An automatic safety-based test case generation approach based on systems-theoretic process analysis

Article · October 2016

DOI: 10.18419/opus-8908

2 authors:

Asim Abdulkhaleq
Bosch
23 PUBLICATIONS **345** CITATIONS
SEE PROFILE

Stefan Wagner
247 PUBLICATIONS **3,841** CITATIONS
SEE PROFILE

An Automatic Safety-Based Test Case Generation Approach Based on Systems-Theoretic Process Analysis

Asim Abdulkhaleq, Institute of Software Technology, University of Stuttgart, Germany
Stefan Wagner, Institute of Software Technology, University of Stuttgart, Germany

Software safety remains one of the essential and vital aspects in today's systems. Software is becoming responsible for most of the critical functions of systems. Therefore, the software components in the systems need to be tested extensively against their safety requirements to ensure a high level of system safety. However, performing testing exhaustively to test all software behaviours is impossible. Numerous testing approaches exist. However, they do not directly concern the information derived during the safety analysis. STPA (Systems-Theoretic Process Analysis) is a unique safety analysis approach based on system and control theory, and was developed to identify unsafe scenarios of a complex system including software. In this paper, we present a testing approach based on STPA to automatically generate test cases from the STPA safety analysis results to help software and safety engineers to recognize and reduce the associated software risks. We also provide an open-source safety-based testing tool called *STPA TCGenerator* to support the proposed approach. We illustrate the proposed approach with a prototype of a software of the Adaptive Cruise Control System (ACC) with a stop-and-go function with a Lego-Mindstorms EV3 robot.

CCS Concepts: •**Software and its engineering** → **Formal software verification; Software safety; Software testing and debugging;**

Additional Key Words and Phrases: safety-critical software, STPA safety analysis, software safety Formal software verification, test case generation

ACM Reference Format:
Asim Abdulkhaleq and Stefan Wagner, 2016. An Automatic Safety-Based Test Case Generation Approach Based on Systems-Theoretic Process Analysis. *ACM Trans. Softw. Eng. Methodol.* , , Article (May 2016), 31 pages.
DOI: 0000001.0000001

1. INTRODUCTION

Software has become an indispensable part of many modern systems and often performs the main safety-critical functions. Hence, software safety must be analysed in a system context to gain a comprehensive understanding of the roles of software and to identify the software-related risks that can cause hazards in the system. A software failure may lead to catastrophic results such as injury or loss of human life, damaged property or environmental disturbances. Therefore, it becomes essential to test the software components for unexpected behaviour before using them in practice [Minister of Defence 1991]. The Toyota Prius, the General Motors airbag and the loss of the Mars Polar Lander (MPL) mission [JPL 2000] are well-known software problems in which the software played an important role in the loss, although the software had been successfully verified against all functional requirements.

Software testing is a crucial process to assess the quality of the software and determine whether it meets its specified requirements. The term software safety testing

Author's addresses: A. Abdulkhaleq and, S. Wagner, Institute of Software Technology, University of Stuttgart, Universitätsstrae 38, 70569 Stuttgart, Germany

DOI: 0000001.0000001

[NASA-GB- 8719.13 2004] was introduced and implies that software testing should not only address functional requirements, but the software safety requirements as well. Therefore, the process for testing safety-critical software combines conventional testing and safety analysis approaches to focus the testing efforts in a specific way to address the safety of the software and test the critical risky situations. Fault Tree Analysis (FTA) [Vesely et al. 1981] and Failure Mode, Effects and Criticality Analysis (FMECA) [International 1967] are the approaches commonly used for the purpose of safety-based testing. However, these approaches focus only on single component failures and they have limitations to cope with complex systems including software. Leveson [Leveson 2011] noted that the primary safety problem in software-intensive systems is not software failure but the lack of appropriate constraints on software behaviour. The solution is to identify the required constraints and enforce them in the software and overall system design. Therefore, a new safety analysis technique called STPA [Leveson 2011] has been developed to overcome the limitations of the traditional techniques in addressing the unsafe scenarios of complex systems.

1.1. Problem Statement

The complexity of safety-critical software makes exhaustive software testing impossible. In addition, existing testing approaches and tools do not incorporate the information derived from safety analysis sufficiently. Integrating the information from safety analysis into software testing creates a number of challenges. First, defining appropriate software safety requirements with traditional safety analysis techniques is difficult due to their limitations to cope with complex systems including software. Second, the traditional techniques such as FTA and FMECA do not provide any kind of model to represent the system behaviour. Third, the safety requirements usually are written in natural language which makes it hard to map them directly to the test model.

1.2. Research Objectives

The overall objective of this research is to fill the aforementioned gap by investigating the possibility of automatically generating safety-based test cases from the information derived during the safety analysis. This will help safety and software engineers to recognize the unexpected software behaviours that may contribute to an accident and avoid them.

1.3. Contributions

The main contribution of this paper is an automatic safety-based testing approach to generate safety-based test cases using the information derived from STPA safety analysis. We provide four main contributions: (1) We develop an algorithm based on STPA to derive unsafe software scenarios and automatically translate them into a formal specification in LTL (Linear Temporal Logic) [Pnueli 1977]. (2) We explore how to build the safe software behavioural model based on the STPA control structure diagram. (3) We develop an algorithm to automatically extract the safe test model and check its correctness by automatically transforming it into an SMV representation (Symbolic Model Verifier) [McMillan 1993] and verify it against the STPA safety requirements using the NuSMV model checker [Cimatti et al. 1999]. (4) We develop an algorithm to automatically generate the traceability matrix between the STPA software safety requirements and the test model and generate the safety-based test cases for each safety requirement from the test model.

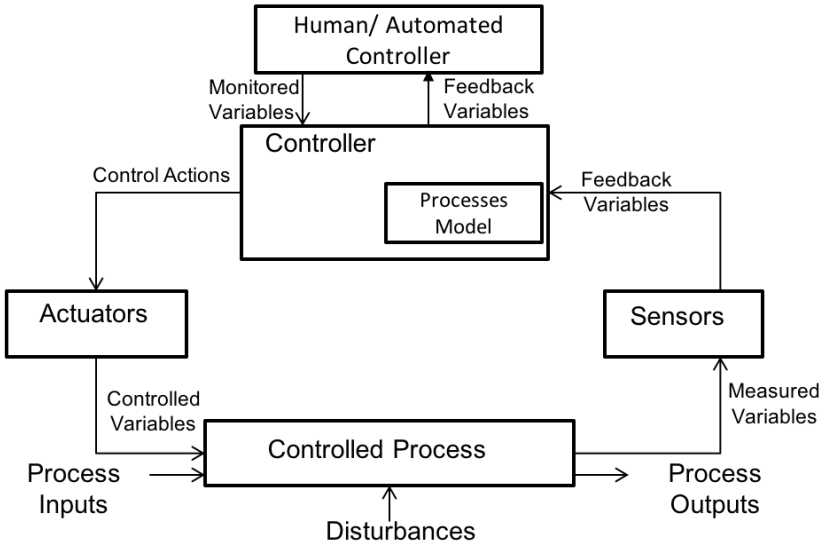

Fig. 1. A general feedback control structure of a software system

2. BACKGROUND

2.1. STPA Safety Analysis & Software Safety

STPA (Systems-Theoretic Processes Analysis) [Leveson 2011] is a top-down process based on the accident model called STAMP (Systems-Theoretic Accident Model and Processes). STPA is developed for generating detailed safety requirements of complex and modern systems to prevent the occurrence of unsafe scenarios in the systems. In STPA, the system is seen as a set of interrelated components which interact with each other to provide a dynamic equilibrium through feedback loops of information and control. STPA has three main steps: (1) Establish the fundamentals of the analysis (e.g. system-level accidents and the associated hazards) and draw the control structure diagram of the system (shown in Fig. 1). (2) Use the control structure diagram to identify the potential unsafe control actions. (3) Determine how each potentially unsafe control action could occur by identifying the process model and its variables for each controller and analysing each path in the control structure diagram.

An extended approach to STPA is proposed by Thomas [Thomas 2013] for identifying the unsafe control actions which are identified in STPA Step 2 based on the combinations of process model variables of each controller in the control structure diagram.

The basic components in STPA are safety constraints, unsafe control actions, unsafe scenarios, control structure diagram and process models. A control structure diagram is made up of basic feedback control loops. An example is shown in Fig. 1. When put together, they can be used to model the high-level control structure of a particular system.

Definition 2.1 (*A Control Structure Diagram*). The Control Structure Diagram (CSD) of a software system S can be expressed with five-tuples (CO, AC, SO, CP, CA), where CO is a set (one or more) of the software controllers which control the controlled processes (CP) by issuing control actions to the actuators, AC is a set of the actuators which implement the control actions (CA) of the controller, CP is a set of the controlled processes which are controlled by controllers (COs). SO is a set of sensors which send the feedback about the status of the controlled process.

Each controller in the control structure diagram must contain a model of the assumed state of the controlled process, called the process model [Leveson 2011]. A process model contains one or more variables, the required relationships among the variables, the current state and the logic of how the process can change state. This model is used to determine what control actions are needed. It is updated through various forms of feedback [Leveson 2011].

Definition 2.2 (*A Software Controller*). A software controller CO_i can be expressed formally as a two-tuple $CO_i = (CA, PM)$, where CA is set of the control actions and PM is the process model of the controller which has a set of Process Model Variables (PMV), which are a set of states that have an effect on the safety of CA: $PMV = \bigcup(\mathcal{P}_1 \dots \mathcal{P}_n)$.

In [Abdulkhaleq et al. 2015], we classified the process model variables of the software controller that affect the safety of the critical control actions into three types: 1) *Internal variables* which change the status of the software controller, 2) *Interaction interface variables* which receive and store the data/command/feedback from the other components in the system, and 3) *Environmental variables* of the environmental components that interact with or are controlled by the software controller.

To support the safety engineering process based on STPA, we developed an extensible platform called XSTAMPP[1] [Abdulkhaleq and Wagner 2015b] which 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 three main steps of STPA and provides an internal representation in XML for each STPA component to support possible future integration with other tools.

2.2. Software Safety Testing

Software testing is one of the most important phases during the software development process to confirm that the software complies with its requirements, and ensure that the software performs all required functions correctly. A popular testing approach called Model-based Testing (MBT) [Dalal et al. 1999; Apfelbaum and Doyle 1997] aims at automatically generating test cases using models extracted from software requirements. The model-based testing process involves creating a suitable model of the software's behaviour based on requirements or an existing specification to generate the test cases.

A number of software behaviour models are in use today, several make good models for testing such as control flow charts, finite state machines, SpecTRM-RL [Leveson 2000], and sequence event diagrams. Finite state machines are commonly used in software behaviour modeling and testing to generate test cases [Apfelbaum and Doyle 1997]. The finite state model includes a set of states, a set of input events and the transition between them. The main challenge of software testing is to generate suitable test cases that cover all requirements and functions of the software.

Software safety testing [NASA-GB- 8719.13 2004; Lutz 2000] is a crucial process in developing safety-critical systems to verify whether a software system meets its safety requirements. Safety-critical software should be tested extensively to ensure that the potential software-related hazards have been eliminated or controlled to a low level of risk.

3. RELATED WORK

In the following, we will discuss our own prior work and the related work.

