An Industrial Case Study on the Evaluation of a Safety Engineering Approach for Software-Intensive Systems in the Automotive Domain

Asim Abdulkhaleq[a,*], Sebastian Vöst[a,b], Stefan Wagner[a], John Thomas[c]

[a] *Institute of Software Technology, University of Stuttgart, Germany*
[b] *BMW Group, Munich, Germany*
[c] *MIT, Cambridge, MA, U.S.A*

Abstract

Safety remains one of the essential and vital aspects in today's automotive systems. These systems, however, become ever more complex and dependent on software which is responsible for most of their critical functions. Therefore, the software components need to be analysed and verified appropriately in the context of software safety. The complexity of software systems makes defining software safety requirements with traditional safety analysis techniques difficult. A new technique called STPA (Systems-Theoretic Process Analysis) based on system and control theory has been developed by Leveson to cope with complex systems. Based on STPA, we have developed a comprehensive software safety engineering approach in which the software and safety engineers integrate the analysis of software risks with their verification to recognize the software-related hazards and reduce the risks to a low level. In this paper, we explore and evaluate the application of our approach to a real industrial system in the automotive domain. The case study was conducted analysing the software controller of the Active Cruise Control System (ACC) of the BMW Group.

Keywords: STAMP, STPA, safety analysis, automotive software system, verification

1. Introduction

Safety is an important property of today's complex systems. Modern systems have achieved greater capabilities through growing reliance on increasingly capable software. Software has become a crucial part of modern safety-critical systems, and the amount of software in such systems is increasing. For example, a modern BMW 7 series car has something close to 100 million lines of software in it and runs on more than 100 microprocessors [1]. Modern vehicles

*Corresponding author, Asim Abdulkhaleq
Email. Asim Abdulkhaleq@informatik.uni-stuttgart.de

come with various complex systems (e.g. electronic stability control, navigation systems) which rely on software to control the main functions. An unforeseen behavior of software may result in catastrophic consequences such as injury or loss of human life, damaged property or environmental disturbances. Over the last ten years, the number of accidents and losses related to software flaws has increased. The Toyota Prius and the General Motor airbag are two commonly known software problems that occurred in the automotive domain in the last five years. Recently, Google's self-driving car and the Tesla autopilot are the two latest software-related accidents in the automotive domain.

Software safety is a crucial aspect during the development of modern safety-critical systems. However, safety is a system-level property; therefore, software safety must be considered at the system level to ensure the whole system's safety [2]. Furthermore, the software design and implementation must be verified against the software safety requirements which are identified during safety analysis at the system level. Software components in modern software-intensive systems and their interactions with other components, and especially their complexity, become a challenge in ensuring the safety of the whole system. In particular, the safety analysis and verification of each of these software components at the system level are demanding. For this issue, Abdulkhaleq, Wagner and Leveson [3] proposed a comprehensive approach to safety engineering of software-intensive systems based on STPA called *STPA SwISs* (STPA for Software-Intensive Systems), which combines the STPA safety analysis activities with the verification and testing activities. The *STPA SwISs* approach offers seamless safety analysis and verification activities to help the software and safety engineers to recognize the associated software risks at the system level.

To support the safety engineering process based on STPA, we developed an extensible platform called XSTAMPP [4]. XSTAMPP[1] is an open-source platform written in Java based on the Eclipse Plug-in-Development Environment (PDE) and Rich Client Platform (RCP). XSTAMPP supports performing the main steps of STPA and automatically transforms the STPA-generated safety requirements into formal specifications in LTL (Linear Temporal Logic) [5].

1.1. Problem Statement

STPA is a rather new technique, however. Little is yet known about its difficulties and benefits in the context of software safety, especially in the automotive domain. Moreover, the *STPA SwISs* safety engineering approach is based on STPA and the *STPA SwISs* approach has not yet been evaluated with a real industrial software system.

1.2. Research Objectives

The overall objective of this research is to explore the application of the *STPA SwISs* approach for safety engineering based on STPA using a real industrial system in the automotive domain. This objective includes deriving

[1]`http://www.xstampp.de`

appropriate software safety requirements at the system level and generate safety-based test cases directly from the information derived during the STPA safety analysis to recognize the software risks.

1.3. Contribution

We contribute an investigation of the application of STPA SwISs in terms of deriving the software safety requirements at the system level by *STPA SwISs* and deriving safety-based test cases for each STPA-generated software safety requirement. We started by applying STPA to the system specifications of ACC stop-and-go to derive the software safety requirements at the system level. We used XSTAMPP [4] to document the results of applying STPA and transformed the STPA safety requirements automatically into formal specifications in LTL. Based on the results of STPA, we constructed a safe behavioral model of the ACC as a Simulink statechart. To ensure the correctness of the resulting model and STPA results, both are reviewed by two BMW experts. Furthermore, we automatically converted the safe behavioral model into SMV (Symbolic Model Verifier) [6] by using our tool *STPA TCGenerator* (STPA Test Case Generator)[2] which is a model-based safety testing tool. We also verified the SMV model against the STPA results by using the NuSMV model checker [7]. We used the safe behavioral model as input to *STPA TCGenerator* to generate safety-based test cases. Finally, we selected 20 of the generated safety-based test cases to be executed on the ACC system.

1.4. Context

We conducted the case study at the German company BMW Group, which is a luxury automobile and motorcycle company. We applied the *STPA SwISs* approach to the BMW active cruise control system with stop-and-go function of the new car model. The case study was performed at the headquarters of the BMW Group in Munich, Germany.

1.5. Outline

The remainder of this paper is structured as follows: The background of the safety analysis, software verification and our previous work on the comprehensive approach for safety engineering for software-intensive systems are presented in section 2. Related work is discussed in section 3. The case study design and how to apply the *STPA SwISs* approach to the active cruise control system with stop-and-go are presented in Section 4. The results are presented in section 5. Finally, conclusions and future work are provided in section 6.

1.6. Terminology

We define the most relevant terms in Tab. 1 to ensure a consistent terminology in this paper.

[2]`https://sourceforge.net/projects/stpastgenerator/`

Table 1: Terminology

Terminology	Definition
Accident	Accident (Loss) results from inadequate enforcement of the behavioral safety constraints on the process [2].
Hazard	Hazard is a system state or set of conditions that, together with a particular set of worst-case environmental conditions, will lead to an accident [2].
Unsafe Control Actions	The hazardous scenarios which might occur in the system due to provided or not provided control action when required [2].
Safety Constraints	The safety constraints are the safeguards which prevent the system from leading to losses (accidents) [2].
Process model variables	The process model variables are the safety-critical variables of the controller in the control structure diagram (e.g. internal variables, internal states, interaction and environmental variables) which have an effect on the safety of issuing the control actions [2] [8].
Causal Factors	Causal factors are the accident scenarios that explain how unsafe control actions might occur and how safe control actions might not be followed or executed [2] [8].
Safety-based Test Cases	The safety-based test cases are set of the test cases which are automatically generated from information derived during the STPA safety analysis process.

2. Background

2.1. STPA Safety Analysis

Developing a safe software demands that the development process of software shall take into account the complexity of software components; especially on how to define appropriate software safety requirements and how to verify the software design and implementation. The complexity of software makes defining appropriate software safety requirements difficult by traditional safety analysis techniques such as Fault Tree Analysis (FTA) [9] and Failure Mode and Effect Criticality Analysis (FMECA) [10]. Rather than focusing on creating software safety requirements, most traditional techniques focus on failures and analyze an existing design with some or all of the requirements already defined. Previous

