the program code. The reduction is done according to a given slicing criterion. In a first step, the code is decomposed based on a data and control flow analysis. The result is called *Control Flow Graph*(CFG), where nodes correspond to program statements and edges relate to the control flow. The elements of the graph that form the slice are computed according to the *slicing criterion* which is defined as a pair $c = \{s, v\}$, where s is a statement in the program and v represents a set of variables.



```
x=5;
if (x>2) then
        y=6;
else
        y=7;
        z++;
output y;
```

Figure 6.1: Example of a control flow graph (middle) derived from a C program (left) and the corresponding program dependence graph (right) [Yatapanage 12]

Following research is often based on a so-called *Program Dependence Graph* (PDG). Here, statements and expressions represent the nodes of the graph, edges define the dependencies between the nodes (see [Ottenstein 84] for the first approach with this topic). Figure 6.1 shows an example of a CFG and a PDG from a given program.

Having this graph representation, a slice is received via a backward traversal starting from the node of interest, which is defined by the slicing criterion. Figure 6.2 shows the general procedure to create a slice from program code.

For debugging purposes usually *backward slicing* is used, which means that the resulting slice contains those elements that directly or indirectly influence the node of interest.

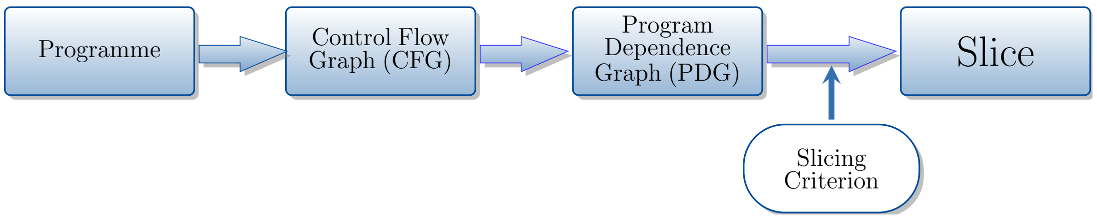


Figure 6.2: The steps to create a slice from program code

Forward slicing [Bergeretti 85] starts from the point of interest and traverses the graph to find out which other nodes are affected by the start node. This method is used mainly for dead code removal and software maintenance. Slicing methods can also be distinguished in *static* and *dynamic slicing*. A dynamic slice is computed with respect to a specific test case or rather a certain set of input variables. Static slices are extracted without making assumptions about the input.

Apart from the methods that operate directly on program code, several approaches that use specifications written in other languages were published in the last decade. For example, [Androutsopoulos 13] surveys the domain of state-based model slicing, which refers to slicing on the basis of finite state machine based models. Especially interesting in that context is the work of [Thrane 08], which apply slicing to UPPAAL models to achieve a syntactic reduction. Their experiments proved the advantages of using slicing methods in combination with the existing internal reduction strategies. Unfortunately, this work doesn't seem to be continued or even be integrated into the UPPAAL toolbox.

[Yatapanage 10] apply slicing to models written in the graphical *Behaviour Tree* language, whereby a behaviour tree is derived from the requirements the system must fulfil. For the automation of the slicing procedure, the slicing criterion is not manually defined but derived from a property (often) written in a formal language. That implies, that the criterion not necessarily corresponds with one node in the PDG but can refer to several statements. This technique of *property-directed* slicing was first introduced by [Hatcliff 00] and will be applied to the behaviour-based system used in this thesis as well.

An approach which includes the external environment into the verification process is presented in [Matsubara 12]. The paper introduces a variable dependence graph (VDG), which is similar to a PDG with a difference in the data unit. Before the actual slicing procedure, the parts of the system that are relevant to the given property are defined manually. Irrelevant parts are treated as part of the external environment and are not modelled in detail. The authors also developed a tool which supports the slicing and verification process by graphically representing the results.

For behaviour-based systems the need to identify context-specific relevant parts is already discussed in Sec. 4.1. Influence graphs, which include those nodes that influence a certain one, seem to be similar to a slice as defined in the former. The remainder of this section, discusses the application of the slicing technique to behaviour-based systems and presents the developed tool support on an application example.

6.2.1 Slicing of Behaviour-Based Systems

In this thesis, the verification of behaviour-based systems refers to the verification of the network structure. The formalization is done by modelling each behavioural component as a set of finite-state-machines, and connect those sets according to the connections in the network. The size of the formal model is therefore related to the size of the network. This implies, that a reduction of the network would decrease the model size.

In Sec. 4.1, the network structure is presented as behaviour dependence graph. With having this structure automatically when developing behaviour-based control systems, the first two steps of the regular slicing approaches - the creation of the data flow graph and the program dependence graph (see Fig. 6.2) can be omitted.

The slicing criterion must be defined in relation to the property, that should be checked on the formal model. For the definition of the slicing criterion, the approach at hand is based on the property-directed slicing technique introduced by [Hatcliff 00]. According to this, a slicing criterion can refer to several nodes in the graph. This is important, as queries for models of behaviour-based systems often contain more than one element (see Sec. 5.4 for examples).

A slicing criterion for a behaviour dependence graph must therefore contain the set of graph nodes $B(p)$ that corresponds to the formal query related to the property p. The slicing criterion is defined as follows.

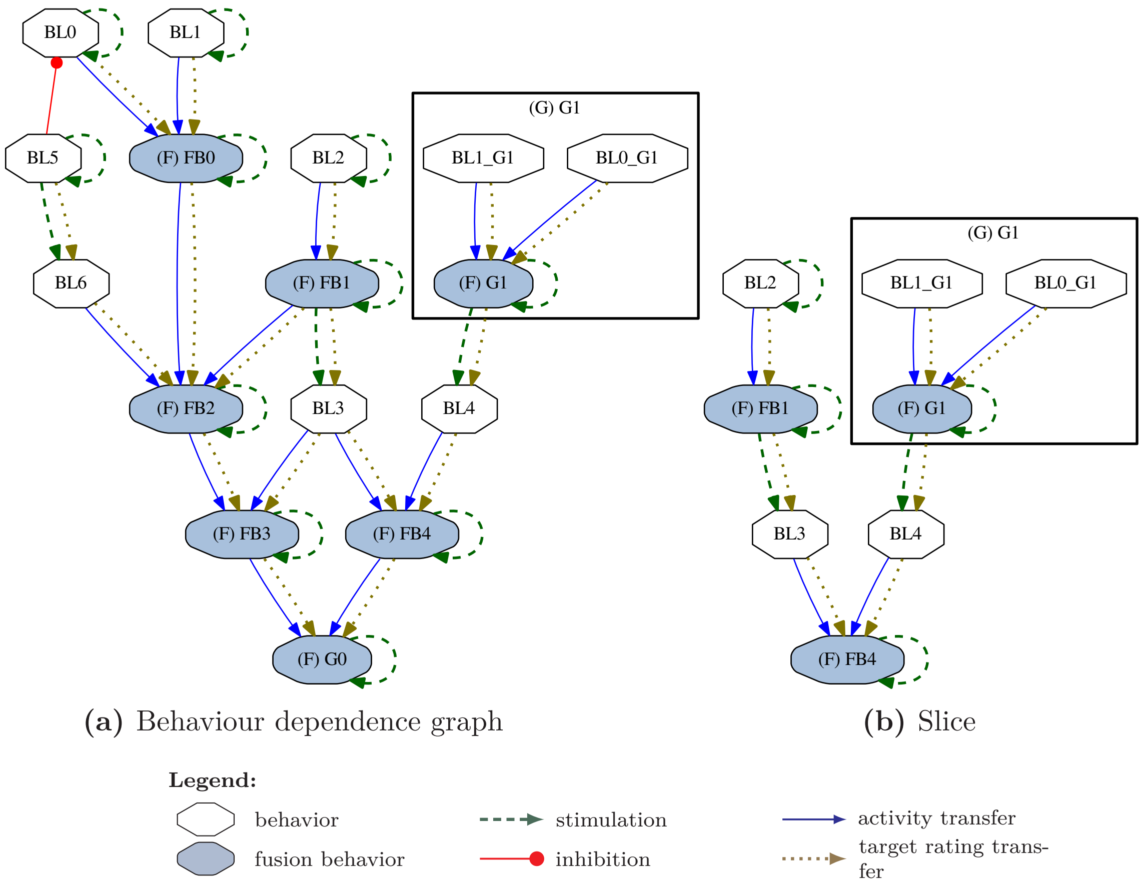
Definition 6.1: Slicing Criterion

A slicing criterion C for a behaviour dependence graph G_{BD} is a non-empty set of nodes $\{n_0, ..., n_k\}$ where each n_i, with $i \in [0, k]$, is a node in the behaviour dependence graph. The slicing criterion $C(p)$ for the property p contains all nodes in $B(p) \subseteq G_{BD}$.

The slice $S(p)$ for the property p is extracted from the BDG G_{BD} via a backward traversal starting from the nodes included in the slicing criterion. Here, every node is included into the final slice only once to avoid infinite traversals. The resulting slice is a graph $S(p) \subseteq G_{BD}$ that contains all components corresponding to $C(p)$ and all components that influence them. Usually, only one slice emerges containing all nodes of $C(p)$. If there arise two or more slices this implies that the behavioural components of $C(p)$ have no common basis. In order to minimise the model as much as possible, it is useful to split the property and prove each with its corresponding slice. In contrast to influence graphs presented in Sec. 4.1 a slice can have more than one *start nodes* and the resulting graph contains connections of all types.

According to Chap. 5.3, where an extension of the formal model is presented, properties can also include environmental conditions and states of sensor components. As those components are not part of the BDG they have no influence on the slicing process. The environment is still integrated in the model as soon as a safety behaviour is contained in the slice.

Figure 6.3 gives an example for the slicing of behaviour networks. The complete network in Fig. 6.3a contains 16 behavioural components where three are encapsulated in a group *(G) G1*. The property to check is `Is it possible, that behaviour (F) FB4`



(a) Behaviour dependence graph **(b)** Slice

Figure 6.3: The original graph 6.3a and the slice 6.3b created according to the slicing criterion $C(p) = \{(F)FB4\}$

`gets active?`. From this, the slicing criterion $C(p) = \{(F)FB4\}$ is formulated. The corresponding slice is pictured in Figure 6.3b.

The sliced graph $S(p)$ consists of only eight behaviour modules which is only half the number of the original graph. The generated formal model of the sliced graph consists of 30 automata with a total of approximately 55 locations and 71 edges in contrast to the model of the full graph with 64 automata (95 locations, 143 edges).

6.2.2 Tool-Assisted Slice Generation

In this section, the advantages of the slicing procedure shall be pointed out on the example presented in the previous chapter (see Sec. 5.4). Furthermore, the extension of FINSTRUCT's *VerificationView*, that allows to execute the process and to display the results are discussed.

The property to prove is again

P1: *(F) Fast Front Slow Down* shall be active if an obstacle is present in front of the robot.

It was verified on a small part of RAVON's safety control, namely the group *(G) Fast Front Slow Down*. This group was chosen manually in order to keep the formal model as small as possible. This is additional effort which can be reduced when executing the previously described slicing procedure.

In order to simplify the application of slicing techniques to behaviour-based networks, the functionality is integrated into FINSTRUCT's *VerificationView*. The tool is enhanced by the following features:

SV 1 Automatic slice creation for a property inserted as query

SV 2 Possibility to define the initial behavioural component for the slice manually in the graph view

SV 3 Display a slice

SV 4 Creation of a formal model from the slice

With the integrated automated slicing process, the model checking process containing the creation of the formal model and the submission of a formal property stays the same as described in Chap. 5. The slice corresponding to the given property is calculated in the background and the formal model of the slice is used as input for the model checker. This can also be done for several queries at the same time, which highlights one of the major advantages of applying the slicing procedure to the verification process of behaviour-based networks. The scope of the network must not be defined manually for each property in order to keep the resulting formal model small. It is done automatically and guarantees to output a minimal network including all relevant components.

For the developer, it might be of interest to view the slice displayed in the context of the overall network. This functionality is also integrated in the network view of FINSTRUCT. Figure 6.4 shows a detail of the group *(G) Full Speed Safety Control* (cp. Fig. 2.6) with the slice related to the example. All components belonging to the slice are shown in full colour, whereas other components are greyed out. Here, the slicing criterion was extracted from the property $C(p) = \{$*(F) Fast Front Slow Down*$\}$. The slice is calculated using this element as starting position in the behaviour dependence graph. The resulting slice includes the starting fusion behaviour of the group *(G) Fast Front Slow Down* and the four inner groups, that deliver data to it. All behavioural components inside the four groups are connected to the respective fusion behaviour and are therefore included in the slice as well. Due to readability constraints, only one of the groups is exemplarily unfolded.