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

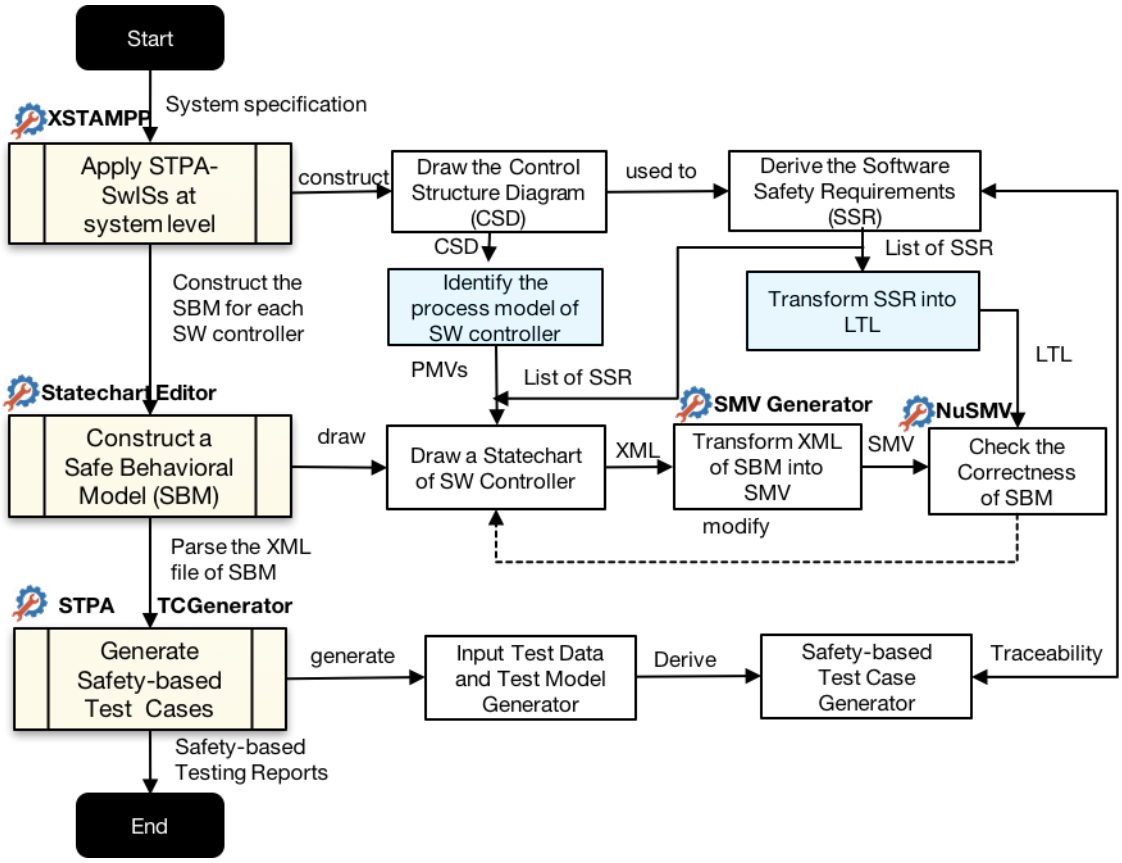

Fig. 2. An overview of the proposed approach

3.1. Risk-Based Software Testing

There are quite a few software safety test techniques in the literature that combine safety analysis principles with model-based testing. Most of them use the term ”Risk-based Testing”, which combines risk analysis approaches such as FTA and/or FMEA or the Markov chain with software testing approaches (e.g. model-based testing) to create a prioritization criterion for generating test cases.

Redmill [Redmill 2004] explored the benefits of risk-based testing as the basis of test planning in the software testing process and how to understand the risks of the system to focus test efforts. He does not show how to generate the test cases from the risk analysis approach.

Zimmermann et al. [Zimmermann et al. 2009] proposed a refinement-based approach to the reliability analysis of software-critical systems. They used model-based statistical testing as a model-based testing technique and the Markov chain model to model the system under test. They also used FTA and FMEA as risk-based analysis techniques to identify the critical situations that represent high risk.

Kools et al. [Kloos et al. 2011] proposed a model-based testing approach which uses the information derived from FTA in combination with a system model to generate the risk-based test cases. They used FTA to select, generate and prioritize the test cases. They derived the test cases from the combination of fault trees and a basic system behaviour model called the ”base model”.

Our approach uses a similar principle of combining the risk analysis approaches with model-based testing but the difference is that our approach uses STPA for safety analysis based on system and control theory rather than reliability theory like FTA and FMEA. STPA copes with the analysis of complex, modern systems and tackles the dynamic behaviour of the system by treating safety as a control problem. Furthermore, STPA provides an abstract model of the system under analysis called the safety control structure which views all main interacting components including the software components of the system. This allows us to directly construct the test model from the control structure diagram and constrain its transitions with the STPA-generated safety requirements.

3.2. STPA SwISs Approach for Software-Intensive Systems

Developing safety-critical software requires a more systematic software and safety engineering process that enables the software and safety engineers to recognize the potential software risks. For this purpose, we proposed a comprehensive software safety engineering approach based on STPA for Software-Intensive Systems, called STPA SwISs [Abdulkhaleq et al. 2015]. STPA SwISs provides a concept of deriving the software safety requirements by STPA at the system level, modeling them as a safe behavioural model and automatically generating test cases from this model by using an existing model-based tool such as ModelJUnit [Utting and Legeard 2007]. The STPA SwISs approach is carried out in three major steps: 1) Deriving the software safety requirements at the system level, and generating the unsafe scenarios based on the extended approach to STPA by Thomas [Thomas 2013], and expressing the corresponding safety requirements in formal specifications using LTL; 2) Modeling STPA results with a safe behavioural model. A safe behavioural model is a UML statechart notation that models the process model variables of a software controller in the STPA control structure diagram as states and the control actions as the state actions, and it is constrained by the STPA-generated software safety requirements (transitions); and 3) Generating the safety-based test cases by using an existing model-based tool.

Our preliminary algorithm [Abdulkhaleq et al. 2015] for deriving test cases from STPA results relied on using an existing model-based testing tool called ModelJUnit to drive the test cases. This algorithm is effective in deriving test cases but it has some limitations: 1) ModelJUnit requires that a behavioural model be written as a Java class, which represents the finite state machine of the system; 2) The ModelJUnit tool has not been developed with the purpose of safety-based testing and deriving the test cases from the safety analysis results; 3) there is no way to verify and check the correctness of a test input model of ModelJUnit against the safety analysis results; and 4) The ModelJUnit tool does not provide a traceability matrix between the safety requirements and the generated test cases.

4. THE PROPOSED APPROACH

In this section, we propose an automatic safety-based test case generation approach for deriving test cases directly from the STPA safety analysis results. The proposed approach follows the main steps of the STPA SwISs approach and provides a high degree of automation of each step.

Figure 2 shows the main steps of the approach which includes four steps: 1) deriving the software safety requirements of a software controller by following the STPA SwISs approach [Abdulkhaleq et al. 2015] and automatically expressing them in LTL, 2) constructing the safe behavioural model of the software controller with the statechart notations in Simulink, 3) transforming the safe behavioural model into an input model of the NuSMV model checker and checking the correctness of the generated model against the STPA and safety requirements expressed in LTL; and 4) automat-

ically generating a safety-based test model and deriving the safety-based test cases from this model. In the following sections, we describe the four major activities in more detail:

4.1. Deriving Software Safety Requirements

This step starts by applying STPA to the system specification to identify STPA software safety requirements and the potentially unsafe scenarios which the software can contribute to. The algorithm starts by establishing the fundamentals of analysis by determining the system-level accidents (ACC) and the associated system-level hazards (HA) which the software can lead to or contribute in. Next, the algorithm demands that the safety control structure diagram of the system shall be constructed from the system specifications. The software here is the controller in the control structure diagram.

For each software controller component in the control diagram, its software safety requirements can be derived by performing the following steps:

(1) Identify all safety-critical Control Actions (CAs) that can lead to one or more of the associated hazards (HA).
(2) Evaluate each CA with four general types of hazardous behaviours to identify the Unsafe Control Actions ($UCAs$): (a) a control action required for safety is not provided, (b) an unsafe action is provided, (c) a potentially safe control action is provided too early, too late or out of sequence and (d) a safe control action is stopped too soon or continued too long.
(3) Translate the identified $UCAs$ manually into informal textual Software Safety Requirements (SSR).
(4) Identify the process model and its variables and include them in the software controller in the control structure diagram to understand how each UCA could occur. The process model describes the states of the software controller (only critical states which are relevant to the safety of the control actions) and their variables describe the software communication, input and output.
(5) Automatically generate the critical set of combinations of the process model variables for each control action (CA). Each combination should be evaluated within two contexts (C_1= **Providing** CA or C_2 = **Not Providing** CA) to determine whether the control action is hazardous in that context or not. A control action CA could be considered hazardous in context C if only a combination of process variables related to CA leads to a system-level hazard $H \in HA$. The context C_1 = **Providing** CA has three types of sub-contexts: *context incorrectness*, in which the unsafe control action commanded incorrectly and caused a hazard (any time), *context real-time execution*, in which the unsafe control action commanded in a wrong timing (too early or too late) or sequence, and *context execution mechanism*, in which the unsafe control action commanded in a wrong mechanism of execution (applied too long or stopped too soon).
(6) Identify the potentially unsafe critical combination of unsafe software control action and evaluate it to identify the potential unsafe scenarios of the software.

Definition 4.1 (*Refined Unsafe Control Action*). The refined unsafe control action ($RUCA$) is a four-tuple (CA, Cs, C, TC), where CA is a control action which causes a hazard $H \in HA$, Cs = $\bigcup(\mathcal{P}_1 = v_1, \ldots \mathcal{P}_n = v_n)$ which is a critical set of combinations of the relevant process model variables PMV of CA, C is a context where providing or not providing the control action CA is hazardous, and TC is the type of context **providing** of control action CA (**any time**, **too early** or **too late**).

To automatically translate each critical combination of process model variables for each control action CA into the unsafe software scenarios, we set the following rules:

Rule 1: Each refined unsafe control action ($RUCA$) in the context of **Providing** (C_1) of a control action CA_i can be expressed as:

$RUCA_i$ = <CA> **provided** <TC> **is hazardous when** <Cs= $\bigcup(\mathcal{P}_1 = v_1, \ldots \mathcal{P}_n = v_n)$> occurred.

Rule 2: Each refined unsafe control action ($RUCA$) in the context of **Not Providing** (C_2) of a control action CA_i can be expressed as:

$RUCA_i$ = <CA> **Not provided is hazardous when** <Cs= $\bigcup(\mathcal{P}_1 = v_1, \ldots \mathcal{P}_n = v_n)$> occurred.

By using the rules 1 and 2, we refine the unsafe control actions which are identified based on the combination set of process model variables. The software safety requirements are generated automatically from the refined unsafe control actions. Based on definition 3, we identify the following rules which are used to automatically generate the Refined Software Safety Requirements ($RSSR$):

Rule 3: Each $RUCA_i$ in the context **Providing** (C_1) of control action CA_i can be transformed automatically into a new software safety requirement as follows:

$RSSR_i$ = <CA> **must Not be Provided** <TC> **when** <Cs= $\bigcup(\mathcal{P}_1 = v_1, \ldots \mathcal{P}_n = v_n)$> occurred.

Rule 4: Each $RUCA_i$ in the context **Not Providing** (C_2) of control action CA_i can be transformed automatically into a new software safety requirement as follows:

$RSSR_i$ = <CA> **must be Provided when** <Cs= $\bigcup(\mathcal{P}_1 = v_1, \ldots \mathcal{P}_n = v_n)$> occurred.

