Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 235

Development of an Eclipse Plug-in for the STPA TCGenerator Tool

Ting Luk-He

Course of Study:	Informatik
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	M.Sc. Asim Abdulkahleq
Commenced:	May 9, 2016
Completed:	November 8, 2016
CR-Classification:	D.2.2, I.3.4, I.1.2

Abstract

Software safety is becoming one of the most important topics in modern systems, as software plays an important role in main critical-functions of systems. However, not only a software functional error, but also an unexpected software behavioral flaw may lead to catastrophic results. Therefore, software must be tested, which includes not only functionality requirements, but also software safety requirements. Based on the concept of STPA SwISs [4], the software XSTAMPP [3] and STPA TCGenerator [1] were developed each in Eclipse and Netbeans by Abdulkhaleq and Wagner. XSTAMPP provides a base platform for safety engineering, which can be easily extended with new functions and approaches based on the STAMP model. STPA TCGenerator is a tool for automatically generating software test cases from software safety requirements. This work presents an extended plug-in STPA TCGeneratorPlugin for XSTAMPP based on the tool STPA TCGenerator. The main functions of STPA TCGenerator are 1) automatically generating the safe behavioral model and the SMV model, 2) verification of SMV model, 3) build safe test model and 4) automatically generating safety-based test cases. STPA TCGeneratorPlugin contains all functions of STPA TCGenerator and can be integrated into XSTAMPP, so that the safety-based test cases can be generated directly in XSTAMPP rather than using a different tool. Furthermore, several improvements for STPA TCGenerator are implemented, such as graphical visualization, algorithm of generating traceability matrix and representation of test case results.

Kurzfassung

Softwaresicherheit ist zu einem der wichtigsten Themen in modernen Systemen geworden, da die Software eine wichtige Rolle in den kritischen Funktionen der Systeme spielt. Jedoch kann nicht nur ein Softwarefehler, sondern auch ein unerwarteter Softwareverhaltensfehler zu katastrophalen Ergebnissen führen. Daher muss Software getestet werden, welche neben den Funktionalitätsanforderungen auch die Softwaresicherheitsanforderungen umfasst. Demzufolge wurde die Software XSTAMPP [3] und STPA TCGenerator [1] auf Basis des Konzepts von STPA SwISs [4] in Eclipse und Netbeans entwickelt. XSTAMPP bietet eine Basisplattform für die Sicherheitstechnik, die mit neuen Funktionen und Ansätzen auf Basis des STAMP-Modells leicht erweitert werden kann. STPA TCGenerator ist ein Werkzeug für die automatische Erzeugung von Softwaretestfällen aus Softwaresicherheitsanforderungen. Diese Arbeit stellt ein erweitertes Plug-in STPA TCGeneratorPlugin für XSTAMPP basierend auf dem Tool STPA TCGenerator vor. Die wichtigsten Funktionen von STPA TCGenerator sind 1) die automatische Erzeugung des sicheren Verhaltensmodells (Safe Behavioral Model) und des SMV-Modells,

2) die Überprüfung der SMV-Modell, 3) die Zusammenstellung sicherer Testmodelle (Safe Test Model) und 4) die automatische Erstellung der sicherheitsbasierten Testfällen. STPA TCGeneratorPlugin enthält alle Funktionen von STPA TCGenerator und kann in XSTAMPP integriert werden, so dass, anstatt mit einem anderen Tool, die sicherheitsgerichteten Testfälle direkt in XSTAMPP erzeugt werden können. Darüber hinaus sind einige Verbesserungen für STPA TCGenerator implementiert worden, wie z.B. die grafische Visualisierung, der Algorithmus zur Erzeugung der Traceability Matrix und die Darstellung von Testfallergebnissen.

Acknowledgment

I would first like to thank my supervisor Asim Abdulkhaleq, for his very careful review, his comments and his supports during my bachelor thesis. Abdulkhaleq's office was always open, whenever I ran into a trouble spot or had a question about my project or writing. He consistently allowed this work to be my own work, but steered me in the right direction whenever he thought I needed it.

I would also like to thank Prof. Stefan Wagner, for his generous advice, inspiring guidance and encouragement throughout my project for this work.

A special thank of mine goes to Lukas Balzer, who always gave me a lot of help when I had a problem with specific programming issues.

Finally, I must express my very profound gratitude to my parents and to my partner for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of the project and writing this thesis.

This accomplishment would not have been possible without them. Thank you.

Contents

List of Abbreviations

EFSM	Extended Finite State Machine
GUI	Graphical User Interface
LTL	Linear temporal logic
NuSMV	A new symbolic model checker
SBM	Safe Behavioral Model
SMV	Symbolic Model Checker
SSR	Software Safety Requirement
STPA	System Theoretic Process Analysis
STPA SwISs	STPA for Software-Intensive Systems Approach
STPA TCGenerator	STPA Test Cases Generator
STPA TCGeneratorPlugin	STPA Test Cases Generator Plug-in
STAMP	Systems-Theoretic Accident Model and Processes
XSTAMPP	An eXtensible STAMP Platform

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 Motivation

Nowadays software is applied in many modern systems and often plays an important role in main safety-critical functions of systems. However, an unexpected software flaw may lead to catastrophic results such as injury or loss of human life, major equipment failure or environmental damage [1]. Leveson [13] noted that the primary safety problem in software-intensive systems is not the software failure, but the lack of appropriate constraints on software behaviour. Therefore, software must be comprehensively tested, which includes not only functionality requirements, but also the software safety requirements. For this purpose, an accident model STAMP (Systems-Theoretic Accident Model and Processes) was developed by Leveson [13], with which the failure behavior and the system hazards are identified by theoretical consideration of the system. Based on that, STPA (Systems-Theoretic Process Analysis) [13] has been developed as a new hazard analysis technique to identify system safety requirements. Based on STPA, Abdulkhaleq, Wagner and Leveson [4] developed an aproach called STPA SwISs (STPA for Software-Intensive Systems Approach) for deriving software safety requirements rather than overall complex system safety requirements. Based on STPA SwISs, Abdulkhaleq and Wagner developed a tool called XSTAMPP (an eXtensible STAMP Platform) [3] in Eclipse, which is commonly used for documenting the STPA safety analysis process as well as performing the safety verification activities with model checkers such as NuSMV [15] and SPIN [10]. Furthermore, they developed a tool called STPA TCGenerator [1] with the concept of linking safety analysis and testing, which generates test cases automatically from the XSTAMPP analysis results.

1.2 Problem Statement

The basis of this work is the STPA TCGenerator tool, which has been developed by Abdulkhaleq and Wagner [1]. However, STPA TCGenerator has following problems:

- It is a standalone software with different architecture to the Eclipse plug-in.

- The traceability-matrix between the safety requirements and the generated safety based test cases is not optimal.
- There is a problem with graphical visualization of test models.

1.3 Research Objectives

The goals of this work are:

- Migration of STPA TCGenerator from Netbeans into Eclipse as a new plug-in for XSTAMPP called STPA-TCGeneratorPlugin, so that the safety analyst can generate the safety based test cases in XSTAMPP rather than using a different tool.
- Overcoming the shortcomings in the current version of STPA TCGenerator:
  1) Improve the tractability matrix between the STPA software safety requirements and the generated safety based test cases.
  2) Improve the visualizing and documenting of the safe test model and the safety based test cases.

1.4 Outline

This Paper consists of seven sections, which are listed below:

Chapter 2 – Background: In this section, the theoretical background and the relevant tools of STPA TCGeneratorPlugin will be described here.

Chapter 3 – Analysis and Design of STPA TCGenerator as Plugin: The mode of operation, the architecture and the structure of the STPA TCGeneratorPlugin will be shown in this section.

Chapter 4 – GUI Design of STPA TCGenerator Plug-in: The GUI prototype of STPA TCGeneratorPlugin will be designed and shown as figures in this section.

Chapter 5 – Implementation of STPA TCGenerator Plug-in: In this section, the concrete implementation of STPA TCGeneratorPlugin will be presented. Furthermore, the important improvements of STPA TCGeneratorPlugin will be described here.

Chapter 6 – Evaluation Example: The functionality and usability of STPA TCGeneratorPlugin will be assessed through a concrete use case ACC Simulator.

Chapter 7 – User Manual: The guide for setting up and using STPA TCGeneratorPlugin will be illustrated here.

Chapter 8 – Conclusion and Future Work: The contents of this paper will be summarized in this section and a view of possible future extension will be discussed at the end.

2 Background

2.1 STPA Safety Analysis

STPA is the abbreviation of Systems-Theoretic Process Analysis, which is a new safety analysis technique that was developed based on the model of accident causation named STAMP (Systems-Theoretic Accident Model and Process) by Leveson [13]. The goal of safety analysis can be described as the inspection of potential safety flaw that are causes of accidents, which should be identified and prevented in design or operation before damage occurs. Before STPA was developed, some other safety analysis techniques were already widely used, such as Fault Tree Analysis (FTA) [21], Failure Mode and Effect Criticality Analysis (FMECA) [18] and Hazard and Operability Analysis (HAZOP) [11]. The most important reason for developing STPA was to use the new causal factors identified through STAMP, so that STPA hazard analysis would consider not only the electromechanical components, but also entire accident process [13]. Furthermore, STPA can be used before a design has been created, so that the system design can be guided by the information of STPA.

STAMP identifies general causes of accidents by using the basic systems and the control theory, which is relied on three basic concepts: safety constraints, a hierarchical safety control structure and process models. Accidents in STAMP are caused by the behavior in a complex process that oppose the safety constraints. The safety constraints are required in various levels of the hierarchical control structure and are considered in all processes such as design, development, manufacturing, and operations.

Based on STAMP, STPA can be processed in three main steps:

1. Establish fundamentals such as hazards and accidents,
2. Identify the potential hazardous control actions in the system and draw the safety-control structure diagram (Figure 2.1) of the system. According to STAMP the causal of accidents (unsafe control actions) can be classified in four groups:
    a) A control action required for safety is *not* provided or not followed.
    b) An unsafe control action *is* provided.

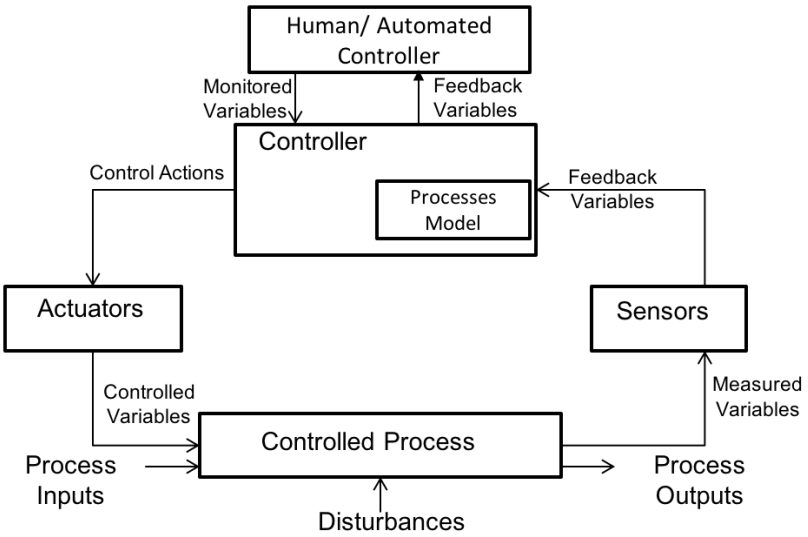


Figure 2.1: Safety-Control Structure Diagram [1]

c) The necessary control actions were provided, but at the wrong time (too early or too late) or in the wrong sequence.

d) A control action required for safety is stopped too soon or applied too long.

Figure 2.1 illustrates a generic diagram of control structure. A controller receives variables from the sensor and determines which control actions should be performed based on the process model. Finally the required control actions must be implemented for the controlled process by the actuator.

3. Determine how each potentially hazardous control action identified in step 1 could occur. This could be achieved by following methods:

   a) For each hazard control action, check all parts of the control structure to identify if they could cause it.

   b) The designed controls should be considered with time-development vision, that is, the hazard controls could be degraded over time.

2.2 Software Testing

Software testing is the process of verifying software according to a specific sequence of steps, comparing the software's actual and expected outputs [7]. Software testing aims at assessing whether all software requirements can be achieved and software functions are executed correctly. The more hazards can be found during the test phase, the better

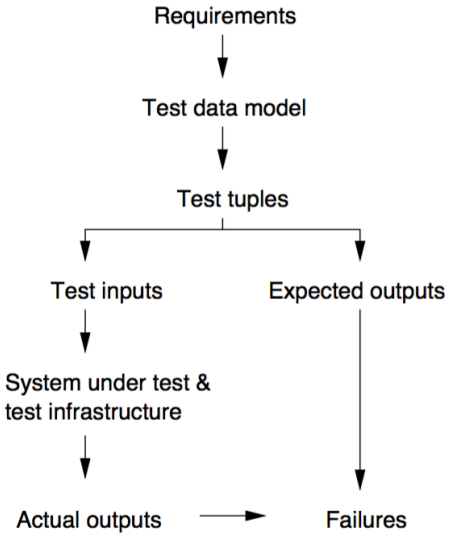


Figure 2.2: Data flow of a generic test-generation system [6]

is the software testing design. However, it can be a difficult task to design a systematical and comprehensive test, since software is a more complex product than traditional products.

In this case, a popular test approach called Model-based Testing (MBT) [6] was developed. Model-based Testing generates a suit of test cases automatically based on models extracted from software requirements. Figure 2.2 illustrates the architecture of a generic test generation system. From system safety analysis, we obtain a number of requirements, which will be used as input data for generating test cases. From these requirements, a test model can be extracted, which is derived from a requirements data model. A requirements data model determines the set of all possible values for a parameter. Compared to that, a test-generation data model indicates a set of valid and invalid values that will be supplied for the parameter in a test. In a test case, the values for each parameter can be chosen sequentially according to some specific constrains. For example, two parameters might accept the same values, but cannot at the same time. We call these test cases a test set in the test-generation data model and combine different test sets to a test suit.

A good test suit must envelope all requirements and functions of software. Therefore the test-generation model plays an important role in the generation of test cases. Some good test models are in use today, such as control flow charts, finite state machines, SpecTRM-RL [12] and sequence event diagrams. Among them finite state machines are widely used in software behavior modeling and generating test cases [1].

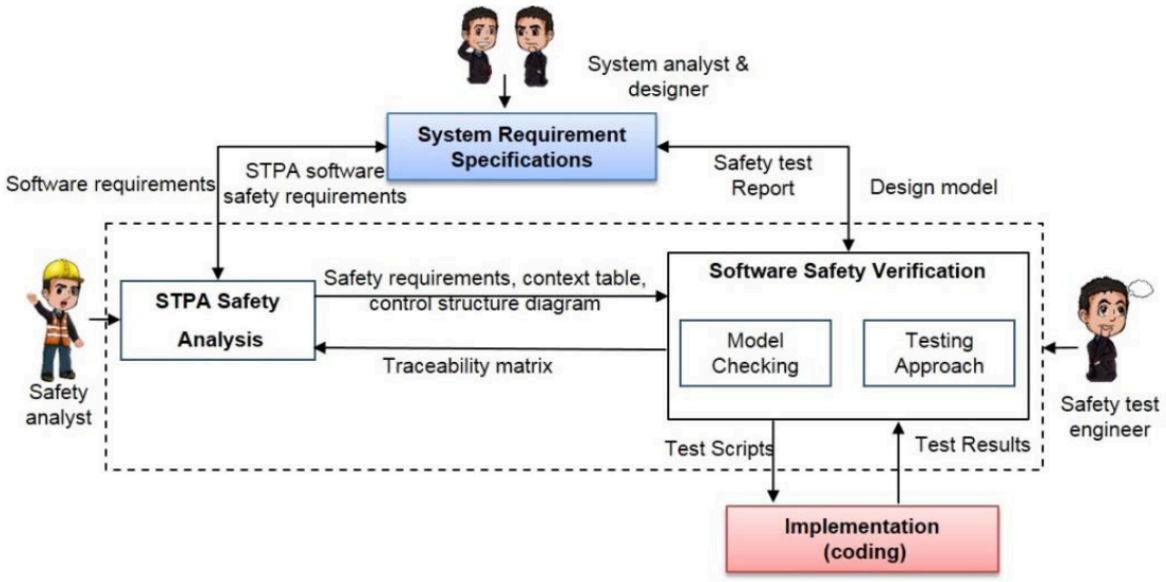


Figure 2.3: Overview of STPA SwSIs [4]

2.3 STPA SwISs: STPA for Software-Intensive Systems Approach

Based on STPA approach Abdulkhaleqa, Wagner and Leveson [4] developed a comprehensive safety engineering approach called STPA SwISs. Since STPA has been developed to analyze safety requirements for overall complex systems, but not specific for software intensive system, STPA is not very suitable for treatment of software components without any adjustment. STPA SwISs is a comprehensive safety engineering approach based on STPA, including software testing and model checking approaches for the purpose of developing safe software. Based on STPA, STPA SwISs derives software safety requirements, transforms them into formal specifications in model, verifies model using model checking and verifies software safety by automatically generated test cases. This approach can be embedded within a defined software engineering process or applied to existing software systems, allowing software and safety engineers the integration of the analysis of software risks to their verification.