The example shows again the possibility to dramatically reduce the model when using slicing. The overall group contains 112 components, whereas the slice includes only 14. The advantage of using the bigger group is to be able to create several queries containing components of different groups without changing the view.

The slicing functionality can also be used independent from query definition. The network view provides the possibility to mark single behavioural components as starting point for the slicing procedure. This enables the user to easily detect dependencies in the network structure. This view can also be used for the analysis tasks discussed in Sec. 4.1.

6.3 Model Reduction Using Abstraction

Abstraction is a very important technique for handling the state-space explosion problem. It means reducing the original programme or model in a way, that a property can be checked on the abstract model and a positive answer will hold for the original, as well.

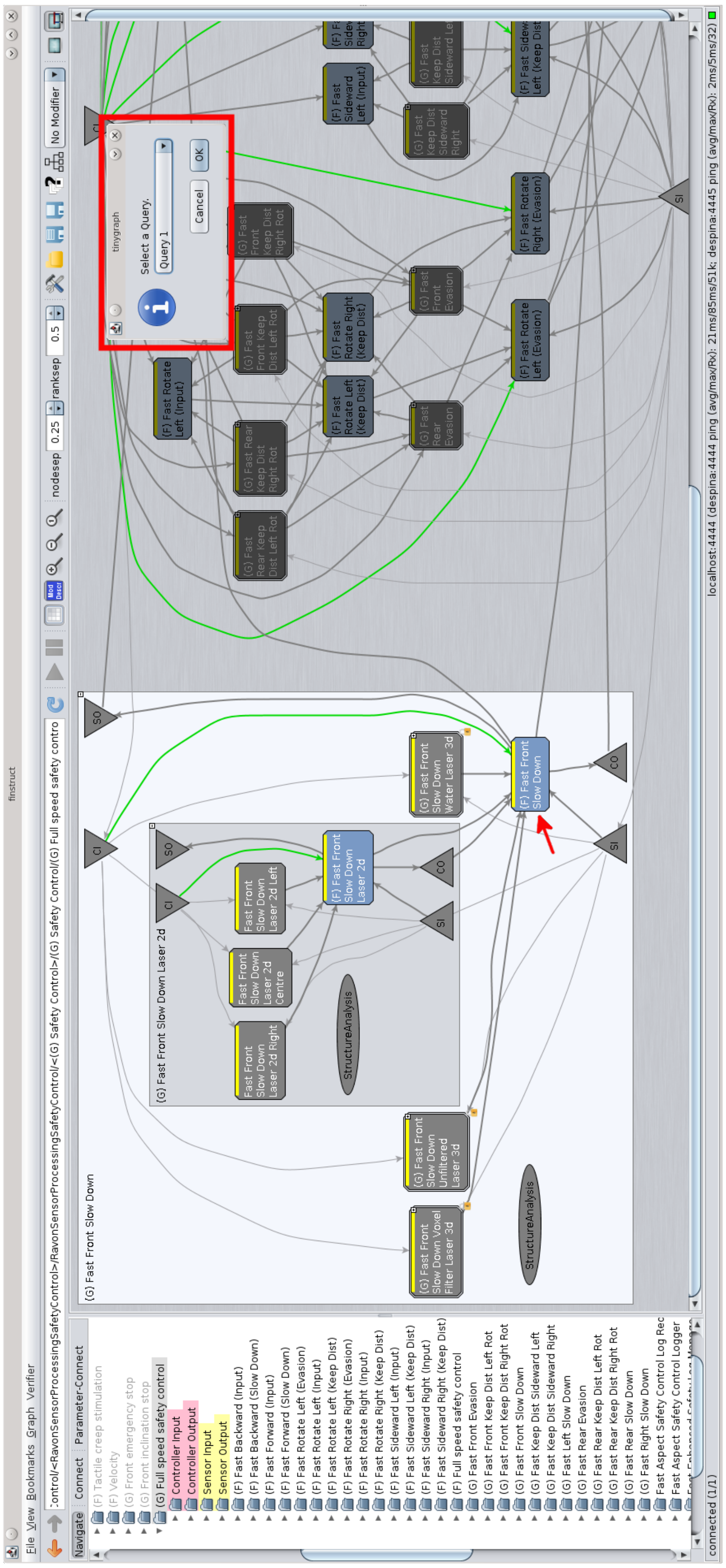

finstruct
File View Bookmarks Graph Verifier
No Modifier
nodesep 0.25 ranksep 0.5
Navigate Connect Parameter-Connect
(G) Fast Front Slow Down
(G) Fast Front Slow Down Laser 2d
(F) Fast Front Slow Down
(F) Fast Front Slow Down Laser 2d
Fast Front Slow Down Laser 2d Left
Fast Front Slow Down Laser 2d Centre
Fast Front Slow Down Laser 2d Right
(G) Fast Front Slow Down Water Laser 3d
(G) Fast Front Slow Down Unfiltered Laser 3d
(G) Fast Front Slow Down Voxel Filter Laser 3d
StructureAnalysis
tinygraph
Select a Query.
Query 1
Cancel
OK
CI
CO
SI
SO
(F) Fast Rotate Left (Input)
(F) Fast Rotate Right (Keep Dist)
(F) Fast Rotate Left (Keep Dist)
(F) Fast Rotate Right (Evasion)
(F) Fast Rotate Left (Evasion)
(F) Fast Sideward Left (Input)
(G) Fast Front Evasion
(G) Fast Rear Evasion
(G) Fast Front Keep Dist Right Rot
(G) Fast Front Keep Dist Left Rot
(G) Fast Rear Keep Dist Right Rot
(G) Fast Rear Keep Dist Left Rot
(G) Fast Keep Dist Sideward Right
(G) Fast Keep Dist Sideward Left
(F) Tactile creep stimulation
(F) Velocity
(G) Front emergency stop
(G) Front inclination stop
(G) Full speed safety control
Controller Input
Controller Output
Sensor Input
Sensor Output
connected (1/1)

Figure 6.4: Representation of a slice in FINSTRUCT's network view showing a part of the network given in Fig. 2.6. All behavioural components not included in the slice are greyed out. The small window in the upper right corner (red bordered) shows the dialog that is used to choose the query for which the slice should be visualised. The red arrow points to the element included in the slicing criterion.

In the case the check fails and the negative result could be due to the reduction and the process must be restarted under different preconditions. [Henzinger 02] describe the traditional flow for the model checking process of abstracted models as follows

Abstraction Choose a finite set of predicates and build an abstract model of the given program as a finite automaton. The states of the automaton represent truth assignments for the chosen predicates.

Verification Check the abstract model for the desired property. A positive result is equivalent to a positive result for the original program. A negative result produces a counterexample, which leads to the next step.

Refinement If the counterexample corresponds to a concrete counterexample in the original program, a program error is found. In the other case, the model must be refined, for example new predicates must be added and the process starts again.

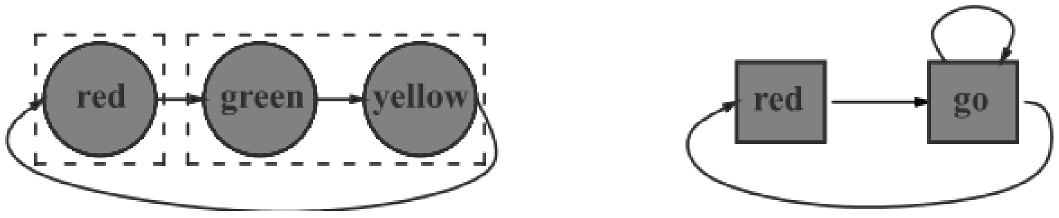


Figure 6.5: Abstraction on the example of a traffic light [Clarke 00]

An example for data abstraction is given in Fig. 6.5. A traffic light has three states, *red*, *yellow* and *green*. In order to check if cars can collide on a crossing, it is sufficient to prove whether they are allowed to drive or not. Therefore, an abstract model can be built which summarises the states *green* and *yellow* and map them to a state *go*. As the model allows cars to stop and go, respectively, the property is evaluated true, which holds for the original model with three states as well.

As example for the case where the abstraction is not fine enough, consider the property that the traffic light will always eventually come to state red. It can be easily seen in the original model, that the property is fulfilled there. Checking the property on the abstract model fails, returning a counterexample. For example there exists an infinite trace $< red, go, go, ... >$ that will never reach state *red*. As the counterexample contains abstracted elements, it does not correspond to a concrete counterexample. In that case, the abstract model needs to be refined.

Several research is done introducing different abstraction and refinement techniques and methods that automate the complete process of abstraction, checking and refinement [Clarke 00, Saïdi 00, Henzinger 02, Bouajjani 04]. The following section shows how the basic ideas of abstraction can be applied to behaviour-based systems.

6.3.1 Abstraction of Behaviour-Based Networks

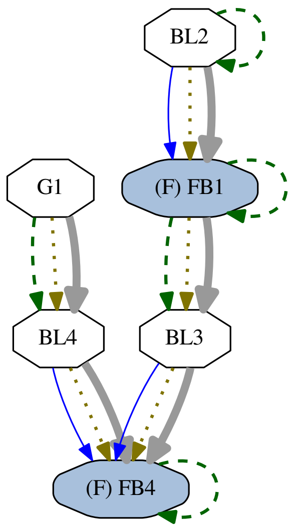
Working with large behaviour-based systems one structuring element is frequently used: *behavioural groups*. Behavioural groups encapsulate a number of behaviours (see Sec. 2.2 for the implementation in the iB2C). On the one hand, this is useful for displaying purposes, as large networks can not be represented in a proper way otherwise. On the other hand, groups combine a number of correlating components, which implies that the interface of a group provides a behaviour which is a combination of several sub-behaviours. This factor will be used for the generation of an abstract model of the system.

Abstraction: In a first step, the internals of all sub-groups of the behaviour-based network are neglected and represented by just one standard behavioural component for each group. The result is a network without nesting. In order to minimise it as much as possible, the slicing approach can be applied before. In the following, the network can be modelled formally as described in Chap. 5.

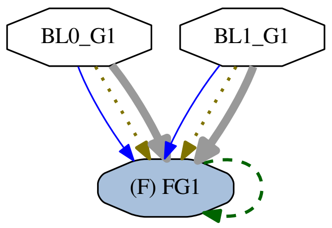
Verification: The formal model is checked for the respective property. In the case the property is fulfilled or the counterexample does not contain any component that represent folded groups, the result correlates to the original model. If the result depends on the state of the behavioural component that encapsulates a group, the network must be refined.

Refine: The state of a behavioural group is determined by the state of its corresponding fusion behaviour. Therefore, a model of the network influencing the fusion behaviour's activity or target rating must be build. The property to be checked includes the activity or target rating status of the fusion behaviour.

As a group may include further nested groups, this process must possibly be repeated. If the sub-requirements are fulfilled for every group component, the starting requirement is fulfilled, too. If not, the counterexamples point to a faulty part of the network.



(a) Abstract network with folded inner groups



(b) Refinement of the inner group of the nested graph

Figure 6.6: Application of abstraction and refinement on the example of the network depicted in Fig. 6.3a

As example, the network introduced in Sec. 6.2.1, Fig. 6.3b is used. Figure 6.6a depicts the abstracted network. The group *(G) G1* is represented by the component *G1*. The property to prove is

P1: It shall be possible, that fusion behaviour *(F) FB4* gets active when component *BL2* is inactive

Here, the model checker discovers, that the property is fulfilled if *G1* is active. As the behavioural component is modelled as described in Sec. 5.2.1 where the activity value switches non-deterministically between active and not active $\{1, 0\}$ the property is fulfilled. But, the internals of the folded group may determine its output otherwise. Therefore, the abstracted component is essential for the result of the model checker and it must be checked if it fulfils the requirements.

This means for the example, the group must be unfolded and the internal network must be checked. Figure 6.6b shows the refined behavioural group *(G) G1*. It is either possible to check the previous query on the refined model, or to use a new query with only the refined group. The latter allows to keep the formal model as small as possible, and will be used here. The property to check is

P2: It shall be possible, that *(F) G1* gets active

The corresponding query is evaluated to false, meaning the component can never get active. This implies that the initial requirement is also not fulfilled and the network need improvements. The reason for this failure must be discovered with further analysis. In this case, this is due to the fact that the input behaviours *BL0_G1* and *BL1_G1* can never get active, as they have no stimulating edges and are therefore inactive.

The presented approach enables the piecewise verification of nested behaviour-based networks. The network is verified starting with the highest layer and descending into the deeper layer according to the result of the model checker. The method avoids rechecking as far as possible since the inner groups structures are verified individually provided that no influences from outside of the group exist. The following section gives an example of the verification of a large behaviour-based network.