4.2. Automatically Formalizing Safety Requirements in Linear Temporal Logic (LTL)

In [Abdulkhaleq and Wagner 2015a], we described an algorithm to formalize the safety requirements in LTL. Here, we extend it to include software safety requirements that include timing. By using rules 3 and 4, each refined software safety requirement $RSSR_i$, which is identified from the refined unsafe control action $RUCA_i$, can be transformed automatically into a formal specification in LTL.

Rule 3 defines three types of software safety requirements, which means that the control action CA_i must not be provided in the type of context TC = **any time, too early or too late** when the critical combination Cs_i of the relevant process model variable values occurred. Each type of software safety can be transformed automatically into formal specification by the following rules:

Rule 3.1: Each $RSSR_i$ derived from the context of providing control action CA_i **any time** can be automatically transformed into LTL as:

LTL_i = G (Cs_i -> ! ($controlAction$ == CA_i)), where Cs_i= $\bigcup(\mathcal{P}_1 = v_1 \wedge \ldots \mathcal{P}_n = v_n)$.

Rule 3.1 means that it always (G) the software controller should not (!) provide a control action CA_i when the values of the critical combination Cs_i have been occurred.

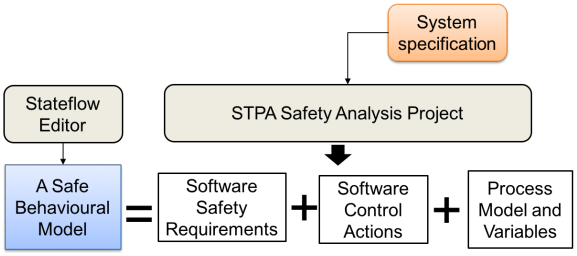

Fig. 3. A safe software behavioural model

Rule 3.2: Each $RSSR_i$ derived from the context of providing control action CA_i **too early** can be automatically transformed into LTL as:

LTL_i = G (((*controlAction* == CA_i) -> Cs_i) & ! ((*controlAction* == CA_i) U Cs_i)).

Rule 3.2 means that a software controller should always (G) not provide control action CA_i before the occurrence of critical combinations set Cs_i still not become true in the execution path.

Rule 3.3: Each $RSSR_i$ derived from the context of providing control action CA_i **too late** can be automatically transformed into LTL as:

LTL_i = G ((Cs_i-> (*controlAction* == CA_i)) & ! (Cs_i U (*controlAction* == CA_i))).

Rule 3.3 means that the software controller should always (G) not provide a control action CA_i while the occurrences of the critical set of combinations has become previously true in the execution path.

Rule 4 defines one type of the software safety requirements which is the context of not providing a control action CA_i when it is required. This type can be expressed into LTL by the following rule:

Rule 4.1: Each $RSSR_i$ derived from the context of **Not providing** of control action CA_i can be automatically transformed into LTL as:

LTL_i = G (Cs_i -> (*controlAction* == CA_i)), where Cs_i= $\bigcup(\mathcal{P}_1 = v_1 \wedge \ldots \mathcal{P}_n = v_n)$.
This rule means that the occurrence of a critical set of combination values always implies that the software controller must provide the control action CA_i.

4.3. Constructing a Safe Software Behavioural Model

To generate the safety-based test cases, the information derived from the STPA safety analysis must be integrated into a suitable model which should visualize the process model variables of each software controller and their relations in a control structure diagram. For this purpose, we select the stateflow [MathWorks 2016] diagram notations to visualize the automation model of each software controller. The stateflow diagram is a visual notation for describing dynamic behaviour, including the hierarchy, concurrency and communication information. The idea here is to build a model from STPA results with a modeling editor (e.g. Simulink) that supports the export of the statechart notations as XML specifications.

Definition 4.2 (*Safe Behavioural Model (SBM)*).
Let SBM be a Safe Behavioural Model (shown in Fig.3) which can be expressed by a three-tuple (PMV, T, CA), where PMV is a set or subset of software controller states S (process model variables), T is a set of transitions which are extracted from the refined software safety requirements $RSSR$, and CA is the set of the critical software control actions.

Each transition T_i of the safe behavioural model is expressed with the syntax T_i = ***IE [SSR] / TA***, where ***IE*** is the input event that causes the transition T_i, ***SSR*** is a safety requirement which is a Boolean condition that constrain the transformation from the current state to the next state, and ***TA*** is an action that will be executed when the Boolean expression is valid. Each state in the stateflow model has three optional types of actions: *Entry, During and Exist* actions. Entry actions execute when the state is entered, During actions execute when the state is active, an event occurs and no valid transition to another state is available, and Exit actions execute when the state is active and a transition out of the state occurs [MathWorks 2016]. These actions are used to determine how to change the current state of the software controller to the next state.

We identify the rules of constructing a safe software behavioural model from the STPA results as follows:

— The safe behavioural model should contain all internal state process variables of the software controller in the STPA control structure diagram: $PMV \subset S \in SBM$, where S is a set of software controller states.
— All process model variables of the software controller in the STPA control structure diagram should be declared in the safe behavioural model.
— The safe behavioural model should constrain the transitions using the STPA software safety requirements (constraints) which are identified based on the rules 3 and 4.
— Define an enumeration data type variable named *controlAction* in the safe behavioural model which takes all control actions of the software controller in the STPA control structure diagram as its value.
— The *controlAction* variable will be used as an entry action of internal states of the safe behavioural model to show which control action will be issued when the software controller enters a state.

4.4. Automatically Transforming a Safe Software Behavioural Model into an SMV Model

To check the correctness of the safe behavioural model and ensure that the safe behavioural model of the software controller satisfies all STPA software safety requirements, the safe behavioural model must be verified against the generated LTL formulae. For this purpose, we developed an algorithm that automatically transforms the SBM model created in the Simulink editor into an input language of a model checker such as SMV (Symbolic Model Verifier), automatically parses the LTL formulae from the STPA data model and includes them into an SMV model. To verify the SMV model against the STPA software safety requirements, we use the NuSMV model checker. In case that the SMV model does not satisfy a given LTL of a software safety requirement, the NuSMV model checker will produce a counterexample that contains the information where the model violates the given LTL. Based on the counterexample's information, the safe behavioural model diagram should be modified and transformed again into an SMV model until it satisfies all STPA-generated software safety requirements.

The algorithm of generating the SMV model is divided into three sub-algorithms: 1) *generate STPA data model* which parses XML specifications of the STPA project created in XSTAMPP into the corresponding Java objects that represent all STPA

ALGORITHM 1: Generate STPA Data Model

Input: P : A STPA file with extension .haz or hazx
Data: DM= A Java data model to store STPA results, CAs= a list of control actions, PMV= a list of process model variables and their values, $SSR =$ a list of software safety requirements, and $LTLs =$ a list of generated LTL formulae of SSR.
Output: DCs= a list of the data model of controllers $CO \in$ P.
Description:

1: **Parse** XML specifications of STPA project P into DM Java objects that represent all STPA data.
2: **Extract all data:** $DM \leftarrow P$
3: **for each** SW Controller in DM **do**
4: **Extract:** $CAs \leftarrow CA_i$,
$PMVs \leftarrow PMV_i$,
$SSRs \leftarrow SSR_i$,
$LTLs \leftarrow LTL_i$
5: **Add** $DC_i \leftarrow CAs$, $PMVs$, $LTLs$, and $SSRs$.
6: **Add** $DCs \leftarrow DC_i$.
7: **end for**
8: **Return** DCs

data (shown in algorithm 1); 2) *generate stateflow (safe behavioural model) data model* which parses XML specifications of a stateflow model into the Java objects that represent all stateflow data and generate a Tree of Stateflow states (TSf) in which a node represents one stateflow state (shown in algorithm 2); and 3) *generate SMV model* which transforms the STPA data model and stateflow data model into SMV specifications (shown in algorithm 3).

Algorithm 1 shows the process of parsing the STPA project created by XSTAMPP. To parse the XML specifications of the STPA project into Java objects, we use Java Architecture for XML Binding (JAXB) [Ort and Mehta 2003] technology. The algorithm process accepts the STPA project file P with extensions .hazx or .haz as input. Then, it parses the XML specification of the STPA project into the corresponding Java objects which represent all data in an STPA project. For each software controller in the control structure diagram, a Java Data Model object DM will be created to store the information about the software controller such as its critical control actions, process model and its variables, software safety requirements and the generated LTL formulae.

Algorithm 2 shows how to parse the XML specifications of the stateflow model stored in a Simulink/Matlab file. The input of this algorithm is an XML file of the Simulink stateflow file (Sf) which contains XML specifications of the stateflow model. To parse the stateflow file which we created in the Simulink editor with the extension *.slx*, we first generate XML specifications of the Simulink stateflow from the Simulink/Matlab editor by using the Matlab command:

```
save_system('Stateflow.slx', 'Output.xml', 'ExportToXML', true)
```

The structure of the stateflow model allows a multilevel hierarchy of states in which a state $S_{i,j}$ can contain sub-states with different types, where i inducts the number of the level hierarchy of the stateflow model ($i = 0...n$), j is the number of states, and n is the total number of levels in the stateflow model. Therefore, the process of algorithm 2 traverses recursively the stateflow data model object based on the depth-first search algorithm to consider all sub-states of the superstate and add them to the tree of stateflow. Each stateflow model has two kinds of state decomposition: OR states (exclusive) and AND (parallel) states [MathWorks 2016]. The stateflow semantics allow every state to has a state decomposition that indicates what type of sub-states the su-

ALGORITHM 2: Generate a Tree of Stateflow Data

Input: Sf : A Simulink stateflow file with the extension .xml
Data: DM_{Sf} = A Java data model to store all data of Stateflow in Sf, S= A list of states of stateflow Sf.
Output: T_{Sf}= a tree which represents all information of stateflow states $\in sf$.
Description:

1: **Parse** XML specifications of Sf into DM_{Sf} Java objects that represent all data of the Simulink stateflow model.
2: **Extract all data:** $DM_{Sf} \leftarrow Sf$
3: **Extract all states at level 0:**
$S \leftarrow DM_{Sf}$.Stateflow.getStates().
4: **Create a state root node** $\leftarrow root$
5: **Set ParentID** $root \leftarrow$ ParentID $\notin DM_{Sf}.States.IDs$,
Name $root \leftarrow$ 'root'
6: **for each State** s **in** S **do**
7: **Create a state child node** $\leftarrow node$
8: **Set ParentID** $node.parentID \leftarrow root.ID$.
9: **Set Data** $node.name \leftarrow s.name$, $node.Id \leftarrow s.SSID$,
$node.setDecomposition \leftarrow s.getDecomposition()$
10: **if** $s.hasChildren()$==true **then**
11: $node.isHasChildren(true)$
12: **traverseChildren (**$node$ **,** s **)**
13: **end if**
14: **Add** $root.addChild$ **(** $node$ **)**
15: **end for**
// Extract all transitions between the states.
16: T_{Sf}**.setTransitions(**$DM_{Sf}.getTransitions()$**)**
// Extract all variables of stateflow.
17: T_{Sf}**.setVariables(**$DM_{Sf}.getVariables()$**)**
T_{Sf}.root = $root$
18: **Return** T_{Sf}.

perstate can contain. All sub-states of a super-state $S_{i.j}$ should have the same type of decomposition of the parent state.