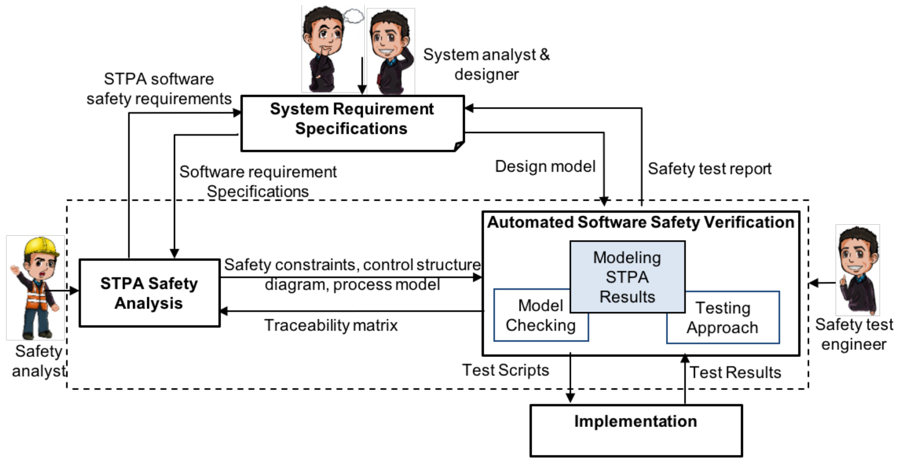

System analyst & designer
STPA software safety requirements
System Requirement Specifications
Software requirement Specifications
Design model
Safety test report
Automated Software Safety Verification
Modeling STPA Results
Model Checking
Testing Approach
STPA Safety Analysis
Safety constraints, control structure diagram, process model
Traceability matrix
Safety analyst
Safety test engineer
Test Scripts
Test Results
Implementation

Figure 1: STPA SwISs: A comprehensive safety engineering approach for software-intensive systems

work has identified a number of limitations of traditional automotive safety analysis techniques when it comes to complex software behavior [11] in terms of identifying unsafe interactions between systems, anticipating human error and other behaviors dependent on human interaction, identifying design flaws, and producing requirements To overcome these limitations, a new safety analysis approach called STPA [2] has been developed. STPA is a unique safety analysis created for complex systems, including software-intensive systems. The STPA safety analysis process is carried out in three major steps: (1) Establish the fundamentals of the analysis (e.g. system description, system-level accidents, system-level hazards, safety and design requirements) and draw a high-level safety-control structure diagram in which the system is viewed as interacting components; (2) identify the potentially unsafe control actions of the system that could lead to one or more system-level; and (3) Identify accident scenarios that explain how unsafe control actions might occur and how safe control actions might not be followed or executed.

2.2. STPA SwISs: A Comprehensive Safety Engineering Approach for Software

Software is an integral part of the system and it must deal with the hazards identified during safety analysis at the system level to ensure the whole system's safety. Moreover, the software implementation and design must also fulfill the software safety requirements. STPA is a new technique, however. It has not yet been placed in the software development process of safety-critical systems and used for software verification. Traditionally, formal verification and testing are complementary approaches which are used in the development process to verify the functional correctness of software. To address the aforementioned challenges of software safety, Abdulkhaleq, Wagner and Leveson [3] developed a comprehensive safety engineering approach based on STPA for software-intensive systems. The approach offers seamless safety analysis and verification activities. This will allow the software and safety engineers to work together to derive the software safety requirements at the system level, verify them and generate safety-based test cases. Instead of the exhaustive testing of all software component behavioral, which is not possible, the approach focuses on identifying appropriate software safety requirements that must be tested and/or verified to ensure that the software components satisfy these requirements. Figure 1 shows the proposed approach for safety engineering based on STPA. To help in identifying unsafe control action and safety requirements, context tables were used as proposed by Thomas [12].

The STPA SwISs approach as described in [3] is carried out in the following major steps: **Step 1**: Deriving the software safety requirements at the system level by applying STPA to the system specification and requirements. **Step 2**: Modeling STPA results with a safe behavioral model. A safe behavioral model is a UML statechart notation constrained by the STPA results. **Step 3**: Verifying the safe behavioral model against the STPA results by using model checking to ensure that the safe behavioral model satisfies the STPA-generated safety requirements. **Step 4**: Generating and executing safety-based test cases. This step is divided into two steps: *(4.1)* Using the safe behavioral model as an input to the model-based testing tool to generate safety-based test cases (e.g. STPA TCGenerator); and *(4.2)* Executing the safety-based test cases on the system under analysis and generating the safety verification report. This report shows the results of software safety verification activities of each software safety requirement derived by STPA, the list of the generated safety-based test cases, the test coverage measure, the traceability metric between the STPA results and the generated safety-based test cases.

The output of the safety engineering approach is a safety report which the engineers will use to modify the software design and implementation to ensure the safe operation of the software.

3. Related Work

In this section, we will describe our primary work as follows: Abdulkhaleq and Wagner [13] applied STPA to a well-known example of a safety-critical system in the automotive domain: adaptive cruise control (ACC). ACC is an automotive feature that allows a vehicle's cruise control system to adapt the vehicle's speed to the traffic environment. This case study was performed based on an existing case study with MAN Truck & Bus AG [14] in which the authors conducted an exploratory case study applying safety cases for the ACC system. We compared the results of STAMP/STPA with the safety cases on the same system.

Thomas [15] introduced an extended approach to STPA with the purpose of identifying unsafe control actions in STPA Step 1 based on the combinations of process model variables of each controller in the control structure diagram. A combination of process model variables is called a context. Two contexts of control actions are proposed: *Provided control action and not provided control* *action.* The control action will be hazardous only in a certain context. The main problem of context tables is the difficulty in defining the combination for a large number of values of the process model variables which have an effect on the safety of control actions. To solve this problem, Abdulkhaleq and Wagner [16] developed an algorithm based on the concept of combinatorial testing [17] to automatically generate the context tables and to allow safety analysts to identify a minimal combination of process model variables. The safety analysts can add and apply constraints and Boolean relations to the generated context tables to ignore some unnecessary combinations from these tables. Furthermore, they explain how to automatically generate the hazardous rules and refined unsafe control actions based on the results of the context table. The hazardous rules will be automatically translated into the refined safety constrains and expressed them into formal specifications in LTL. Both algorithms are implemented as an Eclipse plug-in called XSTPA[3] (Extended Approach to STPA) which integrates with the XSTAMPP platform.

In [18], Abdulkhaleq and Wagner proposed a safety verification methodology based on the STPA safety analysis. We applied STPA to vehicle cruise control software to identify the software safety requirements at the system level and verify these safety requirements at the design level. Abdulkhaleq and Wagner [16] also extended the software safety verification approach by investigating the possibility of verifying the software safety requirements based on the model extracted directly from the source code of the software. They integrated the STPA safety analysis with a software model checker.

Recently, Abdulkhaleq and Wagner developed a safety-based test case generator tool called *STPATCGenerator* [4]which generates test cases based on the results of STPA safety analysis results which are modeled in a Simulink stateflow. Furthermore, they developed an Eclipse plug-in called STPA verifier [5] to verify the STPA safety requirements with model checking tools such as SPIN [19] and NuSMV [7] in XSTAMPP. The STPA-generated safety requirements in XSTAMPP are automatically transformed into formal specifications in LTL (linear Temporal Logic).

4. Case Study Design

In the following, we will describe the case study design which contains the main research questions that drive the case study, data collection and analysis procedures, as well as how we ensure the validity of the results. Our case study design follows Runeson and Höst's guidelines [20].

4.1. Study Goal and Research Questions

The goal of this case study is to explore the applicability and feasibility of the *STPA SwISs* approach of software safety engineering based on STPA in a

[3]`http://www.xstampp.de/xstpa.html`
[4]`http://www.xstampp.de/STPATCGenerator.html`
[5]`http://www.xstampp.de/STPAVerifier.html`

real industrial environment. We use two research questions to structure the study design.

RQ1) How effective is using the *STPA SwISs* approach to derive the software safety requirements at the system level? This research question focuses on investigating how *STPA SwISs* helps to derive the appropriate software safety requirements at the system level to help the software and safety engineers to recognize the software risks.

RQ2) How useful is generating the safety-based test cases from the STPA results? We want by this research question to investigate how the safety-based test cases generated from STPA results can help to test the system against each software safety requirement to ensure that the system satisfies the STPA software safety requirements.

4.2. Case and Subjects Selection

We select one case which is representative of an automotive system and with which the subjects have experience. The subjects in this case study were all authors and 1 internal ACC system expert at BMW who contribute in different roles. Two of the authors contribute in performing the case study at the location of BMW. Asim Abdulkhaleq works in the role of STPA safety analysis and safety engineering expert. Sebastian Vöst acts as the automotive system requirements and testing expert and assumes the role of system analyst and tester. The internal ACC system architecture expert takes the role of ACC system designer. Stefan Wagner and John Thomas act as the software and safety engineering experts. They assume the roles of supporting the case study and review the results at different stages.

We chose the active cruise control system with stop-and-go function as a case study object because it is a software-intensive system with strong safety implications.