6.4 Application Example

The RAVON control system serves as an example for large behaviour-based networks. The before described applications of verification techniques have been done on small parts of the network, as their application on the complete network always lead to problems with memory consumption. This section will show how the previously introduced reduction techniques can be used to verify a large network piecewise on the example of the safety control of RAVON. The whole verification process can be executed with the help of the tool FINSTRUCT. For displaying purposes, the representation of the network parts will be done using GRAPHVIZ in this work.

RAVON's safety control is responsible for avoiding collisions by forcing the robot to slow down, evade obstacles or to initiate an emergency stop. It is integrated in the overall control system as a safety layer which is able to override control commands. As introduced in Sec. 2.3.1, the safety control distinguishes three drive modi related to the actual robot speed. The respective networks have an equivalent structure with differences in the use of sensors and sensor processing algorithms. For the example discussed in this section, the focus is on the network for the full speed mode as depicted in Fig. 2.6. The verification could have also be done on the whole safety control network or one layer above, but this would lead to problems in representing and understanding the correlations within the limits of this thesis.

The network of the group *(G) Full Speed Safety Control* is already to large to be verified without using reduction techniques and is therefore sufficient to show the introduced reduction approaches. The group facilitates the calculation of velocity and steering angles for evasion and slow down actions. In previous examples, subgroups of this group have been verified.

Here, the requirement *The robot shall slow down in case of front obstacles* shall be proven on the complete group. The fusion behaviour that fuses the components responsible for slowing down the robot is *(F) Fast Forward (Slow Down)* (see the part of Fig. 2.6 highlighted in green). This fusion behaviour should be active in case of obstacles in front of the robot and transfer adapted control commands. Furthermore, the behavioural component *(F) Fast Forward (Input)* which passes the original control data to the fusion behaviour should be inhibited by the safety groups. The exact property can be formulated as

P1: In case of obstacles in front of the robot, *(F) Fast Forward (Slow Down)* (F_FF_SD) shall be active and *(F) Fast Forward (Input)* (F_FF_Input) shall be inactive.

The property is translated to the formal query

```
E<> !(F_FF_SD_a_value==1) and !(F_FF_Input_a_value==1) and scenario_now.obstacle_front
```

and checked by UPPAAL's verifier. For all verification tasks in this section, the 64 bit version of UPPAAL's standalone verification tool VERIFYTA is called with `verifyta -u -S 2 model.xml query.q`, where `-S 2` indicates that the optimisation of the memory consumption is set to "most" and `-u` shows a summary after the verification. The calculations are done on an AMD Opteron^{TM} 6134 processor with 2.3 GHz and about 120 GB of memory.

Proving this property on the model of the complete group containing 118 behavioural components exceeds the available memory. Applying the **slicing method** described in Sec. 6.2, the network can be reduced to 32 behavioural components. Verifying the property on the slice reveals, that the property is not fulfilled. The counterexample shows, that this is the case, when all sensors fail. In order to find out further failure possibilities, the failing of one of the sensors is excluded (similar to the examples in Sec. 5.4). With this configuration, the available memory size is reached before a result could be achieved.

This shows, that using only the slicing approach for model reduction is still not sufficient to verify the whole network. Therefore, the abstraction technique is additionally applied to the network. That leads to a network containing three behavioural components each representing a folded group and two fusion behaviours, see Fig. 6.7. As assumed, the verification of the above query leads to a counterexample which proves that the fusion behaviour is inactive in the case that all other components are inactive. A further query shows, that it is sufficient for the fusion behaviour to become active when one of the components representing a group is active.

Therefore, one of the three groups, namely *(G) Fast Right Slow Down* is refined to verify its inner network. The group contains four subgroups and a fusion behaviour that fuses the outputs of the subgroups. The content of the group *(G) Fast Right Slow Down* with folded subgroups is depicted in Fig. 6.8. Three of the subgroups include a safety behaviour which is connected to its respective groups fusion behaviour. The group *(G) Right Slow Down Voxel Filter Laser 3D* contains only a fusion behaviour.

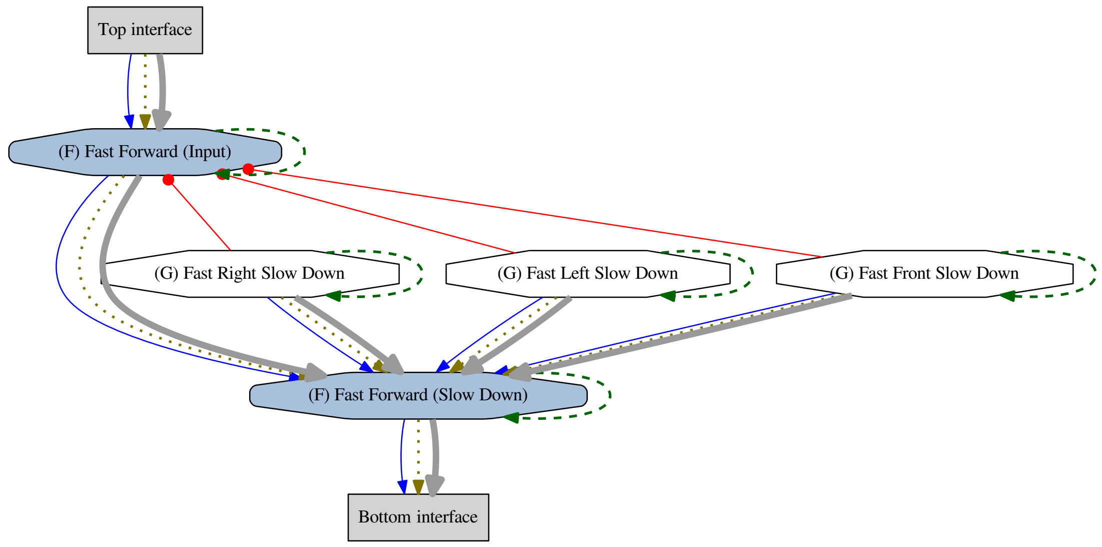


Figure 6.7: Abstraction of the slice $S(P1)$ according to the slicing criterion $C(P1)$ extracted from the network of the group *(G) Full Speed Safety Control*

For a first refinement step, the subgroups of the group are folded and each represented by a behavioural component, respectively. The property to prove for the group *(G) Fast Right Slow Down* is

P2: In case of obstacles in front of the robot, *(F) Fast Right Slow Down* shall be active.

Again, the model checker provides a counterexample showing that the fusion behaviour cannot get active in the case of all connected components being inactive, and just as in the previous step, one active component suffices to fulfil the query. A further refinement step leads to the unfolding of the subgroup *(G) Right Slow Down Laser 2D* (see Fig. 6.9) and the property

P3: In case of obstacles in front of the robot, *(F) Fast Right Slow Down Laser 2D* shall be active.

Here again, the query is not fulfilled. The counterexamples show a trace to the state where the safety behaviour is inactive and the connected virtual sensor and its real sensor work correctly. That means, that the subgroup *(G) Right Slow Down Laser 2D* will never get active in case of front obstacles (except for the case of failing sensors generating false positive results). The same result holds for the other two subgroups, *(G) Right Slow Down Water Laser 3D* and *(G) Right Slow Down Unfiltered Laser 3D*. The subgroup *(G) Right Slow Down Voxel Filter Laser 3D* can also never become active, as its input is not connected to another component. Combining the results of the four groups leads to the conclusion, that the group *(G) Fast Right Slow Down* never gets active in case of front obstacles.

Starting again in the first abstraction layer, the next step is to refine the group *(G) Fast Left Slow Down*. As it is build up from the same basis as the group for the right side, the results are similar. Therefore, the third group *(G) Fast Front Slow Down* must be refined. The group contains four subgroups encapsulating each a set of safety behaviours, which are combined according to their correlating sensors. The network with folded subgroups is depicted in Fig. 6.10.

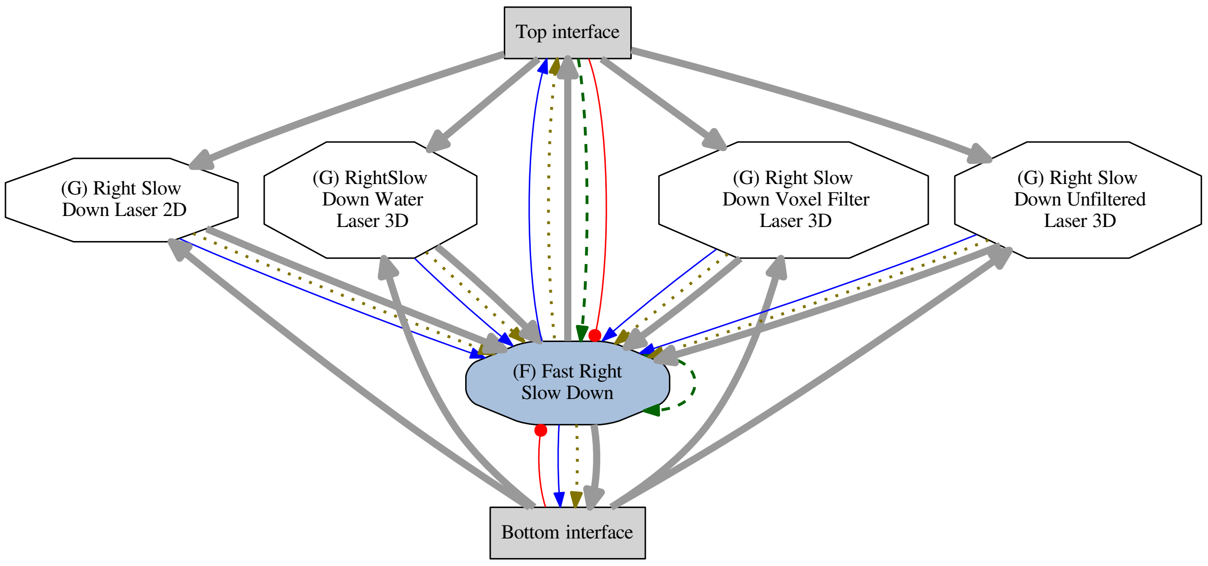

Top interface
(G) Right Slow Down Laser 2D
(G) RightSlow Down Water Laser 3D
(G) Right Slow Down Voxel Filter Laser 3D
(G) Right Slow Down Unfiltered Laser 3D
(F) Fast Right Slow Down
Bottom interface

Figure 6.8: Abstracted network of the group *(G) Fast Right Slow Down*

Here again, the property

P4: In case of obstacles in front of the robot, *(F) Fast Front Slow Down* shall be active.

is verified on the network with folded groups, and, as assumed it is not fulfilled in the case that all four behavioural components are inactive. Once again, a subgroup, namely *(G) Fast Front Slow Down Unfiltered Laser 3d* is refined (see Fig. 6.11). It encapsulates three safety behaviours that react to data provided by one of the virtual sensors, respectively. A fusion behaviour combines the outputs of the components and transfers it to the groups output.

Verifying the property

P5: In case of obstacles in front of the robot, *(F) Fast Front Slow Down Unfiltered Laser 3d* shall be active.

reveals, that the property is not fulfilled in the case of a sensor fail (*false negative*). Excluding this possibility still leads to an example, where the property is not fulfilled. This is the case when all three virtual sensors fail and therefore provide false negatives to the safety behaviour. Forcing one of the virtual sensors to work correctly together with a correctly working sensor, the fusion behaviour gets always active. This result similarly fulfils the initial property Prop. P1. The fusion behaviour *(F) Fast Forward (Slow Down)* is always active, when the 3D laser scanner and one of its corresponding virtual sensors work correctly and obstacles in the front of the robot exist.