The algorithm for generating the tree of the stateflow (shown in algorithms 2 & 3) starts by parsing the XML specifications of the Simulink's stateflow Sf into the Java data model DM_{Sf}. The Java data model DM_{Sf} contains several Java classes as XML elements which have similar attributes of the XML elements in Sf. A tree stateflow object will be created to store a root node, a list of transitions and the list of the stateflow variables. As a stateflow model has no root state, a default node called $root$ will be created to store all information about the super-states at level 0 and assigned its *ParentID* randomly as an integer number that is not assigned to any state in the stateflow model. Each node stores the following data: id, *name*, *parentID*, T a list of transitions, a list of children (sub-states), the order of execution, a list of the state actions (entry, during and exit actions) and type of decomposition state (*OR State* or *AND State*). All superstates at level 0 in the stateflow model are added as the children of the default *root* node. For each state $S_{i,j}$, a node will be created to store all information of the state $S_{i,j}$. If the state $S_{i,j}$ has children, then all its substates will be traversed recursively until no more children exist for the superstate. Then, a state $node$ will be added as a child of the root node. The transitions at this level will be added to a transition list of the stateflow tree T_Sf to be used in the next algorithms: the *SMVGenerator* algorithm, Extended Finite State Machine model (*EFSMGenerator*) and a truth-table of the EFSM model generator.

ALGORITHM 3: traverseChildren(root, s)

Input: $root$: a root node in the tree T_{Sf},
s: a state in a satateflow data model DM_{Sf}
Description:

```
1: if s.hasChildren()==ture then
2:    for each State child in s.getChildren() do
3:       Create a new node node
4:       Set node.setName ← child.getName,
         nodel.setId ← child.getID,
         node.setParentID ← child.getParentID
5:       if child.hasChildren()== ture then
6:          node.setHasChildren (true)
7:       end if
8:          Add root.addChild(node)
9:          traverseChildren(node, child)
10:   end for
11: end if
```

```
1  MODULE main (<module variables>)
2   VAR
3   variables : <range data type>/<enumeration>
4   <nameSub1>: _SubModule1 (variables)
5   ...
6   <nameSubN>: _SubModuleN (variables)
7   states: <All children states>
8   ASSIGN
9    INIT (states=<initialState>)
10   INIT (<variable> =<value>)
11   next(<variable>):= case
12   <var1>=<value> & <tranConditon>:<nextValue>;
13   ...
14   next(states):= case
15   <states>=value & transition:nextState;
16   ...
17   esac;
18  LTLSPEC
19   <List of LTL formulae>
```

Fig. 4. The structure of the SMV model is generated from the stateflow tree and the STPA data object

Figure 4 shows the basic structure of the SMV model as described in [Cavada et al. 2010]. Each SMV module represents a super state in the stateflow model which can contain the following sections: 1) The name of the model with the optional state variable parameters, 2) The declaration of the state variable and their possible values, 3) The initial values of variables and the $states$ variable, 4) The sub-modules of the super module deceleration, 5) The transitions of the module, and a list of the LTL formulae. To represent the states of the stateflow model ($\simeq$ internal state variables of each controller in STPA) in an SMV model, we declare an enumeration variable called *"states"* which contains the names of sub-states of the super- state in the stateflow model.

Based on the principles of the SMV model [Cavada et al. 2010] and stateflow diagram [MathWorks 2016], we develop an algorithm to transform the stateflow (safe

behavioural model) and STPA data objects into an SMV model. Algorithm 4 shows the process of automatically transforming the safe behavioural model and the STPA data model into an input language of the SMV model checker. The algorithm traverses the states of the safe behavioural model recursively and generates the SMV model by parsing the hierarchical levels of the safe behavioural model. The inputs of the algorithm are a tree of stateflow model T_{Sf} which is created based on algorithm 2 and the STPA data model DCs_i of the software controller CO_i, which is generated based on algorithm 1 and a node n in the tree T_{Sf}.

The algorithm process starts by creating an object of the SMV model which is a Java class representing all structure data of the SMV model. The algorithm takes the root node of the safe behavioural model tree as the input at the first time to create the main module of the SMV model, then it declares the variables and maps their data types to the SMV data types. Secondly, the algorithm checks whether the root state $root$ has children states and which of them has children too. In case that a child $node$ of $root$ has children, then the algorithm declares a sub-module for this child $node$ as follows: $name$*: sub_Module (variables as parameters);*

Then, the algorithm takes all variables of the current state $node$ to create a list of the parameters of the sub- module. Next, the algorithm parses the sub-states of the super-state and creates the variable *"states"* with a list of the names of the substates as values. Then, the algorithm parses all local variables of the state as they are declared in the stateflow model. The SMV model does not support the same basic data types (int, double, single) as the data types which are declared in the stateflow model, it supports only a finite range type as integer range $min...max$ value. Therefore, the algorithm should map the data types (int, double or single) into a finite range which starts with 0 and ends with a maximum value of integer data type. The enumeration data types are declared into the stateflow model as a class which is saved in a separate file and not in the XML specifications of stateflow model. Therefore, the algorithm checks each variable with enumeration data type whether it is a process model variable in the STPA data model or not. In case the enumeration variable is a process model variable, the algorithm takes its values as they are defined in the STPA process model variable values. Otherwise, the algorithm creates an empty bracket {} for the values of the enumeration variable and prompts the user to determine the values of this variable. The algorithm will check whether the variables are declared exactly in the process model of the software controller in the STPA control structure diagram to reduce the time and effort of matching these variables during the verification step.

The algorithm will create the *initial* expression of the *"states"* variable. Each data variable will also be initialised with the minimum value of its data type (a variable with a numeric data type with zero, Boolean with FALSE and enumeration variable with the first value). Next, the algorithm will parse all transitions of the current state $node$ and create the *next* expressions for the *"states"* variable. The *next* expressions of $states$ variable refer to the transition relations of current state $node$ with other states in the model (the truthtable). The *next* expressions of the $states$ variable are expressed as follows:

```
next(states):= case
states=<substate> & transition : <nextstate>
...
1: {All sub-states}
esac;
```

To create the *next* expressions for each data variable, the algorithm parses the *entry, during and exit* actions of the current state and extracts all actions of each variable. The *next* expressions of the data variables refer to the values of variables in the next

state. The *next* expressions of each data variable are expressed as follows:

```
next(variable):= case
states = <state> & transition: <nextValue>
....
```

The algorithm will continue parsing the super-state in the tree of the safe behavioural model (stateflow) till all super-states have been visited. The generated SMV specifications of each sub-module and the main module will be saved as a string into a stack object. Finally, the algorithm will fetch the LTL formulae from the STPA data model object and add them at the end of the main-module section. To create the text file of the SMV model with extension **.smv*, the algorithm will read SMV data from a stack object and save it to a file.

To check the correctness of the generated SMV model and the safe behavioural model, we run the NuSMV model checker to verify the whether the SMV model contains errors and verify it against the STPA software safety requirements expressed in the LTL formulae and saved to the SMV model.

4.5. Automatically Constructing the Safe Test Model from the Safe Software Behavioural Model

To generate test cases for a system, the system behaviour should be modelled into a suitable model such as an extended finite state machine model. The Extended Finite State Machine (EFSM) [Harel 1987] is a common and very useful diagram to model the system behaviour and suitable for driving the test cases. EFSM contains nodes which represent the states of the system and the directed arcs which represent the transitions of the system from one state to another [Utting and Legeard 2007].

In [Harel 1987] Harel defined the statecharts language and the semantics of statecharts for complex systems. Simply, each stateflow has a chart which is an independent state machine. Each chart has one or more states which are linked together by arcs labeled with transition information. The states can be also hierarchical states and contains a number of sub-states (children). Each state should have a type of state decomposition *OR_STATE* or *AND_STATE*. The *OR_STATE* decomposition allows only one substate (which has a default transition) to be active, at a time when the parent (superstate) is active whereas the *AND_STATE* decomposition allows all sub-states to be active when the parent (superstate) is active.

After ensuring the correctness of the generated SMV model of the safe behavioural model (stateflow model), the safe behavioural model which uses the notations of the Simulink's stateflow should be transformed into the EFSM notation. For this purpose, we develop an algorithm based on the semantics of the stateflow and EFSM to map the stateflow tree of the safe behavioural model and its truth-table into an EFSM model. The algorithms 5 and 6 show the process of transforming the tree of the stateflow model into a EFSM model. The idea here is to eliminate the hierarchical and concurrent structure of the stateflow model (flattened and broadcast communication) and transform them into the EFSM notations by considering the state decomposition (exclusive or parallel).

The algorithm 5 starts by taking the root node of the stateflow tree T_{Sf} as the root node of the EFSM model and the truth table of the stateflow as the truth table of the EFSM model. The stateflow semantic supports multi-hierarchy levels of states, whereas the EFSM model does not. Therefore, the truth table of the EFSM model must not have any source or destination node as a super-state (a state that has children). The idea here is to investigate the truth table of stateflow and update the destination and source parent state with its substates. At the beginning, the algorithm checks

ALGORITHM 4: generateSMV(T_{Sf}, DCs_i, n)

Input: T_{Sf} : a tree data model of safe behavioural model,
DCs_i: a STPA data model of controller C_i, n: is a node in tree T_{Sf}.
Output: SMV_i: a SMVJava class represents the data of SMV.
Description:

Create a $SMV_i \leftarrow$ SMV model object
if (n.isRoot ()==true) **then**
  Set header of $SMV_i \leftarrow$ **'Module main'**
else
  $SMV_i \leftarrow$ **'Module'** root.getName() (root.getVariables)
end if
Set VAR section of $SMV_i \leftarrow$ **'VAR'**
Parse $SMV_i.setVariables() \leftarrow T_{Sf}.getVariables()$
Map data type of SMV variables into SMV data types.
if (checkInSTPA ($n.getVaraibles()$, DCs_i) **then**
  Declare each variable as $v.getName : var$.getType();
  if (n.isSubModule==true) **then**
    for each s $\in$ n.getChildren() **do**
      Declare "Sub_" + s.getName($n.getVariables()$)
    end for
  end if
  initial each v of SMV_i.InitialVariables $\leftarrow$
  init(v.getName()) :=initial_Value;
  Set ASSIGN section of $SMV_i \leftarrow$ **'Assign'**
  Parse Transitions $T \leftarrow n$.getTransitions()
  Set Next section of T of n state
  $SMV_i \leftarrow$ **'next'** (states) := case
  for each t $\in T$ **do**
    $states := t.Source\&t.Condition : t.Destination;$
    $TRUE : states; esac;$
  end for
  if (n.isRoot ()==true) **then**
    for each v $\in$ n.getVariables **do**
      $SMV_i \leftarrow$ **'next'** (v.getName) := case
      states=n.getSource(): n.getEntry(v.getName)
      TRUE: v.getName; esac;
    end for
  end if
  $SMV_i \leftarrow$ **'esac;'**
  if (n.hasChildren()) **then**
    for s $\in$ $root$.getChildren() **do**
      $SMV_i \leftarrow$ generateSMV(T_{Sf}, DCs_i, s)
    end for
  end if
else
  Show "STPA variables do not match Sf variables"
  $SMV_i \leftarrow$ null
end if
$SMV_i \leftarrow$ "LTLSPEC "DCs_i.getLTL()
Return SMV_i.

whether there is a super-state in the truth table. For each transition $t \in T$ in the truth table, the algorithm will identify its a source state and a destination state and check their decompositions as follows:

— If source state $src \in T_{sf}$ of transition t is a **super-state** with a state decomposition "OR_STATE" or "AND_STATE" and the destination node $dest \in T_{sf}$ is **not super-state**. Each sub-state of src state must be linked to the destination state $dest$ by creating a new transition with the same information of transition $T \in T_{sf}.TruthTable$ for each substate and only update the source with substate.
— If source state $src \in T_{sf}$ is **not super-state** and the destination state $dest \in T_{sf}$ is **super-state** with a state decomposition "OR_STATE". The default state $defaultState$ of super-state $dest$ (a default state is a state which has a default transition) should be identified. A new transition will be created and set its source as src and its destination as the default state of destination.
— If source state $src \in T_{sf}$ is **not super-state** and the destination state $dest \in T_{sf}$ is **super-state** with a state decomposition "AND_STATE". All sub-states of $dest$ state should be identified and linked with the source state. A new transition will be created for each substate of $dest$, where source is src and destination is the substate of destination.
— If source state $src \in T_{sf}$ is **super-state** with a state decomposition "OR_STATE" or "AND_STATE" and the destination state $dest \in T_{sf}$ is **super-state** with a state decomposition "OR_STATE". All sub-states of src state should be identified and linked with a default state of $dest$ state. A new transition will be created for each substate of src and its source is src and its destination is the default state of destination $dest$ state.
— If source state $src \in T_{sf}$ is **super-state** with a state decomposition "OR_STATE" or "AND_STATE" and the destination state $dest \in T_{sf}$ is **super-state** with a state decomposition "AND_STATE". All sub-states of src state should be identified and linked with all sub-states of $dest$ state. A new transition will be created for each substate of src and its source is src and its destination is the sub-state of destination $dest$ state.
— If source state $src \in T_{sf}$ is **not super-state** and the destination state $dest \in T_{sf}$ is **not super-state**. A transition t will be added into the truthtable.

The algorithm runs continuously till no superstate in the truth-table. All substates (without children) in the stateflow model tree will be taken as the states of the EFSM model. Also, all data variables of the stateflow model and the actions of the state (entry, exist, during) will be taken added into the states of EFSM. Please note that in this algorithm we did not consider the semantic of join transitions which are allowed in the stateflow semantics.

4.6. Automatically Generating Safety-Based Test Cases

The final step is to generate the test cases from the safe test model which are constructed from the safe behavioural model.

Kim et al. [Kim et al. 1999] describe how to generate test cases from Unified Modelling Language (UML) statechart diagrams. Based on this work, we develop an algorithm to derive the test cases from the EFSM with depth-first search and breadth-first search mechanism. The test data input of the test case generation algorithm are extracted from the stateflow model (all variables with input scope). For each input test variable, the algorithm prompts the user to identify the initial, minimum and maximum values.

Generating test cases from a model usually leads to an infinite number of possible test cases. Therefore, it is necessary to choose a suitable test coverage criteria to manage the generating process. In our algorithm, we identify three test coverage criteria: 1) *state coverage* which is the number of visited states divided by the total number of the states of the model, 2) *transition coverage* is the number of the executed transitions

ALGORITHM 5: GenerateEFSM(T_{Sf})

Input: T_{Sf} : a tree of stateflow model,
Output: $EFSM$: a Java object represent all data of EFSM
Description:

```
1: Create StateNode root ← T_Sf.getRoot()
2: Get TruthTable truthTable ← T_Sf.getTruthTable()
3: if root.hasChildren()==ture then
4:    Set Initial state ← T_Sf.getInitialState( )
5:    while isHasSuperState(truthTable) do
6:       for Transition t ∈ truthTable do
7:          StateNode src ← t.getSourceNode ()
8:          StateNode dest ← t.getDestinationNode ()
9:          if src.isSuper() & !(dest.isSuper()) then
10:            get children ← src.getChildren()
11:            for child ∈ children do
12:               updateTruthTable (child, dest, t, truthTable)
13:            end for
14:         else
15:            if !(src.isSuper()) & dest.isSuper() then
16:               get children ← dest.getSubSates();
17:               for child ∈ children do
18:                  updateTruthTable (src, child, t, truthTable)
19:               end for
20:            end if
21:         else
22:            if src.isSuper() & dest.isSuper() & OR_STATE then
23:               get srcChildren ← src.getSubSates();
24:               get def ← dest.getDefaultSate();
25:               for s ∈ srcchildren do
26:                  updateTruthTable (s, def, t, truthTable)
27:               end for
28:            end if
29:         else
30:            if src.isSuper() & dest.isSuper() & AND_STATE then
31:               get srcChildren ← src.getSubSates();
32:               get destChildren ← dest.getSubSates();
33:               for s ∈ srcchildren do
34:                  for d ∈ destchildren do
35:                     updateTruthTable (s, d, t, truthTable)
36:                  end for
37:               end for
38:            else
39:               if !(src.isSuper())& !(dest.isSuper()) then
40:                  updateTruthTable (src, dest, t, truthTable)
41:               end if
42:            end if
43:         end if
44:      end for
45:   end while
46: end if
47: Add EFSM.setTruthTable ← truthTable
48: Add EFSM.setStates ← T_Sf.getStates()
49: Return EFSM.
```

ALGORITHM 6: UpdateTruthTable(src, $dest$, t, $truthTable$)

Input: src : a source node of transition t, $dest$: a destination node of transition t, t : a transition in the truth table, $truthTable$: a truthTable of stateflow tree T_{sf}

Description:

1: **create** new Transition t_new
2: **set** data $t\_new \leftarrow t$
3: **update** t_new.setSrc(src)
4: **update** t_new.setDest(dest)
5: **add** $truthTable \leftarrow t\_new$

divided by the total number of the transitions, 3) *STPA safety requirements coverage* in which each STPA software safety requirement should be covered at least in one test case to trace how the STPA-generated software safety requirements are covered into the generated test cases. To measure the STPA SSR coverage , we define a *safety requirements traceability* matrix between the generated safe test model and STPA software safety requirements to manage the quality of the test case generating process and measure the coverage of STPA software safety requirements in the generated safety-based test cases. As the safe test model of the safe behavioural model is constrained with STPA safety requirements (step 2) and contains the process model variables as states, the algorithm will automatically generate the traceability matrix (TM= SSR $\times$ tn, where $SSR \in DCs$ of the STPA data model and TN transition conditions $\in T_{Sf}$).

Algorithm 7 takes the generated Safe Test Model (STM), a Traceability Matrix TM, a list of the test coverage criteria CC, a number of Test steps which is the total number of executions of the algorithm and a stop condition which is a test coverage criteria to stop the execution of the algorithm when it reaches 100%. The process of generating the test cases from the safe test model can be described as follows:

(1) The algorithm starts by selecting a random state as the start state and a state as the end state from the safe test model to generate all possible paths between them.
(2) A new test suite ts will be created to store all the generated test cases.
(3) Generate for each input data variable a random value between its minimum and maximum values which are identified by the user.
(4) By using the depth-first algorithm, all possible paths between the start and end states will be identified. The path here means a sequence of the visited states and their transitions. We also use the breadth-depth-first algorithm combined with depth first algorithm to identify all possible paths PT from start state to achieve a good test coverage criteria.
   — For each transition t in path $pt \in PT$, its transition condition will be transformed into a Java Script function. The test input variables in will be passed as an input of a Java Script function. To execute this function at the run time, we use the Java Script Engine which invokes the function with values of input data parameters and returns the result.
   — For each state s in path pt, the state actions (Entry, During, Exit) will be eliminated and transformed into Java Script functions. These will be executed to update the values of each local loc or output variable out of each state.
   — Create a new test case tc. Each test case will store the information about the sequence path pt such as: *id* is a number of the test case, *id_Ts* which is the number of the test suite, *id_SSR* which is the number of the software safety requirement, *preconditions and actions* which is the sequence of the local variables of states in the path pt and their updated values, and *postconditions* which is thesequence of output variables and their values.

ALGORITHM 7: Generate Safety-based Test Cases(STM, TM,CC, $TestSteps$, $StopConiditon$)

Input: STM : a safe test model extracted from SBM, TM: a traceability matrix, CC: is a list of the test coverage criteria, $TestSteps$ is the total number of execution algorithms, $StopCondition$: a condition to stop the execution process.
Output TS: a list of test suites, each test suite should contain a list one test case TC.
Description:

```
1: Initial step ← 0
2: while step < TestSteps do
3:    Choose start state ← STM.getRandomState()
4:    Choose end state ← STM.getRandomState()
5:    Create a new test suite ts
6:    if StopConiditon < 100.0% then
7:       Randomly Generate_Tes_InputData ()
8:       Walk TC_i ← GenerateTestCasesByDFS (start, end)
9:       Add ts ← TC_i
10:       Walk TC_j ← GenerateTestCasesByBFS (start)
11:      Add ts ← TC_j
12:   else
13:      if StopConiditon==100.0% then
14:         Calculate_Coverage_Criteria()
15:         STOP
16:      end if
17:   end if
18:   ADD TS ← ts
19:   Calculate_Coverage_Criteria()
20:   unvisitedTransitions(STM)
21:   unvisitedStates(STM)
22:   Initial step ← step + 1
23: end while
24: Return TS.
```

(5) Check whether the test case tc has been covered in any test suite. If it hasn't, tc will be added to the test suite ts.
(6) Calculate the test coverage criteria and check the stop condition of the algorithm.
(7) Change status of all states and transitions in the safe test model to unvisited to generate a new sequence path.
(8) The algorithm will be continued (repeat1-8) till the stop condition is achieved (100%) or the number of executions the algorithm has been reached to the total number of the test steps.

Ultimately, the time spent during test case generation process, the values of the test coverage criteria and a list of test suits and their test cases with the related software safety requirements will be automatically saved into a CSV file.

5. TOOL SUPPORT

Here we describe the implementation of the proposed approach for generating test cases based on the information derived from the STPA safety analysis. We use the previous algorithms and rules as the basis for implementing tool support for our safety-based test case generation approach.

To automatically formalize the STPA software safety requirements which are documented in XSTAMPP and transformed into LTL based on rules 1–4, we developed an Eclipse plug-in called XSTPA[2] based on the XSTAMPP architecture. XSTPA automat-

[2]http://www.iste.uni-stuttgart.de/se/werkzeuge/xstpa.html

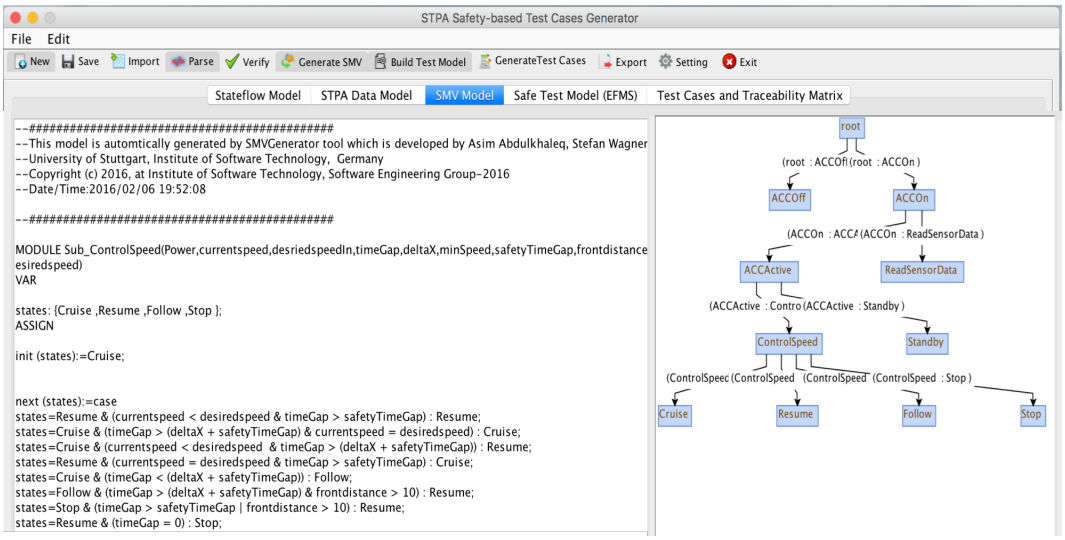

Fig. 5. STPA test case generator tool

ically generates the context tables (combinations between process model variables) by using a Java library for the combinatorial testing algorithm called ACTS[3] [Kuhn et al. 2013] which was developed by the American National Institute of Standards and Technology to generate combination sets of t parameters with n values. Based on rules 3 and 4, XSTPA automatically generates the LTL formulae. The generated LTL formulae will be used to check the correctness of the constructed safe test model which will be used to generate the safety-based test cases for each STPA-generated software safety requirement.