4.3. Data Collection Procedures

The case study follows the 4 steps of the *STPA SwISs* approach to derive the software safety requirements and generate and execute the safety-based test cases on the system environment. Specifically, we apply the approach to the specification document of the active cruise control system with stop-and-go function. We use the XSTAMPP software tool to document the safety analysis results which are exported as PDF files and Excel sheets. We also use the reports generated by formal verification and testing approaches (e.g. STPA verifier and STPA TC Generator). These reports include the verification results of each STPA safety requirement, the generated safety-based test cases and the test execution results.

4.4. Analysis Procedure

The following section describes the top-down process for applying the *STPA SwISs* safety engineering approach to the software controller of the active cruise control system in this case study (shown in Fig. 2):

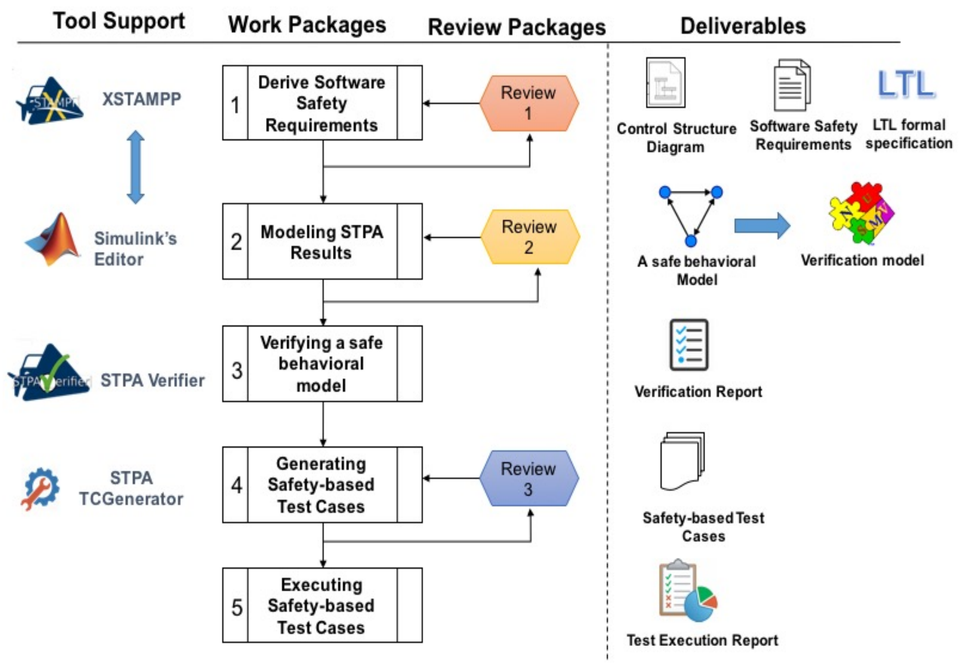


Figure 2: Case study work packages, deliverables and tool support

Deriving the Software Safety Requirements. At first, the safety analyst investigates the existing documents about the case study object, its functional requirements, and its system specification. From these documents, the safety analyst establishes the fundamentals of the analysis (e.g. a list of the system-level accidents that the software can contribute to, a list of the system-level hazards that may lead to one or more of the system-level accidents) and builds a safety control structure diagram of the ACC system and its environment. The safety control structure diagram is a high-level abstraction diagram that contains the main components which interact with the ACC software controller and the necessary information about the input, software control actions and feedback signals. The control structure diagram is visualized by XSTAMPP. One of the internal ACC system designers reviews the control structure model and the STPA-generated software safety requirements and provides feedback to the safety analyst. Based on his feedback and improvement suggestions, the safety analyst modifies the diagram. The safety analyst uses the final control structure diagram to guide the safety analysis process to derive the software safety requirements based on the *STPA SwISs* approach which is described in the following sub-steps: (1) For each software control action that is a safety-critical action, identify the potentially unsafe control actions; (2) Translate the unsafe control actions into informal textual safety requirements; (3) Augment the process model and its variables (states) which affect the safety of one or more control actions of the software controller in the control structure diagram; (4) Generate the combination sets of the process model variable values using context tables, which can be generated using XSTAMPP. Evaluate each control action and whether it is hazardous to provide or not provide the control action in each context. (5) Refine the informal textual safety requirements based on the hazardous combination sets; and (6) Automatically generate the formal specification in LTL of the refined safety requirements by using XSTAMPP. The results of these steps are reviewed by the ACC system expert at BMW.

Modeling of STPA results. Second, the safety analyst builds the safe behavioral model of the software controller of the ACC system from the system specifications and STPA results. The safe behavioral model is a Simulink state chart model which contains the relevant process model variables (states) and transitions which is labeled by the STPA software safety requirements. The safe behavioral model was also reviewed by the ACC system designer and tester. The resultant model is reviewed by the ACC system testing expert at BMW.

Verifying the safe behavioral model. Third, the safety analyst transforms the safe behavioral model into the SMV (Symbolic Model Verifier) specification model [6, 21]by using the *STPA TCGenerator* tool. Then the safety analyst verifies the SMV model against the STPA results by using the NuSMV model checker to ensure that the SMV model satisfies all STPA-generated software safety requirements.

Generating safety-based test cases. Fourth, the safety tester uses the XML specification of the safe behavioral model and the STPA project, which is created by XSTAMPP as input to the *STPA TCGenerator* tool to generate the safe model test and generate test cases for each STPA software safety requirement. The system tester determines the range of test data for each safety-critical variable (process model variable). The *STPA TCGenerator* tool automatically generates the traceability matrix between the software safety requirements and the safe test model and automatically generates the safety-based test cases from this model. The generated test cases are saved automatically in an Excel sheet. The generated safety-test cases are reviewed by the ACC testing expert at BMW.

Test execution of the safety-based test cases on the ACC system environment. Finally, we define criteria of selection safety-based test cases in which each generated software safety requirement should be tested at least in one test case. The safety analyst and system tester will conduct the execution of the test cases based on the final implementation of the ACC software system on the BMW car model.

4.5. Measurements

To answer the research questions, we investigate the safety analysis and the verification and test case generation reports. Moreover, we investigate the execution results of the safety-based test cases.

To answer RQ1, we first summarise the results of deriving software safety requirements. We calculate the total number of the STPA-generated software safety requirements which are derived at the system level and the unsafe software scenarios which are reported by following *STPA SwISs* approach.

To answer RQ2, we investigate the list of the generated unsafe scenarios to evaluate whether these scenarios describe real unsafe scenarios in the ACC system. We investigate the list of test cases generated and the test execution report to evaluate how useful it is to generate test cases directly from the safety analysis. Finally, we calculate the coverage of the software safety requirements by the generated test cases generated by counting the total number of STPA-generated software safety requirements covered by safety-based test cases. We measure the Software Safety Requirements (SSR) coverage in the generated safety-based test cases by using the following equation:

$$\text{SSR Coverage} = \frac{|\#\text{STPA SSR covered by Test Cases}|}{|\#\text{STPA Software Safety Requirements}|} \tag{1}$$

4.6. Validity Procedure

To ensure internal validity, we define an extensive review role after each step. All generated software safety requirements, the control structure diagram of ACC, the safe behavioral model, the SMV model and the safety-based test cases are reviewed by the experts of the approach, the ACC system, and software and system safety. The experts provide valuable feedback and comments to ensure that all steps were conducted correctly and the results obtained were practically reasonable and acceptable.

An external validity considers on how the proposed approach can be generalized to any software systems within the same industry or in a different industry. We perform a single case study with one company, nevertheless, we choose an automotive software of the well-known safety-critical system that has strong safety requirements. To ensure the external validity threat further studies on applying the *STPA SwISs* approach to different software systems in different industry domains are needed.

5. Results

In this section, we first present a detailed description of the case we selected. Then, we describe the results of the case study and answer the research questions.