In order to check the redundancy of the system, the refinement process can be done for the other two subgroups. The result will show, that it is sufficient if one of the sensors and one of its corresponding virtual sensors work correctly to fulfil the initial property. The example shows, that using the proposed reduction techniques facilitates the verification of large behaviour-based networks. The models created from the reduced networks are small enough to be verified using minimal memory and short computation time. Thereby, the overall computation time is also shortened dramatically. However, the technique requires additional effort from the safety expert to manually chose the next refinement

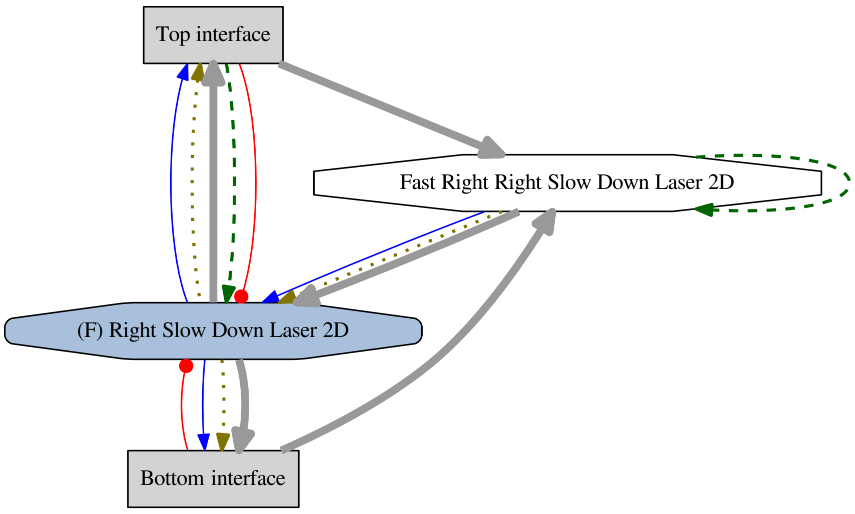

Top interface
Fast Right Right Slow Down Laser 2D
(F) Right Slow Down Laser 2D
Bottom interface

Figure 6.9: Abstracted network of the group *(G) Fast Right Slow Down Laser 2D*

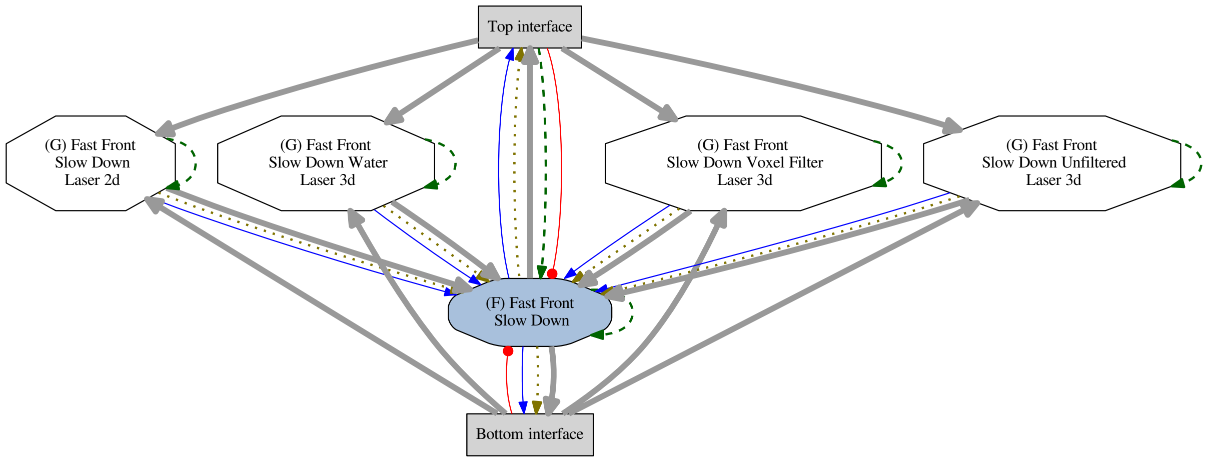

Top interface
(G) Fast Front Slow Down Laser 2d
(G) Fast Front Slow Down Water Laser 3d
(G) Fast Front Slow Down Voxel Filter Laser 3d
(G) Fast Front Slow Down Unfiltered Laser 3d
(F) Fast Front Slow Down
Bottom interface

Figure 6.10: Abstracted network of the group *(G) Fast Front Slow Down*

steps and to define queries for them according to the counterexample. But, extensions to the FINSTRUCT tool support the developer in order to simplify and accelerate the process.

6.5 Combining Fault Tree Analysis and Model Checking

Fault tree analysis and model checking are two radically different methods with the same goal: the analysis of safety critical systems. While fault tree analysis is used by safety engineers to calculate the probability of undesired events and to discover the possible root-causes, model checking proves a property given in a formal language to be correct on a formal model of the system. The advantages of fault tree analysis are their wide acceptance in industry, the comparatively simple creation of fault trees, and the definition of the property to analyse from the specification. The latter is one of the difficulties of model checking, as system specifications are rarely written formally, but include natural

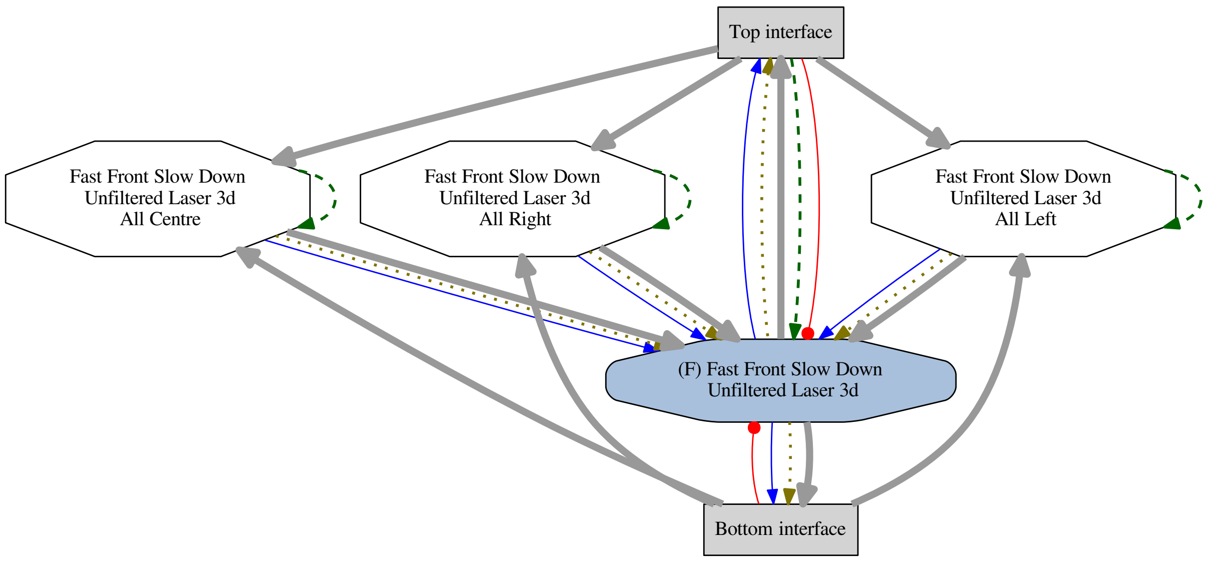


Figure 6.11: Abstracted network of the group *(G) Fast Front Slow Down Unfiltered Laser 3D*

language, figures, etc. A disadvantage of fault tree analysis is that its completeness is not naturally given.

The combination of fault tree analysis and model checking aims at taking the pros of both methods, while eliminating the cons. The great number of different research approaches show the various possibilities in which the methods can be combined. The intention of [Bieber 02] is to use both methods independently depending on the special task, and to represent the results on a common basis in order to combine them and make them available for both, safety engineers and system designers. [Thums 03, Schäfer 03] use model checking techniques to verify the proper design of a fault tree. Other approaches use the results of fault tree analysis, the minimal cut sets, to generate properties to be checked by the model checker [Santiago 05, Cha 12]. A different approach is used by [Akerlund 99], who creates fault trees according to counter-examples generated by the model checker. To overcome the difficulties in manually creating fault trees and formal queries, [Cha 12] present an approach that automates the whole process. In the remainder of this section, the possibilities to use similar approaches for the analysis procedure of behaviour-based systems are discussed.

Model Checking of Fault Tree Events

In Chap. 4.2 the application of fault tree analysis to behaviour-based systems was introduced. By the use of the structured composition of the software, the completeness of the fault tree can be achieved for the software part. But, another problem could be identified when using this technique: the size and complexity of the fault tree. One idea is therefore, to use model checking to eliminate parts of the fault tree by verifying their correctness.

The approach proposed in this thesis models a faulty data signal output of a behavioural component as combination of its data calculation and its activity calculation (see Sec. 4.2.1 and Fig. 4.9). In both cases, the event *structural fault* is introduced which addresses the possibility that the behavioural component is not correctly interconnected with others. This problem is also covered by the model checking approach. Model checking can verify the connection to be correct under given conditions, which would eliminate the event from the fault tree. By way of example, consider the network introduced in Sec. 4.2.2, Fig. 4.16 and Fig. 4.18. The structural fault considering the activity calculation of the component *Fast Front Slow Down Centre* can be excluded by verifying whether it is possible that the component can get active and inactive, respectively in case of obstacles. This result shows, that the state of the component is dependent on the sensor hardware, which is correct. The event can therefore be removed and therewith the size of the fault tree can be reduced.

Fault Tree Design Using Model Checking

Using model checking to design a fault tree has the goal to create a complete and correct tree. Some approaches build a formal model of the underlying system and build the fault tree from this model, others use series of queries for every leaf and use the counter-examples for further division. In the case of behaviour-based systems, the structuring of the software already enables a straight-forward fault-tree creation. The process can be automated in large parts and needs manually treating only in special cases. These are for example fusion behaviours (see Sec. 4.2.1) and behavioural components which interact with the environment (*safety behaviours*). In the case of safety behaviours, a negative result of the model checker provides a witness or backtrace including statements about the relation of the state of sensor hardware and behavioural components in the software network. For example, a behavioural component cannot be active in the case of a faulty sensor. The event *faulty sensor* can then be integrated in the fault tree. This procedure helps to create a correct and complete fault tree on the interface between (sensor) hardware and software.

Combining the Results of Model Checking and FTA

Using both analysis methods provides the possibility to overcome the drawbacks of the single ones. In the case of behaviour-based systems, the abstraction level of the formal model limits the possible queries. These can be covered by fault tree analysis, which allows to integrate any states of behavioural components and to consider data signals, which are neglected in the formal model. However, the evaluation of structural faults must be done manually for fault tree analysis and can be easily executed with model checking. Thanks to the structured representation of behaviour-based software systems, both methods can work on the same basis. The same holds for the representation of the results, counter-examples can be displayed in the network view as well as the set of components belonging to a certain cut set.

6.6 Discussion of Reduction Techniques

In this chapter several approaches to fight the state-space explosion problem are discussed. All aim at the reduction of the formal model. Some techniques reduce the model automatically during state exploration, others focus on the reduction of the original program before it is modelled formally and given to the verifier. Two of the latter, *slicing* and *abstraction* are presented in more detail and their fundamental ideas are transferred to behaviour-based systems. A combination of both reduction techniques is applied to the control system of the RAVON robot. The example shows that using the reduction methods enables to verify large networks that could not have been verified without them.

The application of slicing to the behaviour-based network is a simple, but powerful approach, that uses influence graphs to find out the components that relate to the one specified in a property. The slice can be used in various ways, not only to be the basis of the formal model for formal verification, but also for other analysis tasks (see Chap. 4) or to gain understanding about the network structure e.g. for non-system-experts.

The abstraction approach is also based on the existing structure of behaviour-based networks. It utilises the concept of encapsulating a number of network components in groups. Using this technique, the number of network components can be drastically reduced. The method is the basis for the automation of the reduction and refinement processes for behaviour-based systems. Moreover, the abstraction approach offers further reduction possibilities taking other system conditions into account. For example, environmental conditions like the precise definition of the position of an obstacle can be used to exclude further elements of the network.

7. Conclusion

The increasing number of robotic systems entering our daily life raise the need for methods to analyse and verify the safety of such systems. For decades, this question was limited to either hardware or software, or later to industrial robots that are a combination of both, but acting in a restricted area with great safety arrangements concerning their environment. Today, embedded systems are wide-spread in various devices interacting with humans and in changing environments.

Behaviour-based systems have been described in this thesis as an architecture for the control of robotic systems. They feature among others, redundancy, expandability and reactivity implemented in simple, reusable components, which make them a promising approach for the control of autonomous mobile robotic systems. The complexity of the control network structure of advanced systems has been identified as a downside of this approach.

The requirement of having a flexible architecture for the implementation of robotic systems as well as the need for save and reliable systems formed the topic of the thesis at hand. The presented concepts allow a comprehensive analysis and verification of behaviour-based control systems. They support the developer to gain a deep understanding of the relationship of hardware/software components and the relationship of system events that cause a robot's action. Furthermore, they allow for the identification of root causes of failures and for the formal verification of safety requirements. The work has been divided according to two objectives. The first has been the identification of suitable analysis methods, whereas the second has dealt with the application of formal verification. Figure 7.1 shows the concept.

Analysis is an elastic term with several areas using it to name their special methods. In the context of this thesis, it has been utilised in three different meanings in Chap. 4.

Structural Analysis: The structural analysis targets the determination of the static characteristics of a behaviour-based network. Here, basic checks as the correct connection of edges are combined with more sophisticated ones analysing cycles and dependencies in the network, or the determination of the relevance of a component.