To generate the test cases based on STPA results, we implemented a tool support called *STPA Test Cases Generator* (STPA TCGenerator[4], shown in Fig.5) which parses the STPA file project created in XSTAMPP and the safe behavourial model which is created with Simulink's stateflow editor to generate the SMV model and check the correctness of the safe beahvioural model, eliminate the safe test model and generate safety-based test cases. The *STPA TCGenerator* tool accepts two files as input: an STPA project with extension haz. or .hazx and a stateflow model as xml file. The *STPA TCGenerator* parses the XML specifications of the STPA project and stateflow model into the corresponding Java objects by using Java Architecture for XML Binding (JAXB) [Ort and Mehta 2003] technology. We implemented a Java library called *SMV Generator* which contains all necessary methods for transforming the XML specifications of the STPA project and stateflow model into an SMV model. To check the correctness of the generated model against the LTL formulae of STPA software safety requirements, *STPA TCGenerator* uses the binary files of the NuSMV model checker to verify the generated SMV model.

For generating and visualizing the safe behavioural tree and the safe based model, we implemented a Java library called *buildTestGraph* which contains all the necessary methods and functions to visualize the safe test model and its truth table. *buildTestGraph* uses the Java library called JGraphT [5] to visualize the safe behavioural model

[3]http://csrc.nist.gov/groups/SNS/acts/index.html
[4]https://sourceforge.net/projects/stpastgenerator/
[5]http://jgrapht.sourceforge.net

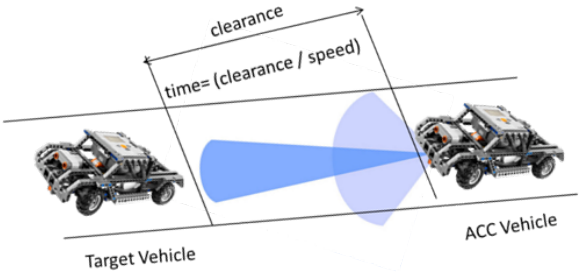

Fig. 6. A mechanism of the simulator of ACC with Stop and Go function

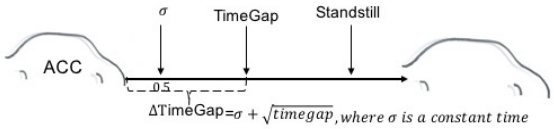

Fig. 7. The ACC system with a Stop and Go function scenarios

tree T_{sf}. *STPA TCGenerator* provides a test input data table which contains all input variables extracted from the safeflow model of SBM to allow the user to assign the initial, maximum, and minimum values for each variable.

To generate the test cases, *STPA TCGenerator* provides a test configuration view in which the user has to assign the test steps, select the test coverage criteria (state, transitions, STPA software safety requirements), and the test algorithm (depth-first search or breadth-depth-first search or both), and select a test coverage criterion as a stop condition. The generated test suites and the test cases will be viewed into a tree and saved automatically in an Excel sheet. The first prototype of *STPA TCGenerator* and the results of the illustrative example are available online in our repository[6].

6. AN ILLUSTRATIVE EXAMPLE: A SIMULATOR OF THE ACC SYSTEM WITH STOP AND GO FUNCTION

To illustrate the proposed approach, we developed a simulator software written in ANSI-C to simulate the adaptive cruise control system with stop and go function by using two LEGO EV3 Mindstorm robots[7]. The simulator was developed by a bachelor student within 6 months. The ACC with stop and go function [Venhovens and Naab 2000] is an extended version of the normal adaptive cruise control system. It maintains a certain speed and keeps a safe distance from the vehicle ahead based on the radar sensors. The ACC with stop and go function will bring the vehicle to a complete stop when the vehicle ahead comes to a standstill or there is a stationary object in the lane.

Figure 6 shows the mechanism of the simulator of the ACC with stop and go. The ACC simulator maintains a constant time gap to vehicles ahead. It uses a forward ultrasonic sensor with a range of up to 255 centimeters, which is located in the front of the robot to detect the distance of the robot ahead of it and can automatically maintain the pre-set time gap. It adjusts the robot speed by increasing or decreasing the value of current speed to keep a safe distance. If the robot ahead is completely stopped, then the ACC simulator will slow down the robot vehicle to a standstill. If the vehicle ahead starts moving again, then the ACC simulator will automatically start to move again

[6]https://sourceforge.net/projects/stpastgenerator/.
[7]http://www.iste.uni-stuttgart.de/en/se/forschung/werkzeuge/acc-simulator.html

Table I. Examples of potentially unsafe control actions of the ACC software controller

Control Action	Not providing causes hazard	Providing causes hazard	Wrong timing or order causes hazard	Stopped too soon or Applied too long
Accelerate Speed	The ACC software does not accelerate the speed when the robot vehicle ahead is so far in the lane. [**Not Hazardous**]	**UCA-1.1**: The ACC software accelerates the speed of robot unintendedly when the time gap to the robot vehicle ahead is smaller than the desired time gap. [**H-1**] [**H−2**]	**UCA-1.2**: The ACC software accelerates the speed before the robot vehicle ahead starts to move again. [**H-1**] [**H-2**]	**UCA-1.3**: The ACC software accelerates the speed too long so that it exceeds the desired speed of the robot. [**H-2**]
Decelerate Speed	**UCA-1.5**: The ACC software does not decelerate the speed when the robot vehicle ahead is too close in the lane. [**H-1**]	The ACC software decelerates the speed of robot unintendedly when the time gap to the robot vehicle ahead is larger than desired time gap. [**Not Hazardous**]	The ACC software decelerates the speed when the robot vehicle ahead starts to move again. [**Not Hazardous**]	**UCA-1.6**: The ACC software decelerates the speed long enough so that it cannot bring the robot to fully stop when the robot ahead is stopped. [**H-3**]
FullyStop	**UCA-1.4**: The ACC software does not bring the robot to a complete stop at a standstill when the robot vehicle ahead is fully stopped. [**H-1, H-3**]	The ACC software stops the robot suddenly when the distance to the robot ahead is too close. [**Not Hazardous**]	The ACC software does not accelerate the speed after the robot vehicle ahead starts to move again. [**Not Hazardous**]	N/A

and maintain a constant time gap between the robot ahead. Our simulator algorithm is the ACC simulator starts first read the distance data from the ultrasonic sensor and then compute the time gap by using the following equation:

$$currentTimegap = |\frac{Frontdistance}{CurrentSpeed}| \tag{1}$$

Second, the simulator computes the standstill time, which is the time at which the ACC vehicle must decrease the speed or stop when the vehicle ahead is close or fully stopped. It is calculated as

$$\Delta Timegap = stillstandtime + \sqrt{currentTimegap} \tag{2}$$

Third, the simulator will compare the value of the time gap with the following scenarios (shown in Fig.7):

— *TimeGap* $>$ *(*Δ*TimeGap + safeTimeGap)*. This indicates that the vehicle ahead is so far from the point t_σ. The simulator will accelerate the speed of the vehicle robot till the desired speed. The simulator adjusts (increase/decrease) the current speed by using the following equation:

$$currentSpeed + /- = \sqrt{speed^2 + 2 * (Time)}, \tag{3}$$

where Time = ((ΔTimegap + safeTimeGap) - TimeGap)

— *(TimeGap* $>$ *safeTimeGap)* && *(timeGap* $<$ *(*Δ*TimeGap + safeTimeGap))*. This indicates that the vehicle robot ahead is approaching within the period of time gap between $[t_\sigma\ t_{safeTimeGap}]$. The simulator will put the ACC system in *follow* mode.

Follow mode means that there is a vehicle in front in the lane. The simulator will automatically adjust the current speed by using the equation 3.

— *TimeGap* == *safeTimeGap*. This indicates that the vehicle robot ahead is approaching within the desired time gap and there is a safety distance between them. The simulator will put the ACC system in the cruise mode. *Cruise* mode means that the vehicle robot ahead is approaching in safe time gap. Then, the simulator will set the current speed as the desired speed.

— *TimeGap* < *safeTimeGap*. This indicates that the vehicle ahead is moving within the time between $[t_{safeTimeGap}\ t_0]$. The simulator will reduce the speed of the vehicle by using the equation 3.

— *TimeGap* == *0*. This indicates that the vehicle ahead has fully stopped. Then the simulator will bring the vehicle to a complete stop at the standstill distance and change the ACC mode to stop. If the front vehicle starts to move again, then the simulator will change the ACC mode to resume. *Resume* mode means that the current speed of the ACC vehicle will be accelerated to the desired speed. The simulator uses the following equation to achieve that:

$$currentSpeed+ = accelerationratio, \tag{4}$$

where the accelerationratio is set to 4 cm/sec;

6.1. Deriving Software Safety Requirements of the ACC Simulator

To derive the software safety requirements, we applied the STPA SwISs Step 1 to the system specification requirements. We used the XSTAMPP software tool to document the results of STPA and generate the formal specification of the STPA results. The results are saved in a STPA project file called *ACCSimulator.hazx*.

As a result, we identified the system-level accidents that the simulator software can lead (or contribute to). For example, **ACC-1 : The ACC robot crashes the robot ahead**. The system-level hazards which can lead to this accident are:

— H_1: The ACC software does not keep a safe distance from the a vehicle robot ahead.
— H_2: The ACC software provides an unintended acceleration when the vehicle in front is too close.
— H_3: The ACC software does not stop the vehicle when the vehicle ahead is fully stopped.

We built the control structure diagram of the ACC simulator (shown in Fig. 8). It contains the main interconnecting components of the ACC simulator at a high level, such as the *ACC simulator software controller unit*, *the electronic motors*, *the robot vehicle* as the controlled process, and *the Ultrasonic and speed sensors*. The ACC software controller receives the distance data from the ultrasonic sensor and current speed data from the speed sensor. Based on this information, the software will calculate the time gap and determine if the vehicle robot ahead is present. The ACC software will adjust the speed of the robot based on the above sensors and issues one of the critical safety control action: accelerate, decelerate, or fullystop. Each one of these control actions will be evaluated based on the four general hazardous types (columns of table I). Table I shows the examples of the potential unsafe control actions of the ACC simulator.