Figure 2.3 illustrates the general process of the STPA SwISs approach, which can be performed in three steps:

1. Specify software safety requirements using STPA at the system level:
   A. Abdulkhaleqa, S. Wagner and N. Leveson [4] developed an algorithm for deriving software safety requirements based on STPA [13] and extended STPA [19]

approach. This algorithm can be divided into the following steps: 1) Identify all safety-critical Control Actions(CA) that cause system hazards (HA), 2) Classify all Unsafe Control Actions(UCAs) in four groups as described in 2.1 and translate the UCAs into the Software Safety Requirements(SSR) in natural language, 3) Identify process model and its variables which affect the safety of control actions and lead to UCAs, 4) Determine the combination of different process model variables if they have an effect on the same UCA and 5) A set of unsafe software scenarios that lead to UCAs must be identified and translated into safety requirements in formal specifications using LTL.

2. Modeling software controller's behavior under the constrain of software safety requirements:
   A safe behavioral model (SBM) [4] includes only the process model variables that have an effect on the safety of control actions of the software controller. The model is visualized by using UML statechart notation. Each state presents a process model variable and the state actions are control actions, which are constrained by the software safety requirements.

3. Verifying software safety by formal verification and automatically generated test cases:
   The safety test engineer should perform the verification activities as below:
   1) Verifying the safe behavioral model (SBM) with STPA software safety requirements and specifying SBM in LTL [9], which should be verified by using model checker such as NuSMV [15]. 2) Generating safety-based test cases: The verified safe behavioral model is used as input and the output is test cases that are grouped in test suits. 3) Generating and executing test scripts: using generated test cases as input, executable test scripts are generated and then executed. The output of this step is a safety verification report [4].

2.4 Tool Support

2.4.1 XSTAMPP: an eXtensible STAMP Platform

XSTAMPP is a rich client platform developed by Abdulkhaleq and Wagner in 2015 [2][3]. Based on the Eclipse Plug-in-Development Environment (PDE) and Rich Client platform (RCP), XSTAMPP provides core functionality and components for easily extending new plugins. The Goal for the development of XSTAMPP was to create a base platform for safety engineering, which can be easily extended with new functions and approaches based on the STAMP model. The architecture of XSTAMPP is shown in Figure 2.4. XSTAMPP consists of four major components as described in [3]:

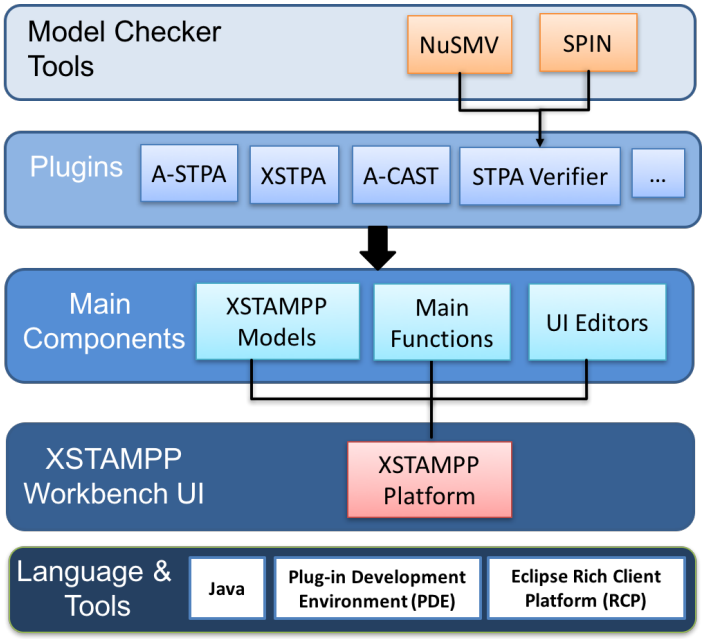


Figure 2.4: Architecture of XSTAMPP [3]

1. **Eclipse Rich Client Platform(RCP):**
   The basis of XSTAMPP is the Eclipse Rich Client Platform(RCP) that provides the application framework for further extension and integration of new software modules.

2. **Plug-ins and Plug-in Development Environment (PDE):**
   A number of custom extension points are provided in PDE, which can be extended as new plugins in software. Before the publishing of this paper, XSTAMPP involves following plug-ins: A-STPA, A-CAST, XSTPA and STPA-Verifier[5]. Eclipse plug-in Development Environment will be described in detail in section 2.4.4.

3. **XML Elements XSD Specification Template:**
   By using STPA and extended STPA, STAMP data lists (e.g. hazards list, accidents list and safety requirements), STAMP diagrams (e.g. safety control structures) and STAMP tables (e.g. unsafe control actions table) are identified and represented as XML elements. By using XSD file, all XML elements can be saved and restored with extension *.haz for the whole project.

4. **UI Editors Workbench User Interface:**
   The views and UI Editors are integrated independently in the Workbench User Interface. Each UI Editor represents an STAMP component (e.g. STAMP data lists, STAMP diagrams and STAMP Tables) which can be edited and restored in the UI Editor.

2.4.2 STPA Verifier

Abdulkhaleq and Wagner [4] proposed a comprehensive approach for software safety verification based on STPA. Since the correctness of the software safety requirements model plays an important roll for further steps of software testing, based on the approach mentioned in section 2.3, Balzer [5] developed a plug-in for XSTAMPP called STPA Verifier to verify automatically the correctness of safety constrains. This plug-in is based on the concepts of the STPA SwISs approach (see in 2.3) and provides a graphical user interface. STPA Verifier is performed as below:

1. Input an software safety requirements model in form of LTL (Linear Temporal Logic)[9] or CTL (Computation Tree Logic) [8] into STPA Verifier.
   LTL is a temporal logic with which it can formulate logical expressions and their validity over time. CTL is another temporal logic in use of model check. Compared to LTL, CTL does not observe the execution of program as a linear sequence but as a tree, that means the program can go from one state to more possible states.
2. Check the input model with exists model checker Spin [10] or NuSMV [15] against a selected system model.
3. Generating a verification report.

2.4.3 STPA TCGenerator

STPA TCGenerator[1] is a tool for generating test cases based on the results of system safety analysis. This tool was developed by Abdulkhaleqa and Wagner [1] in Netbeans. The main functions of STPA TCGenearator are listed as below:

1. Parse the STPA Data Project from XML into a Java model
2. Parse the Simulink Stateflow from XML into a Java model
3. Automatically generate SMV model from STPA Data model and Simulink Stateflow model
4. Verify the SMV model against the generated LTL for the STPA safety requirements
5. Generate the extended finite state model from verifed SMV model
6. Allows the user to add the input test date
7. Generate the safety based test cases and

[1] STPA TCGenerator: http://www.xstampp.de/STPATCGenerator.html

8. Build matrix between safety requirements and test cases.

The whole process of STPA TCGenerator is illustrated in the flow chart Figure 2.5.

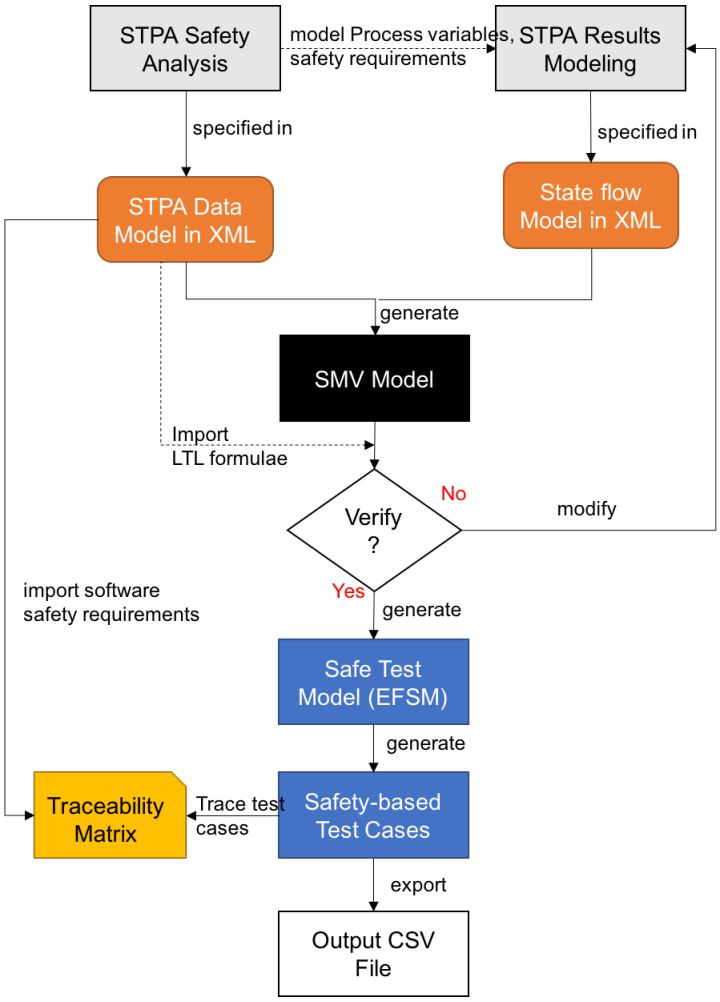


Figure 2.5: Process Flow Chart of STPA TCGenerator [1]

By A-STPA project we obtained STPA Data Model as the result of safety analysis in XML. With the help of Simulink, STPA Data Model was simulated into Stateflow model (also called safe behavioural model SBM) in XML. Both models should be input into STPA TCGenerator and transformed into Java Model by using JAXB [16]. In the next step an SMV model (see in Figure 2.6) will be automatically generated from STPA Data model and safe behavioral model. The correctness of SMV model will be verified with model checker NuSMV against the generated LTL for the STPA safety requirements. According to the results of verification the save behavioural model and SMV model should be adjusted and rechecked. After the verification of SMV model we generate the safe test model (Extended Finite State Model) that has a tree structure. The tree nodes are the

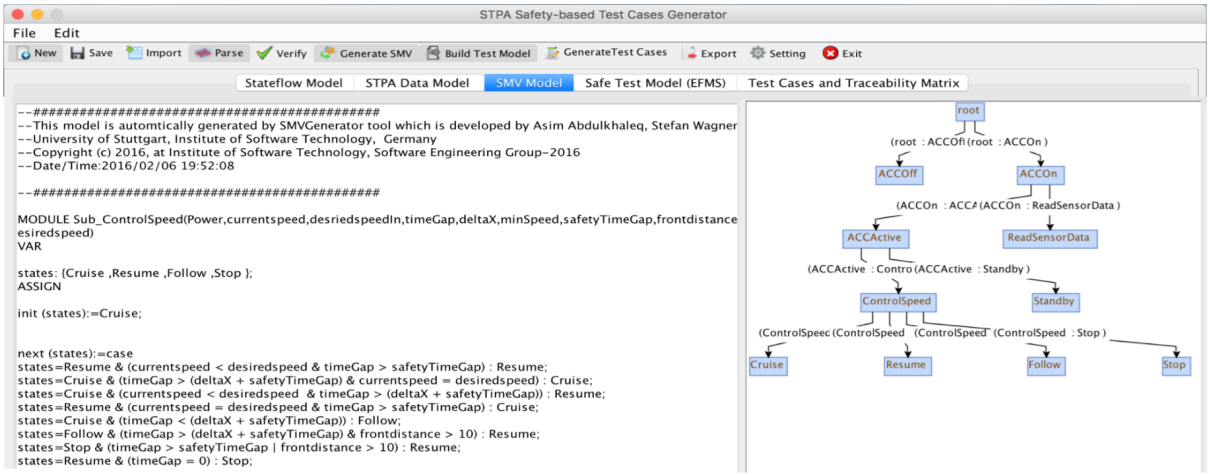


Figure 2.6: STPA TCGenerator in Netbeans: SMV Model and Tree Graph of Safe Behavioral Model(SBM)[1]

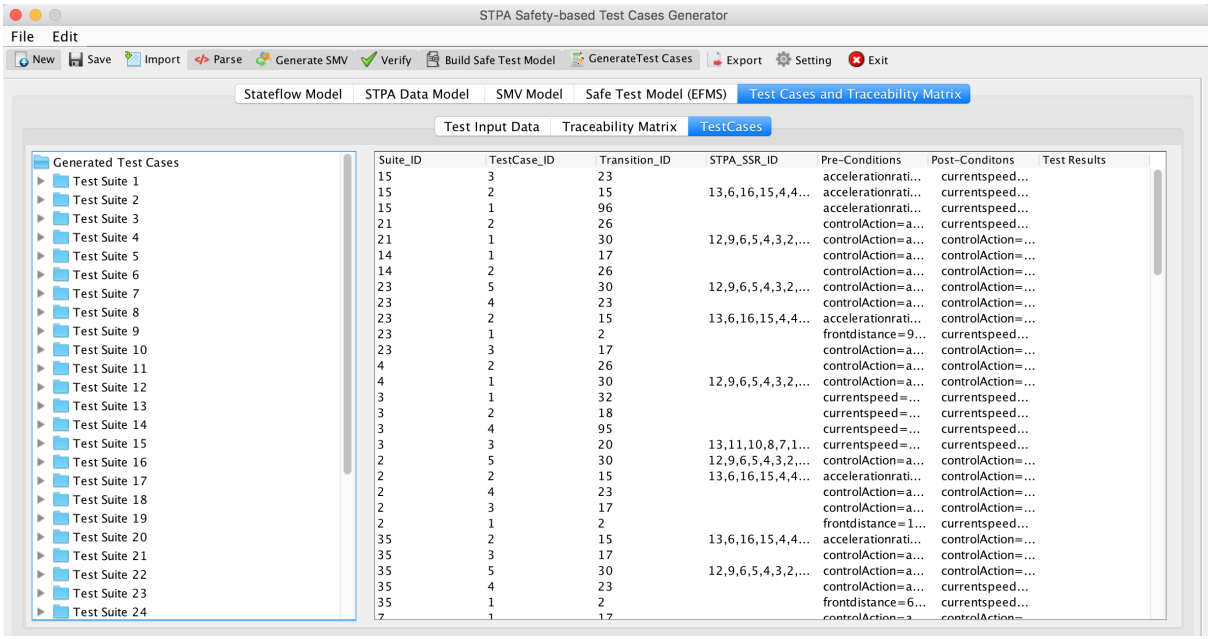


Figure 2.7: STPA in Netbeans: Result of Test Cases [1]

reachable states and the edges are transactions between states. By traverse of the tree we obtain the safety-based test cases as result, which will be saved as test suits in files (see in Figure 2.7). At the same time a traceability matrix was built between STPA data model and safety-based test cases in order to estimate the quality of generated test cases.