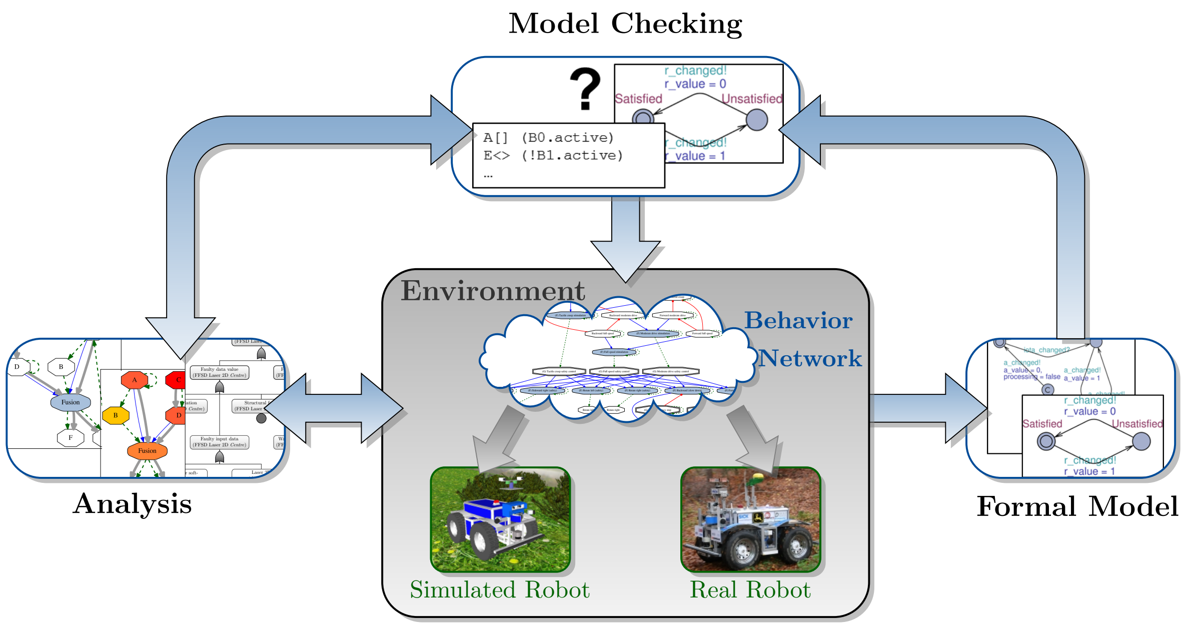


Figure 7.1: The concept of this thesis. The behaviour-based network together with its execution platforms (simulated or real robot) embedded in its environment is verified and examined using several analysis techniques. For formal verification a formal model is created. The results of the Uppaal model checker can be fed back to the control system and/or used as reference point for analysis. Vice versa analysis methods can be used to detect sources of failures detected in the formal verification process.

The techniques and outcomes of the structural analysis have proven to be useful for the other analysis methods presented in this work.

Hazard Analysis: *Fault tree analysis* has been chosen in the work at hand as a technique to identify system components and events that lead to a safety hazard. A special focus was set on the creation of fault trees combining hardware and software events. A concept to model the behavioural network as a fault tree has been presented and the application has been shown on the example of the outdoor robot RAVON.

Data Analysis: The problem of detecting and tracing faults in the network has been addressed using the term *data analysis*. As the interaction of single components make it difficult to clearly identify the root causes of a robot failure, several techniques have been proposed to support the developer. Among these are static analysis techniques that use data derived during test cases for offline analysis as well as online methods.

All these methods aim at the verification of the system with respect to safety. Formal verification is a special kind of verification using mathematical methods to prove a system's correctness. Model checking, as one method to formally verify systems, has been chosen for the verification of behaviour-based systems in this thesis. The basis for model checking techniques is a formal system model, which needs to be extracted from the (software) system. In the case of robotic systems, it is useful to represent the combination of software and hardware in the formal model in order to make statements about their correct interaction. Chapter 5 has dealt with these problems.

Formal Modelling: The Uppaal toolbox has been selected as model checker for the work at hand. Accordingly, a formal model of the behaviour-based control system has been built as a set of finite-state machines. Every behavioural component is modelled as a set of five FSM's, which represent the component's interface (stimulation, inhibition, activity, target rating and activation). The individual automata are connected with each other via sending and receiving edges. An automated process supports the developer in the creation of the formal model out of a behavioural network.

Hardware and Environment Integration: *Safety behaviours* have been identified as the interface between software and hardware. Their task is to evaluate sensor data and generate the corresponding control data. The automata modelling a behavioural component have been adapted to model these special characteristics. Furthermore, the sensor chain including the real sensor component and the corresponding virtual sensor are modelled as finite-state automata. Failure automata have been introduced to model possible failures. Additionally, a scenario automaton has been modelled to define the environmental conditions where the robot is situated.

In order to facilitate the application of model checking, the process has been automated as far as possible and tools supporting the modelling, generation of queries and backtracking of negative model checking results have been developed. The generation of safety relevant queries and the handling with the model checker's results have been shown on the example of the autonomous outdoor robot RAVON.

The special issue of handling large and complex networks has been addressed in Chap. 6. Especially for the model checking method, the *state-space explosion problem* has been identified as existing for behaviour-based systems as well. Current research targeting this topic has been investigated and could be divided into two areas, namely methods to optimise the model checking technique and methods to reduce the formal model. The latter has shown to be most suitable for the work at hand. Two reduction techniques, *abstraction* and *slicing*, have been adapted to the use with behaviour-based systems. Their benefit could be proven on the RAVON example. Also addressed in this context is the combination of fault tree analysis and model checking. Several combination possibilities have been discussed. They do not only target the reduction of the system to analyse but also determine ways to benefit from the advantages of both methods.

Great importance has been attached to the topic of automating analysis methods and graphical support. As especially for large and complex networks, the manual analysis is not feasible, several tools have been presented throughout this work that reduce the analysis effort and visualise the results. The examples using the autonomous mobile off-road robot RAVON which is controlled by a network with more than 400 behavioural components, have shown the applicability of the presented concept to a complex real-world system.

7.1 Results and Discussion

The work at hand has provided a number of methods to verify a behaviour-based robot control system according to safety properties. The application examples have demonstrated their applicability to complex robotic systems on the example of the autonomous robot RAVON. The analysis techniques of Chap. 4 have provided valuable information about the system, either to identify weak points in the system or to determine sources of failure. The trouble of handling large networks manually has been overcome by the development of tools that support the analysis process. Nevertheless, the problems which had already been identified in related research could also be seen here. The results of the analysis process are strongly dependent on the experience of the executing person. This problem could be mitigated by automated methods that identify faults in the system according to the system specification, or by methods that automatically detect faulty system states. Additionally, the complexity of creating a fault tree out of the behaviour-based control software is faced with providing a standardised model to represent a behavioural component as fault tree element.

It has been shown in Chap. 5, that the integration of sensor hardware components into the model checking process enables not only to determine correct software, but also their correlation with hardware (and possible hardware failures). The approach to additionally model environmental conditions, simplifies the transfer of safety requirements used for the analysis procedure in Chap. 4 to properties that shall be verified formally. The emerging problem of formulating properties formally to be processed by the model checker has been overcome by the development of a tool that supports formal query generation. The developed tool also visualises the results of the model checking process which encounters the problem of interpreting the abstract results on the real system.

The application on the real robot system has shown the limits of this verification approach, which become noticeable by the large memory consumption and long verification times. This topic has been addressed in Chap. 6. The introduced techniques aim at the reduction of the formal model which have been implemented as reduction of the behavioural network to be verified before formally modelling it. This approach has the advantage that the special properties of behaviour-based systems can be exploited to achieve a maximal reduction without losing relevant information. Furthermore, the reduction techniques are not only usable for formal verification but can also be applied before the execution of the other analysis techniques. The application example of Sec. 6.4 has shown the improvements in the model checking process using reduction techniques.

In summary, the analysis and formal verification techniques presented in the work at hand all have their strengths and weaknesses. While the analysis methods of Chap. 4 are intuitive and comprehensible also to non-experts, the lack of completeness and the complexity of the result representation can lead to the missing of faults. In contrast, the results of the model checker are easy to interpret due to the developed tool support which allows to easily identify faults in the system. However, the abstraction level of the formal model forces to exclude certain system characteristics from verification. A combination of the different techniques allows for a comprehensive analysis and verification of behaviour-based systems acting in unstructured environment. The knowledge of relations between system events allows for the estimation of the criticality of hazards. Understanding the relation of system components facilitates the identification of failure sources. Finally, formal verification enables to prove that certain properties hold for the given system.

7.2 Future Work

The presented concept for the analysis and verification of behaviour-based systems can be extended in several directions. The process of fault tree creation is currently a manual process guided by the developed modelling approach. The automation of the creation process should be feasible using the results of the structural analysis and the modelling guidelines and would be a great facilitation for the developer. The interpretation of the results can be evaluated utilizing the approach presented in [AlTarawneh 14, AlTarawneh 15] as basis, where a further extension could be to additionally show the minimal cut sets of the software fault tree analysis in the behaviour-based network structure.

Another task for future work is the handling of the complexity of the resulting fault trees. The reduction techniques presented in Chap. 6 have already been applied successfully to reduce the formal model for the model checking process in the context of this thesis. Their application in combination with fault tree modelling needs to be investigated and integrated in an automated process.

The formal model of a behaviour-based system is currently limited to boolean values, which reduces e.g. the description of the activity of a behavioural component to *active* or *not active*. This abstraction is sufficient for many verification tasks, but for example, the special methods to fuse data coming from several behaviours cannot be covered with that approach. The extension of the behavioural signals from the set $\{0, 1\}$ to a more fine-grained interval would blow up the state space using the Uppaal verifier. A combination of the presented reduction techniques that reduce the system to the task specific network parts and an automated choice of the level of detail of behavioural signals could mitigate this problem. Another possibility of facing problems caused by the high abstraction level is to integrate a formal definition of the task a behavioural component has to complete. Such a formal definition could be integrated in the existing formal model and considered during the model checking process. This would allow for checking more sophisticated queries.

The different techniques presented in this thesis build the basis for a comprehensive analysis and formal verification of behaviour-based system. They assist the developer in the understanding of the relationship of system events and the interaction of hardware/software components, the identification of failure causes, and the formal verification of safety requirements. The combination of these methods helps to overcome their weaknesses and enables to profit from their advantages. In Sec. 6.5 several possibilities to combine fault tree analysis with model checking have already been described.

The concept of integrating hardware into the formal model of the behaviour-based system has been shown on the sensor processing chain of a robot. A logical next step is the integration of actuators in order to have a formal model that covers the complete robot system. The presented approach facilitates the integration of new components and the formulation of real-world questions to formal queries.

Bibliography

[0sha06 06] "Report No. 0552652", Occupational Safety & Health Administration, 2006. Washington, DC.

[Akerlund 99] O. Akerlund, S. Nadjm-Tehrani, G. Stålmarck, "Integration of Formal Methods into System Safety and Reliability Analysis", in *Proceedings of 17 th International Systems Safety Conference*. Florida, USA, August 1999, pp. 326–336.

[Al-Zokari 10] Y. I. Al-Zokari, T. Khan, D. Schneider, D. Zeckzer, H. Hagen, "CakES: Cake Metaphor for Analyzing Safety Issues of Embedded Systems", in *Scientific Visualization: Advanced Concepts*, ser. Dagstuhl Follow-Ups, H. Hagen, Ed., vol. 2. Wadern, Germany: Schloss Dagstuhl–Leibniz Center for Informatics, 2010.

[Al-Zokari 12] Y. I. Al-Zokari, Y. Yang, D. Zeckzer, P. Dannenmann, H. Hagen, "Towards Advanced Visualization and Interaction Techniques for Fault Tree Analyses Comparing existing methods and tools", in *PSAM11 - ESREL2012*. 2012.

[Alemzadeh 15] H. Alemzadeh, R. K. Iyer, Z. T. Kalbarczyk, N. Leveson, J. Raman, "Adverse Events in Robotic Surgery: A Retrospective Study of 14 Years of FDA Data", *PLoS ONE*, vol. 11, no. 4, 2015. DOI:10.1371/journal.pone.0151470.

[Allenby 01] K. Allenby, T. Kelly, "Deriving safety requirements using scenarios", in *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*. 2001, pp. 228–235.

[AlTarawneh 14] R. AlTarawneh, J. Bauer, S. R. Humayoun, A. Ebert, P. Liggesmeyer, "ESSAVis++: an interactive 2D*plus*3D visual environment to help engineers in understanding the safety aspects of embedded systems", in *ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. Rome, Italy, June 17–20 2014, pp. 201–204.

