A Software Safety Verification Method Based on System-Theoretic Process Analysis

Asim Abdulkhaleq and Stefan Wagner

Institute of Software Technology, University of Stuttgart,
Universitätsstraße 38,70569 Stuttgart, Germany
{Asim.Abdulkhaleq,Stefan.Wagner}@informatik.uni-stuttgart.de
http://www.iste.uni-stuttgart.de/se.html

Abstract. Modern safety-critical systems are increasingly reliant on software. Software safety is an important aspect in developing safety-critical systems, and it must be considered in the context of the system level into which the software will be embedded. STPA (System-Theoretic Process Analysis) is a modern safety analysis approach which aims to identify the potential hazardous causes in complex safety-critical systems at the system level. To assure that these hazardous causes of an unsafe software's behaviour cannot happen, safety verification involves demonstrating whether the software fulfills those safety requirements and will not result in a hazardous state. We propose a method for verifying of software safety requirements which are derived at the system level to provide evidence that the hazardous causes cannot occur (or reduce the associated risk to a low acceptable level). We applied the method to a cruise control prototype to show the feasibility of the proposed method.

Keywords: STPA approach, software safety analysis, temporal logic, safety verification, formal verification methods.

1 Introduction

A safety-critical system is a system that can cause undesired loss or harm to human life, property, or the environment, whereas safety-critical software is software that can contribute to such loss or harm [1]. A software cannot directly cause loss or harm, but it may control some equipment that may cause accidents [2]. Therefore, many examples of safety systems which have failed due to software related faults: the loss of Ariane 5 [4], Therac-25 [3], and more recently Boeing 777-200 [8] and the Toyota Prius. Many software related accidents and major losses are the result of incompleteness or other flaws in the software requirements, not coding errors [1]. Safety is a system problem; therefore, to understand the safety aspects of software, it is necessary first to understand the general field of system safety.

STPA (System-Theoretic Process Analysis) [5] is an approach developed by Leveson to identify safety requirements and constraints at the system level. In STPA, the system is seen as a set of control loops (comprising interacting components involving software) which interact with each other. STPA uses the existing knowledge about a system to guide the safety analysis process; therefore, it is not necessary to have knowledge about the details of implementation.

1.1 Problem Statement

Typically, software verification focuses on proving the functional correctness of software and demonstrating that the software fully satisfies all functional requirements [16]. However, they cannot make it safe and the correctness of software cannot ensure the software is safe, or reduce the risk. Therefore, the software must be analyzed regarding the safety aspect and verified against its safety requirements at the system level [7]. As STPA is a new technique, which has proven to be effective on establishing the safety requirements and constraints at the system level (e.g. Space Shuttle Operations [18], Japanese Exploration Agency (JAXA) [19]); it has not been used for identifying software safety requirements in the system context and verifying the software against them.

1.2 Research Objectives

The overall objective of this research is to fill this gap and investigate the possibility of verifying safety-critical software against safety requirements and constraints which are derived at the system level by using STPA. To control the associated risk of the safety-critical software, we first need to identify the potential hazards and then demonstrate that a potential hazardous cause cannot occur, i.e., the software cannot contribute to an unsafe state. The main purpose of applying STPA to software in the context of a system in our method is to understand the software hazardous causes early to develop corresponding software safety requirements which should be taken into consideration. The second purpose is to reduce the amount time and effort of safety analysis and verification at the code level.

1.3 Contribution

For that, we propose a method which provides a link between the safety analysis at the system level and safety verification at the code level. This method enables the safety analyst to extract the software safety requirements at the system level and verify them at the code level.

2 Background

We give background information on the three main topics which we use in the proposed method: STPA, safety verification, and formal specification and verification:

2.1 STPA

STPA [5] is a top-down system engineering approach to system safety; therefore, it can be applied early in the system development process or before a design has been created to generate high-level safety requirements and constraints. In contrast to traditional safety analysis techniques, which are based on reliability theory, STPA is more powerful in terms of identifying more causal factors and hazardous scenarios, particularly those related to software, system design and human behaviour [6]. STPA identifies systematic failures such as software design errors, hardware design errors, requirements specification errors and other operational procedures.