5.1. Case Study Description

We carried out the case study on an automotive software system of the German company BMW Group. Active Cruise Control with stop-and-go (ACC) [6] is an extended version of the adaptive cruise control system which keeps the

[6] `http://www.bmw.com/articles/active_cruise_control_stop_go.html`

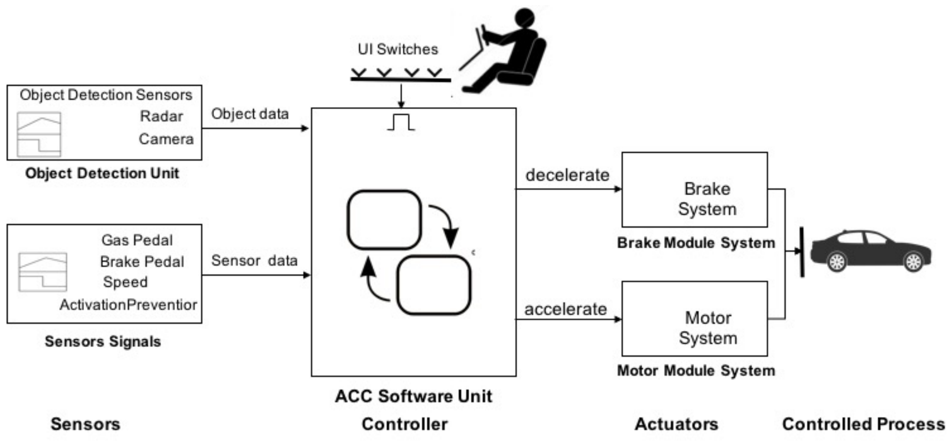


Figure 3: The block diagram of BMW's ACC System with stop-and-go function

vehicle at a safe distance from the vehicle in front at all times. It keeps the vehicle speed constant within a range of 30 to 210 km/h and automatically adapts the following distance to the vehicle in front. The stop-and-go function controls the speed when the car slows down to a standstill and restarts the engine automatically after a short interval (< 3 seconds). When there is a traffic jam or the traffic comes to a halt, the ACC system with stop-and-go will apply the brakes until the vehicle comes to a standstill and then automatically will move on as soon as the road is clear.

The ACC system with stop-and-go has four main components (shown in Fig. 3) : 1) *ACC software controller* unit which receives data from the object detection unit to automatically adapt the vehicle's speed (e.g. accelerate or decelerate or fully stop or resume speed) to the traffic environments in a critical situation. 2) *Object detection* unit which contains two components: (2.1) radar sensor with a long-range of up to 150 m which continually measures the distance between the vehicle and the objects ahead; and (2.2) a front-mounted camera on the rear-view mirror to provide an enhanced interpretation of the traffic situation ahead. 3) *A brake system* unit which is an actuator that receives a deceleration command from the ACC software unit to reduce the current speed by the deceleration ratio. 4) *A motor system* unit which is an actuator that receives an accelerate signal command from the ACC software unit to increase the current speed of the vehicle by the acceleration ratio. The ACC system gets the data/feedback from different sensors (e.g. gear sensor, door-locks sensors, the driver belt sensor, gas pedal sensor, brake pedal sensor and activation preventer sensor. The ACC system sends acoustic and optical warnings to the driver to take action.

5.2. Deriving software safety requirements at the system level

Before applying *STPA SwISs* to the active cruise control system with stop-and-go function, we established the fundamentals of the analysis. We used

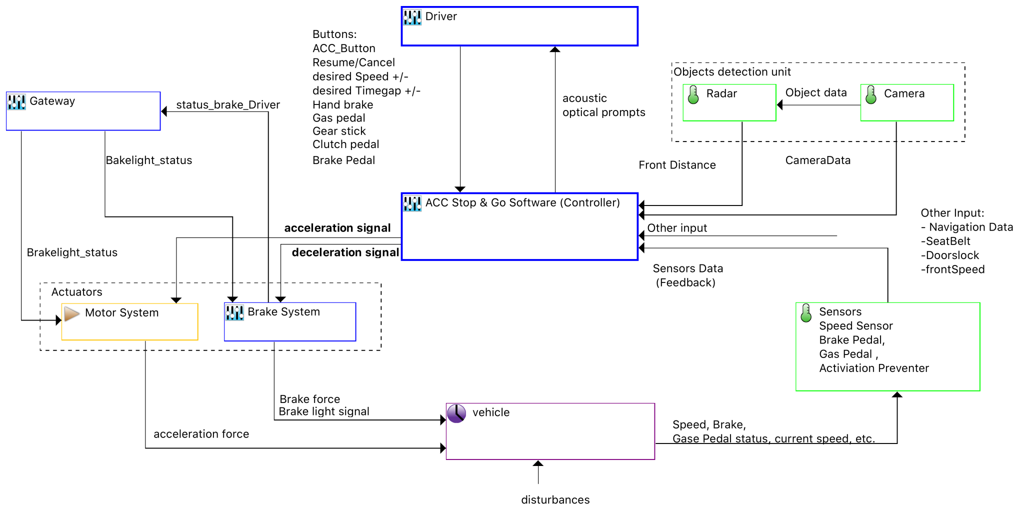


Figure 4: The safety control structure diagram of the ACC system with stop-and-go function

the XSTAMPP tool to document the results of this step. All the results and materials of this case study are available on our repository [7]. Here we summarise our results as follows: We identified 6 system-level accidents which the software of the ACC system can lead or contribute to (shown in Tab. 2). We also identified 9 system-level hazards which can lead to the accidents (shown in Tab. 3). We linked the system-level hazards to the accidents.

We drew the high-level safety control structure diagram of the ACC system (shown in Fig. 4). The diagram shows the main components which interact with the ACC system. The main components of this diagram are: 1) ACC software as a controller component, which controls the controlled process (vehicle) by issuing control actions to the actuators; 2) The motor system and brake system are the actuator components which implement control actions of the ACC software controller; 3) The vehicle is the controlled process which is controlled by an ACC software control while the ACC system is active; and 4) a set of sensor components which send feedback about the status of the controlled process to the controller. The ACC software controller issues two safety-critical control actions: *acceleration signal* and *deceleration signal* to control the speed of the vehicle. We used this diagram to identify the potentially unsafe control actions of an ACC software controller in the ACC system.

We identified 21 unsafe control actions for the safety-critical control actions (shown in Tab. 4): *acceleration signal* (10 unsafe control actions) and *deceleration signal* (11 unsafe control actions). We evaluated each item in table 4 to check whether it can contribute or lead to any system-level hazards. If an item is hazardous, we assign one or more system-level hazards to it. Otherwise,

[7]https://sourceforge.net/projects/sptaswiss-casestudy/files/

Table 2: Examples of the system level accidents

ID	Accident	Description
1	ACC Stop & Go vehicle collides with a moving vehicle in the lane while the ACC system is active.	If there is a vehicle slowing down in the front of the ACC Stop & Go and the ACC vehicle does not reduce the speed or even bring a vehicle to the complete stop.
2	A vehicle is approaching too close behind the ACC Stop & Go vehicle and suddenly the ACC stop & Go vehicle is stopped without illuminating the brake light.	A vehicle is approaching behind the ACC Stop & Go vehicle and suddenly the ACC stop & Go vehicle is stopped while the vehicle behind is too close without illuminating the brake light

Table 3: Examples of system level hazards

ID	Hazards	Accidents
1	ACC Stop & Go system does not keep a safe distance from a slowed-down object in front.	1,3, 4, 5
2	ACC Stop & Go system provides an unintended acceleration while the moving vehicle is too close.	1,3, 4
3	ACC Stop & Go system does not stop the vehicle when the traffic comes to a halt and the speed of the forward vehicle is zero (stationary).	3,4
4	ACC Stop & Go system does not keep a safe distance from the non-fixed objects in its lane.	5

we assign *not hazardous* to it. We translate each hazardous item manually to the corresponding software safety requirement. Table 5 shows examples of the informal textual software safety requirements.