[AlTarawneh 15] R. AlTarawneh, S. R. Humayoun, J. Schultz, A. Ebert, P. Liggesmeyer, "LayMan: A Visual Interactive Tool to Support Failure Analysis in Embedded Systems", in *Proceedings of the 2015 European Conference on Software Architecture*. Dubrovnik/-Cavtat, Croatia, September 7–11 2015, pp. 68:1–68:5.

[Alur 93] R. Alur, C. Courcoubetis, D. Dill, "Model-Checking in Dense Real-Time", *Information and Computation*, vol. 104, no. 1, pp. 2–34, May 1993.

[Ames 15] A. D. Ames, P. Tabuada, B. Schürmann, W. Ma, S. Kolathaya, M. Rungger, J. W. Grizzle, "First steps toward formal controller synthesis for bipedal robots", in *Proceedings of the 18th International Conference on Hybrid Systems, Computation and Control, HSCC'15*. Seattle, WA, USA, April 14–16 2015, pp. 209–218.

[Androutsopoulos 13] K. Androutsopoulos, D. Clark, M. Harman, J. Krinke, L. Tratt, “State-Based Model Slicing: A Survey”, *ACM Computing Surveys (CSUR)*, vol. 45, no. 4, pp. 53:1–53:36, 2013.

[Arkin 98] R. Arkin, *Behaviour-Based Robotics*, MIT Press, 1998. ISBN-10: 0-262-01165-4; ISBN-13: 978-0-262-01165-5.

[Armbrust 10] C. Armbrust, T. Braun, T. Föhst, M. Proetzsch, A. Renner, B.-H. Schäfer, K. Berns, “RAVON – The Robust Autonomous Vehicle for Off-road Navigation”, in *Using robots in hazardous environments: Landmine detection, de-mining and other applications*, Y. Baudoin, M. K. Habib, Eds., Woodhead Publishing Limited, 2010, ch. 15. ISBN-10: 1 84569 786 3; ISBN-13: 978 1 84569 786 0.

[Armbrust 11] C. Armbrust, **L. Kiekbusch**, K. Berns, “Using Behaviour Activity Sequences for Motion Generation and Situation Recognition”, in *Proceedings of the 8th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2011)*, J.-L. Ferrier, A. Bernard, O. Gusikhin, K. Madani, Eds., Institute for Systems and Technologies of Information, Control and Communication (INSTICC). Noordwijkerhout, The Netherlands: SciTePress - Science and Technology Publications, July 28–31 2011, pp. 120–127.

[Armbrust 12] C. Armbrust, **L. Kiekbusch**, T. Ropertz, K. Berns, “Verification of Behaviour Networks Using Finite-State Automata”, in *KI 2012: Advances in Artificial Intelligence*, B. Glimm, A. Krüger, Eds. Saarbrücken, Germany: Springer, September 24–27 2012.

[Armbrust 13a] C. Armbrust, **L. Kiekbusch**, T. Ropertz, K. Berns, “Quantitative Aspects of Behaviour Network Verification”, in *Proceedings of the 26th Canadian Conference on Artificial Intelligence*, ser. Lecture Notes in Computer Science (LNCS), O. Zaiane, S. Zilles, Eds., vol. 7884. Regina, Saskatchewan, Canada: Springer, May 28–31 2013.

[Armbrust 13b] C. Armbrust, **L. Kiekbusch**, T. Ropertz, K. Berns, “Tool-Assisted Verification of Behaviour Networks”, in *Proceedings of the 2013 IEEE International Conference on Robotics and Automation (ICRA 2013)*. Karlsruhe, Germany, May 6–10 2013, pp. 1813–1820.

[Armbrust 14a] C. Armbrust, “Design and Verification of Behaviour-Based Systems Realising Task Sequences”, Dissertation, Robotics Research Lab, Department of Computer Science, University of Kaiserslautern, November 26 2014. http://www.dr.hut-verlag.de/978-3-8439-2261-6.html.

[Armbrust 14b] C. Armbrust, G. D. Cubber, K. Berns, “ICARUS - Control Systems for Search and Rescue Robots”, in *Field and Assistive Robotics - Advances in Systems and Algorithms*, K. Berns, S. A. Mehdi, A. Muhammad, Eds., Aachen: Shaker Verlag, 2014, pp. 1–16. ISBN-13: 978-3-8440-2753-2.

[Armbrust 15] C. Armbrust, **L. Kiekbusch**, T. Ropertz, K. Berns, “Soft Robot Control with a Behaviour-Based Architecture”, in *Soft Robotics - Transferring Theory to Application*, A. Verl, A. Albu-Schäffer, O. Brock, A. Raatz, Eds., Springer-Verlag, 2015, ch. 8, pp. 81–91.

[Arnold 13] J. Arnold, R. Alexander, “Testing Autonomous Robot Control Software Using Procedural Content Generation”, *Computer Safety, Reliability, and Security: 32nd International Conference, SAFECOMP*, pp. 33–44, September 2013. DOI: 10.1007/978-3-642-40793-2_4.

[Backström 95] T. Backström, M. Döös, “A comparative study of occupational accidents in industries with advanced manufacturing technology”, *International Journal of Human Factors in Manufacturing*, vol. 5, no. 3, pp. 267–282, 1995.

[Behrmann 04] G. Behrmann, A. David, K. G. Larsen, “A Tutorial on UPPAAL”, in *Formal Methods for the Design of Real-Time Systems*, ser. LNCS, M. Bernardo, F. Corradini, Eds., vol. 3185. Springer Berlin / Heidelberg, 2004, pp. 200–236. ISBN: 978-3-540-23068-7;.

[Behrmann 06] G. Behrmann, A. David, K. G. Larsen, “A Tutorial on UPPAAL 4.0”, November 28 2006. Revised and extended version of [Behrmann 04].

[Bengtsson 98] J. Bengtsson, B. Jonsson, J. Lilius, W. Yi, “Partial order reductions for timed systems”, in *Proceedings of the 9th International Conference on Concurrency Theory.* September 1998.

[Bensalem 09] S. Bensalem, M. Gallien, F. Ingrand, I. Kahloul, T.-H. Nguyen, “Toward a More Dependable Software Architecture for Autonomous Robots”, *Special issue on Software Engineering for Robotics of the IEEE Robotics and Automation Magazine*, vol. 16, no. 1, pp. 67–77, March 2009.

[Bergeretti 85] J.-F. Bergeretti, B. A. Carré, “Information-flow and Data-flow Analysis of While-programs”, *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 37–61, January 1985.

[Berry 00] G. Berry, “The Foundations of Esterel”, in *Proof, Language, and Interaction*, G. Plotkin, C. Stirling, M. Tofte, Eds., Cambridge, MA, USA: MIT Press, 2000, pp. 425–454.

[Bieber 02] P. Bieber, C. Castel, C. Seguin, “Combination of Fault Tree Analysis and Model Checking for Safety Assessment of Complex System”, in *Dependable Computing EDCC-4*, ser. Lecture Notes in Computer Science, A. Bondavalli, P. Thevenod-Fosse, Eds., Springer Berlin Heidelberg, 2002, vol. 2485, pp. 19–31.

[Bluvband 05] Z. Bluvband, R. Polak, P. Grabov, “Bouncing failure analysis (BFA): the unified FTA-FMEA methodology”, in *Reliability and Maintainability Symposium, 2005. Proceedings. Annual.* Jan 2005, pp. 463–467.

[Böhm 10] P. Böhm, T. Gruber, “A Novel HAZOP Study Approach in the RAMS Analysis of a Therapeutic Robot for Disabled Children”, in *Computer Safety, Reliability, and Security, 29th International Conference SAFECOMP.* Vienna, Austria, September, 14–17 2010, pp. 15–27.

[Bouajjani 04] A. Bouajjani, P. Habermehl, T. Vojnar, “Abstract Regular Model Checking”, in *Proceedings of the 16th International Conference on Computer Aided Verification (CAV ’04)*, ser. Lecture Notes in Computer Science, vol. 3114. Boston, MA, USA: Springer-Verlag, July 13–17 2004, pp. 372–386.

[Brooks 91] R. A. Brooks, “New Approaches to Robotics”, Artificial Intelligence Laboratory, MIT, Tech. Rep., 1991.

[Bush 05] D. Bush, “Modelling support for early identification of safety requirements: A preliminary investigation”, *In Fourth International Workshop on Requirements for High Assurance Systems*, August 2005.

[Carlson 12] C. S. Carlson, *Effective FMEAs*, John Wiley & Sons, Inc., 2012, ch. Understanding the Fundamental Definitions and Concepts of FMEAs, pp. 21–55. ISBN = 9781118312575,.

[Cha 12] S. Cha, J. Yoo, “A safety-focused verification using software fault trees”, *Future Generation Computer Systems*, vol. 28, no. 8, pp. 1272–1282, October 2012.

[Chen 12] Y. Chen, C. Ye, B. Liu, R. Kang, “Status of FMECA research and engineering application”, in *Prognostics and System Health Management (PHM)*. May 2012, pp. 1–9. ISSN=2166-563X.

[Clarke 00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, “Counterexample-Guided Abstraction Refinement”, in *Computer Aided Verification: 12th International Conference*, E. A. Emerson, A. P. Sistla, Eds., vol. 1855. Springer Berlin Heidelberg, July 15–19 2000, pp. 154–169.

[Clarke 01] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, “Progress on the state explosion problem in model checking”, in *Informatics*, Springer. 2001, pp. 176–194.

[Clarke 08] E. M. Clarke, “The Birth of Model Checking”, in *25 Years of Model Checking*, ser. Lecture Notes in Computer Science (LNCS), O. Grumberg, H. Veith, Eds., Springer Berlin Heidelberg, 2008, vol. 5000, pp. 1–26.

[Clarke 12] E. M. Clarke, W. Klieber, M. Nováček, P. Zuliani, “Model Checking and the State Explosion Problem”, in *Tools for Practical Software Verification*, ser. Lecture Notes in Computer Science, B. Meyer, M. Nordio, Eds., Springer Berlin Heidelberg, 2012, vol. 7682, pp. 1–30.

[Clarke 99] E. M. Clarke, O. Grumberg, D. A. Peled, *Model Checking*, MIT Press, 1999. ISBN-10: 0-262-03270-8; ISBN-13: 978-0262032704.

[Corke 11] P. Corke, Ed., *Ensuring Correct Behavior - Formal Methods for Hardware and Software Systems*, ser. IEEE Robotics & Automation Magazine, vol. 18, no. 3. September 2011. ISSN 1070-9932, http://www.ieee-ras.org/ram.

[de Assis 16] P. O. A. de Assis, “Improving the Consistency and Completeness of Safety Requirements Specifications”, Dissertation, TU Kaiserslautern, 2016. to be published.

[Denger 08] C. Denger, "SafeSpection–A systematic customization approach for software hazard identification", Dissertation, Technische Universität Kaiserslautern, 2008.

[Dhaussy 11] P. Dhaussy, J.-C. Roger, F. Boniol, "Reducing State Explosion with Context Modeling for Model-Checking", in *HASE*. 2011, pp. 130–137.

[Dhillon 15] B. S. Dhillon, *Robot System Reliability and Safety: A Modern Approach*, CRC Press, 2015. ISBN: 978-1-4987-0644-5.

[Ellson 03] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, G. Woodhull, "Graphviz and dynagraph – static and dynamic graph drawing tools", *Graph Drawing Software*, pp. 127–148, 2003.

[Espiau 95a] B. Espiau, K. Kapellos, M. Jourdan, "Formal Verification in Robotics: Why and How?", in *Robotics Research - The Seventh International Symposium*, G. Giralt, G. Hirzinger, Eds. Springer, 1995, pp. 225–236.

[Espiau 95b] B. Espiau, K. Kapellos, M. Jourdan, D. Simon, "On the Validation of Robotics Control Systems - Part I: High Level Specification and Formal Verification", Unité de recherche INRIA Rhône-Alpes, Rapport de recherche 2719, November 1995.

[Finucane 10] C. Finucane, G. Jing, H. Kress-Gazit, "LTLMoP: Experimenting with Language, Temporal Logic and Robot Control", in *Intelligent Robots and Systems (IROS), IEEE/RSJ International Conference*. October 2010, pp. 1988 – 1993.

[Firesmith 07] D. G. Firesmith, "Engineering Safety and Security Related Requirements for Software Intensive Systems", in *Software Engineering - Companion, 29th International Conference*. May 2007, pp. 169–169.

[Forouher 14] D. Forouher, J. Hartmann, E. Maehle, "Data Flow Analysis in ROS", in *ISR/Robotik 2014; 41st International Symposium on Robotics; Proceedings of*. June 2014, pp. 1–6.