STPA is implemented in four steps [6]: (1) establish the fundamentals of analysis; (2) identify potentially hazardous control actions; (3) use the identified potentially hazardous control actions to create safety requirements and constraints; and (4) determine how each potentially hazardous control action could occur. In step 1, the safety analyst must identify the accidents or losses which will be considered, hazards associated with these accidents, and specify safety requirements (constraints). After establishing the fundamentals, the safety analyst must draw a preliminary (high-level) functional control structure of the system. In step 2, the analyst has to use the control structure as a guide for investigating the analysis to identify the potentially unsafe control actions. Then he or she translates them to corresponding safety constraints. In step 3, the analyst has to identify the process model variables for each controller (automated controller or human) in the control loop and analyze each path to determine how each potentially hazardous control actions could occur. At the end of the process, a recommendation for the system design should be developed for additional mitigations.

2.2 Software Safety Verification

The first step of safety verification is to verify that the software requirements are consistent with or satisfy safety constraints. Safety verification exists to provide evidence that associated risk has been reduced or eliminated [1]. Safety verification is not the same as functional verification. Functional verification assures that the software fully satisfies its specifications, while safety verification uses the results of the safety analysis process to assure that the software meets the safety requirements [20]. The safety verification can be done in two ways [1]: (1) static analysis which looks over the code and design documents of the system (e.g. fault tree, formal verification); and (2) dynamic analysis requires the execution of the software to check all of the systems safety features. Static analysis is the same as a structured code review. Systems can be proven to match requirements, but it will not catch any safety states that the requirements miss [1]. The dynamic analysis has the ability to catch unanticipated safety problems, but it cannot prove that a system is safe (e.g. software testing).

SFTA (Software Fault Tree Analysis) [1] is a static analysis technique which is primarily used to discover all potential faults such as faulty inputs or software bugs that could occur in software. SFTA has also been used for verifying software code. Leveson stated in [1] that SFTA is applicable only to small-sized software. Because the complete generating of a tree is not possible for large software.

2.3 Formal Specification and Verification Techniques

Formal verification is a very active area of research, and many promising techniques and methodologies have been invented for verifying computing systems. Theorem proving and model checking are common methods used today. Formal verification entails a mathematical proof showing that a system satisfies its desired property or specification. To do this, the property of interest must be modeled in a mathematical structure (e.g. temporal logic). Temporal Logic has been proposed by Pnueli [9] as an appropriate formalism in the specification and verification of concurrent programs. Many different versions of temporal logic have been used in the verification process such as Linear-Time Temporal Logic (LTL), and Computation Tree Logic (CTL) [23] which have been broadly used to express safety properties in a formal notation. An LTL formula consists of atomic propositions, Boolean operators ($\neg, \vee, \wedge, \leftrightarrow, \rightarrow, true, false$) and temporal operators ($\bigcirc$ next, $\Box$ always, $\Diamond$ eventually, $\mathcal{U}$ until, $\mathcal{R}$ release). CTL is an extension of classical logic that allows reasoning about an infinite tree of state transitions. Model checking is a very popular formal verification technique and has been used widely in the verification of software. It first involves building a finite state machine as a formal model of a system, and then verifying whether the property, written in some temporal logic, holds or not through an exhaustive search of the system state space. Model checkers can be used also for testing purposes to generate test cases [10].

3 Software Safety Verification Method Based on STPA

The safety analysis of safety-critical software provides the safety requirements which need to be tested. Safety verification shall be performed to verify a correct incorporation of software safety requirements [24]. Verification must show that hazards have been eliminated or controlled to an acceptable level of risk. Figure 1 shows the proposed method of software safety verification based on STPA at the system level. The method includes three main steps: (1) safety analysis of software at the system level; (2) formalization of safety requirements and constraints; and (3) verification and testing at the code level.

3.1 Safety Analysis of Software at the System Level (Step 1)

This step aims at analyzing the software in the context of the system to identify the potential hazardous causes of software that could lead or contribute to an accident. At this step, the safety analyst will apply STPA to the requirements specification of the whole system. Then he/she will extract the requirements relevant to the software in the context of the system. The safety control structure of a system will include the software in the control loop as the main component