We evaluated each item in table I to check whether it can contribute or lead to any system-level hazards ($H_1 - H_3$). If an item is hazardous, then we assign one or more system-level hazards to it. We translate each hazardous item manually to the corresponding software safety requirement by using the guide words, e.g., *have to, must be, or should*. Table II shows examples of the informal textual software safety requirements.

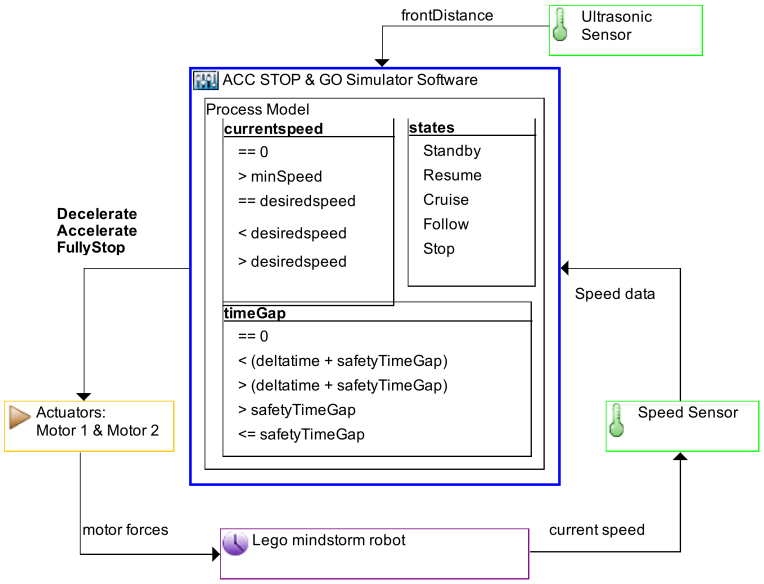

Fig. 8. The control structure diagram of ACC with the safety-critical process model variables

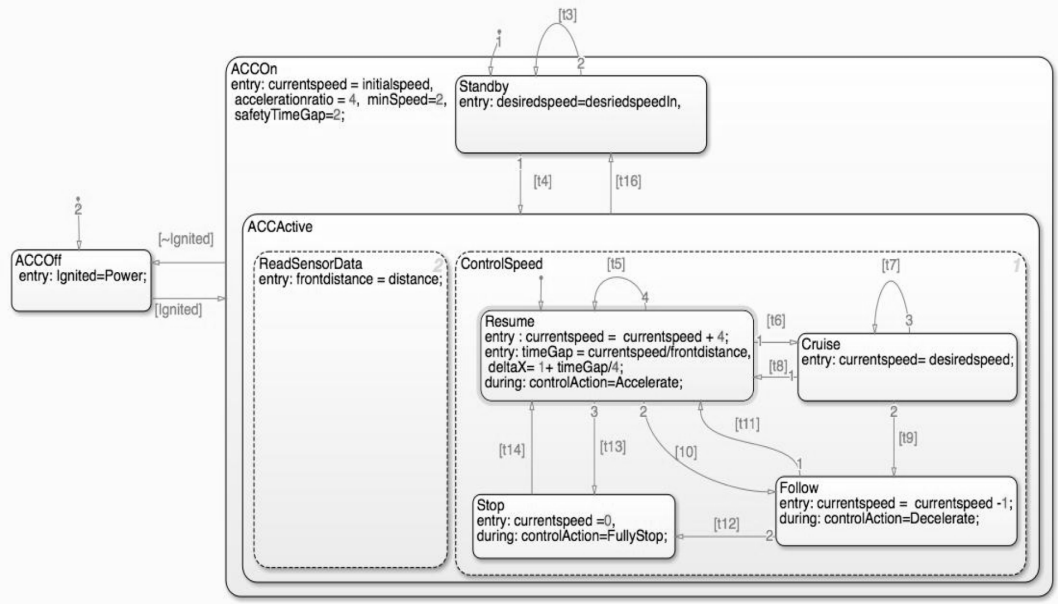

Fig. 9. The safe behavioural model of the ACC software controller

To refine the informal textual software safety requirements which are shown in table II, we identified the process model of the ACC software controller and its critical variables which have an effect on the safety of the ACC software control actions. Figure 8 shows the control structure diagram and process model variables of the ACC software. The ACC software has three safety-critical process model variables: *Internal variables* such as currentSpeed (5 values), Timegap (5 values), *Internal states variable* such as

Table II. Examples of software safety requirements at the system level

Related UCAs	Corresponding Safety Constraints
UCA-1.1	**SSR1.1**- ACC software should not accelerate the speed of the robot when the target robot vehicle is too close in the lane.
UCA-1.2	**SSR1.2**- ACC software should not accelerate the speed when the robot ahead is fully stopped.
UCA-1.3	**SSR1.3**-ACC software should not increase the speed than the desired speed.
UCA-1.4	**SSR1.4**-ACC controller should stop the robot at standstill point (shown in Fig. 7) when the robot ahead is fully stopped.

Table III. Examples of the context table of *providing* the control action *accelerate*

Control Actions	Process Model Variables			Is it a hazardous Control Action?
accelerate	CurrentSpeed	TimeGap	ACC Mode	providing
	CS> minSpeed	TimeGap <(Δ Timegap + safetyTimeGap)	follow	No
	CS<=desiredSpeed	TimeGap == 0	follow	Yes, H2, H1
	CS<desiredSpeed	TimeGap >safetyTimeGap	follow	No
	CS<desiredSpeed	TimeGap < (Δ TimeGap + safetyTimeGap)	follow	Yes

Table IV. Examples of refined software safety requirements in XSTAMPP based on critical combinations

Related UCAs	Refined Safety Constraints
RUCA-1.1	**RSSR1.1**- Accelerate command must not be provided when ACC mode is Standby and timeGap is greater than (deltaX + safetyTimeGap) and the current speed is less than desired speed.
RUCA-1.2	**RSSR1.2**- Accelerate command must not be provided when timeGap is less than (deltaX +TimeGap).
RUCA-1.3	**RSSR1.3**-Accelerate command must not be provided when current speed is greater than or equal to desired speed.
RUCA-1.4	**RSSR1.4**-FullyStop command must provided when the timeGap is 0.
RUCA2.1	**RSSR2.1**- Decelerate command must not be provided too late when ACC mode is follow and timeGap is less than safetyTimeGap and currentSpeed is greater than desired speed.

ACC mode (states) with 5 values, and *the environmental variables* such as front distance. Each safety control action provided by the ACC software should be evaluated to determine whether it will be hazardous or not when the combination set of relevant values of the process model variables (context) occur.

We used XSTPA to generate the critical combinations (context tables) for each safety-critical action in the two contexts *when the control action is provided* and *it is not provided* and causes hazard. For each control action, the total number of combinations between the process model variables is ($5 \times 5 \times 5$ =125) combinations. We reduced the number of combinations by applying pairwise test coverage to the generated combination sets. The number of critical combinations is reduced to 25 for each control action.

Based on the generated combination sets, we evaluated each control action in two contexts *Providing* and *Not Providing*. Table III shows examples of the context table of providing the control action *accelerate* based on the combinations of the values of the critical process model variables. As a result, we identified 32 unsafe scenarios for all

Table V. Examples of LTL formulae of the refined software safety requirements at the system level

Refined SSRs	Corresponding LTL formula
RSSR1.1	**LTL1.1**- [] ((state=Standby) && (timeGap > deltaX+safetyTimeGap) && (currenSpeed<desiredSpeed) -> ! (controlAction==Accelerate)).
RSSR1.2	**LTL1.2**- [](currentSpeed > desiredSpeed) && (TimeGap <(deltaTime + safetyTimeGap)) $->$! (controlAction==Accelerate).
RSSR-1.3	**LTL1.3**-[](currentSpeed >=desiredSpeed) $->$! (ControlAction==stop) .
RSSR-1.4	**LTL1.4**- [](timeGap $==0$) $->$ (controlAction==FullyStop).
RUCA2.1	**LTL.2.1**- [] ((state==Follow) && (timeGap <safetyTimeGap) && (currentSpeed >= desiredSpeed) $->$!(controlAction==Decelerate))

the control actions *accelerate (18 scenarios)*, *decelerate (7 scenarios)* and *FullyStop (7 scenarios)*.

From the critical combinations of each control action, we used XSTAMPP to automatically refine the informal textual software safety requirements into formal textual software safety requirements (shown in Table IV). Based on the rules 3-4, XSTAMPP also automatically generates the LTL formula for each refined software safety requirement. Table V shows the examples of the corresponding LTL formula of each software safety requirement. We used the generated-LTL formulae to verify the safe behavioural model which is constructed from the STPA results.

6.2. Automatically Generating SMV Model

We created a Simulink/Matlab stateflow model to visualize a safe behavioural model of the ACC simulator (shown in Fig. 9). The safe behavioural model is saved in a Simulink file called *ACCSimulator.slx*. It contains 10 states (2 of them are decomposition with AND_STATE) and 20 transitions. It shows the relationship between the process model variables in the safety control structure diagram which describes the critical states of the software and how the software issues the critical safety control actions(e.g. accelerate, decelerate, etc.)

To validate the correctness of the safe behavioural model, we generated a verification input of the NuSVM model checker. For that, we first derived the XML specifications of *ACCSimulator.slx* Simulink's stateflow file. The XML specifications are saved in an XML file called *ACCSimulator.xml*. Second, we took the STPA project file ACCSimulator.hazx and *ACCSimulator.xml* as input to the tool *STPA TCGenerator*. The tool parses both files and generates the SMV model which maps all states, transitions and data variables, and LTL formulae of STPA software safety requirements of the safe behavioural model to SMV model specifications and automatically saved them into a file named as *ACCSimulator.smv*.

We updated the default values of each input data variable which are declared in the generated SMV model (e.g. *initial speed (10.0), desired speed (45.0), initial front-distance (150.0)*). The STPA TCGenerator tool runs the NuSMV 2.6.0 model checking tool to verify the generated SMV model file. The NuSMV model succeeded in verifying the generated SMV model within 0.29 seconds and no further errors were reported. NuSMV consumed 42.10 megabytes to store 2.31828e+17 states and performed 2.97418e+09 transitions. As a result, all LTL formulae were satisfied and there is no counterexample generated because the safe behavioural model itself was built from STPA software safety requirements.

6.3. Safety-based Test Case Generation from the Safe Test Model

After validating the correctness of the safe behavioural model, we used the STPA TC-Generator to generate a hierarchical tree of the safe behavioural model which shows

```
1  [Test Case ID] 2
2  [Test Suite ID] 2
3  [Related STPA SSRs]
4    SSR4, SSR6, SSR15, SSR16
5  [PreConditons]
6    desiredSpeed=45.0
7    frontdistance=120.32
8    currentSpeed=44.0
9    state=Resume
10 [Actions]
11   controlAction=Accelerate
12 [PostConditons]
13  currentSpeed=45.0
14  state=Cruise
15 [Comment]
```

Fig. 10. An example of a generated safety-based test case

Table VI. The safety-based test cases generated by STPA TCGenerator tool

ID	Test Algorithm	Test Steps	Test Suite	Test Cases	Time (in Sec)	State Coverage	Transition Coverage	STPA SRR Coverage
1	DFS	10	1	119	3	6/7 = 85.7%	23/32=71.9%	32/32=100%
2	BFS	10	4	24	1	6/7 = 85.7%	17/32= 53.1%	32/32=100%
3	Both	10	5	249	2	7/7 = 100%	18/32= 87.5%	32/32=100%

the hierarchy levels of the safe test model. The *STPA TCGenerator* tool parses the tree of the safe behavioural model recursively by considering super state decompositions *AND_STATE* (parallel) and *OR_STATE* (exclusive) to automatically generate the safe test model as an extended finite state machine. As a result, the generated safe test model contains 7 states (after removing the super states) and 32 transitions (after maintaining the transitions of super states). The tool automatically generates the traceability matrix between STPA software safety requirements and the safe behavioural model and shows them in a table. All input data variables with their data type, initial, minimum, maximum values which are shown in the test input configuration view.