[Fu 15] J. Fu, H. G. Tanner, J. Heinz, "Concurrent Multi-Agent Systems with Temporal Logic Objectives: Game Theoretic Analysis and Planning through Negotiation", *IET Control Theory and Applications*, vol. 9, no. 3, pp. 465–474, February 2015.

[Gansner 00] E. R. Gansner, S. C. North, "An open graph visualization system and its applications to software engineering", *Software Practice and Experience*, vol. 30, no. 11, pp. 1203–1233, 2000.

[Gänßinger 11] Y. Gänßinger. "Extension of Ravon's Simulation Environment and the GUI for System Analysis". Master Project Report (unpublished); supervised bei Lisa Kiekbusch, 2011.

[Griesmayer 05] A. Griesmayer, R. Bloem, M. Hautzendorfer, F. Wotawa, "Formal Verification of Control Software: A Case Study", in *Proceedings of the 18th International Conference on Innovations in Applied Artificial Intelligence*, ser. IEA/AIE'2005. London, UK, UK: Springer-Verlag, 2005, pp. 783–788.

[Guiochet 02] J. Guiochet, A. Vilchis, “Safety analysis of a medical robot for tele-echography”, in *Proceedings of the 2nd IARP IEEE/RAS joint workshop on Technical Challenge for Dependable Robots in Human Environments*. Toulouse, France, 2002, pp. 217–227.

[Guiochet 13] J. Guiochet, Q. A. D. Hoang, M. Kaâniche, D. Powell, “Model-Based Safety Analysis of Human-Robot Interactions: the MIRAS Walking Assistance Robot”, in *International Conference on Rehabilitation Robotics (ICORR)*. Seattle, United States, June 2013, pp. 1–7.

[Guo 10] Z. Guo, D. Zeckzer, P. Liggesmeyer, O. Mackel, “Identification of Security-Safety Requirements for the Outdoor Robot RAVON Using Safety Analysis Techniques”, *Software Engineering Advances, International Conference on*, pp. 508–513, 2010.

[Hatcliff 00] J. Hatcliff, M. B. Dwyer, H. Zheng, “Slicing Software for Model Construction”, *Higher-Order and Symbolic Computation*, vol. 13, no. 4, pp. 315–353, 2000.

[hazop01 01] “British Standard BS: IEC61882:2002 Hazard and operability studies (HA-ZOP studies)- Application Guide”, British Standards Institution, August 2001. ISBN: 0 580 37625 7.

[Hendriks 04] M. Hendriks, G. Behrmann, K. G. Larsen, P. Niebert, F. W. Vaandrager, “Adding Symmetry Reduction to Uppaal”, in *Formal Modeling and Analysis of Timed Systems: First International Workshop, FORMATS 2003*, K. G. Larsen, P. Niebert, Eds. Marseille, France: Springer Berlin Heidelberg, 2004, pp. 46–59. ISBN 978-3-540-40903-8, DOI 10.1007/978-3-540-40903-8_5.

[Henzinger 02] T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre, “Lazy abstraction”, *ACM SIGPLAN Notices*, vol. 37, no. 1, pp. 58–70, 2002.

[Hoang 12] Q. A. D. Hoang, J. Guiochet, D. Powell, M. Kaâniche, “Human-robot interactions: Model-based risk analysis and safety case construction”, *6th European Congress on Embedded Real-Time Software and Systems*, 2012.

[Holzmann 97] G. J. Holzmann, “The Model Checker SPIN”, *IEEE Transactions on Software Engineering*, vol. 23, pp. 279–295, 1997.

[IFR 15] “Industrial Robot Statistics, World Robotics 2015 Industrial Robots”, 2015. `http://www.ifr.org/industrial-robots/statistics/`, visited: July 2016.

[Ingrand 02] F. Ingrand, F. Py, “Online Execution Control Checking for Autonomous Systems”, in *Proceedings of the 7th International Conference on Intelligent Autonomous Systems (IAS-7)*. Marina del Rey, California, USA, March 25–27 2002.

[ISO/IEC-Guide51 99] ISO/IEC-Guide51, *ISO/IEC-Guide51, Safety aspects - Guidelines for Their Inclusion in Standards*, International Organization for Standardization, 1999.

[Johnson 11] B. Johnson, H. Kress-Gazit, “Probabilistic Analysis of Correctness of High-Level Robot Behavior with Sensor Error”, in *Proceedings of Robotics: Science and Systems VII (RSS 2011)*. Los Angeles, California, USA, June 27–30 2011.

[Johnson 13] B. Johnson, H. Kress-Gazit, “Analyzing and revising high-level robot behaviors under actuator error”, in *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*. Nov 2013, pp. 741–748.

[Johnson 15] B. Johnson, H. Kress-Gazit, “Analyzing and Revising Synthesized Controllers for Robots with Sensing and Actuation Errors”, *The International Journal of Robotics Research*, vol. 34, no. 6, pp. 816–832, 2015.

[Juhasz 11] T. Juhasz, S. Dzhantimirov, **L. Kiekbusch**, K. Berns, U. Schmucker, “Verteilte Echtzeitsimulation eines autonomen Fahrzeugsystems”, in *Digitales Engineering und Virtuelle Techniken zum Planen, Testen und Betreiben Technischer Systeme*, Fraunhofer IFF, Magdeburg. 2011, pp. 179–183.

[Juhasz 12] T. Juhasz, D. Schmidt, **L. Kiekbusch**, U. Schmucker, K. Berns, “Modellierung und verteilte Simulation eines autonomen Mobilbaggers”, *Tagungsband Fachtagung Digitales Engineering zum Planen, Testen und Betreiben Technischer Systeme*, pp. 123–130, June 26–28 2012.

[Kaiser 03] B. Kaiser, P. Liggesmeyer, O. Mäckel, “A New Component Concept for Fault Trees”, in *Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software* -, ser. SCS ’03, vol. 33. Darlinghurst, Australia: Australian Computer Society, Inc., 2003, pp. 37–46.

[Kazanzides 08] P. Kazanzides, G. Fichtinger, G. D. Hager, A. M. Okamura, L. L. Whitcomb, R. H. Taylor, “Surgical and Interventional Robotics - Core Concepts, Technology, and Design”, *Robotics & Automation Magazine, IEEE*, vol. 15, no. 2, pp. 122–130, June 2008. DOI: 10.1109/MRA.2008.926390.

[Khodabandehloo 96] K. Khodabandehloo, “Analyses of robot systems using fault and event trees: case studies”, *Reliability Engineering & System Safety*, vol. 53, no. 3, pp. 247 – 264, 1996. Safety of Robotic Systems.

[Kiekbusch 14] **L. Kiekbusch**, C. Armbrust, K. Berns, “Formal Verification of Behaviour Networks Including Hardware Failures”, in *Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13)*. Padova, Italy, July 15–19 2014.

[Kiekbusch 15] **L. Kiekbusch**, C. Armbrust, K. Berns, “Formal Verification of Behaviour Networks Including Sensor Failures”, *Robotics and Autonomous Systems*, vol. 74, Part B, pp. 331–339, December 2015. DOI:10.1016/j.robot.2015.08.002.

[Kim 05] M. Kim, K. C. Kang, “Formal Construction and Verification of Home Service Robots: A Case Study”, in *ATVA*. 2005, pp. 429–443.

[Kress-Gazit 11] H. Kress-Gazit, T. Wongpiromsarn, U. Topcu, “Correct, Reactive, High-Level Robot Control”, *IEEE Robotics & Automation Magazine*, vol. 18, no. 3, pp. 65 –74, 2011.

[Lahijanian 10] M. Lahijanian, J. Wasniewski, S. B. Andersson, C. Belta, “Motion Planning and Control from Temporal Logic Specifications with Probabilistic Satisfaction Guarantees”, in *Proceedings of the 2010 IEEE International Conference on Robotics and Automation (ICRA 2010)*. Anchorage, Alaska, USA, May 3–8 2010, pp. 3227–3232.

[Lamine 02] K. B. Lamine, F. Kabanza, “Reasoning about Robot Actions: A Model Checking Approach.”, in *Advances in Plan-Based Control of Robotic Agents*, ser. Lecture Notes in Computer Science, M. Beetz, J. Hertzberg, M. Ghallab, M. E. Pollack, Eds., vol. 2466. Springer, 2002, pp. 123–139.

[Lari 15] A. Lari, F. Douma, I. Onyiah, “Self-Driving Vehicles and Policy Implications: Current Status of Autonomous Vehicle Development and Minnesota Policy Implications”, *Minnesota Journal of Law, Science & Technologie*, vol. 16, no. 2, pp. 735–769, June 2015. ISSN: 1552-9541.

[Laval 13] J. Laval, L. Fabresse, N. Bouraqadi, “A methodology for testing mobile autonomous robots”, in *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on.* November 2013, pp. 1842–1847.

[Leveson 83] N. G. Leveson, P. R. Harvey, “Analyzing Software Safety”, *IEEE Transactions on Software Engineering*, vol. 9, no. 5, pp. 569–579, September 1983.

[Liggesmeyer 09] P. Liggesmeyer, *Software-Qualität*, Second ed., Heidelberg, Germany: Spektrum-Verlag, 2009.

[Lowry 97] M. Lowry, K. Havelund, J. Penix, “Verification and Validation of AI Systems that Control Deep-Space Spacecraft”, in *Foundations of Intelligent Systems*, ser. LNCS, Z. Ras, A. Skowron, Eds., vol. 1325. Springer Berlin / Heidelberg, 1997, pp. 35–47. ISBN: 978-3-540-63614-4.

[Lyons 15a] D. M. Lyons, R. C. Arkin, S. Jiang, D. Harrington, F. Tang, P. Tang, “Probabilistic Verification of Multi-robot Missions in Uncertain Environments”, in *Tools with Artificial Intelligence (ICTAI), 2015 IEEE 27th International Conference on.* Nov 2015, pp. 56–63.

[Lyons 15b] D. M. Lyons, R. C. Arkin, S. Jiang, T.-M. Liu, P. Nirmal, “Performance Verification for Behavior-Based Robot Missions”, *IEEE Transactions on Robotics*, vol. 31, no. 3, pp. 619–636, June 2015.

[Lyu 96] M. R. Lyu, *Handbook of Software Reliability Engineering*, M. R. Lyu, Ed., Hightstown, NJ, USA: McGraw-Hill, Inc., 1996.

[Maas 14] R. Maas, E. Maehle, “Health Signal Generation in the ORCA (Organic Robot Control Architecture) Framework”, in *44. Jahrestagung der Gesellschaft für Informatik, Informatik 2014, Big Data - Komplexität meistern.* Stuttgart, Germany, 22.–26. September 2014, pp. 1327–1338.

[Martin-Guillerez 10] D. Martin-Guillerez, J. Guiochet, D. Powell, C. Zanon, “A UML-based Method for Risk Analysis of Human-robot Interactions”, in *Proceedings of the 2Nd International Workshop on Software Engineering for Resilient Systems*, ser. SERENE ’10. New York, NY, USA: ACM, 2010, pp. 32–41.

[Matarić 92] M. J. Matarić, “Behavior-Based Systems: Main Properties and Implications”, in *Proceedings of the IEEE International Conference on Robotics and Automation*, IEEE. Nice, France, May 1992, pp. 46–54. from the Workshop on Architectures for Intelligent Control Systems.

[Matsubara 12] M. Matsubara, K. Sakurai, F. Narisawa, M. Enshoiwa, Y. Yamane, H. Yamanaka, "Model Checking with Program Slicing Based on Variable Dependence Graphs", *FTSCS*, pp. 56–68, 2012.

[McDermid 94] J. A. McDermid, D. J. Pumfrey, "A development of hazard analysis to aid software design", in *Computer Assurance, 1994. COMPASS '94 Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security. Proceedings of the Ninth Annual Conference on.* June 1994, pp. 17–25.

[McDermid 95] J. A. McDermid, M. Nicholson, D. J. Pumfrey, P. Fenelon, "Experience with the application of HAZOP to computer-based systems", in *Computer Assurance, 1995. COMPASS '95. Systems Integrity, Software Safety and Process Security. Proceedings of the Tenth Annual Conference on.* June 1995, pp. 37–48.

[Nazier 12] R. Nazier, T. Bauer, "Automated Risk-Based Testing by Integrating Safety Analysis Information into System Behavior Models.", in *ISSRE Workshops.* IEEE, 2012, pp. 213–218.

[Onofrio 15] R. Onofrio, F. Piccagli, F. Segato, "Failure Mode, Effects and Criticality Analysis (FMECA) for Medical Devices: Does Standardization Foster Improvements in the Practice?", *6th International Conference on Applied Human Factors and Ergonomics (AHFE 2015) and the Affiliated Conferences, AHFE 2015*, vol. 3, pp. 43–50, 2015.