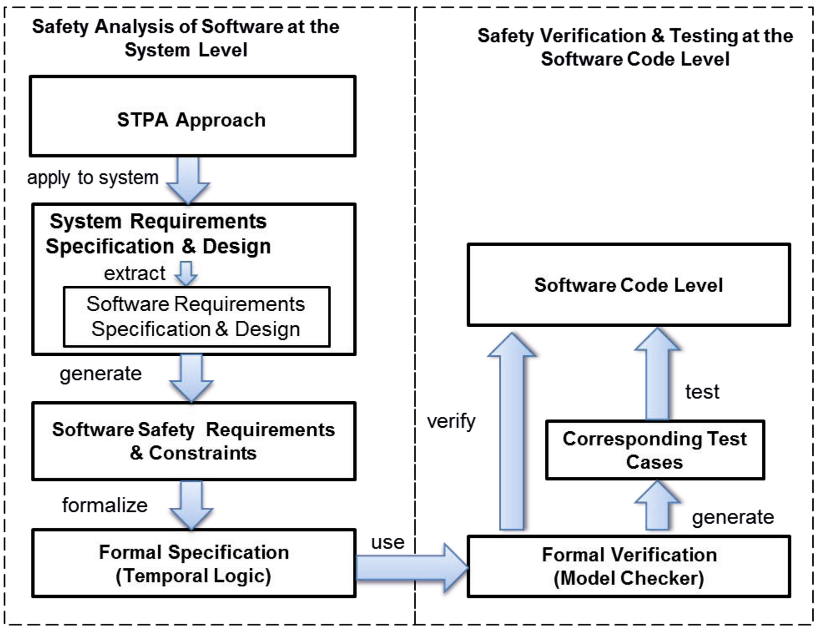

Safety Analysis of Software at the System Level
STPA Approach
apply to system
System Requirements Specification & Design
extract
Software Requirements Specification & Design
generate
Software Safety Requirements & Constraints
formalize
Formal Specification (Temporal Logic)
use
Safety Verification & Testing at the Software Code Level
Software Code Level
verify
test
Corresponding Test Cases
generate
Formal Verification (Model Checker)

Fig. 1. Overview of software safety verification method

(controller) to depict the interactions between software and other parts of the system (SW/HW components). Each unsafe control action in this context will be documented with four types of hazardous control actions [5]: *Not Providing Causes Hazard*, *Providing Causes Hazard*, *Wrong Timing/Order*, and *Stopped Too Soon/Applied Too Long*. At the end of this step, the safety analyst will translate each hazardous control action of software identified by STPA into the corresponding safety constraints and requirements in the system context.

3.2 Formalization of Safety Requirements and Constraints (Step 2)

Up to this step, the safety requirements of the software are identified. These safety requirements must be formalized using temporal logic (e.g. LTL, CTL) to be able to verify them in the next step.

3.3 Verification and Testing at the Code Level (Step 3)

This step aims to verify the software at the code level against the safety requirements which are expressed in the formal specification in step 2. After formalizing the safety requirements, this step can be done in two different ways: 1) using a model checker for formal verification [22], or 2) using a model checker to generate corresponding test cases [17]. A model checker takes as input a model of the software and the property of interest, which is written in temporal logic and then effectively explores the entire state space of the model. The model checker generates counterexamples which can easily be turned into complete test cases with the safety requirements (input and expected output).

4 Case Study: Vehicle Cruise Control

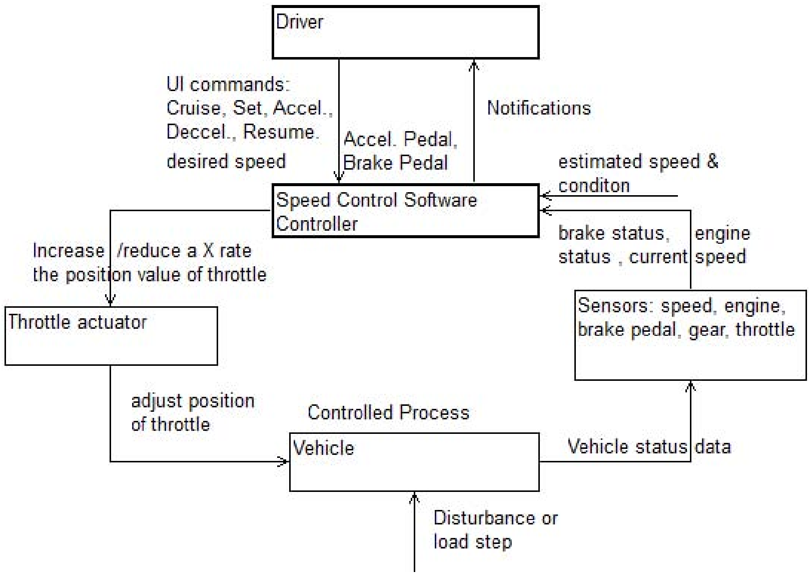
To illustrate the application of the method, we applied three steps of the method to the prototype of a vehicle cruise speed controller. The cruise control (speed control) system is a system which automatically controls the speed of a motor vehicle based on a preset value of a steady speed given by the driver. The speed controller unit is a program for judging the control scheme of the cruise control. In [11], we have applied STPA to the adaptive cruise control system at the abstract system level. Here, we focus on the safety analysis of software in the system context. In accordance with the proposed method, the first step involves applying STPA to the cruise control system. We use the A-STPA [12] tool to document the STPA analysis results. In the following, we will describe in detail the software safety verification based on a safety analysis at the system level.