In this work STPA TCGenerator will be migrated from Netbeans into Eclipse and integrated as a new plug-in into XSTAMPP as next step.

2.4.4 Eclipse Plug-in Development Environment

The Plug-in Development Environment (PDE) provides tools to create, develop, test, debug, build and deploy Eclipse plug-ins, fragments, features, update sites and RCP products [17]. A software component in Eclipse is called plug-in. Developers are allowed to extend eclipse applications with additional functions via plug-ins. For Example, a new plug-in create new menu entries or toolbar entries.

Plug-ins are *extendable* by using extensions and extension points. A plug-in can provide one or more extension points, so that other plug-ins can be added to the functionality of the plug-in. A plug-in may also provide extensions to connect to other plug-ins.

Plug-ins are *sharable*. A plug-in can be exported as a directory or as a jar which can be added to other applications. Plug-ins can be grouped into features which can be distributed and installed into applications.

Eclipse Rich Client Platform (Eclipse RCP) indicates that the Eclipse platform is used as a basis to create feature-rich stand-alone applications. XSTAMPP (see in Sec. 2.4.1) was developed based on PDE and RCP, that contains multiple plug-ins, which can be added, replaced or removed to alter the functionality of the product.[3]

3 Analysis and Design of STPA TCGenerator as Plugin

This chapter deals with the analysis of the architecture of STPA TCGeneratorPlugin as plug-in in XSTAMPP and the implementation in the form of a concrete use case and sequence diagram. The aim of this chapter is to give an insight into the basic concepts for the GUI Design in Chapter 4 and the implementation in Chapter 5.

3.1 Architecture and Process Flow of STPA TCGeneratorPlugin

3.1.1 Architecture of STPA TCGeneratorPlugin

This section describes the architecture of the STPA TCGeneratorPlugin as well as an overview of the most important channels of communication between the major components.

The architecture of the plug-in STPA TCGeneratorPlugin is shown in Figure 3.1. Java and Eclipse RCP provide the basic libraries for the XSTAMPP platform. The XSTAMPP platform provides the basic components, the models and the interface for the development and integration of the plug-in STPA TCGeneratorPlugin. The graphical user interface (GUI) of the STPA TCGeneratorPlugin is developed based on SWT, JFace and Javax Swing.

This plug-in STPA TCGeneratorPlugin is developed bases on STPA TCGenerator in Netbeans, which has a connection with other tools. By Simulink created Safe Behavioral Model (SBM) and via A-STPA derived safety-based STPA data model (STPA data Model) must be converted into java model by using JAXB [16]. From the SBM and the STPA data Model generated SMV model will be verified either by local installed model checker

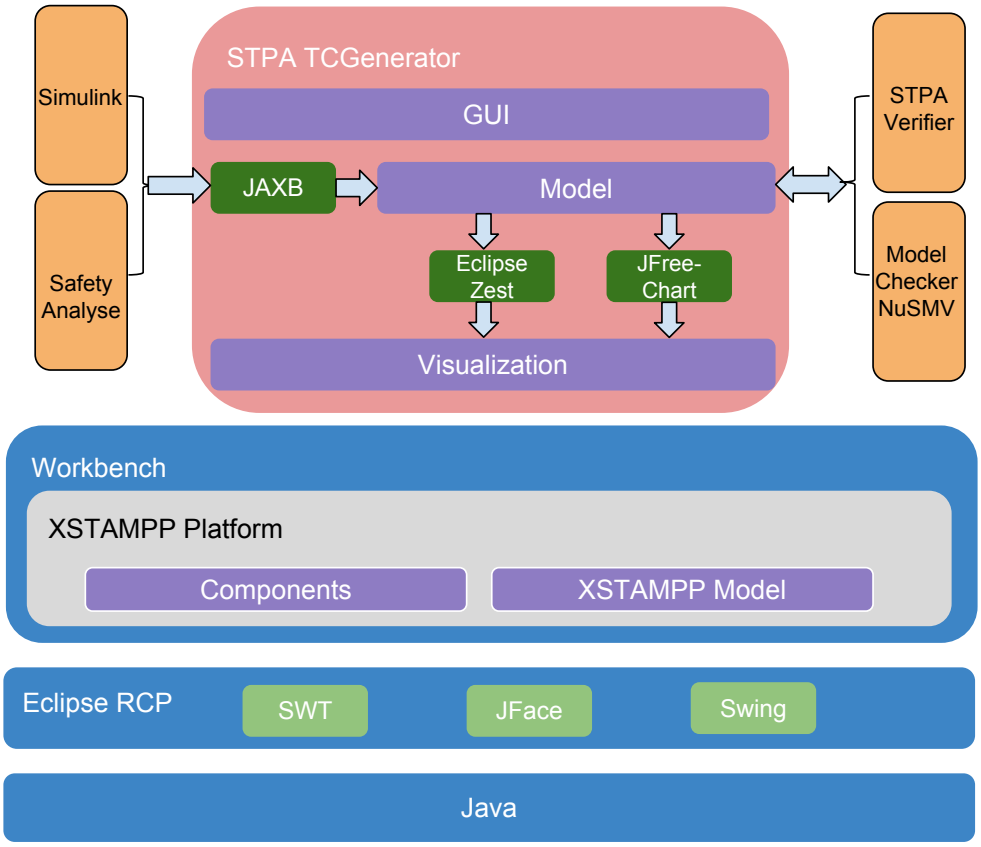


Figure 3.1: Architecture of STPA TCGeneratorPlugin

NuSMV or by another plug-in STPA Verifier. The models can be visualized as tree graph and histogram each by using Eclipse Zest[1] and JFreeChart[2].

3.1.2 Process Flow of STPA TCGeneratorPlugin

In comparison of STPA TCGenerator in Netbeans, the process flow in STPA TCGeneratorPlugin was improved by adding the validation between STPA data model and safe behavioral model before generating the SMV model. The improved process flow in STPA TCGeneratorPlugin is illustrated in Figure 3.2. Since safety behavioral model (SBM) was generated from state flow model, which had been simulated based on the STPA data model, the validation between SBM variables and STPA process model variables is

[1]Eclipse Zest: https://wiki.eclipse.org/Zest
[2]JFreeChart: http://www.jfree.org/jfreechart/

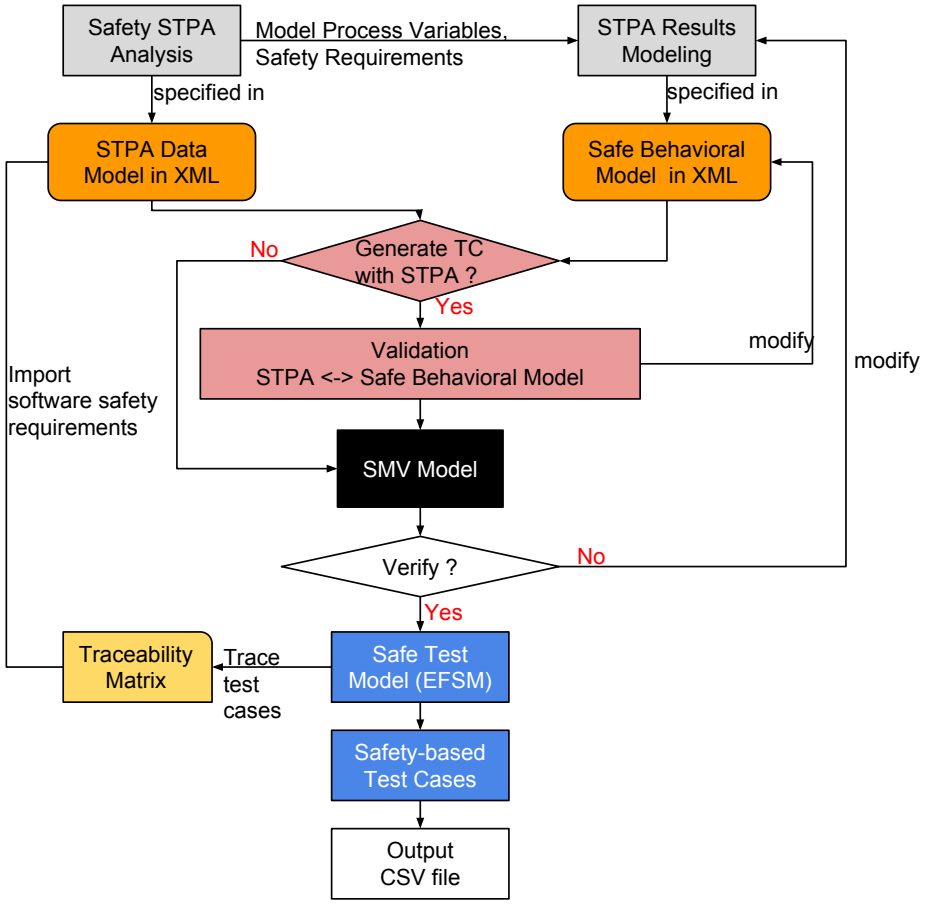


Figure 3.2: Process Flow of STPA TCGeneratorPlugin

necessary. It is verified, whether all states and control action in STPA data model can be found in SBM with exactly the same name and if data variables in SBM can be found in STPA data model.

3.2 Use Case Diagram

An use case diagram represents the relationship between the user and different use cases in the system in which the user is involved. Figure 3.3 illustrates the user's interaction with the STPA TCGeneratorPlugin, so that the main functions of STPA TCGeneratorPlugin and the available options are represented intuitively. The most important functions of STPA TCGeneratorPlugin are listed as below:

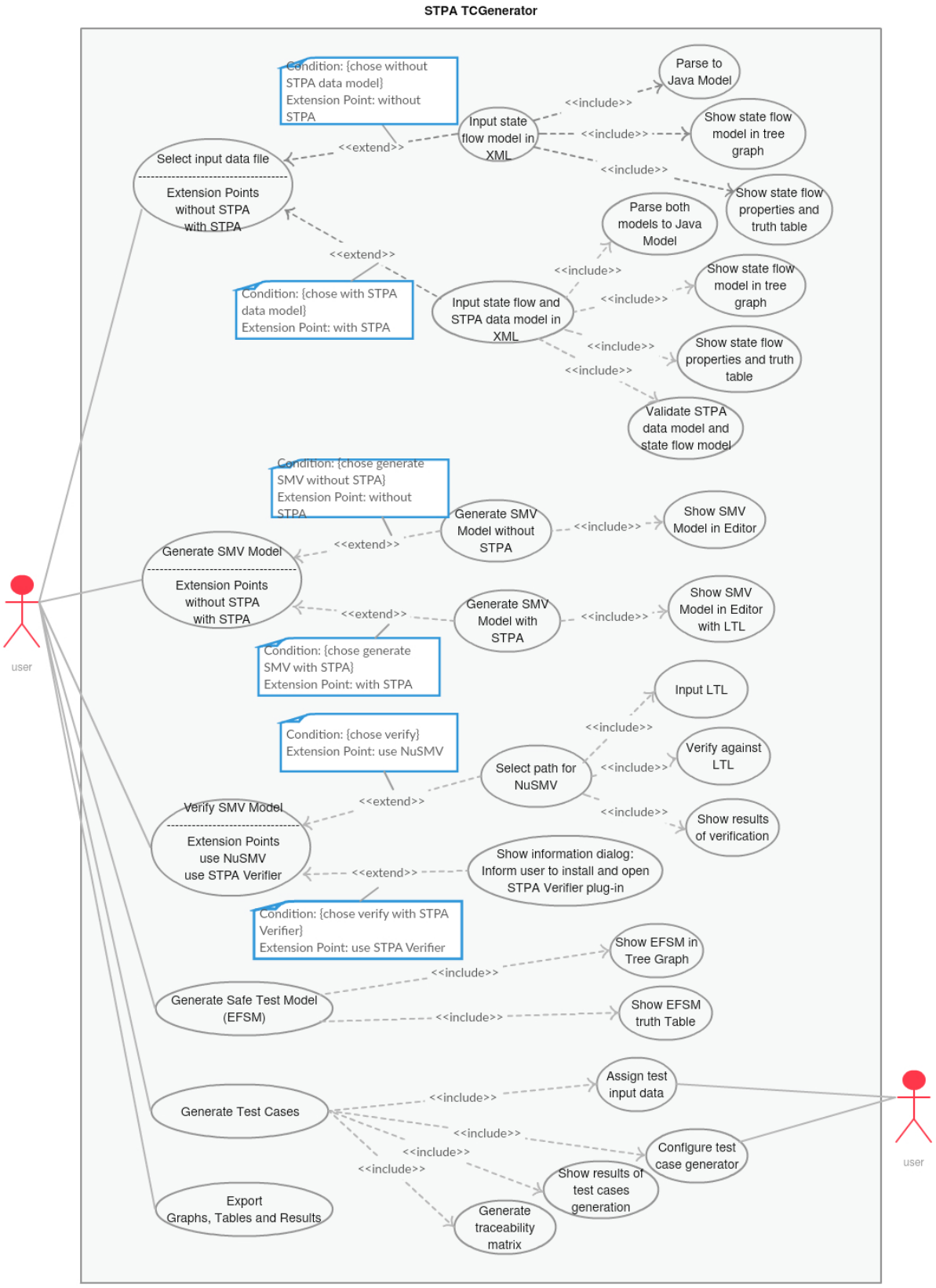


Figure 3.3: Use Case Diagram for STPA TCGeneratorPlugin

1. Input and Preparing state flow model (and STPA data model) to generate SMV model:
   The user is able to choose the path of state flow XML file via document browser.
2. Automatically Generate SMV Model with or without STPA data model,
3. Verify SMV Model with two options:
   a) Check with local installed model checker NuSMV: NuSMV should be installed in system and the user must give the path through document explorer.
   b) Check with plug-in STPA Verifier: pre-condition of this option is that the STPA Verifier plug-in must be installed.
4. Automatically Generate Safe Test model,
5. Automatically Generate Test Cases and allow the user to
   a) assign test input data and
   b) configure test case generator with the number of test steps, test coverage and test algorithm.
6. Export the graphics, tables and generated safety-based test cases in PNG and CSV.

3.3 Sequence Diagram

A Sequence diagram is an interaction diagram that shows how objects operate with one another and in which order. It is typically associated with use case realizations in the logical view of the system under development. The sequence diagram of STPA TCGeneratorPlugin is illustrated in Figure 3.4.

STPA TCGeneratorPlugin is an extended plug-in of XSTAMPP, so that this plug-in is started by right clicking on an STPA project in XSATMPP. After configuration for STPA TCGeneratorPlugin, the default perspective of STPA TCGeneratorPlugin will be shown, which contains a tree graph of the save behavioral model (SBM), a view of SBM properties and a view of the SBM truth table. If we generate test cases with STPA project, the view of validation between STPA data model and safe behavioral model is also shown in an additional view. Furthermore, the user is allowed to update variables in tables of views.

In the next step, SMV model is generated in two options-with or without STPA data model. The SMV model will be revealed in Editor in both cases. If the SMV model is generated with the consideration of STPA data model, the LTL table must be shown in another view.

The generated SMV model can be verified with or without STPA verifier. The user must in both cases give the path of NuSMV in configuration and start the verification. To open the STPA verifier, the user must have been installed this plug-in at first. The results or errors of verification will be shown in console, error log as well as in LTL table if the SMV model was generated with STPA data model.

After that, the user is able to generate safe test model (EFSM) and then generate test cases. Before generating test cases, the user is allowed to assign test input data and configure test case generator. The results of the generated test cases are displayed in different views and the traceability-matrix between safety requirements and test cases is generated at the same time.

At the end the user is able to export all results with export button in the toolbar and then close STPA TCGeneratorPlugin with the button "close TCGenerator" in the toolbar.