[Ottenstein 84] K. J. Ottenstein, L. M. Ottenstein, "The Program Dependence Graph in a Software Development Environment", *SIGPLAN Not.*, vol. 19, no. 5, pp. 177–184, April 1984.

[Pelánek 08] R. Pelánek, "Fighting state space explosion: Review and evaluation", in *Proceedings of Formal Methods for Industrial Critical Systems (FMICS).* 2008.

[Proetzsch 07] M. Proetzsch, K. Berns, T. Schuele, K. Schneider, "Formal Verification of Safety Behaviours of the Outdoor Robot RAVON", in *Proceedings of the Fourth International Conference on Informatics in Control, Automation and Robotics (ICINCO 2007)*, J. Zaytoon, J.-L. Ferrier, J. Andrade-Cetto, J. Filipe, Eds. Angers, France: INSTICC Press, May 9–12 2007, pp. 157–164.

[Proetzsch 10a] M. Proetzsch, *Development Process for Complex Behavior-Based Robot Control Systems*, ser. RRLab Dissertations, Verlag Dr. Hut, 2010. This publication is available at `http://www.dr.hut-verlag.de/978-3-86853-626-3.html`.

[Proetzsch 10b] M. Proetzsch, F. Zimmermann, R. Eschbach, J. Kloos, K. Berns, "A Systematic Testing Approach for Autonomous Mobile Robots Using Domain-Specific Languages", in *KI 2010: Advances in Artificial Intelligence*, ser. Lecture Notes in Computer Science, R. Dillmann, J. Beyerer, U. Hanebeck, T. Schultz, Eds., vol. 6359. Springer Berlin / Heidelberg, 2010, pp. 317–324.

[Rafe 13] V. Rafe, M. Rahmani, K. Rashidi, "A Survey on Coping with the State Space Explosion Problem in Model Checking", *International Research Journal of Applied and Basic Sciences*, vol. 4, no. 6, pp. 1379–1384, 2013. ISSN 2251-838X.

[Reichardt 09] M. Reichardt, **L. Wilhelm**, M. Proetzsch, K. Berns, “Applications of Visualization Technology in Robotics Software Development”, in *4th Human Computer Interaction and Visualization (HCIV) Workshop 2009*. Kaiserslautern, Germany, March 2 2009.

[Reichardt 13] M. Reichardt, T. Föhst, K. Berns, “On Software Quality-motivated Design of a Real-time Framework for Complex Robot Control Systems”, *Electronic Communications of the EASST*, vol. 60: Software Quality and Maintainability 2013, August 2013. This publication is available at `http://journal.ub.tu-berlin.de/eceasst/article/view/855`.

[Rohr 12] S. Rohr, “Visualisation of Traces Originating from Model Checking”, Robotics Research Lab, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, thesis of master’s project, 2012. unpublished; supervised by Christopher Armbrust.

[Ropertz 12] T. Ropertz, “Graphical Support for the Verification of Behaviour Networks”, Master’s thesis, Robotics Research Lab, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2012. unpublished; supervised by Christopher Armbrust and Lisa Kiekbusch.

[Saïdi 00] H. Saïdi, “Model Checking Guided Abstraction and Analysis”, in *Proceedings of the 7th International Symposium on Static Analysis*, ser. SAS ’00. London, UK, UK: Springer-Verlag, 2000, pp. 377–396.

[Saini 12] D. K. Saini, “Software Testing for Embedded Systems”, *International Journal of Computer Applications*, vol. 43, no. 17, April 2012.

[Santiago 05] I. B. Santiago, J.-M. Faure, “From Fault Tree Analysis to Model Checking of Logic Controllers”, in *16th IFAC World Congress*. Praha, Czech Republic, July 2005.

[Schäfer 03] A. Schäfer, “Combining Real-Time Model-Checking and Fault Tree Analysis”, in *FME 2003: Formal Methods*, ser. LNCS, K. Araki, S. Gnesi, D. Mandrioli, Eds., vol. 2805. Springer Berlin / Heidelberg, 2003, pp. 522–541. ISBN: 978-3-540-40828-4; this publication is available at `http://dx.doi.org/10.1007/978-3-540-45236-2_29`.

[Schäfer 11] B.-H. Schäfer, *Control System Design Schemata and their Application in Off-road Robotics*, ser. RRLab Dissertations, Verlag Dr. Hut, 2011. `http://www.dr.hut-verlag.de/978-3-86853-865-6.html`.

[Scherer 05] S. Scherer, F. Lerda, E. M. Clarke, “Model Checking of Robotic Control Systems”, in *Proceedings of the 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS 2005)*. Munich, Germany, September 5–8 2005.

[Schneider 05] K. Schneider, T. Schuele, “Averest: Specification, Verification, and Implementation of Reactive Systems”, in *Proceedings of the Fifth International Conference on Application of Concurrency to System Design (ACSD 2005)*, J. Desel, Y. Watanabe, Eds. Saint-Malo, France: IEEE Computer Society, June 7–9 2005.

[Schneider 09] D. Schneider, M. Trapp, “Runtime Safety Models in open Systems of Systems”, in *Proceedings of UbiSafe 2009*. 2009.

[Scholl 01] K.-U. Scholl, J. Albiez, B. Gassmann, “MCA - An Expandable Modular Controller Architecture”, in *Proceedings of the 3rd Real-Time Linux Workshop*. Milano, Italy, November 26–29 2001.

[Seward 00] D. Seward, C. Pace, R. Morrey, I. Sommerville, “Safety analysis of autonomous excavator functionality”, *Reliability Engineering & System Safety*, vol. 70, no. 1, pp. 29–39, 2000.

[Silva 12] J. Silva, “A Vocabulary of Program Slicing-Based Techniques”, *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, pp. 12:1–12:41, 2012.

[Slomp 10] G. H. Slomp, “Reducing UPPAAL models through control flow analysis”, September 2010. Master’s thesis at University of Twente, The Netherlands.

[Steiner 12] M. Steiner, P. Keller, P. Liggesmeyer, “Modeling the Effects of Software on Safety and Reliability in Complex Embedded Systems”, in *Computer Safety, Reliability, and Security*, F. Ortmeier, P. Daniel, Eds., Springer Berlin Heidelberg, 2012, vol. 7613, pp. 454–465. Proceedings of the 3rd International Workshop in Digital Engineering.

[Tatar 14] M. Tatar, J. Mauss, “Systematic Test and Validation of Complex Embedded Systems”, *Embedded Real Time Software and Systems (ERTS 2014)*, February 5–7 2014.

[Thrane 08] C. Thrane, U. Sørensen, K. G. Larsen, “Slicing for UPPAAL”, *NWPT 2008*, p. 100, 2008.

[Thums 03] A. Thums, G. Schellhorn, “Model checking FTA”, in *FME 2003: Formal Methods*, ser. Lecture Notes in Computer Science, K. Araki, S. Gnesi, D. Mandrioli, Eds., vol. 2805. Springer Berlin Heidelberg, 2003, pp. 739–757. ISBN: 978-3-540-40828-4.

[Thums 04] A. Thums, “Formale Fehlerbaumanalyse”, Dissertation, Universität Augsburg, Augsburg, Dezember 2004.

[Tip 95] F. Tip, “A Survey of Program Slicing Techniques”, *Journal of Programming Languages*, vol. 3, no. 3, pp. 121–189, 1995.

[Ulusoy 13] A. Ulusoy, S. L. Smith, X. C. Ding, C. Belta, D. Rus, “Optimality and Robustness in Multi-Robot Path Planning with Temporal Logic Constraints.”, *The International Journal of Robotics Research (IJRR)*, vol. 32, no. 8, pp. 889–911, 2013.

[van de Molengraft 11] R. van de Molengraft, M. Beetz, T. Fukuda, “Robot Challenges: Toward Development of Verification and Synthesis Techniques”, *IEEE Robotics & Automation Magazine*, vol. 18, no. 3, pp. 22–23, September 2011.

[Vesely 81] W. E. Vesely, F. F. Goldberg, N. H. Roberts, D. F. Haasl, *Fault Tree Handbook*, Washington, DC: U.S. Nuclear Regulatory Commission, 1981. ISBN: 978-1500178208.

[Visser 03] W. Visser, K. Havelund, G. Brat, S. Park, F. Lerda, “Model Checking Programs”, *Automated Software Engg.*, vol. 10, no. 2, pp. 203–232, Apri 2003.

[Vonwirth 15] P. M. Vonwirth, “Graphical Interface for Query Generation in Behaviour-Based Networks Covering Environmental Influences”, Bachelor’s thesis, Robotics Research Lab, University of Kaiserslautern, February 2015. unpublished; supervised by Lisa Kiekbusch.

[Walker 96] I. D. Walker, J. R. Cavallaro, “Failure mode analysis for a hazardous waste clean-up manipulator”, *Reliability Engineering & System Safety*, vol. 53, no. 3, pp. 277 – 290, 1996. Safety of Robotic Systems.

[Webster 11] M. Webster, M. Fisher, N. Cameron, M. Jump, “Model Checking and the Certification of Autonomous Unmanned Aircraft Systems”, University of Liverpool Department of Computer Science, Tech. Rep. ULCS-11-001, 2011.

[Webster 14] M. Webster, C. Dixon, M. Fisher, J. Salem, K. Koay, K. Dautenhahn, “Formal Verification of an Autonomous Personal Robotic Assistant”, in *Formal Verification and Modeling in Human-Machine Systems*. The AAAI Press, March 2014, pp. 74–79. ISBN-13: 978-1-57735-655-4.

[Weiser 81] M. Weiser, “Program Slicing”, in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE ’81. Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449.

[Weiser 84] M. Weiser, “Program Slicing”, *IEEE Trans. Software Eng.*, vol. 10, no. 4, pp. 352–357, 1984.

[Weißmann 11] M. Weißmann, S. Bedenk, A. Buckl, “Model Checking Industrial Robot Systems”, in *Model Checking Software*, Springer, 2011, pp. 161–176.

[Wilhelm 08] **L. Wilhelm**, “Oscillation Detection in Behaviour-Based Robot Architectures”, Diploma thesis, Robotics Research Lab, Department of Computer Sciences, University of Kaiserslautern, November 2008. unpublished.

[Wilhelm 09] **L. Wilhelm**, M. Proetzsch, K. Berns, “Oscillation Analysis in Behavior-Based Robot Architectures”, in *Autonome Mobile Systeme*, ser. Informatik aktuell, R. Dillmann, J. Beyerer, C. Stiller, J. M. Zöllner, T. Gindele, Eds. Springer, 2009, pp. 121–128.

[Woodman 12] R. Woodman, A. F. T. Winfield, C. Harper, M. Fraser, “Building safer robots: Safety driven control”, *International Journal of Robotics Research*, vol. 31, no. 13, pp. 1603–1626, 2012.

[Woodman 13] R. Woodman, “Novel approaches for the safety of human-robot interaction”, Dissertation, University of the West of England, 2013.

[Xu 05] B. Xu, J. Qian, X. Zhang, Z. Wu, L. Chen, “A Brief Survey of Program Slicing”, *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 2, pp. 1–36, March 2005.

[Yatapanage 10] N. Yatapanage, K. Winter, S. Zafar, “Slicing behavior tree models for verification”, in *Theoretical Computer Science*, Springer, 2010, pp. 125–139.

[Yatapanage 12] N. Yatapanage, “Slicing Behavior Trees for Verification of Large Systems”, Dissertation, Griffith University, 2012.

[Zeller 14] M. Zeller, K. Höfig, M. Rothfelder, *Computer Safety, Reliability, and Security*, Florence, Italy: Springer International Publishing, September 2014, ch. Towards a Cross-Domain Software Safety Assurance Process for Embedded Systems, pp. 396–400. ISBN: 978-3-319-10557-4.

[Zhang 09] Q. Zhang, Y.-F. Zhang, S.-Y. Qin, “Modeling and Analysis for Obstacle Avoidance of a Behavior-Based Robot with Objected Oriented Methods”, *Journal of Computers*, vol. 4, no. 4, 2009.

Index

Lisa Kiekbusch

Curriculum vitae

DOCTORAL STUDIES

01/2009 -11/2016	**PhD student, Robotics Research Lab, University of Kaiserslautern**
Thesis	**Analysis and Verification of Complex Robot Systems using Behaviour-Based Control**
01/2009 – 01/2016	Researcher, Robotics Research Lab, University of Kaiserslautern
06/2010 – 06/2014	Scholarship from Carl-Zeiss-Stiftung

EDUCATION

Diploma Thesis	Oscillation Detection in Behaviour-Based Robot Architectures Robotics Research Lab, University of Kaiserslautern
10/2002 – 11/2008	Student of technical computer science, University of Kaiserslautern