4.1 Applying STPA to the Cruise Control

The results of applying STPA to the cruise controller at the system level are as follows:

1. **Analysis of Fundamentals:** The safety analyst must first establish the following fundamentals:
   - **Software Description:** The software of the cruise control (controller) maintains the vehicle speed automatically without pressing the accelerator pedal. It does it by sending the adjust throttle position to move as necessary to maintain the specified speed under varying conditions. The throttle is moved by a throttle actuator. The cruise control software maintains the speed of the vehicle on the occurrence of one of these events: (1) driver engages the brake, (2) the engine stops running, (3) the driver turns the ignition off, and (4) the driver turns the cruise control off. The controller receives signals from several sensors such as rotation sensor, brake pedal sensor, gear box sensor, and engine status sensor.
   - **Software Level Goals:** G.1: Control the speed of the vehicle.
   - **Accident:** AC1: The accident to be considered is a sudden acceleration of the vehicle which leads to a crash with another vehicle and the occupants are injured while the cruise control is in operation.
   - **The Related Software Level Hazards:**
     H.1: Unintended acceleration or deceleration of the vehicle when the cruise control is in active mode.
     H.2: The current speed value on the user interface is different from the actual speed of vehicle.

- **Design Requirements:**
  DR.1: The target speed must be between 40 $km/h <= vt <= 100\ km/h$.
  DR.2: The software shall notify the driver when trouble is detected.
  DR.3: It shall keep the acceleration rate within 0.25 and 0.4 m/sec.
- **Safety Constraints:**
  SC1: The controller shall keep the current speed of the vehicle below or equal to the desired speed.
  SC2: The cruise control shall not engage at the speed < 25 mph (40kph).
- **Safety Control Structure Diagram:** Figure 2 shows the control structure diagram which depicts the interaction between the software and other components in the system.



Driver
UI commands: Cruise, Set, Accel., Deccel., Resume.
desired speed
Accel. Pedal, Brake Pedal
Notifications
estimated speed & conditon
Speed Control Software Controller
brake status, engine status , current speed
Increase /reduce a X rate the position value of throttle
Throttle actuator
Sensors: speed, engine, brake pedal, gear, throttle
adjust position of throttle
Controlled Process
Vehicle
Vehicle status data
Disturbance or load step

Fig. 2. Safety control structure of cruise control software at the system level

2. **Identify Unsafe Control Actions:** Based on the control structure diagram (Fig. 2), we can identify the potentially unsafe control actions of the software at the system level which can lead to critical error. For example, the control Action: *Provide throttle position command (out)* can be documented as follows:
   - **Not Providing Causes Hazard:** *Throttle position command provided but not received by the throttle actuator when the cruise control is engaged (on)* [**H1**]
   - **Providing Causes Hazard:** *The throttle position is commanded while the cruise control is inactive (off)* [**H1**]. *The throttle position commanded with incorrect value of the throttle position* [**H1**].

Table 1. Examples of safety requirements which are derived by STPA

#code	Hazardous control actions	Safety Requirements and Constraints
SR1	Throttle position command provided but not received by the throttle actuator when the cruise control is engaged (on).	The throttle actuator must receive the adjustment throttle position command when it is commanded by controller.
SR2	The throttle position command is commanded while the cruise control is disengaged (off).	The controller must prevent rogue commands to the throttle when cruise control is off.
SR3	The throttle is commanded with incorrect X throttle position value.	The controller must be able to detect incorrect voltages issuing from the throttle position sensor (0.9-4.0v).