To understand how each unsafe control action can occur and to identify the accident causes, we identify the safety-critical process model variables of the ACC software controller (shown in Fig. 5). The ACC software controller has a process model with 11 critical process model variables. These variables have an effect on the safety of the control actions. We classify the process model variables into three types of process model variables (shown in Tab. 6) as follows:

- **Internal state variables** which indicate the internal states of the software controller of the system such as **ACCMode** which is a process model variable that indicates the status of ACC (active or inactive)and **states** which is a process variable indicating the operational modes of the ACC system. It has five states: *stop, standby, accelerate, cruise and decelerate.*

Table 4: Examples of potentially unsafe control action *acceleration* of the ACC software controller

Not providing causes hazard	Providing causes hazard	Wrong timing or order causes hazard	Stopped too soon or applied too long
ACC software controller does not provide the acceleration signal when the road is clear and the vehicle ahead is so far. [**Not Hazardous**]	**UCA1.1.** ACC software controller provides unintended accelerate signal when a slowed down object ahead is too close. [**H-1**][**H-2**]	**UCA1.3.** ACC software controller provides an acceleration signal before the ACC is engaged and there is an object in the lane approaching too close. [**H-2**]	**UCA1.4.** ACC software controller provides acceleration signal to motor unit too long which increases the current speed beyond the desired speed.[**H-6**]

Table 5: Examples of corresponding software safety constraints at system level

UCA ID	ID	Corresponding Safety Constraints
UCA1.1	SR1.1	The ACC software controller should not provide an acceleration signal when a slowed down vehicle ahead is approaching too close.
UCA1.3	SR1.4	The ACC software controller should not provide an acceleration signal before the ACC system is engaged.
UCA1.4	SR1.3	The ACC software controller should increase the speed within the limit range of speed value (30 ...210 km/h).

- **Internal variables** which change the status of the controller such as **timeGap** which is calculated by an ACC software controller based on the front speed, current speed and front distance between the ACC vehicle and a vehicle in front of it and **currentSpeed** which indicates the current speed of the ACC vehicle.

- **Interaction Interface variables** which receive and store the data or command or feedback from the other components in the system such as *Brake status* which indicates the status of the brake pedal, *Gas Pedal* which indicates the status of the gas pedal, *resume_cancel button* which indicates the status of the resume_cancel button that actives ACC with last desired speed or deactivates ACC (inactive), *ACC button* which indicates the status of the ACC button, and the *Activation preventer* which is an

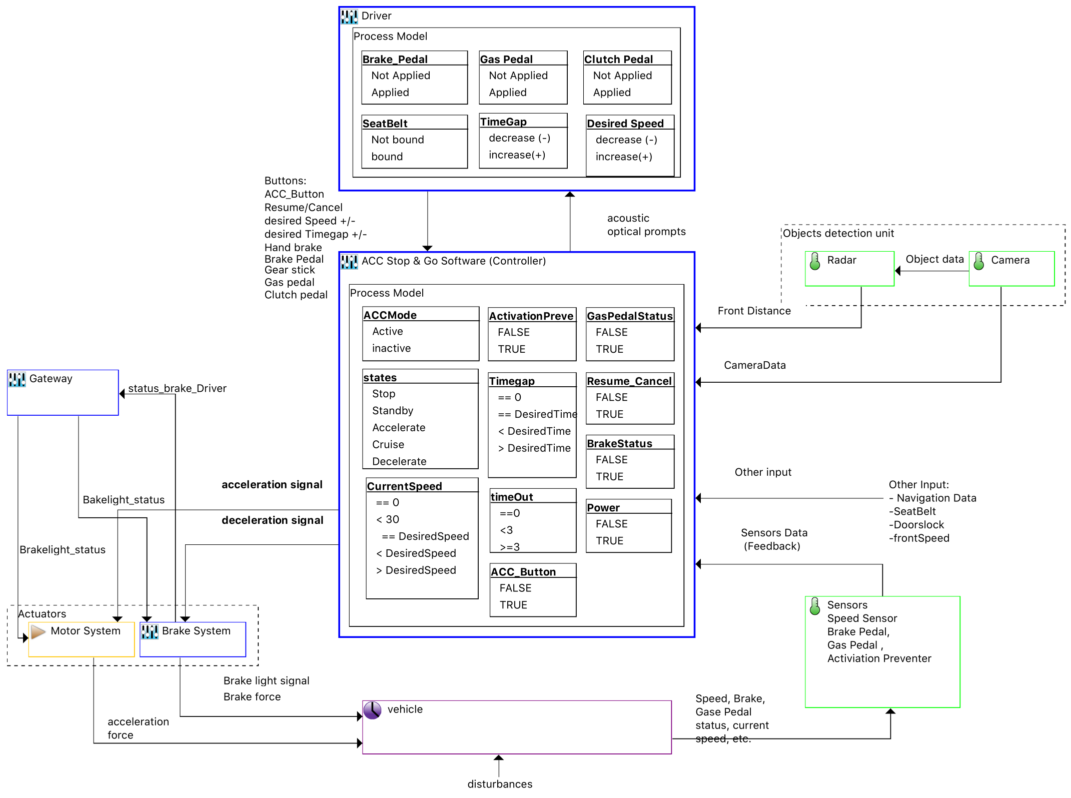


Figure 5: The safety control structure diagram of the ACC system with the safety-critical process model variables

aggregated variable that indicates the status of ACC activation preventer sensors (e.g. driver belt, door lock, gear, etc.). The ACC activation preventer is a set of the ACC deactivation variables. If the driver presses any ACC activation preventer button, then the ACC will be automatically deactivated or can not be activated.

Based on the concept of context table which is proposed by Thomas [15] and its improvements by Abdulkhaleq and Wagner [16, 3], we refine the unsafe control actions in table 4 based on the process model variables. First, we identify the dependencies between the control actions and the process model variables which have an effect on the safety of the control action to generate the context table for each control action (shown in Fig. 6).

Second, we identify the combination sets of relevant values of the process model variables (context) for each control action (shown in Tab. 7) to determine whether or not the control action in this context will be hazardous. We examine the combinations set in two contexts: *Provided control action causes hazard* and *Not Provided control action causes hazard*. The total number of all combination

Table 6: The dependency matrix between the control actions and the process model variables

Control Action	Relevant process model variables	Context
Acceleration Signal	Activation Preventer, Brake, CurrentSpeed, GasPedal, States, TimeGap	Provided/ Not Provided
Deceleration Signal	Activation Preventer, Brake, CurrentSpeed, GasPedal, States, TimeGap	Provided/ Not Provided

sets between the process model variables is calculated by the following equation:

$$\text{Total. No} = \text{Activaiton Preventer } x \text{ GasPedal } x \text{ states } x \text{ TimeGap } x \text{ CurrentSpeed } x \text{ BrakeStatus} \quad (2)$$

For the ACC stop-and-go software controller, the total number of all combination sets of process model variables is $= 2\ x\ 2\ x\ 5\ x\ 5\ x\ 6\ x\ 2 = 1200$ combinations of the process model variable values. To automatically generate the combinations and reduce their number, we used XSTPA[8] plugin in XSTAMPP which uses the combinatorial testing algorithm [17] to automatically generate the context table and identify a minimal combination of process model variables for large and complex systems. XSTPA also automatically refines the unsafe control actions which are identified in STPA Step 1 and transforms the hazardous combinations in context tables into the LTL specifications.

To reduce the number of combination sets in XSTPA, we select the combinatorial testing algorithm (e.g. pairwise algorithm). The pairwise algorithm is a testing criterion which requires that for each pair of process model variables of the software controller, every combination of valid values of these two variables be covered by at least one combination set. The algorithm takes the two longest variable values. For example, the ACC stop-and-go software controller has the following process model variables: the *currentSpeed* (6 values) and *states* (5 values) are the two longest variables values. Based on that, we reduce the total number of combinations as $= 6\ x\ 5 = 30$ combinations. Next, we generate the context tables with 30 combinations for each control action (acceleration, deceleration) in two contexts: *provided* and *not provided.*

We use two strategies to generate the context tables and ignore irrelevant combinations:

- **Assumption 1:** We assume that the ACC system is active and all the sensors which make the ACC system automatically deactivate are off.
- **Assumption 2:** We assume that the ACC system is active and one of the sensors (e.g. brake pedal) which deactivates the ACC system is on.