Before running the tool to generate the test cases, we set the number of test steps to 10 and selected the three test coverage criteria (state, transition and STPA software safety requirements test coverage criteria). We selected the STPA software safety requirements coverage as the stop condition of the test case generating algorithm. We also set the test input value for each input data variable: *power* (true), desired speed (45 cm/sec), initial speed (10 cm/sec), front distance (150 cm). Finally, we ran the STPA TCGenerator tool three times to generate safety-based test cases from the test model, respectively : 1) depth-first search, 2) breadth-depth-first search and 3) the combined algorithm. Table VI shows the results of the generated safety-based test cases by each test algorithm. We could achieve 100% coverage of all the STPA software safe requirements which are linked to the safe test model in the traceability matrix. Figure 10 shows an example of the format of documenting each safety-based test case.

Based on the traceability matrix between the model and the STPA software safety requirements, the *STPA TCGenerator* provides an *individual coverage* (how many test cases TC covered each SSR) by each test algorithm (shown in Fig. 11).

7. DISCUSSION

The manual construction of test cases is a hard, time-consuming and error-prone activity that requires deep knowledge and expertise. Furthermore, the manual building of a test model from system specifications with the purpose of generating test cases still needs a proof of its correctness to ensure that the test model captures all specifications. One solution is to automatically construct a test model for a given system and prove its correctness by transforming it into an intermediate model that can be used as an input model of a formal verification approach (e.g. model checker) to verify the generated model against its specifications. In addition, the specifications should also be mapped from informal text to the formal specifications. For this issue, we transformed the test model into the SMV model and verified it by using the NuSMV model checker tool. However, the model transformation process also needs a proof of the correctness of the resultant model, even though the model checker did not induct any error. In our proposed approach, this issue remains as an open issue for future work.

Traditionally, formal verification and testing are used to assess the functional requirements and they do not directly concern safety. The idea behind the proposed approach is to concentrate on the use of the STPA safety analysis to identify the hazardous situations that the software can lead or contribute to and formalise the STPA software safety requirements into a formal specification. The information derived from the STPA safety analysis will be modelled into the test model.

A second issue is that the automation of the test case generation process can lead to a large number of test cases that cover the same information. Reducing the number of generated test cases is a major factor in evaluating the effectiveness of an automated testing tool and the quality of the generated test cases. Therefore, we added a new test coverage criteria (e,g. STPA software safety requirements) to stop the test case generating algorithm when this criteria becomes 100% to ensure that each STPA safety requirement is covered at least in one test case. Furthermore, the first prototype of the *STPA TCGenerator* tool supports to generate test cases for each software safety requirement by automatically generating a traceability matrix between the STPA software safety requirements and the safe test model. The traceability matrix contains all relevant transitions of each software safety requirement in the safe test model.

A limitation is that the process model variables in the STPA control structure diagram visualized by XSTAMPP have no data types. Furthermore, XSTAMPP does not support multi-levels hierarchies of the process model of the software controller in the control structures. That makes ensuring and checking the consistency between the hierarchy levels of the process model in STPA and the stateflow model in Simulink a big challenge. For example, the process model variable *ACC* Active in the *ACC* software controller has sub-process model variables such as control speed and *read sensor data* which will be activated when the ACC state is active. Therefore, it requires human effort to define the process model hierarchy and map it to the Simulink stateflow model hierarchy level. Another limitation is that the semantics of the stateflow model saves the information inside the state (e.g. the name of the state and its entry, during, exit actions) in one attribute as a string, which makes the process of extracting information very buggy if this information is not separated by semicolons and requires an effort to trace them.

8. CONCLUSION

In this paper, we introduced an automatic approach to generate safety-based test cases based on the STPA safety analysis. Our approach concentrates on generating a set of test cases for each STPA software safety requirement. The generated test cases will be used to verify the safety of the software-intensive system under analysis. We

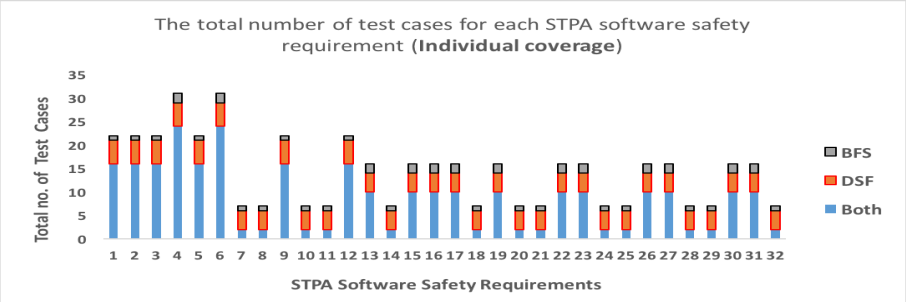

Fig. 11. The total number of test cases for each STPA software safety requirement

also implemented an open-source tool support that automates the safety-based test cases generating approach. Furthermore, we illustrated the proposed approach with safety-critical software of an ACC system with stop-and-go function. The results show that deriving test cases based on the safety requirements is a practical and effective approach to generate different test cases to recognize software risks and assure the software quality.

As a future work, there are many interesting directions and trends to extend the research of safety-based testing for software-intensive systems and the automated tool support. We plan to improve the tool by considering the other stateflow semantics which were not addressed in our approach such as inner transitions and connective and history junctions transitions. Furthermore, we aim to limit the number of the generated test cases, to improve the traceability matrix by adding information about the maximum number of test cases for each software safety requirement and also the priority value to generate a reasonable test case for each software safety requirement.

To support software and safety engineers who use the XSTAMPP platform, we plan to integrate it into the XSTAMPP platform to derive the test cases for each STPA software safety requirement within the XSTAMPP platform. Furthermore, we plan to improve the process model in the control structure diagram by allowing the safety analyst to define the data type of each process model variable and draw the multi-hierarchy levels of the process model variables. Finally, we plan to evaluate the proposed approach and the tool support on a real software-intensive system with an industrial partner.

ACKNOWLEDGMENTS

The authors would like to thank Prof. Nancy Leveson, MIT, for her very careful review of our paper, and for the comments, corrections and suggestions that ensued.

We would also like to express gratitude to Lukas Balzer, University of Stuttgart, who worked with us to improve and build the XSTAMPP platform; Yannic Sowoidnich, University of Stuttgart, who developed XSTPA; Rick Kuhn, National Institute of Standards and Technology, USA, who provides us the Automated Combinatorial Testing Tool (ACTS). We are grateful for their help, effort and time; and Kornelia Kuhle, University of Stuttgart, for her feedback on the text.

REFERENCES

Asim Abdulkhaleq and Stefan Wagner. 2015a. Integrated Safety Analysis Using Systems-Theoretic Process Analysis and Software Model Checking. *In Proc. 2015 SAFECOMP Computer Safety, Reliability, and Security* (2015).

Asim Abdulkhaleq and Stefan Wagner. 2015b. XSSTAMPP: An eXtensible STAMP platform as tool support for safety engineering. In *2014 STAMP Conference, MIT*.

Asim Abdulkhaleq, Stefan Wagner, and Nancy Leveson. 2015. A Comprehensive Safety Engineering Approach for Software-Intensive Systems Based on STPA. *Procedia Engineering* 128 (2015), 2 – 11. Proceedings of the 3rd European STAMP Workshop 5-6 October 2015, Amsterdam.

Larry Apfelbaum and John Doyle. 1997. Model Based Testing. In *Software Quality Week Conference*. 296–300.

Roberto Cavada, Alessandro Cimatti, A. Charles Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri, and Andrei Tchaltsev. 2010. NuSMV 2.6 User Manual. (jan 2010).

Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. 1999. NUSMV: A New Symbolic Model Verifier. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV '99)*. Springer-Verlag, London, UK, UK, 495–499.

S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. 1999. Model-based Testing in Practice. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 99)*. ACM, New York, NY, USA, 285–294.

David Harel. 1987. Statecharts: a visual formalism for complex systems. *Science of Computer Programming* 8, 3 (1987), 231 – 274.

SAE International. 1967. *Society for Automotive Engineers, Design Analysis Procedure for Failure Modes, Effects and Criticality Analysis (FMECA), ARP926*. Warrendale, USA.

JPL. 2000. Report of the Loss of the Mars Polar Lander and Deep Space 2 Missions. (2000).

Y.G. Kim, H.S. Hong, D.-H. Bae, and S.D. Cha. 1999. Test cases generation from UML state diagrams. *Software, IEE Proceedings* - 146, 4 (Aug 1999), 187–192.

J. Kloos, T. Hussain, and R. Eschbach. 2011. Risk-Based Testing of Safety-Critical Embedded Systems Driven by Fault Tree Analysis. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. 26–33.

D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. 2013. *Introduction to Combinatorial Testing* (1st ed.). Chapman & Hall/CRC.

Nancy Leveson. 2000. Completeness in Formal Specification Language Design for Process-control Systems. In *Proceedings of the Third Workshop on Formal Methods in Software Practice (FMSP '00)*. ACM, New York, NY, USA, 75–87.

N.G. Leveson. 2011. *Engineering a Safer World: Systems Thinking Applied to Safety*. MIT Press.

Robyn R. Lutz. 2000. Software Engineering for Safety: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE '00)*. ACM, New York, NY, USA, 213–226.

MathWorks. 2016. The MathWorks, Inc. Simulink, 2015. Version R2015b. (Jan 2016).

Kenneth L. McMillan. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA.

Minister of Defence. 1991. Hazard Analysis and Safety Classificaiton of the Computer and Programmable Electronic System Elements of Defence Equipment, Interim Defence Standard 00-56, Issue 1. (1991).

NASA-GB- 8719.13. 2004. *NASA Software Safety Guidebook*. NASA.

Ed Ort and Bhakti Mehta. 2003. Java Architecture for XML Binding (JAXB). (mar 2003).

Amir Pnueli. 1977. 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, 46–57.

Felix Redmill. 2004. Exploring Risk-based Testing and Its Implications: Research Articles. *Softw. Test. Verif. Reliab.* 14, 1 (March 2004), 3–15.

John Thomas. April 2013. *Extending and automating a systems-theoretic hazard analysis for requirmenets genreation and analysis*. dissertation. MIT.

Mark Utting and Bruno Legeard. 2007. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

P. Venhovens and Adiprasito B. Naab, K. 2000. Stop and go cruise control. *Seoul 2000 FISITA World Automotive Congress* 2000 (jun 2000), 396.

W. Vesely, F.F. Goldberg, N.H. Roberts, and D.F. Haasl. 1981. *Fault Tree Handbook NUREG-0492*. U.S. Nuclear Regulatory Agency, Washington.

F. Zimmermann, R. Eschbach, J. Kloss, and T. Bauer. 2009. Risk-based Statistical Testing: A refinement-based approach to the reliability analysis of safety-critical systems. *Proceedings of the Spring TAV Workshop* (2009).