- **Wrong Timing or Order Causes Hazard: Late**: *the command provided too late* **[H1]**. **Early**: *The command provided too early* **[Not Hazardous]**
- **Stopped Too Soon or Applied Too Long:** *N/A*

Each hazardous cause will be translated into software-level safety constraints (see in Table 1.)

3. **Identify Causal Factors:** Figure 3 shows the process models of the speed controller and human operator as an example. The main process variables of the cruise control controller are the cruise control states, throttle control, and the speed control. These process model variables can be used to analyze each hazardous control actions which could happen. At the end of this step, the corresponding safety constraints will be refined.

4.2 Formalising the Safety Requirements

After identifying the safety commitments, we can translate them into formal specifications to be able to verify them with formal verification methods in the next step. Based on the classification of safety requirements for formal verification which are described in [13] by Friedemann, we mapped the four types of the hazardous control action classifications to the formal specification.

Mapping Safety Requirements to a Formal Specification: The mapping process starts with taking the set of the process model variables of the software (software states variables) which are identified in the last step of STPA to understand the main states of the software. To translate the safety requirements into a formal specification, first, we write them as informal textual requirements, i.e. if we consider the safety constraint **SC2**:*The cruise control shall not engage*

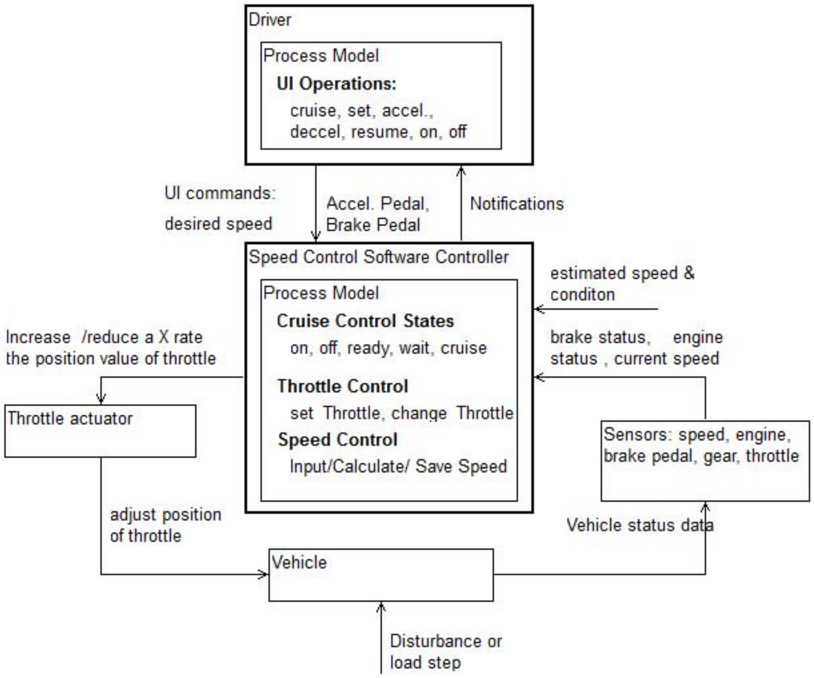

Driver
Process Model
UI Operations:
cruise, set, accel., deccel, resume, on, off
UI commands: desired speed
Accel. Pedal, Brake Pedal
Notifications
Speed Control Software Controller
Process Model
Cruise Control States
on, off, ready, wait, cruise
Throttle Control
set Throttle, change Throttle
Speed Control
Input/Calculate/ Save Speed
estimated speed & conditon
brake status, engine status , current speed
Increase /reduce a X rate the position value of throttle
Throttle actuator
Sensors: speed, engine, brake pedal, gear, throttle
adjust position of throttle
Vehicle status data
Vehicle
Disturbance or load step

Fig. 3. The process model of the cruise control software at the system level

at the speed $<$ 25 *mph* (40 *kph*). Second, we translate the textual descriptions to formal textual description by using the control flow statements (*IF- Then*, *Wait- Until*, *Wait- For*, *Do- Until*), for example:

IF *CruiseControl (inactive)* **and** *Read_speed(actual_speed) <25 mph* ***Then*** *CruiseControl (inactive)*

Finally, we translate them into an LTL specification, for example:

$\Box$ (CruiseControl (off) $\wedge$ $Read\_Speed(speed\_data) < 25\, mph) \rightarrow \Box$ CruiseControl (off)

Examples of formal specifications of software safety requirements which are derived from the STPA safety analysis at the system level are:

- **SR1:** $\Box\, CruiseControl(cruise) \rightarrow \Box\, (change\_Throttle \wedge$ set_Throttle (position_Throttle))
- **SR2:** $\Box$ CruiseControl (off) $\rightarrow \Box\, \neg(set\_Throttle(position\_Throttle) \vee change\_Throttle)$
- **SR3:** $\Box\, set\_Throttle(position\_Throttle) \rightarrow$ $\Box\, (position\_Throttle > min\_value \;\wedge\; position\_Throttle < max\_value)$

4.3 Verification and Testing of the Safety Requirements

To verify the safety requirements of the cruise control software, we used the Symbolic Model Verifier (SMV) which was developed by McMillan [14] and the SMV specification of the cruise control which was written by Ammann, Black, and Majurski [10]. We use SMV to either check the safety requirements or to generate the test set. First, we run SMV to test whether the model of the system satisfies the new safety requirements which we derived by STPA. As a result, the cruise control model did not satisfy SR1, SR2, and SR3, because the SMV specification model of the system does not include any state for controlling the throttle or any constraint about the restricted value of the throttle (the rate of throttle position value 0.09v - 4.0v). Therefore, we update the SVM specification model of the cruise control prototype and run SMV again. The SMV specifications of SR1, SR2 and SR3 are :

```
SPEC AG (CruiseControl= cruise) -> AG (change_Throttle &
set_Throttle.position_Throttle > 0)
SPEC AG (CruiseControl= off ) -> AG !(set_Throttle.position_
Throttle > 0 | change_Throttle)
SPEC AG (set_Throttle.position_Throttle > 0) -> AG (position_
Throttle > min_value & position_Throttle < max_value)
```

Now, SR1, SR2, and SR3 can be verified by SMV and the system model satisfies them. Then counterexamples can be generated by SMV to derive a new test suite.

5 Related Work

There is a large body of existing work on using formal verification for safety properties, safety verification by using SFTA, and generating test cases by using a model checker. Here, we discuss some of the most closely related work.

Leveson et al. [7] explain software fault tree analysis as a method for safety verification at the code level to be used on a more complex language involving such features as concurrency and exception handling. They consider the application of the safety analysis procedures to requirements modeling and specification languages. Kristen et al. [15] investigates how the results of one safety analysis technique, fault trees, are interpreted as software safety requirements by using interval logic to be used in the program design process. They interpreted fault trees as temporal formulas and how such formulas can be used for deriving safety requirements for software. Friedemann[13] propose the classification of safety requirements for the formal verification of software models of industrial automation systems. He expresses the safety requirements by using computation tree logic. The main reason of developing this classification is to handle difficulties in formal specification of safety requirements by software engineer. Recently, Black[17] shows how to generate test cases by using model checking.

The previous work focused on using the formal verification methods to verify that the software fulfills its specification (functional correctness). However, not all software errors can lead to the critical error that may lead to an accident. Safety properties (e.g. deadlocks, unexpected behavior, etc.) also are a special interest in formal verification. There have been a lot of work to verify safety properties at the code level. Verifying safety requirements, which are derived by using the traditional hazard analysis techniques such as FTA and SFTA, is an active area in verifying safety-critical systems. However, FTA and SFTA are difficult to apply to different parts of the safety related system (e.g. interaction between components). They evaluate only the possibility of occurrence, not the likelihood and appear to be scalable for software, but they have limitation. For example, the root node can only describe a known failure and they are labor-intensive and thus costly for large-size software [21].

Since safety of software cannot be analyzed without taking into account the system context, we take the advantages of STPA at the system level to construct a method for verifying safety requirements derived at the system level by using STPA. The method starts with safety analysis of software in the context of the system level by using STPA to derive the high-level safety requirements and constraints. The potential hazards identified during STPA will be translated into a set of verifiable safety requirements. The safety verification uses these verifiable safety requirements to prove that the software satisfies these requirements.