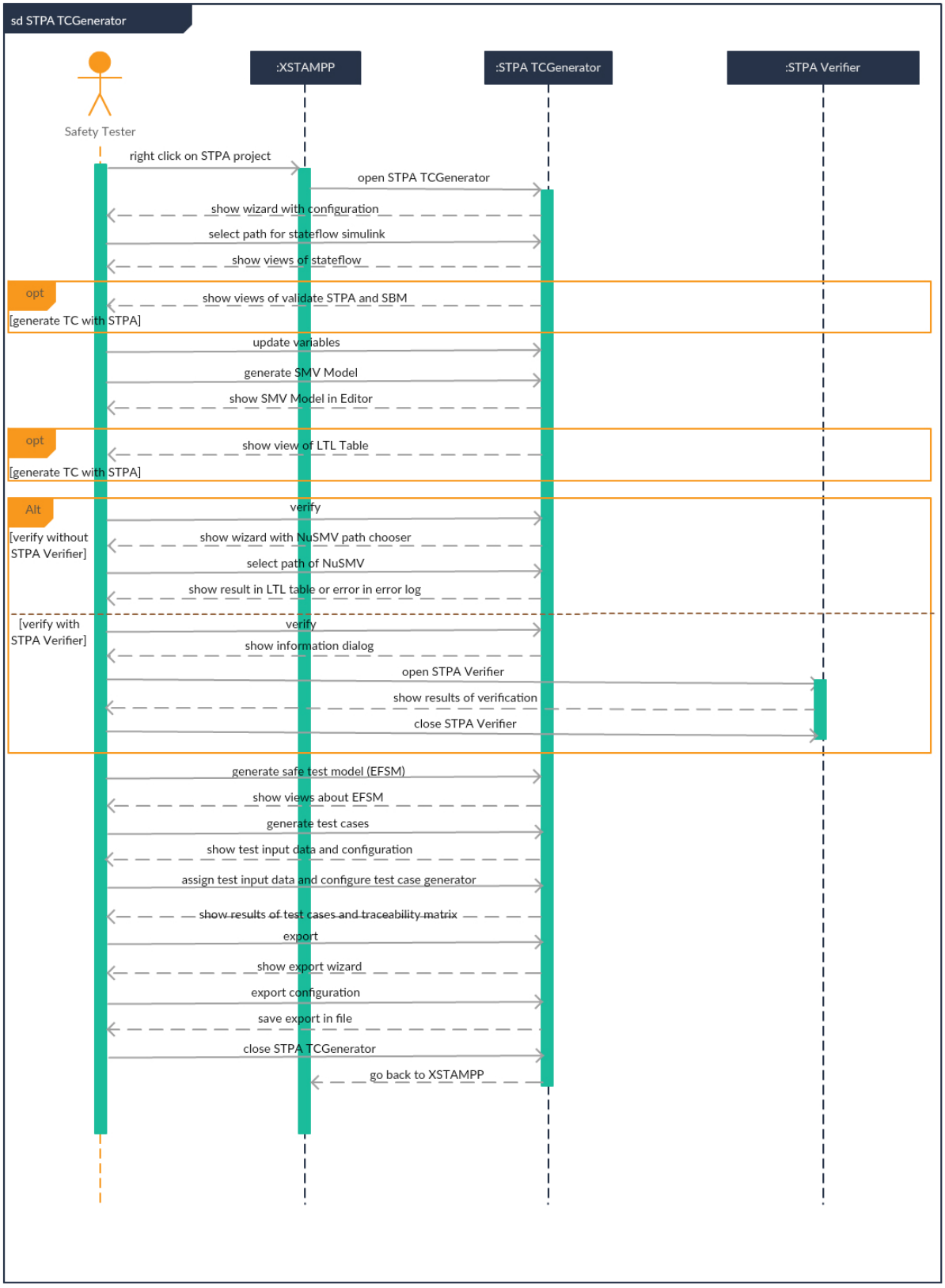


Figure 3.4: Sequence Diagram of STPA TCGeneratorPlugin

3.4 Class Diagram

This chapter shows an overview of the packages' structure and logical relationship of the STPA TCGeneratorPlugin project. This project uses a model-view-controller design pattern for implementing user interfaces. The relationship between controller and model is shown in Figure 3.5. This figure only illustrates the most important models, handlers and jobs, with which the models are generated.

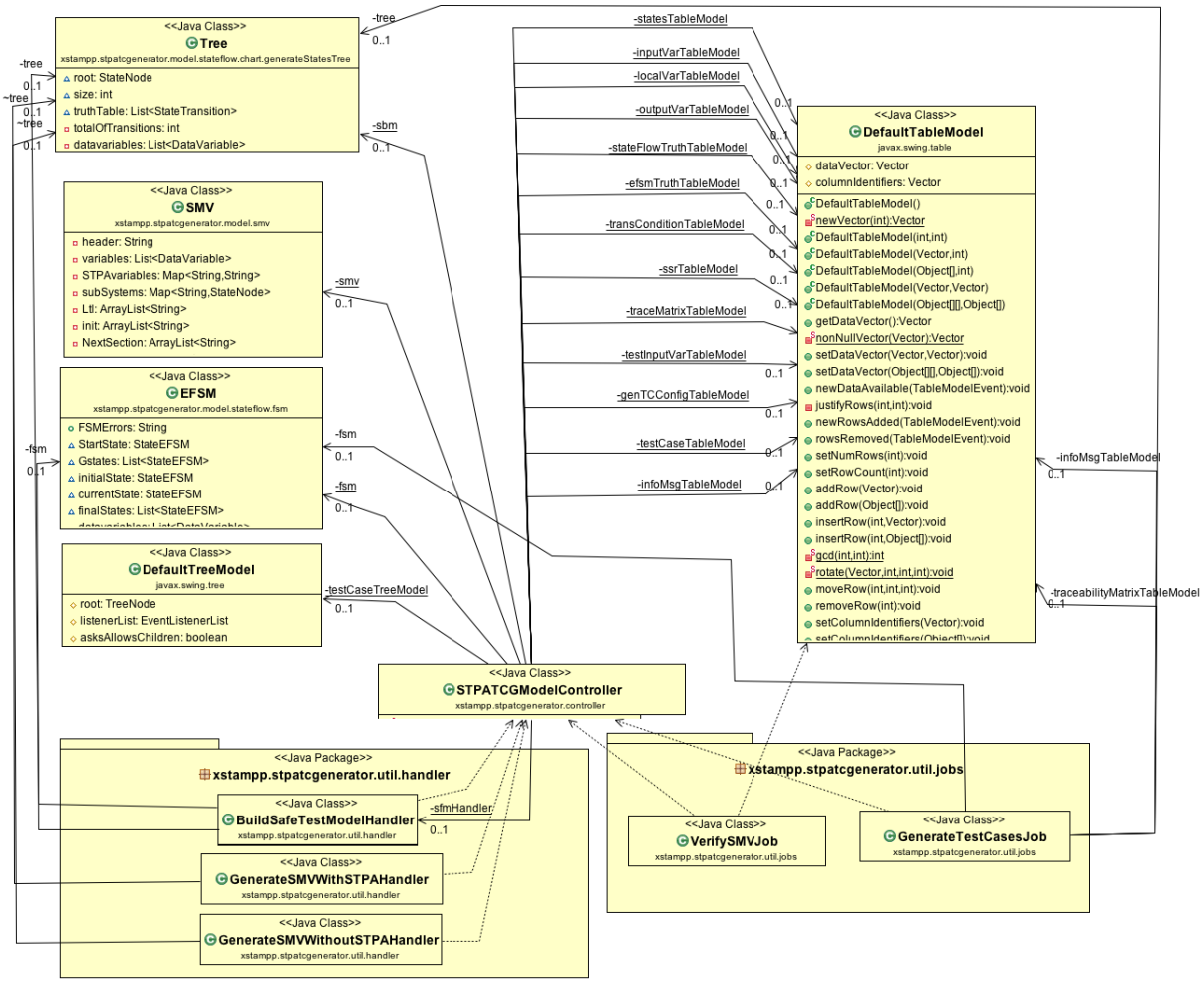


Figure 3.5: Class Diagram: Model-Controller

The generated safe behavioral tree model, SMV model, extended finite state machine model and test cases tree model are related to STPATCGModelController and are able to output as view or file. Figure 3.6 illustrates the most important views, editors and their relationship between the controller (such as handlers and jobs).

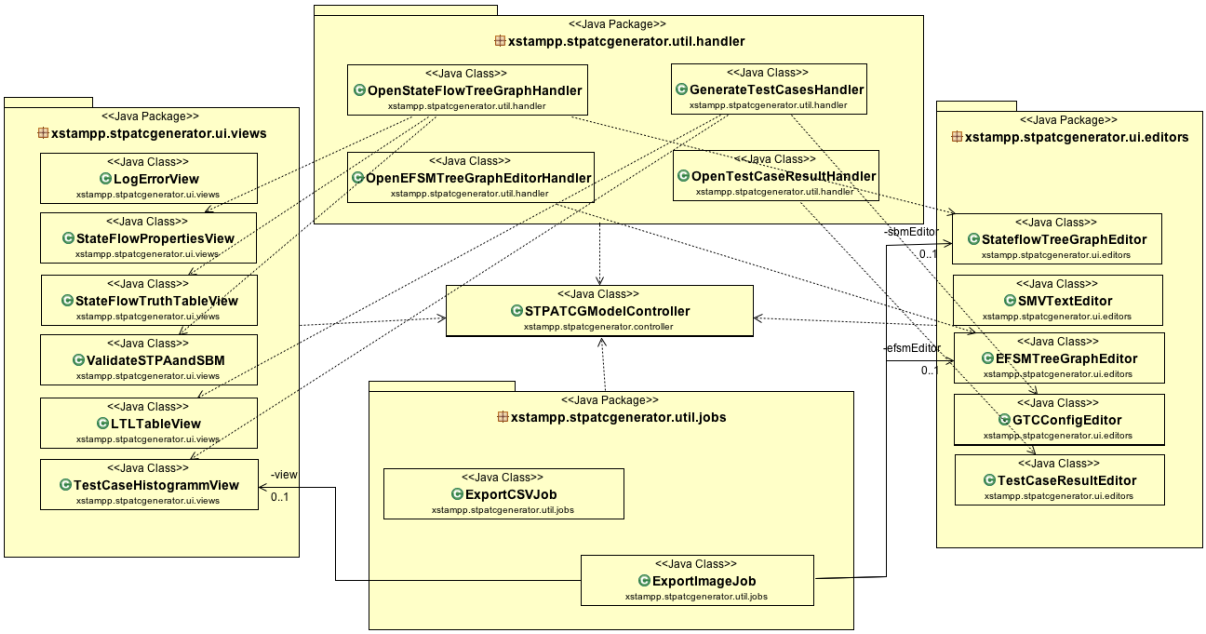


Figure 3.6: Class Diagram: View-Controller

4 GUI Design of STPA TCGenerator Plug-in

This Chapter presents a design concept as well as the end effect of the graphical user interface (GUI) for STPA TCGeneratorPlugin. Its implementation and adaptation will be discussed in chapter 5.

4.1 Design Concept

There are three important principles for user interface design [14]:

1. Place users in control of the interface,
2. Reduce the users' memory and
3. Make the user interface consistent.

Since STPA TCGeneratorPlugin is an extended plug-in based on XSTAMPP [3], the GUI of STPA TCGeneratorPlugin is also designed based on the style of XSTAMPP, so that the user has a consistent experience of software. Figure 4.1 illustrates the GUI design of STPA TCGenerator.

4.1.1 Toolbar

The plug-in STPA TCGeneratorPlugin has an explicit toolbar itself. This toolbar is placed in the main toolbar of XSTAMPP and contains the main functions of STPA TCGeneratorPlugin, which are organized in the process order of STPA TCGenerator. With the help of the toolbar, users feel in control of this tool and the memory load of users is reduced, as there is no need to remember the process order.

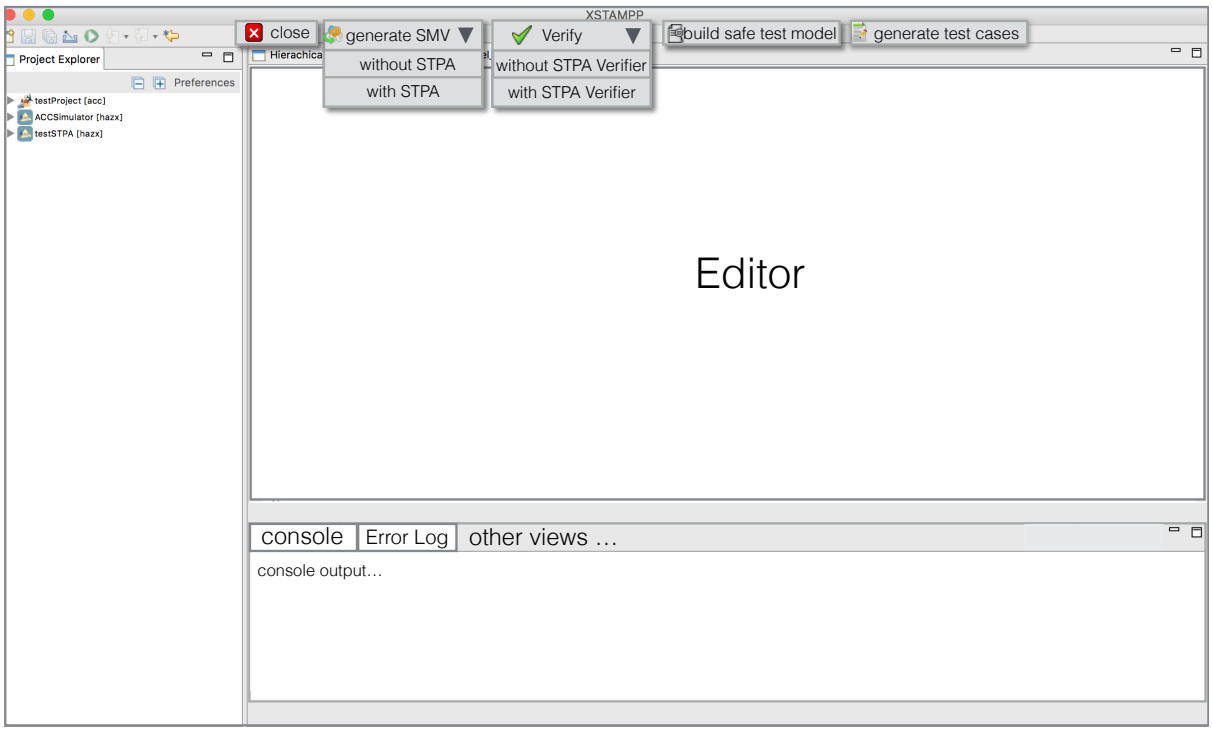


Figure 4.1: GUI Design of STPA TCGeneratorPlugin

4.1.2 Editor

All editors are opened in the significant place of a perspective, so that the editor is usually used to display important and editable parts of an application. Users are able to find the important information at a glance and perform an appropriate execution. STPA TCGeneratorPlugin encloses following editors:

- Safe behavioral model graph editor
- Extended finite state model graph editor
- SMV text editor
- Configuration of generating test cases editor and
- The test case result editor

The concrete functions of each editor will be described in Chapter 6 with an use case example in detail.

4.1.3 Views

The views are usually used for displaying properties and reporting status of a running process. STPA TCGeneratorPlugin involves two standard views: console view and error log view. Console view reports the current running process of a program and error log view displays errors which occur in a process. With the help of these views, users are able to grasp the status of running processes and feel in control of the software. Furthermore, STPA TCGeneratorPlugin provides some other views to show properties of important variables, so that users are allowed to make some modifications on variables. The detailed information and functions of each view will be declared in chapter 6.