Table 7 shows examples of the context table of providing the control action *acceleration signal.* The hazardous rules are automatically generated from the

[8]urlhttp://www.xstampp.de/XSTPA.html

Table 7: Examples of the context table of providing the control action *acceleration signal*

Process model variables				Hazardous ?		
Activation preventer	States	CurrentSpeed	TimeGap	at any time	too early	too late
Off	Decelerate	>DesiredSpeed	Unknown	no	no	no
Off	Stop	Unknown	==0	yes	yes	yes
Off	Standby	Unknown	==DesiredTime	no	no	no
Off	Accelerate	>DesiredSpeed	<DesiredTime	yes	no	no
Off	Cruise	==DesiredSpeed	>DesiredTime	no	no	no

context table. We evaluated each hazardous rule and linked it to one or more unsafe control actions which are identified in STPA Step 1 to automatically refine the unsafe control actions with the process model variables and generate the refined software safety requirements. We identified 86 refined unsafe control actions for the ACC control actions. For example, the unsafe control action $UCA1.1$ ACC software controller provides unintended acceleration signal when a slowed down object ahead is too close can be refined as $RUCA1.1$: *ACC software controller provides the acceleration signal while ACC activation preventer is off, the brake pedal is not pressed, the state is stop, the gas pedal is not pressed, the current speed is unknown and time gap is ==0* . These 86 refined unsafe control actions are automatically transformed into the refined software safety requirements by our tool XSTAMPP/XSTPA. For example, the $RUCA1.1$ can be transformed into the refined software safety requirement $RSSR1.1$ as *The acceleration signal must be not provided any time or too late or too early when activation preventer is off, the brake pedal is not pressed, the state is stop, the gas pedal is not pressed, the current speed is unknown and time gap is ==0.* Table 8 shows examples of the refined software safety requirements based on process model variables which were generated by XSTPA.

For each refined software safety requirement, an LTL formula will be generated automatically. For example, the LTL formula of the $RSSR1.1$ can be expressed as:

$LTL1.1$ = *[ ] ((ActivationPreventer==off && brake==off && states==stop && gaspedal=off && currentspeed==unknown && timegap==0) –> ! (ControlAction==accelerationsignal))*

The LTL formulae will be used to verify the safe test model against the STPA results. Table 9 shows examples of the generated LTL formulae of the software safety requirements.

We also identified 123 causal scenarios that lead to the 21 unsafe control actions which are found in STPA Step 2 by analysing the control loops in the control structures diagram in Fig. 5. Table 10 shows examples of the causal

Table 8: Examples of refined software safety constraints based on process model variables

RUCA	ID	Refined Software Safety Constraints	LTL
RUCA1.1	RSSR1.1	The ACC software controller should not provide an acceleration signal when the activation preventer is off, the state is Stop, the gas pedal is not pressed, the brake pedal is not pressed, the current speed is unknown and the time gap is equal 0.	LTL1.1
RUCA1.2	RSSR1.2	The ACC software controller should not provide an acceleration signal when the activation preventer is off, the state is stop, the gas pedal is not pressed, the brake pedal is not pressed, the current speed is less than desired speed and the time gap is less than desired time.	LTL1.2
RUCA1.3	RSSR1.3	The ACC software controller should not provide an acceleration signal when the activation preventer is off, the state is Decelerate, the gas pedal is not pressed, brake pedal is not pressed, the current speed is greater than the desired speed and the time gap is less than the desired time.	LTL1.3

scenarios of the ACC stop-and-go controller. For example, a causal scenario of the unsafe control action $UCA1.1$: *ACC software controller provides unintended accelerate signal when a slowed down vehicle ahead is too close.* is defined as $CS1.1$: *The ACC software controller receives incorrect data from radar in front which leads to wrong estimation of time gap while a vehicle ahead is too close.*

5.3. Modeling STPA Results

After deriving the software safety requirements of the ACC stop-and-go system, we created a Simulink/Matlab stateflow model to visualize the STPA results with a safe behavioral model (shown in Fig.6). The safe behavioral model contains the process model variables of the ACC software controller (shown in Fig. 5) and shows the relationship between the process model variables and hierarchy levels between the process model variables and its values. This model constrains the transitions between the process model based on the STPA results. The model contains 18 states (4 super-states and 14 sub-states) and 23 transitions. The model has different types of state decomposition: AND_STATE (with dashed line) which shows the parallel relation that allows all sub-states to be active and OR_STATE (with solid lines) which shows the exclusive relation that allows only one state to be active at a time.

Table 9: Examples of the corresponding LTL specifications of the software safety requirements

ID	LTL Formulas
RSSR1.1	[](((ActivationPreventer==off) && (Brake==Notpressed) &&(States==Stop) && (GasPedal==NotPressed) && (CurrentSpeed==Unknown) && (TimeGap==0))−> !(controlAction==accelerationsignal))
RSSR1.2	[](((ActivationPreventer==off) && (Brake==Notpressed) &&(States==Accelerate) && (GasPedal==NotPressed) && (CurrentSpeed¡DeisredSpeed) && (TimeGap<DesiredTime))−> !(controlAction==accelerationsignal))
RSSR1.3	[](((ActivationPreventer==off) && (Brake==Notpressed) &&(States==Decelerate) && (GasPedal==NotPressed)&& (CurrentSpeed¿DeisredSpeed) && (TimeGap<DesiredTime))−> !(controlAction==accelerationsignal))

5.4. Verifying the safe behavioral model against the STPA results

To check the correctness of the safe behavioral model which is constructed within Simulink's stateflow against the STPA process model and the STPA-generated software safety requirements, we first used the Matlab command line to derive the XML specifications of the safe behavioral model of the ACC stop-and-go system. The XML specifications are saved in an XML file called *ACC-StopandGo.xml*. Second, we used the *STPA TCGenerator* tool to automatically transform the safe behavioral model into the verification model in an SMV (Symbolic Model Verifier) model. The tool takes two input files: An STPA project of the ACC system which documents the results of step 1 and the XML specification file of the safe behavioral model. The tool will parse both files and generate the SMV model which maps all states, transitions, and data variables of the safe behavioral model and the LTL formulae of STPA-generated software safety requirements to SMV model specifications and automatically save them to a file named *ACCStopandGo.smv*.

To verify the generated SMV model against the STPA software safety requirements, we used the STPA verifier plug-in [9] which is an Eclipse plug-in to verify the STPA safety requirements with model checking tools such as SPIN and NuSMV. As a result, all LTL formulae of the STPA-generated software safety requirements are stratified except 5 of them are not stratified and counterexamples are generated. We updated the safe behavioral model based on the counterexample results and generated an updated SMV model. We verified the updated SMV model against the LTL formulae. Finally, all LTL formulae were stratified by the updated SMV model.

[9] `http://www.xstampp.de/STPAVerifier.html`

Table 10: Examples of the causal factors for the ACC stop-and-go controller

Component	Causal Factor	Hazard Links
	Missing input: The data of one sensor: brake pedal/clutch pedal/ gear stick/ Activation preventer/ current speed is issued by the ACC software controller, but it does not receive by the actuators.	[H-1][H-3][H-4]
ACC Stop-Go System	Incorrect feedback: The ACC controller receives incorrect data from the sensor brake pedal/clutch pedal/ gear stick/ Activation preventer/ speed	[H-1][H-6]
(Controller)	Missing input: The ACC controller does not receive data of the status of the driver's seat belt.	[H-1][H-6][H-2]
	Inadequate Control Algorithm: The ACC controller issues the acceleration signal instead of the deceleration signal when the vehicle in front is too close in the lane.	[H-2][H-6][H-1]

5.5. Generating safety-based test cases

To automatically generate safety-based test cases, we use the safe behavioral model which is constructed from the STPA safety analysis results and validated the STPA-generated safety requirement as input to *STPA TCGenerator*. STPA TCGenerator parses Simulink's stateflow of the safe behavioral model recursively by considering Simulink's statechart notations (super state decompositions AND_STATE and OR_STATE) to automatically transform the statechart notations into the extended finite state machine notations to generate the safe test model. As a result, the safe test model contains 14 states (after removing the super states) and 56 transitions (after maintaining the transitions of super states by considering the state decomposition type). *STPA TCGenerator* automatically provides the traceability matrix between the STPA-generated software safety requirements and the safe test model. It also shows the input variables of the safe test model (e.g. currentspeed, timeOut, timegap, etc.) with their data type, initial, minimum, maximum values to allow the user to set the test input data and the test configuration.