6 Conclusions and Future Work

In this paper, a method of software safety verification at the system level based on STPA is proposed. We investigated the application of the STPA structure to software, and we found that STPA can be directly used for software. We mapped the results of the STPA safety analysis to a formal specification to be able to verify safety requirements at the software code level. The limitation of the method is that the formal specification is done manually which may lead to much effort to construct and check the potential combinations of relevant states. Therefore, we are exploring the automation of this step and integrate it with our A-STPA tool as future work. Furthermore, we plan in-depth case studies to improve the method by applying it to real safety-critical software in industry. We plan also to investigate the effectiveness of using the proposed method during an ISO 26262 life cycle in the automotive industry.

References

1. Leveson, N.G.: Safeware: System Safety and Computers. ACM, New York (1995)
2. McDermid, J.A.: Issues in developing software for safety critical systems. Reliability Engineering & System Safety 32, 1–24 (1991)
3. Leveson, N.G., Turner, C.S.: An investigation of the Therac-25 accidents. Computer 26(7), 18–41 (1993)
4. Lions, J.L.: ARIANE 5: Flight 501 Failure. Technical Report, Inquiry Board (1996)
5. Leveson, N.G.: Engineering a Safer World: Systems Thinking Applied to Safety. Engineering systems. MIT Press (2011)

6. Leveson, N.G.: An STPA Primer. Engineering systems. MIT Press (2013)
7. Leveson, N.G., Cha, S.S., Shimeall, T.J.: Safety verification of Ada Programs Using Software Fault Trees. IEEE Software 8(4), 48–59 (1991)
8. ATSB Transport safety Investigation Report, In-Flight Upset Event 240 km North-West of Perth, WA Boeing Company 777-200, 9M-MRG (2005)
9. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, pp. 46–57 (1977)
10. Ammann, P.E., Black, P.E., Majurski, W.: Using Model Checking to Generate Tests from Specifications. In: Proceedings of the Second IEEE International Conference on Formal Engineering Methods, ICFEM 1998 (1998)
11. Abdulkhaleq, A., Wagner, S.: Experiences with Applying STPA to Software-Intensive Systems in the Automotive Domain. In: Proc. 2013 STAMP Conference, MIT, USA (2013)
12. Abdulkhaleq, A., Wagner, S.: An Open Tool Support for System-Theoretic Process Analysis. In: Proc. 2014 STAMP Conference, MIT, USA (2014)
13. Friedemann, B.: Classification of Safety Requirements for Formal Verification of Software Models of Industrial Automation Systems. In: Proceedings of the 13th Conference on Software and Systems Engineering and their Applications, ICSSEA 2000 (2000)
14. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Norwell (1992)
15. Hansen, K.M., Ravn, A.P., Stavridou, V.: From safety analysis to software requirements. IEEE Transactions on Software Engineering 24(7), 573–584 (1998)
16. Tracey, N., Clark, J., McDermid, J., Mander, K.: Integrating Safety Analysis with Automatic Test-Data Generation for Software Safety Verification. In: Proceedings of the 17th International Conference on System Safety (1999)
17. Black, P.E.: Test Generation Using Model Checking and Specification Mutation. IT Professional 16(2), 17–21 (2014)
18. Stringfellow, M.V., Leveson, N.G., Owens, B.D.: Safety-Driven Design for Software-Intensive Aerospace and Automotive Systems. Proceedings of the IEEE 98(4), 515–525 (2010)
19. Ishimatsu, T., Leveson, N.G., Thomas, J.P., Fleming, C.H., Katahira, M., Miyamoto, Y., Ujiie, R., Nakao, H., Hoshino, N.: Hazard Analysis of Complex Spacecraft Using Systems-Theoretic Process Analysis. Journal of Spacecraft and Rockets 51(2) (2014)
20. Hardy, T.L.: Essential Questions in System Safety: A Guide for Safety Decision Makers. AuthorHouse (2010)
21. Lutz, R., Nikora, A.: Failure Assessment in System Health Management with Aerospace Applications, 1st edn. John Wiley & Sons (2011), Johnson, S.B., et al (eds.)
22. Brock, B.C., Hunt, W.A.: Formally Specifying and Mechanically Verifying Programs for the Motorola Complex Arithmetic Processor DSP. In: Proceedings of the 1997 IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD 1997, October 12-15, pp. 31–36 (1997)
23. Kapur, R.: CTL for Test Information of Digital ICS. Springer (2002)
24. NASA.: Software Safety, NASA Technical Standard, NASA-STD 8719.13A (1997)