4.2 End Result of GUI

According to the above described GUI Design, the final graphical user interface of STPA TCGeneratorPlugin is illustrated in Figure 4.2. Comparing with the above-mentioned GUI Design, we added the button Open SBM Graph into the toolbar. The reason for this is that, users are able to show the safe behavioral model graph editor and make any modification at any time even if this editor has been closed before.

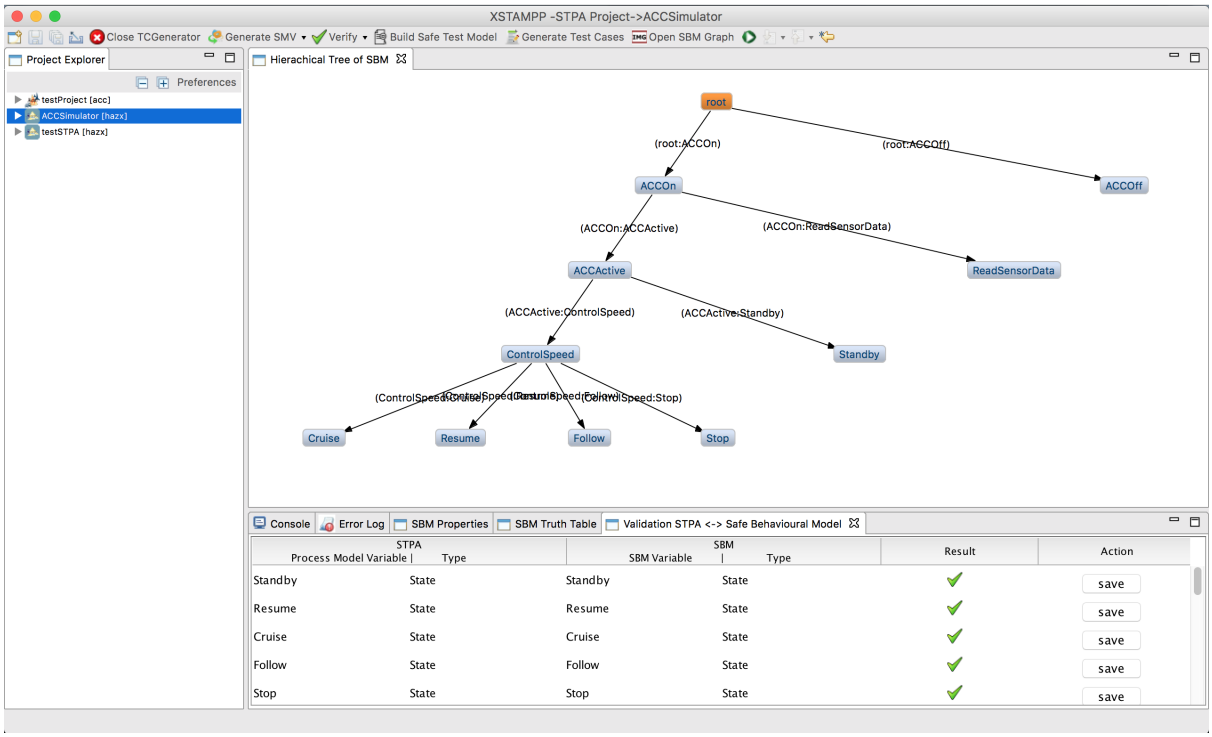


Figure 4.2: GUI of STPA TCGeneratorPlugin in version 1.0.0

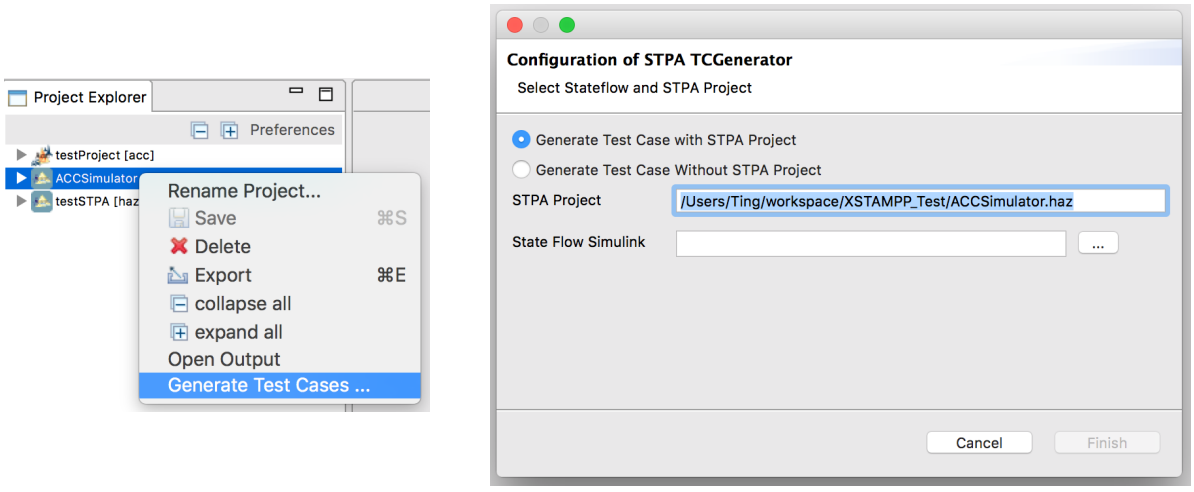
5 Implementation of STPA TCGenerator Plug-in

This chapter gives an overview of functions and windows of STPA TCGeneratorPlugin, which will be evaluated with a practical use case example in chapter 6. Furthermore, the improvements for STPA TCGeneratorPlugin in comparison to the TCGenerator in Netbeans will be discussed in this chapter too.

5.1 Functions

5.1.1 Open STPA TCGeneratorPlugin from STPA Project

STPA TCGeneratorPlugin should be opened with the right click of an STPA Project in XSTAMPP (see Figure 5.1 left). After clicking on the menu item "Generate Test Cases", a



Open STPA TCGenerator Configuration of STPA TCGenerator

Figure 5.1: Open and Configuration of STPA TCGeneratorPlugin

Figure 5.2: Toolbar of STPA TCGeneratorPlugin

configuration wizard (see Figure 5.1 right) is shown and the user is allowed to customize the basic settings and import the state flow Simulink file using the file browser. The path of last selected state flow Simulink file is always saved and automatically shown by a restart of STPA TCGeneratorPlugin.

5.1.2 Overview of Functions

In the main perspective of STPA TCGeneratorPlugin, users can manage the process of generating test cases through an explicit toolbar (see Figure 5.2), which is shown immediately, when the STPA TCGeneratorPlugin was opened. The main functions of generating test cases are all included in the toolbar and described as below:

- **Close TCGenerator**: The STPA TCGeneratorPlugin is closed and the default perspective of XSTAMPP will be shown.
- **Generate SMV**: This is a drop-down menu with two options: Generate SMV with STPA and Generate SMV without STPA. The SMV model is generated either with or without the consideration of STPA project. The functionality and behavior will be discussed with a concrete use case example in detail in chapter 6.1.
- **Verify**: This is a drop-down menu with two options: Verify SMV Model with STPA Verifier and without STPA Verifier. In the last step generated SVM model should be verified with model checker, in order to ensure the correctness of safe behavior model and SMV model. Users are allowed to verify the model either within this plug-in or with an external plug-in STPA Verifier. Details of verification process and the connection with STPA Verifier will be described with a concrete use case example in chapter 6.1.
- **Build Safe Test Model**: A safe test model is generated by this function. Furthermore, a graphical visualization of safe test model is shown in the editor area. The concrete functionality will be explained in chapter 6.3.
- **Generate Test Cases**: Before automatically generating test cases, STPA TCGeneratorPlugin allows user to modify parameters such as input variables, the number of test steps, algorithms, minimum required similarity degree of traceability matrix and so on. All of these can be customized in a configuration editor for generating test cases, which is shown when the button "Generate Test Cases" was clicked. Once

the setting of configuration finished, users should click another button "Generate Test Cases" in the editor. Test cases as well as statistical result will be automatically generated and shown in other windows. The concrete functions and end result of these windows will be declared with a concrete use case example in chapter 6.4.

- **Open SBM Model**: This button provides the function that the user is able to open and modify the safe behavioral model graph any time if it was closed before.

5.1.3 Logging

STPA TCGeneratorPlugin has its own console and error log (see Figure 5.3) that displays all outputs, errors and warnings, which are not only for internal, but also for the external program that is called in an Eclipse typically console.

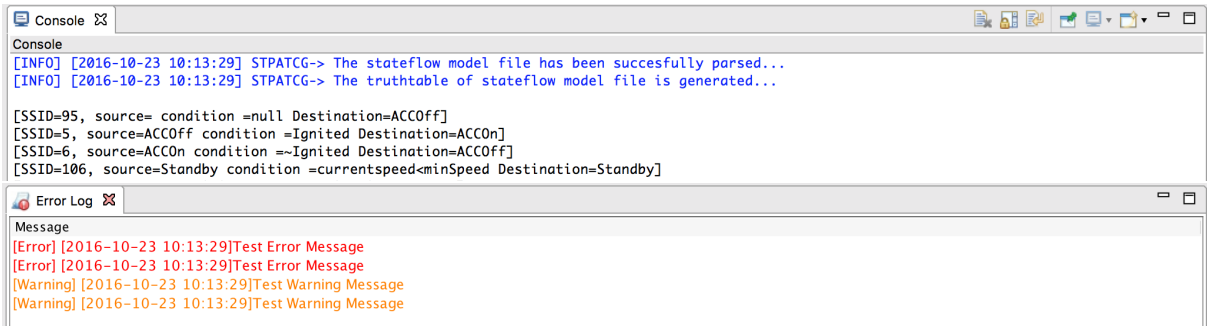


Figure 5.3: Console and Error Log View

5.1.4 Export

Users are allowed to export all tables and graphics to CSV or PNG files. Figure 5.4 illustrates all exportable contents in STPA TCGeneratorPlugin.

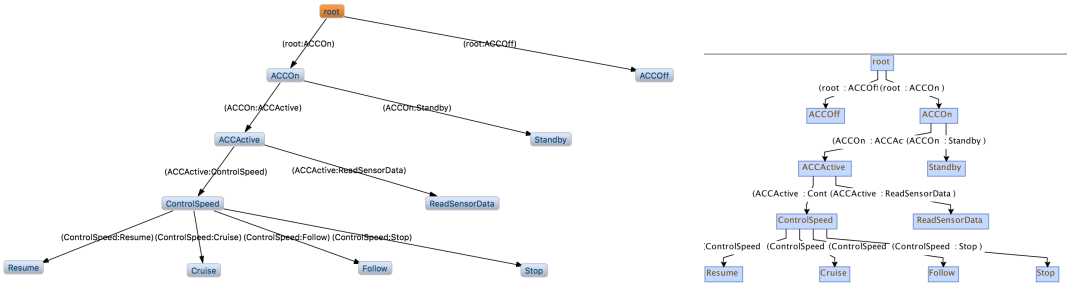


Figure 5.5: Comparison of Safe Behavioral Model Graph in STPA TCGeneratorPlugin (left) and in STPA TCGenerator (right)

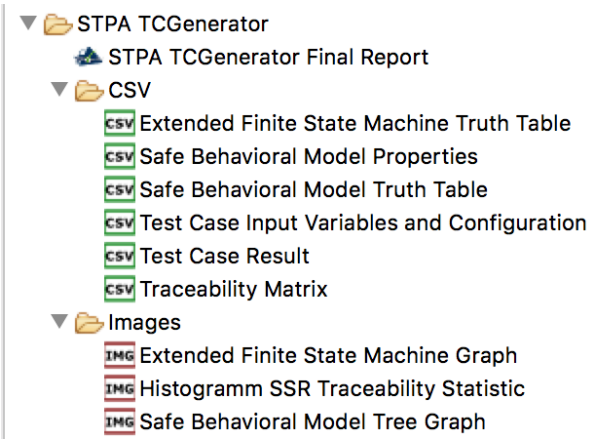


Figure 5.4: Export Wizard of STPA TCGeneratorPlugin

5.2 Improvement of the Graphical Visualization

To give the user an intuitive overview of the safe behavioral model (SBM) and extended finite state machine (EFSM) model, both models should be visualized as tree graph. However, the graph view is not optimal in Netbeans, as some state nodes and transition connections are out of bounds of the graph area and it's hard to move a node to a right position. Therefore, an improvement of both graphics was made by using Eclipse Zest Graph[1] . All state nodes and transition connections are implemented with Zest GraphNode and Zest GraphConnection, which are held in a Zest Graph object. Users are allowed to move any nodes to any position in the editor by mouse clicking and dragging. If the graph is out of the editor bounds, both horizontal and vertical scroll bar

[1]Eclipse Zest: https://wiki.eclipse.org/Zest

are appearing automatically. The result of graph improvements is illustrated in Figure 5.5.

5.3 Improvement of the Traceability Matrix

Abdulkhaleq and Wagner [1] provided an approach to calculate the traceability matrix between extended finite state machine (EFSM) transitions and STPA software safety requirements (SSR). The similarity of traceability is defined as below:

$$Similarity = \frac{Number\,of\,the\,same\,sections\,in\,EFSM\,Transition\,Condition\,and\,SSR}{Max.\,Number\,of\,sections\,in\,SSR\,or\,EFSM\,Transition} \times 100$$

If Similarity is greater than zero, the corresponding EFSM and SSR are added as a pair into traceability matrix. However, this definition is not comprehensive since EFSM transition and SSR consist of control action, source state and transition condition. But, in above-mentioned definition, only transition condition is considered in the calculation of similarity.

Therefore, we improved this algorithm by using a new definition similarity degree, in which not only the transition condition, but also the control action and the source state are considered as features for traceability matrix. With the improved traceability matrix, we are able to limit the number of the generated test cases by adding the minimum similarity degree of traceability matrix, so that the reasonable test cases for each software safety requirement are generated.

5.3.1 Similarity Degree

Similarity degree is used to measure the similarity between STPA safety requirement and EFSM transition, which is calculated with the following function:

(5.1) $$Similarity\,Degree = WCA * SCA + WS * SS + WTC * STC$$

Where

WCA: Weight of Control Action
WS: Weight of Source State
WTC: Weight of Transition Condition
SCA: Similarity of Control Action
SS: Similarity of Source State
STC: Similarity of Transition Condition

$$STC = \frac{Number\ of\ the\ same\ sections\ in\ TC}{Max.\ Number\ of\ TC\ sections\ in\ SSR\ or\ EFSM} \tag{5.2}$$

Algorithm 5.1 declares how to set weights for each section and how to calculate SCA and SS.