We set the *STPA TCGenerator* with the test configurations as follows (shown in Fig. 7): the number of test steps to 20; the test algorithm is the random walk with depth-first and breadth-first search; the test coverage criteria are the state-based, transition-based and STPA software safety requirements test coverage criteria; and the stop condition is STPA software safety requirements. We also set the input value for each test data variable: desiredspeed (30–210

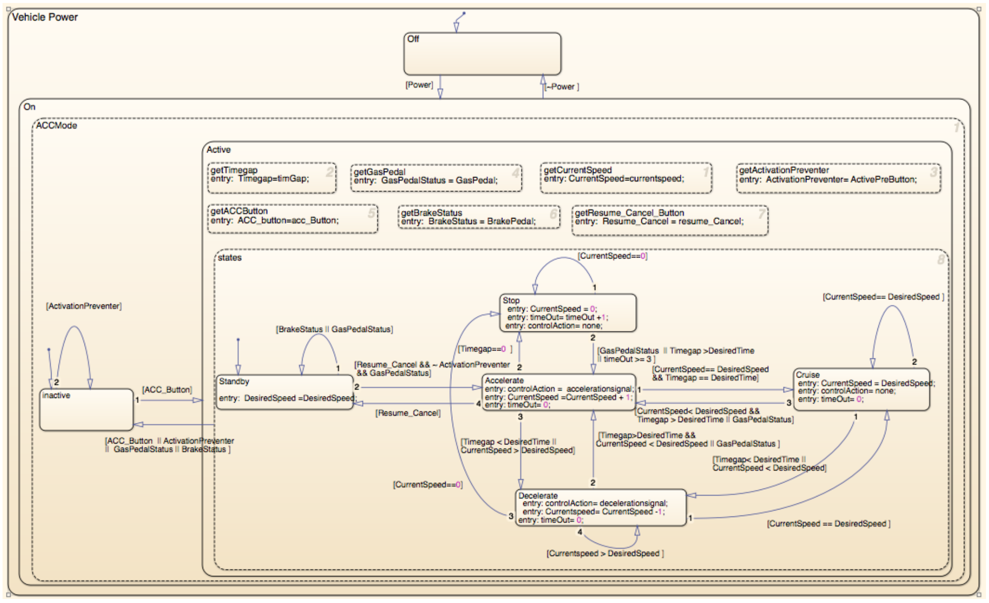

Vehicle Power
Off
[Power]
[~Power]
On
ACCMode
Active
getTimegap
entry: Timegap=timGap;
getGasPedal
entry: GasPedalStatus = GasPedal;
getCurrentSpeed
entry: CurrentSpeed=currentspeed;
getActivationPreventer
entry: ActivationPreventer= ActivePreButton;
getACCButton
entry: ACC_button=acc_Button;
getBrakeStatus
entry: BrakeStatus = BrakePedal;
getResume_Cancel_Button
entry: Resume_Cancel = resume_Cancel;
states
[ActivationPreventer]
inactive
[ACC_Button]
[ACC_Button || ActivationPreventer || GasPedalStatus || BrakeStatus]
[BrakeStatus || GasPedalStatus]
Standby
entry: DesiredSpeed =DesiredSpeed;
[Resume_Cancel && ~ ActivationPreventer && GasPedalStatus]
[Resume_Cancel]
[CurrentSpeed==0]
Stop
entry: CurrentSpeed = 0;
entry: timeOut= timeOut +1;
entry: controlAction= none;
[Timegap==0]
[GasPedalStatus || Timegap >DesiredTime || timeOut >= 3]
Accelerate
entry: controlAction = accelerationsignal;
entry: CurrentSpeed =CurrentSpeed + 1;
entry: timeOut= 0;
[CurrentSpeed== DesiredSpeed && Timegap == DesiredTime]
[CurrentSpeed< DesiredSpeed && Timegap > DesiredTime || GasPedalStatus]
[CurrentSpeed== DesiredSpeed]
Cruise
entry: CurrentSpeed = DesiredSpeed;
entry: controlAction= none;
entry: timeOut= 0;
[Timegap < DesiredTime || CurrentSpeed > DesiredSpeed]
[Timegap>DesiredTime && CurrentSpeed < DesiredSpeed || GasPedalStatus]
[Timegap< DesiredTime || CurrentSpeed < DesiredSpeed]
[CurrentSpeed==0]
Decelerate
entry: controlAction= decelerationsignal;
entry: Currentspeed= CurrentSpeed -1;
entry: timeOut= 0;
[CurrentSpeed == DesiredSpeed]
[Currentspeed > DesiredSpeed]

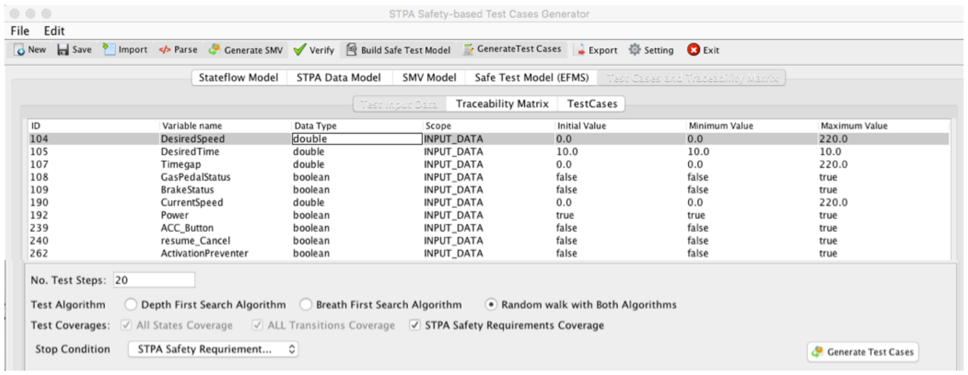
Figure 6: The safe behavioral model of the ACC stop-and-go software controller

kmh), deisredTimegap (2,4 seconds), Timegap (0–10 seconds), currentspeed(0–210 kmh), power (true), brakepedal (false–true), gaspedal (false–true) and Activation preventer (false–true).

As a result, we generated 40 test suites with a total of 230 test cases within 100.0% state coverage, 82.59% transition coverage, and 100 % STPA safety requirement coverage. 180 out of the 230 test cases are safety-based test cases which have a relation with one or more STPA safety requirements in the traceability matrix. The test cases are automatically saved in a CSV file.

5.6. Execution of the safety-based test cases

Based on the available resources (time and hardware) from our industrial partner, we were allowed to execute only 20 test cases. We selected 20 test cases out of 180 which are more relevant for the critical control actions of the ACC software controller and the STPA-generated software safety requirements. We executed the selected test cases by driving the car on the highway. The safety analyst and system tester conducted the execution of the test cases on the car model G11 series 7. They drove first from the university of Stuttgart to the highway because they could not perform the test in the city due to the safety reasons and the BMW car model was under test. The test was performed in a realistic environment on a German highway under cloudy weather. Table 11 shows the examples of the selected safety-based test cases with the execution results. As a result, the ACC stop and go system succeed with 18 out of 20 safety-based test cases. One test scenario was difficult to test. The test scenario


STPA Safety-based Test Cases Generator
File Edit
New Save Import Parse Generate SMV Verify Build Safe Test Model GenerateTest Cases Export Setting Exit
Stateflow Model STPA Data Model SMV Model Safe Test Model (EFMS) Test Cases and Traceability Matrix
Test Input Data Traceability Matrix TestCases
Generate Test Cases

ID	Variable name	Data Type	Scope	Initial Value	Minimum Value	Maximum Value
104	DesiredSpeed	double	INPUT_DATA	0.0	0.0	220.0
105	DesiredTime	double	INPUT_DATA	10.0	10.0	10.0
107	Timegap	double	INPUT_DATA	0.0	0.0	220.0
108	GasPedalStatus	boolean	INPUT_DATA	false	false	true
109	BrakeStatus	boolean	INPUT_DATA	false	false	true
190	CurrentSpeed	double	INPUT_DATA	0.0	0.0	220.0
192	Power	boolean	INPUT_DATA	true	true	true
239	ACC_Button	boolean	INPUT_DATA	false	false	true
240	resume_Cancel	boolean	INPUT_DATA	false	false	true
262	ActivationPreventer	boolean	INPUT_DATA	false	false	true

No. Test Steps: 20

Test Algorithm: Depth First Search Algorithm Breath First Search Algorithm [✓ Random walk with Both Algorithms]

Test Coverages: ✓ All States Coverage ✓ ALL Transitions Coverage ✓ STPA Safety Requirements Coverage

Stop Condition: STPA Safety Requrienment...

Figure 7: The test configuration view in the STPA TCGenerator

was to test the stop function when there is a traffic jam and the vehicle in front moved slowly. As a requirement, the ACC stop-and-go will automatically restart the engine and move off the vehicle, if the stop lasts from 0 to 3 seconds, otherwise it will automatically stop the vehicle until the vehicle in front starts moving again. For this situation, we used a timer to calculate the time of the stop. But it was difficult to measure the stop time of the vehicle in front within 1 or 2 seconds and move again. In another test scenario, we recognized that if the ACC stop-and-go is in the deceleration state (currentspeed > desired speed and there is a vehicle is in front) and the driver pressed ACC button, then the vehicle speed is immediately decelerated too slowly to avoid a collision.

5.7. Discussion

We summarised all results of the case study presented in this paper (shown in Tab. 12). Based on the results of the case study, we answered our research questions as follows: For research question RQ-1, we identified 6 system-level accidents to which the ACC software can contribute to and 9 system-level hazards. We also identified 21 unsafe control actions. Furthermore, we identified 86 refined unsafe scenarios that describe different hazardous events in which the ACC software transits the ACC system into hazardous behaviors. We automatically identified 86 refined software safety requirements based on the process model variables of the control actions of the ACC software controller: *acceleration and deceleration signals*. We also identified 35 accident causes that the ACC software controller can contribute to. The evaluation of *STPA SwISs* substantiates that: 1) the STPA-based analysis approach helps us to identify the hazardous situations of the ACC software controller at the system level and develop detailed software safety requirements; 2) and it help us to transform the informal software safety requirements into formal specifications in LTL to be used for the verification purpose.

Table 11: Examples of the selected test cases and the test execution results

#No.	#Precondition	#Post-condition	Road status	#Test Result

1	currentspeed=100
	DesiredSpeed=83
	DesiredTimeGap=2.4
	TimeGap>=2.4
	Gaspedal=true
	Brakestatus=false
	Power=true
	ACC_Button=false
	Resume_Cancel=false
	ActivatPreven.=false
	currentstate=Accelerate
		controlAction= accelerationsignal State= Accelerate	No vehicle in the lane	Success

2	currentspeed=0
	DesiredSpeed=70
	DesiredTimeGap=2.4
	TimeGap=0.0
	Gaspedal=false
	Brakestatus=true
	Power=true
	ACC_Button=false
	Resume_Cancel=false
	ActivatPreven.=false
	currentstate=stop
	timeOut=1.0
		controlAction= none, State=Stop	A vehicle in front in the lane	incomplete

5.8. Answer the Research Questions

To investigate research question RQ-2, the results reveal that we could generate 180 safety-based test cases from the results of research question RQ-1. All software safety requirements which are identified in step 1 are covered in at least one safety-test case. We obtained 100% for the Software Safety Requirements (SSR) coverage in the generated safety-based test case. That means we can test the ACC system against each software safety requirements with different test cases to measure the safety of the whole system. Deriving test cases directly from the safety analysis results allows us to focus the testing effort to test the critical risky situations. We were able only to execute 20 out of 180 safety-based test cases, however, we could recognize different situations of ACC software behaviors.

Table 12: A summary of the case study results

ID	Item	Total. No
1	System Level Accidents	6
2	System level Hazards	9
3	Unsafe control actions	21
4	Corresponding software safety constraints	21
5	Refined unsafe control actions	86
6	Refined software safety constraints	86
7	Causal factors	35
8	Generated LTL formuale	86
9	Generated test cases	230
10	Safety-based test cases	180

6. CONCLUSIONS AND FUTURE WORK

In this paper, we conducted an industrial case study on evaluation *STPA SwISs* approach. *STPA SwISs* combines the safety analysis activities based on STPA safety analysis with the verification and testing activities. We applied the STPA-based analysis approach to the software controller of the BMW Active Cruise Control System (ACC) at the BMW Group in Munich. The first main contribution is to investigate how to identify the hazardous control actions of the ACC software controller and develop the detailed software safety requirements. The second contribution is to show how to transform the STPA-generated software safety requirements into the formal specifications in LTL to be used in the verification activities. The third contribution is to generate the safety-based test cases directly from the results of the safety analysis to test the ACC system against the STPA-generated software safety requirements. The results showed that the practical effectiveness, scalability and applicability of the approach.

As a limitation, we could only execute a few generated safety-based test cases. Furthermore, we were not able to test some of the test case scenarios to recognize some of the unsafe behaviors such as a bicycle moving in front of the ACC vehicle in the lane or a small stationary obstacle in the lane (e.g. stone).

As a future work, we plan to evaluate the results of this case study with the BMW experts by gathering from them quantitative and qualitative data.

Acknowledgment

The authors would like to express gratitude to Mr. Joham Achatz, ACC system architecture expert at BMW, for his very careful review of the control structure diagram and process model, and for the comments and suggestions that ensued.

The authors also would like to thank Kornelia Kuhle, University of Stuttgart, for her feedback on the text.

References

[1] M. Broy, Challenges in automotive software engineering, in: Proceedings of the 28th International Conference on Software Engineering, ICSE '06, ACM, New York, NY, USA, 2006, pp. 33–42.

[2] N. Leveson, Engineering a Safer World: Systems Thinking Applied to Safety, Engineering Systems, MIT Press, 2011.

[3] A. Abdulkhaleq, S. Wagner, N. Leveson, A comprehensive safety engineering approach for software-intensive systems based on STPA, 2015 European STAMP Workshop at Amsterdam University of Applied Sciences.

[4] A. Abdulkhaleq, S. Wagner, XSTAMPP: An eXtensible STAMP platform as tool support for safety engineering, 2015 STAMP Conference, MIT, 2015.

[5] A. Pnueli, The temporal logic of programs, in: Proceedings of the 18th Annual Symposium on Foundations of Computer Science, SFCS '77, IEEE Computer Society, Washington, DC, USA, 1977, pp. 46–57.

[6] K. L. McMillan, Symbolic Model Checking, Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[7] A. Cimatti, E. M. Clarke, F. Giunchiglia, M. Roveri, NUSMV: A new symbolic model checker, STTT 2 (4) (2000) 410–425.

[8] J. Thomas, F. Lemos, N. Leveson, Evaluating the safety of digital instrumentation and control systems in nuclear power plants, MIT Technical Report.

[9] W. Vesely, F. Goldberg, N. Roberts, D. Haasl, Fault tree handbook (NUREG-0492), U.S. Nuclear Regulatory Agency.

[10] FMECA, Design analysis procedure for Failure Modes, Effects and Criticality Analysis (FMECA), Society for Automotive Engineers.

[11] R. S. Martnez, System theoretic process analysis of electric power steering for automotive applications, Master's thesis, MIT, U.S.A (2015).

[12] J. Thomas, Extending and automating a systems-theoretic hazard analysis for requirements generation and analysis, Ph.D. thesis, MIT (4 2013).

[13] A. Abdulkhaleq, S. Wagner, Experiences with applying stpa to software-intensive systems in the automotive domain, 2013 STAMP Conference at MIT, Boston, USA.

[14] S. Wagner, B. Schatz, S. Puchner, P. Kock, A case study on safety cases in the automotive domain: Modules, patterns, and models, in: 2010 IEEE 21st International Symposium on Software Reliability Engineering, 2010, pp. 269–278.

[15] J. Thomas, Extending and automating a systems-theoretic hazard analysis for requirements generation and analysis, SANDIA National Laboratories report 2012-4080.

[16] A. Abdulkhaleq, S. Wagner, Integrated Safety Analysis Using Systems-Theoretic Process Analysis and Software Model Checking, Computer Safety, Reliability, and Security: 34th International Conference, Springer International Publishing, Cham, 2015, pp. 121–134.

[17] D. Kuhn, R. Kacker, Y. Lei, Introduction to Combinatorial Testing, Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series, Taylor & Francis, 2013.

[18] A. Abdulkhaleq, S. Wagner, A Software Safety Verification Method Based on System-Theoretic Process Analysis, Computer Safety, Reliability, and Security: SAFECOMP 2014 Workshops, Springer International Publishing, Cham, 2014, pp. 401–412.

[19] G. J. Holzmann, The model checker spin, IEEE Trans. Softw. Eng. 23 (5) (1997) 279–295.

[20] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Vol. 14, Springer US, 2009, pp. 131–164.

[21] A. Cimatti, E. M. Clarke, F. Giunchiglia, M. Roveri, Nusmv: A new symbolic model verifier, in: Proceedings of the 11th International Conference on Computer Aided Verification, CAV '99, Springer-Verlag, London, UK, UK, 1999, pp. 495–499.