A concrete calculation example of the similarity degree in Figure 5.6 is declared as below:

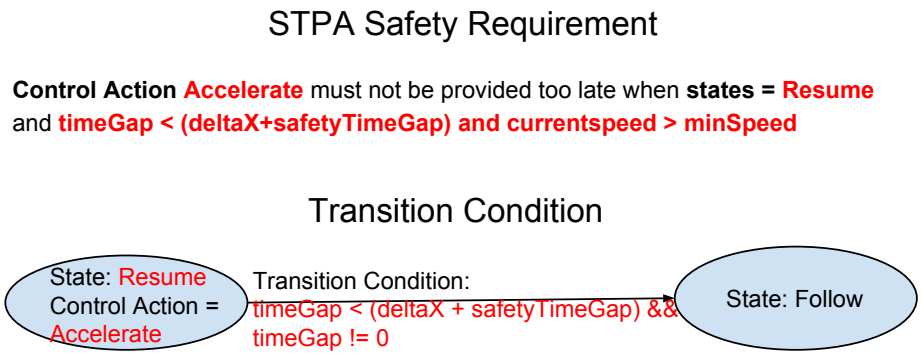


Figure 5.6: Calculate Example for Similarity Degree

$$WCA = 0.33$$
$$WS = 0.33$$
$$WTC = 0.34$$

$$Similarity\ Degree = 0.33 * 1 + 0.33 * 1 + 0.34 * \tfrac{1}{2} = 83\%$$

The algorithm of calculating similarity degree between SSR and efsmTrans is described in algorithm 5.1 and algorithm 5.2. Algorithm 5.1 takes SimpleSTPAConstraint (ssr)

and SimpleEFSMTransition ($efsmTrans$) as input. SimpleSTPAConstaints and SimpleEFSMTransition are two Java classes, both consist of ID, control action, source state and transition conditions, each for STPA safety requirement or EFSM transition. The process of calculating similarity can be described as follows:

1. The algorithm starts by setting values for weight of control action (WCA), weight of the source state (WS) and weight of the transition condition (WTC):
   - If ssr doesn't contain control action, WCA gets the value of 0. WS and WTC take the value of 0.5.
   - Otherwise, WCA, WS and WTC get each of the value 0.33, 0.33 and 0.34.
2. Comparing control action in ssr ($ssrCA$) and EFSM transition ($efsmCA$).
   - If $ssrCA$ equals $efsmCA$, the similarity of control action (SCA) takes the value of 1.
   - Otherwise, SCA takes the value of 0.
3. Comparing source state in ssr ($ssrSS$) and EFSM transition($efsmSS$).
   - If $ssrSS$ equals $efsmSS$, the similarity of the source state (SS) takes the value of 1.
   - Otherwise, SS takes the value of 0.
4. Calculating the similarity between two words list of transition conditions in ssr ($ssrTC$) and in EFSM ($efsmTC$). The concrete approach is described in algorithm 5.2.
5. Calculating the result of similarity with the above declared function 5.1.

Algorithmus 5.1 Calculate Similarity between SSR and efsmTrans

Input:

ssr: Simple STPA Constraint - structured and normalized STPA Safety Constraint

efsmTrans: Simple EFSM Transition - structured and normalized EFSM Transition

Data:

WCA = Weight of Control Action

WS = Weight of Source State

WTC = Weight of Transition Condition

SCA = Similarity of Control Action

SS = Similarity of Source State

STC = Similarity of Transition Condition

ssrCA = control action in ssr

ssrSS = source state in ssr

ssrTC = A list of transition conditions in ssr

efsmCA = control action in EFSM transition

efsmSS = source state in EFSM transition

efsmTC = A list of transition conditions in EFSM

Output:

result = the result of similarity degree

Description:

procedure CALSIMILARITYSSRANDTC(ssr,$efsmTrans$)
  if ssr.getControlAction() is empty **then**
    WCA $\leftarrow 0$
    WS $\leftarrow 0.5$
    WTC $\leftarrow 0.5$
  else
    WCA $\leftarrow 0.33$
    WS $\leftarrow 0.33$
    WTC $\leftarrow 0.34$
  end if
  SCA $\leftarrow 0$
  if ssrCA != $null$ **then**
    if ssrCA.equals($efsmCA$) **then**
      SCA $\leftarrow 1$
    end if
  end if
  SS $\leftarrow 0$
  if ssrSS.equals($efsmSS$) **then**
    SS $\leftarrow 1$
  end if
  STC $\leftarrow$ CALWORDSSIMILARITY($ssrTC$,$efsmTC$)
  result $\leftarrow$ WCA $*$ CAS $+$ WS $*$ SS $+$ WTC $*$ TCS
return result
end procedure

Algorithmus 5.2 Calculate Similarity of Two Words List

Input:
words1: A list of words
words2: A list of words
Data:
score = A variable for counting number of the same words in two lists
NS = Maximum number of sections in two Lists
Output:
similarity = similarity degree of two words list
Description:

```
procedure CalWordsSimilarity(words1,words2)
    score ← 0.0
    for all s1 ∈ words1 do
        for all s2 ∈ words2 do
            if s1.equals(s2) then
                score ← score + 1
            end if
        end for
    end for
    NS ← Max(words1.size(), words2.size())
    similarity ← score/NS
return similarity
end procedure
```

5.3.2 Algorithm of Generating Traceability Matrix

By using similarity degree, the traceability matrix between SSR and EFSM transitions is generated with the algorithm 5.3, which takes a list of STPA safety requirements in nature language (STPAConstraints), a list of EFSM State Transitions (efsmST), an object of extended finite state machine (FSM) and the minimum similarity degree of traceability matrix (minSimilarity) as input. The process of generating traceability matrix can be described as follows:

1. The algorithm starts by splitting and normalization of STPAConstraints and efsmST and saves the result in $issrList$ and $efsmTransList$, which contains the objects in form of SimpleSTPAConstraint or SimpleEFSMTransition.
2. A new array object obj will be created to store all the generated traceabilities.

3. Calculating the similarity for each element ssr in $ssrList$ and $efsmTrans$ in $efsmTransList$. The algorithm of calculating similarity between ssr and $efsmTrans$ was declared in algorithm 5.1.

4. If the similarity between ssr and $efsmTrans$ larger than minimum required similarity ($minSimilarity$), their traceability will be added into traceability matrix table model ($TMTM$).

Algorithmus 5.3 Generating Traceability Matrix

Input:

STPAConstraints:	A list of STPA safety requirements in nature language.
efsmST	A list of EFSM State Transitions
FSM	An object of extended finite state machine
minSimilarity	The minimum required similarity degree of traceability matrix

Data:

ssrList =	A list of normalized and structured STPA constraints. Each element contains id, control action, source state and transition conditions.
efsmTransList =	A list of normalized and structured EFSM transitions. Each element contains id, control action, source state and transition conditions.
obj =	An array object with String

Output: TMTM = Traceability Matrix Table Model

Description:

```
procedure GenTraceabilityMatrix(STPAConstraints, efsmST, FSM, minSimilarity)
    ssrList ←SplitAndNormalizeSSR(STPAConstraints)
    efsmTransList ←SplitAndNormalizeFSMTrans(FSM, efsmST)
    create new obj ← new String[3]
    for all ssr ∈ ssrList do
        for all efsmTrans ∈ efsmTransList do
            similarity ← CalSimilaritySSRandTC(ssr, efsmTrans)
            if similarity >= minSimilarity then
                obj[0] ← efsmTrans.getId()
                obj[1] ← ssr.getId()
                obj[2] ← similarity.toString()
                TMTM.addRow(obj)
            end if
        end for
    end for
return TMTM
end procedure
```

5.4 Improvement of Result Presentation

Two improvements have been implemented for the result presentation.

In STPA TCGeneratorPlugin, users are allowed to choose if the test cases are saved in one test suit or in more test suits. Therefore, the result of the generated test cases are presented in a tree structure either in one or more nodes for the test suits.

Based on the traceability matrix between the model and the STPA software safety requirements, the STPA TCGeneratorPlugin provides an individual coverage (how many test cases TC covered each SSR), which is shown in a view of a histogram. This view was implemented by using JFreeChart [1]. The histogram will be shown in chapter 6.4 with a concrete use case example.

[1] JFreeChart: http://www.jfree.org/jfreechart/

6 Evaluation Example

The evaluation example is based on the example that is presented in paper [1] by Abdulkhaleq and Wagner.

In order to evaluate the functions of the STPA TCGeneratorPlugin, an ACC ("Automatic Cruise Control") with Start / Stop Simulator[1] [2] was developed by Dennis Maseluk and Asim Abdulkhaleq at the University of Stuttgart. For this purpose a Lego MINDSTORM EV3 robot was equipped with a simulation software developed in ANSI-C. The ACC system was then tested by means of an EV3 Ultrasonic sensor in a simulation of a driving situation behind another EV3 robot.

An STPA software safety analysis of the simulator was performed by Abdulkhaleq and Wagner [1] using A-STPA and XSTPA in the XSTAMPP Platform, whereby the basic principles of a software safety analysis were established on the basis of the process, which was presented in chapter 2.1. Figure 6.1 shows the control structure for the ACCSimulator with all the system variables required for the safety analysis, which is required as a basis for the derivation of formal software safety requirements (SSR). All results of software safety analysis were saved in an STPA-project file with the name *ACCSimulator.hazx*.

Based on ACCSimulator STPA-project[3], Abdulkhaleq and Wagner[1] created a Simulink/-Matlab stateflow model to visualize a safe behavioral model of the ACC Simulator. The safe behavioral model is saved in a Simulink file called *NewACCSimulator.xml*.

[1] http://www.iste.uni-stuttgart.de/en/se/forschung/werkzeuge/acc-simulator.html
[2] https://sourceforge.net/projects/acc-with-stop-and-go-simulator/
[3] ACCSimulator STPA-project was created by Abdulkhaleq and Wagner

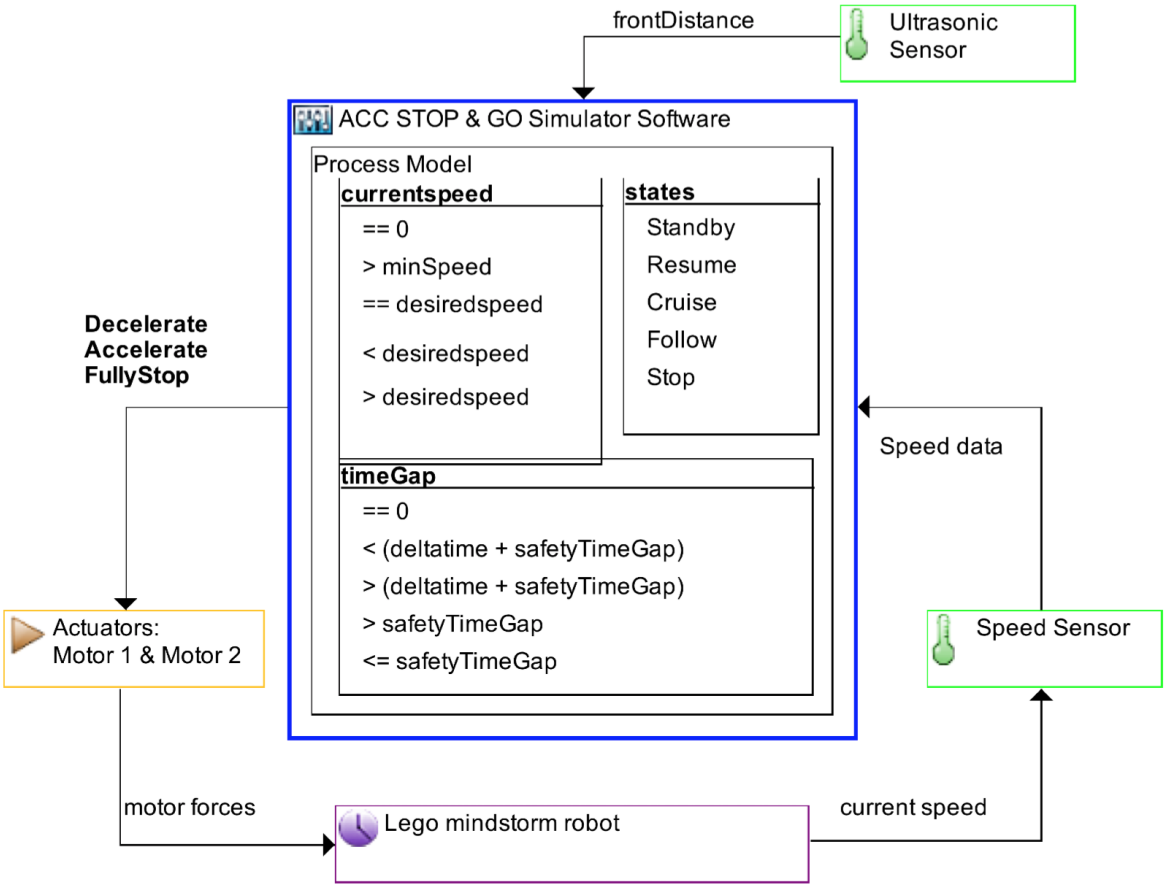


Figure 6.1: Control Structure of ACC Simulator [1]

6.1 Visualization and Validation of the Safe Behavioral Model (SBM)

The safe behavioral model (see Figure 6.2) is parsed automatically from the imported XML file *NewACCSimulator.xml* into a Java-model in the background process. As a result of that, a hierarchical tree graph of SBM (see Figure 6.3) is visualized in editor.

The properties and transition truth table of SBM are shown in tables, in which all states, input variables, output variables and local variables of a safe behavioral model are classified and displayed. Figure 6.4 shows an example of the states table. Users are allowed to modify the name of each state and variables and save the modification by clicking the save button at the end of each row. As a result, the modified name of states

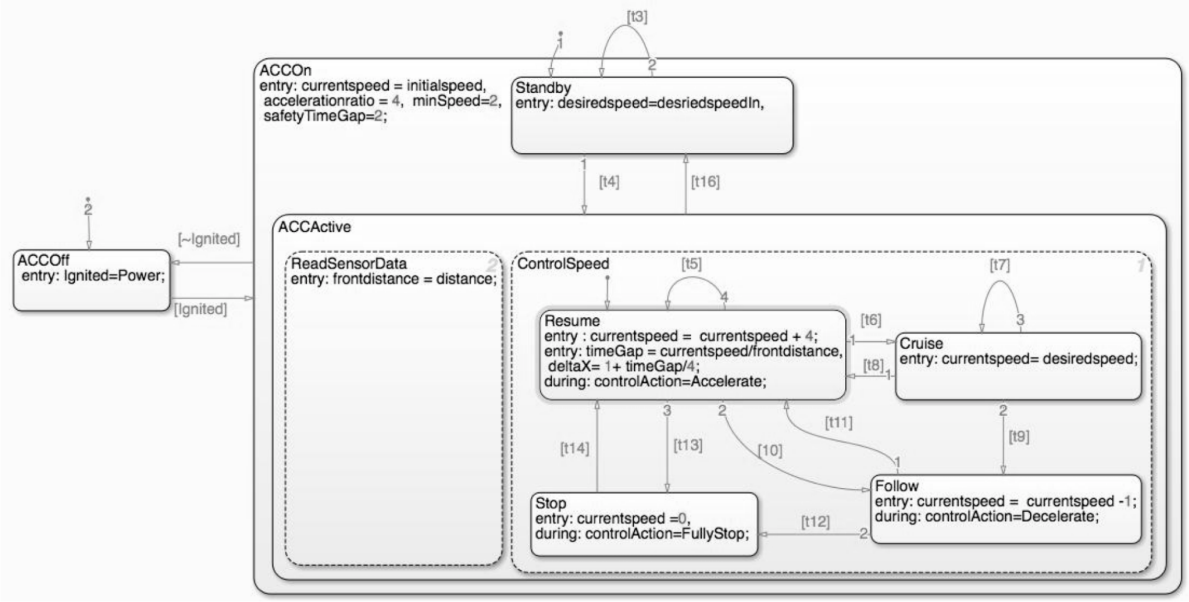


Figure 6.2: Safe behavioral model of the ACC software controller [1]

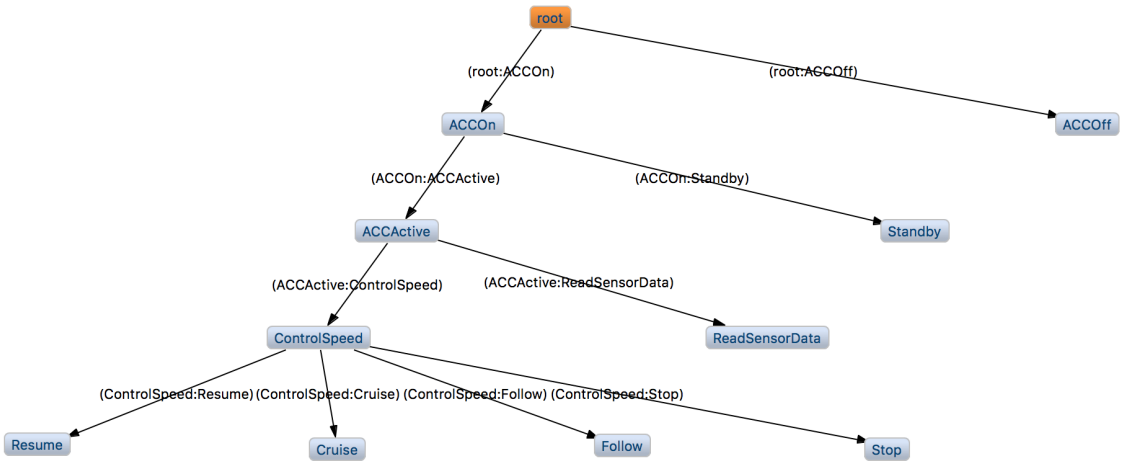


Figure 6.3: Hierarchical Tree Graph of the Safe Behavioral Model

and variables will be changed in the relevant XML file as well. All modifications will be valid by a restart of STPA TCGeneratorPlugin.

Furthermore, a view of validation between STPA and SBM is shown to help the user validating safe behavioral model according to STPA data model before the SMV model is generated. Users are able to modify the name and save the modification by clicking the save button. The modification is saved in the relevant XML file and will be valid by a restart of STPA TCGeneratorPlugin. In our example, an error was found in control

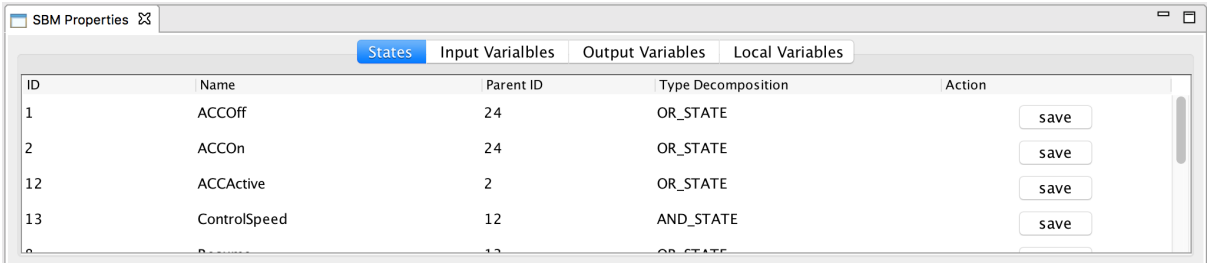


Figure 6.4: Properties Tables of the Safe Behavioral Model

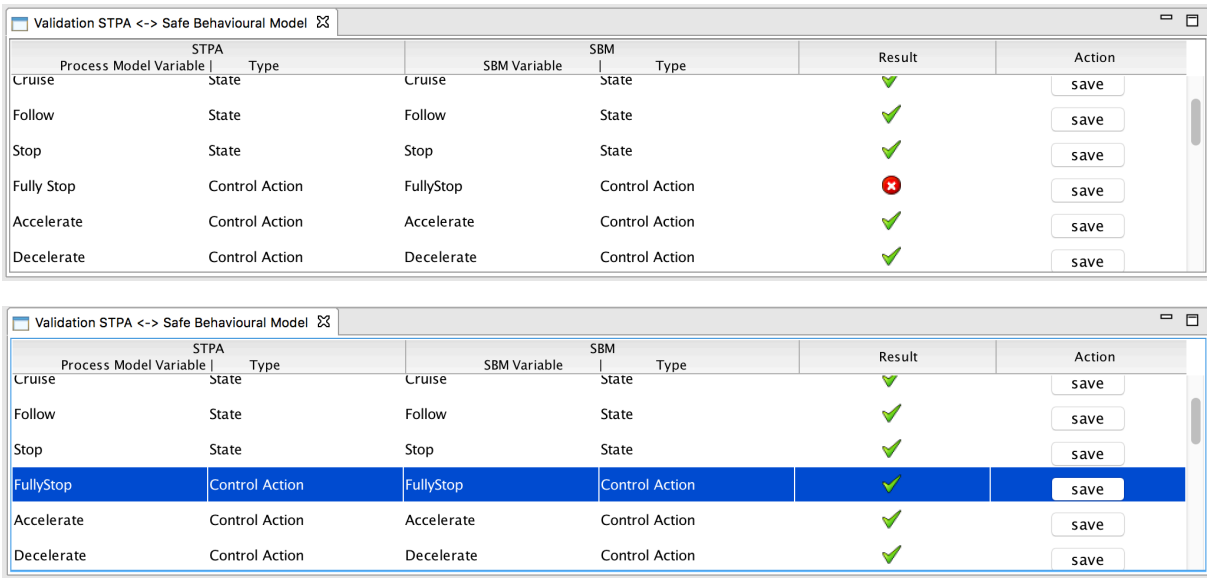


Figure 6.5: Validation of the Safe Behavioral Model – Top: validation result with detected error, Bottom: validation result after correcting error

action variable "Fully Stop", which should be written without the white space. After correction and saving, there is no more error in the validation result. Figure 6.5 shows the above described validation process. The meaning of check result icons is declared in table 6.1.

6.2 Generating and Verification SMV Model

To validate the correctness of the safe behavioural model, a verification input - SMV model is generated for the NuSVM model checker. SMV model can be automatically generated with or without the consideration of the STPA data model. In both cases, all states, transitions and data variables of the safe behavioural model are mapped to SMV model specifications. The generated SMV model is opened in a text editor and saved in a

Table 6.1: Description of Icons in the Validation STPA and SBM Table

Icon	Description
	Check result is correct. That means, the matching state, control action or variable has been found with exactly the same name.
	Check result can be true or false. That means, either a matching process variable in STPA data model was not found or the found process variable has a similar name with the SBM data variable. This can be considered as a warning.
	Check result is wrong. That means, the fitting state or control action cannot be found in SBM or the name of them doesn't match.

```
--#########################################
--This model is automtically generated by SMVGenerator tool which is developed by Asim Abdulkhaleq, Stefan Wagner
--University of Stuttgart, Institute of Software Technology,  Germany
--Copyright (c) 2016, at Institute of Software Technology, Software Engineering Group-2016
--Date/Time:2016/11/06 14:26:39

--#########################################

MODULE Sub_ControlSpeed(Power,currentspeed,desriedspeedIn,timeGap,deltaX,minSpeed,safetyTimeGap,frontdistance,controlAction,initialspeed,a
VAR

states: {Resume ,Cruise ,Follow ,Stop };
ASSIGN

init (states):=Resume;


next (states):=case
TRUE:{Resume};
states=Resume & (currentspeed < desiredspeed & timeGap > safetyTimeGap) : Resume;
states=Cruise & (timeGap > (deltaX + safetyTimeGap) & currentspeed = desiredspeed) : Cruise;
states=Cruise & (currentspeed < desiredspeed  & timeGap > (deltaX + safetyTimeGap)) : Resume;
states=Resume & (currentspeed = desiredspeed & timeGap > safetyTimeGap) : Cruise;
states=Follow & (timeGap > (deltaX + safetyTimeGap) & frontdistance > 10) : Resume;
states=Cruise & (timeGap < (deltaX + safetyTimeGap)) : Follow;
states=Stop & (timeGap > safetyTimeGap | frontdistance > 10) : Resume;
states=Resume & (timeGap = 0) : Stop;
states=Resume & (timeGap < (deltaX + safetyTimeGap) & timeGap != 0) : Follow;
states=Follow & ((timeGap <= safetyTimeGap & frontdistance < 10 )) : Stop;
TRUE: {Resume ,Cruise ,Follow ,Stop };
esac;
```

Figure 6.6: An Example of SMV model - ACCSimulator.smv

file called "ACCSimulator.smv" (see Figure 6.6). In case that the SMV model is generated with the consideration of STPA data model, the software safety requirements (SSR) are added into the SMV model in the form of LTL[9] and a view of LTL Table (see Figure 6.7) is shown with all SSR and their verification result.

SMV model can be verified with or without STPA Verifier plug-in:

1. Verify without STPA Verifier: In this case, the SMV model is verified with model checker NuSMV[1]. the user should choose the installed location for NuSMV program, which will be executed in STPA TCGeneratorPlugin. Once the verification successfully completed, the LTL table is updated with verification result (see Figure

[1]NuSMV: http://nusmv.fbk.eu

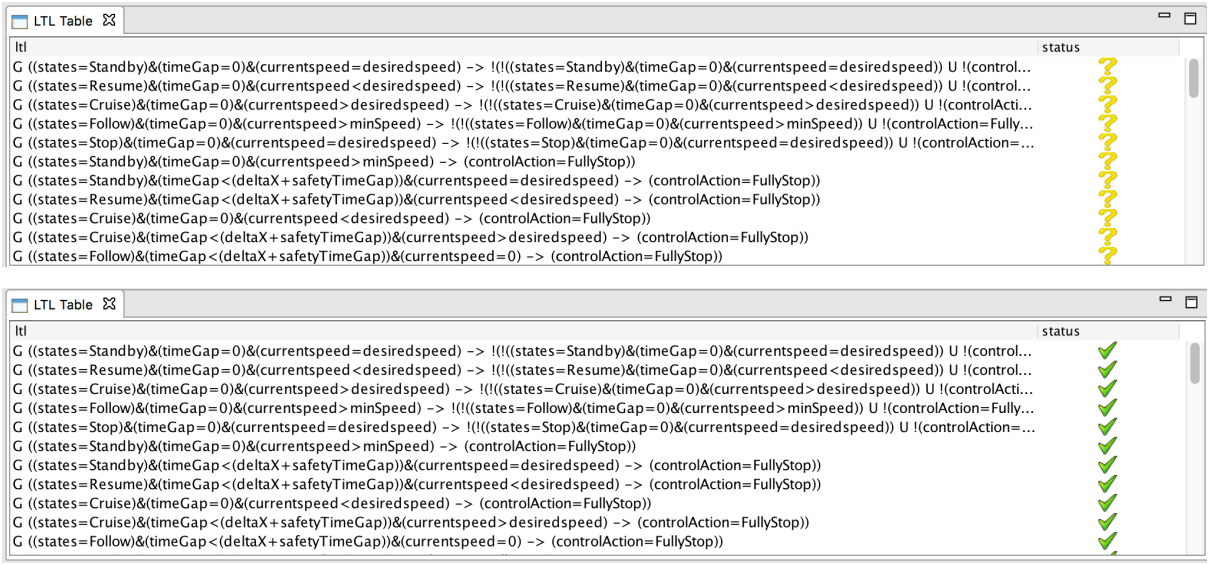


Figure 6.7: LTL Check Result - Top: LTL table before verification, Bottom: Result after verification of LTL

Table 6.2: Description of Icons in the LTL Table

Icon	Description
✓	SSR has been checked and the result is true.
?	SSR has not been checked.
✗	SSR has been checked and the result is false.

6.7). Otherwise, execution errors are shown in the error log view. The meaning of verification result is explained in Table 6.2.

2. Verify with STPA Verifier: The external plug-in STPA Verifier can be opened with the button "Open STPA Verifier". This button is shown in XSTAMPP if STPA Verifier was installed, otherwise a dialog with installation information is popped up.

6.3 Build Safe Test Model

After validating the correctness of the safe behavioral model, a safe test model was generated automatically by eliminating super states (states with parent and children) in the safe behavioral model and the graphical view of safe test model was shown in the editor(see Figure 6.8).

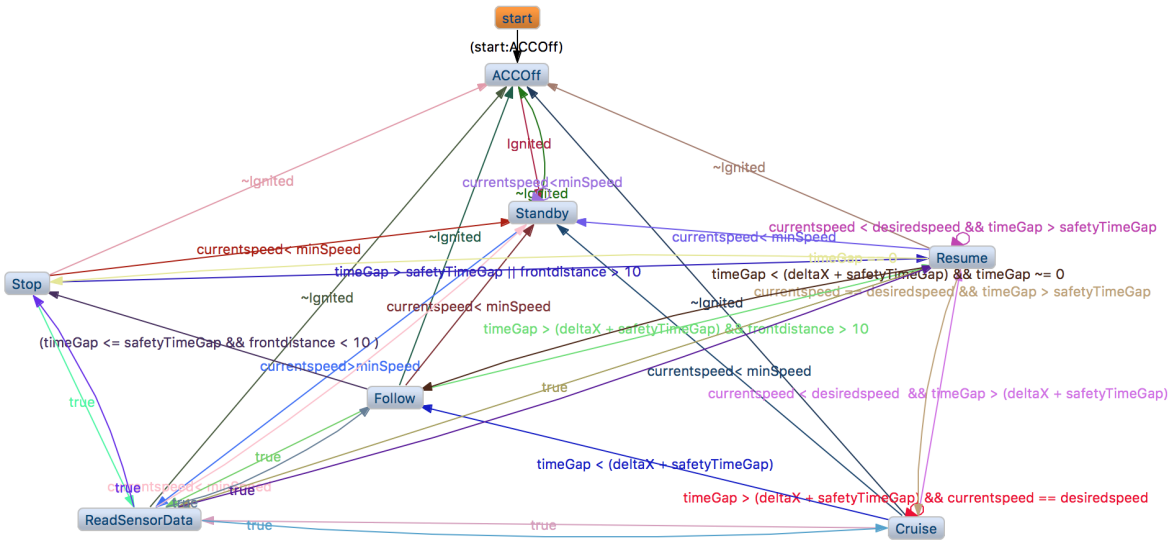


Figure 6.8: Graphical Visualization of Extended Finite State Machine

6.4 Generate Test Cases

STPA TCGeneratorPlugin automatically generates the traceability matrix between STPA software safety requirements and the safe behavioral model and shows them in a table. The minimum degree of traceability matrix was set to 50%. All input data variables with their data type, initial, minimum, maximum values are shown in the test input configuration view. Before generating test cases, we set input data variables and test configuration data exactly the same as in Figure 6.9, then we ran three times of generating test cases, each with depth-first search, breadth-first search and the random walk with both algorithms.

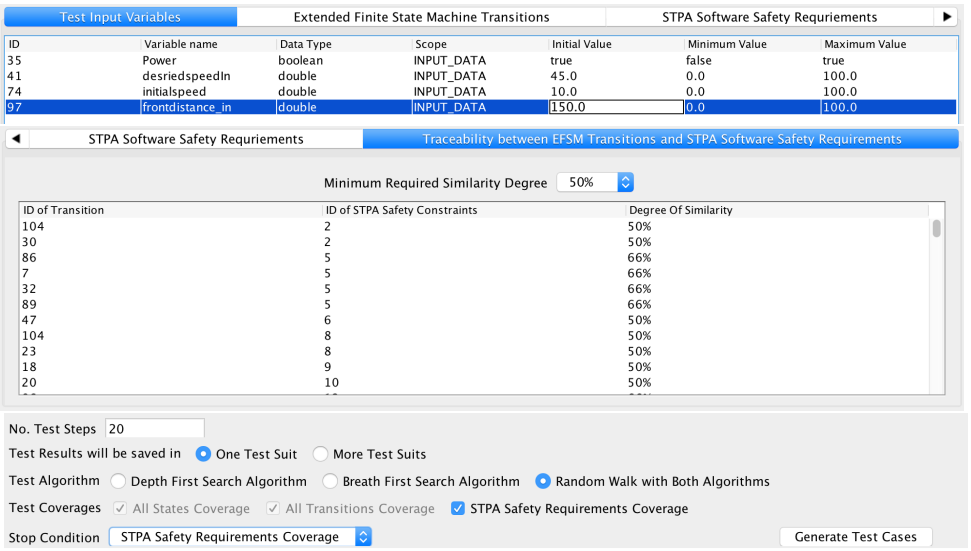


Figure 6.9: Configuration Before Generating Test Cases
top: Input Variables, middle: View of Traceability Matrix , bottom: configuration of generating test cases

Table 6.3: Information about generated test cases by STPA TCGeneratorPlugin

Test Algorithm	Test Suits	Test Cases	Test Steps	Time (in Sec)	State Coverage	Transition Coverage	STPA SSR Coverage
DFS	1	29	20	< 1	$7/7 = 100\%$	$29/32 = 90.62\%$	$44/45 = 97.78\%$
BFS	1	120	20	1	$7/7 = 100\%$	$29/32 = 90.62\%$	$44/45 = 97.78\%$
Both	1	120	20	1	$7/7 = 100\%$	$28/32 = 87.5\%$	$44/45 = 97.78\%$

Results of generated test cases are shown in table 6.3. Furthermore, Figure 6.10 shows the individual coverage (how many test cases covered each SSR) by each test algorithm.

The generated test cases have a relatively good coverage of STPA software safety requirement to 97.78%. The number of test cases is well controlled and all test cases were in a short time completely generated. If user wants to include more STPA software safety requirements and get a higher coverage, the minimum required similarity degree should be reduced before. For example, if the minimum required similarity degree was reduced to 30%, all 55 STPA software safety requirements were traced and the coverage of STPA software safety requirements in test cases can be reached to 100%.

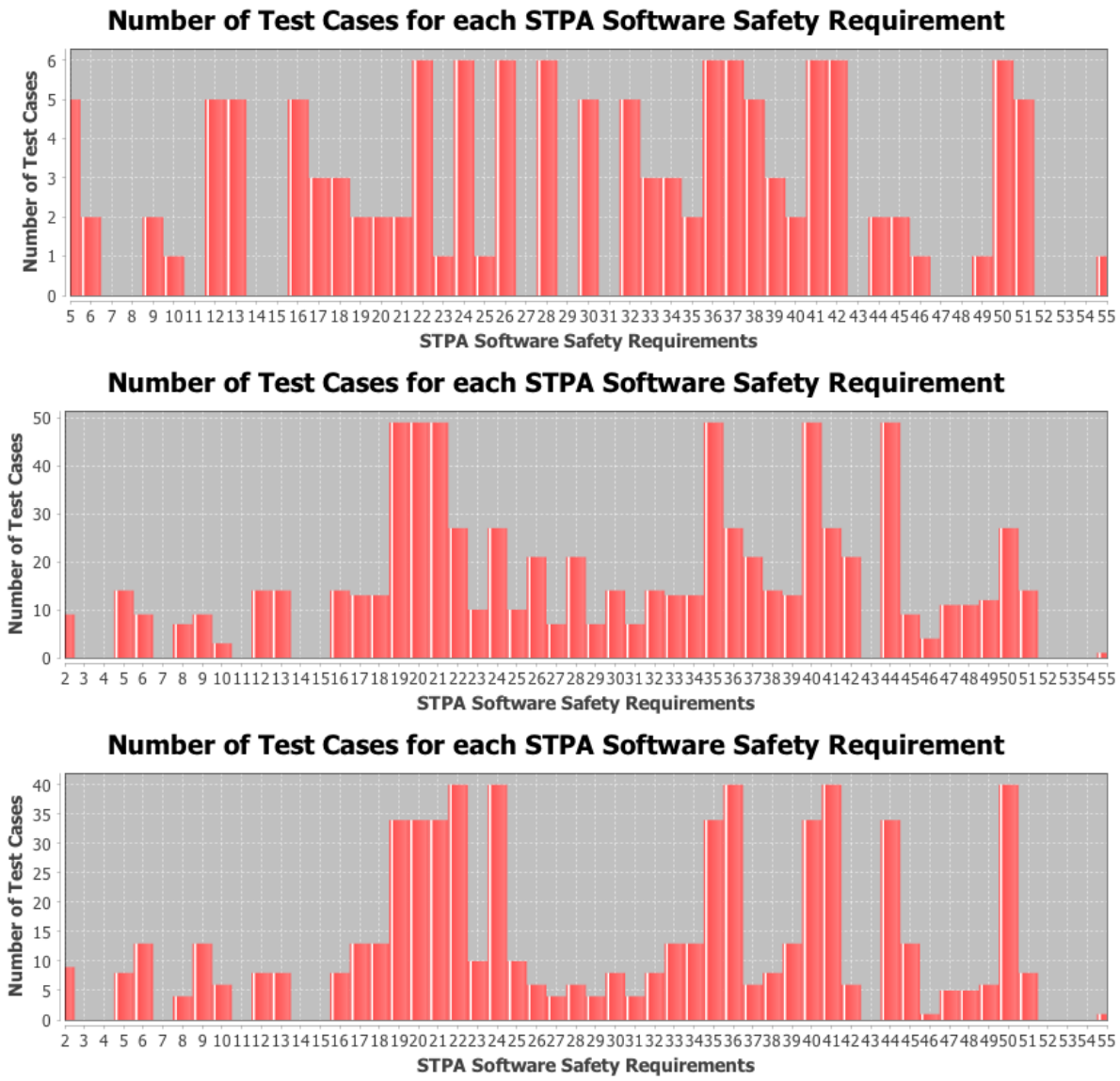


Figure 6.10: The number of test cases for each software safety requirement
Test cases generated with – Top: depth first search, Middle: breadth first search, Bottom: random walk of both search

7 User Manual

In this chapter, the system requirements and the setting up of the STPA TCGeneratorPlugin are presented.

The STPA TCGeneratorPlugin places the following requirements on the system:

- at least 1 GB RAM (recommended 2)
- 200 MB of hard disk space (for XSTAMPP + STPA TCGeneratorPlugin)
- at least one dual core processor (for example, Intel Core i3)

In order to use all functions of the STPA TCGeneratorPlugin, the following additional programs must be installed on the computer:

- Java 7 Runtime [1]
- NuSMV[2]: The current version 2.6.0 is recommended, but at least NuSMV 2.0 is required since older versions do not support BMC.
- XSTAMPP[3]: The STPA-TCGenerator is installed as a plug-in for the XSTAMPP Platform version 2.0.2.
- STPA Verifier[4](optional): The STPA Verifier can be used as external plug-in for checking SVM model, which is generated by STPA TCGeneratorPlugin.

The STPA TCGenerator Plug-in will be available on the XSTAMPP home page under Tools → STPA TCGeneratorPlugin. To download this tool, following steps should be performed:

1. Installation of XSTAMPP: the corresponding archive file must be downloaded and unpacked to the desired installation directory.

[1]http://www.oracle.com/technetwork/java/javase/downloads/index.html
[2]The model checker NuSMV can be downloaded from http://nusmv.fbk.eu/NuSMV/
[3]The XSTAMPP can be downloaded from http://www.xstampp.de/Download.html
[4]The STPA Verifier can be downloaded from http://www.xstampp.de/Download.html

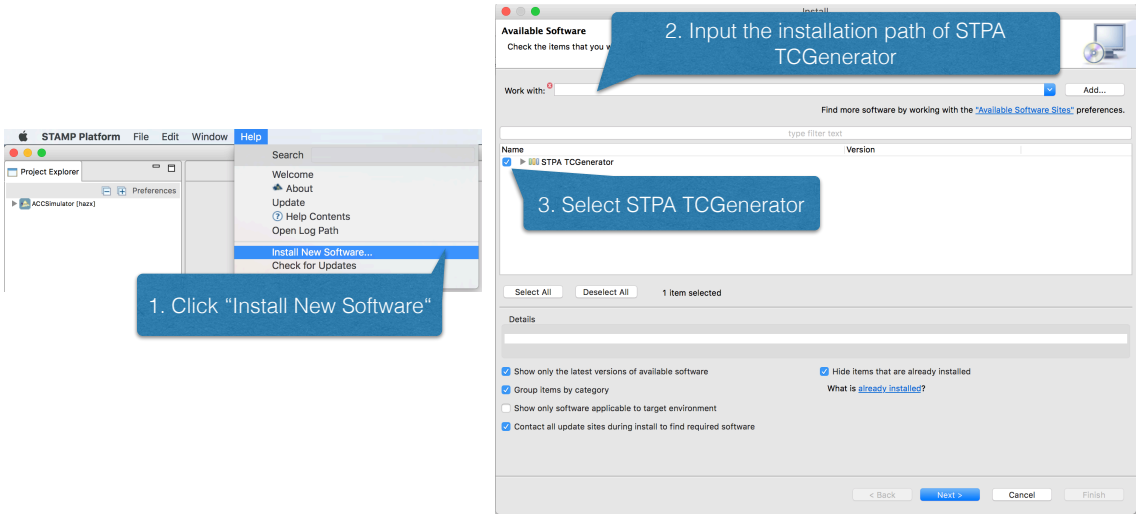
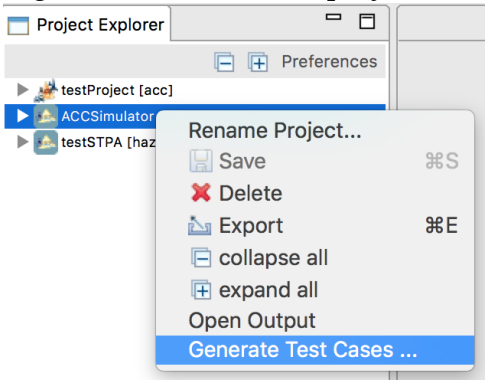


Figure 7.1: Installation STPA TCGenerator Plug-in into XSTAMPP

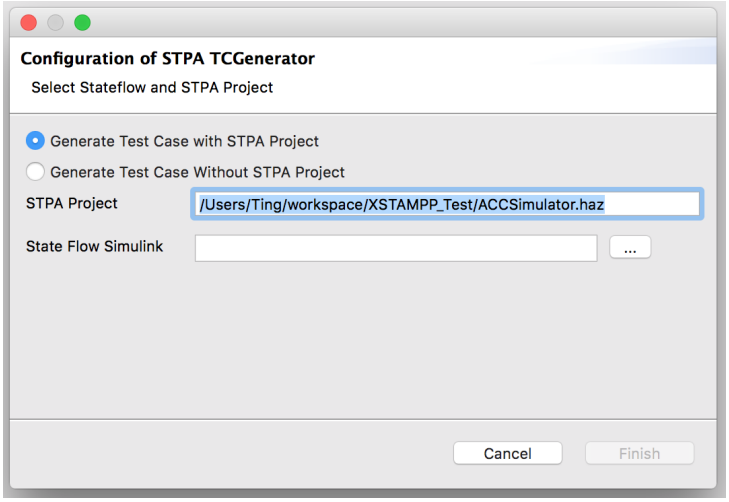
2. Installation of the STPA TCGeneratorPlugin:
   2.1. Download the STPA TCGeneratorPlugin update archive file from http://www.xstampp.de/Download.html
   2.2. Installation of plug-in in XSTAMPP as shown in Figure 7.1.

After the success of the installation, the user can use STPA TCGeneratorPlugin in XSTAMPP as follows:

1. Right click on an STPA project and choose "Generate Test Cases...":



2. Import State Flow Simulink file with the file browser and then click "finish":



3. Generate test cases with following steps:
   Generate SMV → Verify → Build Safe Test Model → Generate Test Cases

4. Export the final report, images and tables with the export icon in the main toolbar.

8 Conclusion and Future Work

8.1 Conclusion

Based on the STPA TCGenerator in Netbeans, STPA TCGenerator plug-in (STPA TCGeneratorPlugin) for XSTAMPP was developed in eclipse. In comparison of STPA TCGenerator, following improvements were made in this work:

- The graphical visualization of safe behavioral model (SBM) and extended finite state machine (EFSM) model were improved by using Eclipse Zest. Thereby, all nodes and connections are illustrated more clearly inside the bounds of graph area and users are able to move each node to a right position.
- The traceability matrix between extended finite state machine (EFSM) transitions and the STPA software safety requirements were improved by using the concept of similarity degree, in which not only the EFSM transition conditions but also the source state and the control actions were considered as influencing factors.
- The result of the generated test cases was represented in table and organized in tree structure. Furthermore, the coverage of STPA software safety requirements (SSR) in test cases was illustrated intuitively with a histogram. Finally, all tables and graphics are allowed to be saved in CSV or image files.

8.2 Future Work

As a future work, there are many possible directions to improve and extend the STPA TCGeneratorPlugin:
First of all, the graphical visualization could be improved by adding zooming functions. In this work, the graph view of the safe behavioral model and the extended finite state machine are generated with a default size. With zooming function, the users are able to view, modify and export the graphics with more detailed information. Furthermore, The connection between STPA TCGeneratorPlugin and STPA Verifier could be more convenient. In such a case, STPA Verifier would be opened directly from

STPA TCGeneratorPlugin and the generated SMV file as well as STPA software safety requirements in the form of LTL are automatically imported into STPA Verifier. Finally, to give the users more freedom of modification of the traceability matrix and to get a better control of test cases, it would be appreciated allowing the user to customize the traceability matrix by adding or removing traceability matrix items.

Bibliography

[1] A. Abdulkhaleq, S. Wagner. “An Automatic Safety-based Test Case Generation Approach based on STPA.” 2016. URL: http://elib.uni-stuttgart.de/handle/11682/8925.

[2] A. Abdulkhaleq, S. Wagner. “A-STPA: Open tool support for system-theoretic process analysis.” In: *Proc. 2014 STAMP Conference at MIT*. Boston, USA, 2014.

[3] A. Abdulkhaleq, S. Wagner. “XSTAMPP An eXtensible STAMP Platform Support for Safety Engineering.” In: *Proc. 2015 STAMP Conference at MIT*. Boston, USA, 2015.

[4] A. Abdulkhaleqa, S. Wagnera, N. G. Leveson. “A comprehensive safety engineering approach for software-intensive systems based on STPA.” In: *Procedia Engineering* 128 (2015), pp. 2–11.

[5] L. Balzer. *Entwicklung eines STPA-Verifiers als Eclipse-Plug-in für die Verifikation von Software- Sicherheitsanforderungen*. University of Stuttgart. 2016.

[6] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, B. M. Horowitz. “Model-based testing in practice.” In: *Proceedings of the 21st International Conference on Software Engineering, ser. ICSE 99*. New York, NY, USA: ACM, 1999, pp. 285–294.

[7] D. Graham, E. V. Veenendaal, I. Evans, R. Black. *Foundations of Software Testing, ISTQB Certification, Reserved Edition*. Cengage Learning EMEA, 2008.

[8] H. P. Gumm. *CTL Model Checking*. 2007. URL: https://www.mathematik.uni-marburg.de/~gumm/Lehre/SS07/ModelChecking/06_CTL_Model_Checking.pdf.

[9] H. P. Gumm. *Lineare Temporale Logik*. 2007. URL: https://www.mathematik.uni-marburg.de/~gumm/Lehre/SS07/ModelChecking/07_LTL_Model_Checking.pdf.

[10] G. J. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison-Wesley Reading. 2004.

[11] *J.E. Troyan, L.Y. Le Vine, HAZOP, Loss Prevention 2:215, 1968*.

[12] N. G. Leveson. “Completeness in formal specification language design for process-control systems.” In: *Proceedings of the Third Workshop on Formal Methods in Software Practice, ser. FMSP ’00*. New York, NY, USA: ACM, 2000, pp. 75–87.

[13] N. G. Leveson. *Engineering a Safe World, Systems Thinking Applied to Safety*. The MIT press, 2012.

[14] T. Mandel. *The elements of user interface design*. John Wiley and Sons, Inc. New York, NY, USA, 1997.

[15] K. L. McMillan. *Symbolic model checking*. Springer, 1993.

[16] E. Ort, B. Mehta. *Java architecture for XML binding (JAXB)*. 2003. URL: http://www.oracle.com/%20technetwork/articles/javase/index-140168.html.

[17] *Plug-in Development Environment Guide*. URL: http://help.eclipse.org/kepler/index.jsp?topic=/org.eclipse.pde.doc.user/guide/tools/export_wizards/export_plugins.htm.

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

[19] J. Thomas. “Extending and automating a systems-theoretic hazard analysis for requirements generation and analysis.” PhD thesis. Massachusetts Institute of Technology, Apr. 2013.

[20] M. Utting, B. Legeard. *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

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

All links were last followed on November 6, 2016.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature