FASR: Automated Identification of Unsafe Control Actions in STPA

Ian Dardik[1], Yining She[1][0000−0002−0071−501X], Sam Procter[2][0000−0003−4379−2362], Keaton Hanna[2][0000−0002−4434−0352], Lutz Wrage[2][0000−0003−4239−4768], and Eunsuk Kang[1][0000−0001−7891−6885]

[1] School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213
{idardik,yiningsh,eunsukk}@andrew.cmu.edu
[2] Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA 15213
{sprocter,kehanna,lwrage}@sei.cmu.edu

Abstract. The *System-Theoretic Process Analysis (STPA)* is a well-established hazard analysis technique that has been applied to a wide range of safety-critical systems. Despite its popularity, there is relatively little automation support for STPA, and most of its steps are carried out manually by a human analyst, which can be time consuming and error prone. This paper investigates the potential use of model-based engineering and formal methods to assist human analysts in efficiently and accurately carrying out STPA. The proposed tool, called *FASR (Formalizing and Automating STPA with Robustness)*, enables automated, complete identification of *unsafe control actions (UCAs)*, leveraging recent advances in *robustness analysis* to identify UCAs as undesirable deviations in the controller's actions. The use of the tool is demonstrated on a case study involving a Braking System Control Unit (BSCU) in an avionics system. As a preliminary exploration of the potential benefits and limitations of the tool, the paper reports on a user study involving nine participants with varying backgrounds in STPA, model-based engineering, and formal methods; the study found that most participants considered the tool a useful aid in identifying UCAs, while suggesting improvements that would make a tool such as FASR usable and applicable to a wider range of systems and analysts.

1 Introduction

The *System-Theoretic Process Analysis (STPA)* is a well-established hazard analysis technique that has been applied to a wide range of safety-critical systems [16]. STPA, like other hazard analysis techniques, is a labor-intensive process: It requires a detailed understanding of the system structure and domain knowledge about potential hazards, and involves reasoning about possible interactions between different parts of the system, which can be a time-consuming and error-prone task. The amount of confidence that one has in the results of an STPA analysis (i.e., whether it has been carried out thoroughly and identified important hazards) is heavily dependent on the level of expertise and knowledge

that the analyst possesses; thus, an analysis error or oversight might lead to an omission of a critical hazard scenario. While a tool cannot replace human knowledge or insight, one that supports the analyst in systematically enumerating hazards or checking for errors in their work may be highly beneficial and provide a higher level of confidence in the analysis output.

This paper investigates the use of model-based engineering and formal methods to provide tool support for STPA analysts. Our proposed tool, called *FASR (Formalizing and Automating STPA with Robustness)*[3], specifically targets the automation of a crucial step in STPA that involves identification of *unsafe control actions (UCAs)*; i.e., actions of a controller that result in its failure to enforce a desired safety constraint and possibly lead the system to a hazardous state. FASR accepts as input (1) a model of the system under analysis (including a controller and a set of processes that it controls), specified as state machines in SysML [17], and (2) a safety invariant. As output, the tool automatically generates a list of UCAs that may cause the system to violate the invariant (i.e., transition into a hazardous state), to be examined by the human analyst for inclusion in the subsequent steps in STPA. Crucially, **FASR provides a formal guarantee that the list of UCAs is complete with respect to the details of the system as modeled**.

A key innovation underlying FASR is the adoption of recent advances in *robustness analysis* [25, 24], a type of model-based, formal analysis used to analyze the ability of a system to remain safe under possible *deviations* in the operating context. This analysis may be used to analyze, for example, whether or not an avionics braking system may enter a hazardous state when the human operator inadvertently deviates from their normative behavior, such as omitting an important action or performing actions out of order (i.e., a deviation). Later in the paper, we describe how the problem of identifying UCAs can be formulated as that of generating unsafe deviations and automated using an existing robustness analysis tool [24].

To identify the potential benefits and limitations of FASR, we performed an exploratory user study involving nine participants with varying backgrounds in formal methods, model-based engineering, and STPA. In the study, participants were asked to use the tool to generate a list of UCAs for a hypothetical Braking System Control Unit (BSCU) in an avionics system; participants were then asked to manually evaluate the quality of the generated UCAs and determine whether they represented legitimate UCAs, sufficient to be included in a human-authored STPA report. Our findings indicate that most of the participants found the tool useful for identifying UCAs, especially those without prior experience with STPA, and increased their confidence in the final analysis output. The study also identified a number of challenges with the use of a model-based, automated tool like FASR, including (i) building a faithful and sufficiently detailed system model and (ii) interpreting the output of the formal analysis.

The contributions of the paper are as follows:

[3] `https://github.com/cmu-sei/fasr`

– An approach to automating the identification of STPA's UCAs through robustness analysis,
– A method for automatically classifying those UCAs according to ways that STPA specifies that control actions can be unsafe, i.e., its guidewords,
– A prototype tool called FASR that implements this approach and classification, and,
– An exploratory user study involving the use of the FASR tool and a report on its findings, including perceived benefits and limitations of the FASR tool as well as future research directions.

2 Background

2.1 STPA

The *System-Theoretic Process Analysis* is a well-established hazard analysis technique that views safety problems as resulting from inadequate control structures: it asks analysts to identify interactions in a critical system that could, if safety constraints are not enforced, lead to an accident or loss [16]. Unlike more traditional, reliability-focused analyses (e.g., Failure Modes, Effects, and Criticality Analysis (FMECA) or Fault Tree Analysis (FTA) [11]), its primary focus is not component failures. In addition to those, it also considers problems like operator errors, interface mismatches, as well as socio-technical concerns. The technique can be applied at different abstraction levels, ranging from organizational to human-computer interaction to low-level automation [16].

STPA has four steps: in the first step, the analyst documents the system, its environment, and the boundary between them. The analyst also describes the losses that can occur in the environment due to the system, as well as the hazards which would cause those losses and safety constraints which would prevent them. In the second step, the analyst models the control structure, including the control actions that controller components send to controlled processes. In the third step, the analyst considers how those control actions could be unsafe according to several *guidewords*, such as *providing* (i.e., if the control action is provided when it should not be) or *not providing* (i.e., if the control action is not provided when it should be). The fourth step requires the analyst to consider if and how the UCAs identified in the third step could occur given the system's implementation.

2.2 Model-Based Engineering

In this paper, we present our approach on a *model* of a system and its components. Analyzing a system model—rather than its implementation—is advantageous for reducing the complexity of the analysis as well as ensuring that the system design is safe before implementation. More generally, the practice of *Model-Based Engineering* (MBE) is well-established in the development of critical systems, where the "model" may range from an informal sketch on a whiteboard to a detailed description written in a domain-specific language with

precise semantics. A wide range of MBE languages and tools exist for a similarly wide range of applications [10], including the use of those models for safety analysis. One of the most popular languages for MBE is SysML [10], which contains a variety of diagram types for modeling different aspects of a system's architecture, usage, and functionality [17].

Model-Based Safety Assessment (MBSA). MBE has been incorporated into safety assessments in various ways, including for manual hazard analyses (a human analyst references a system model) and tool-assisted hazard analyses (e.g., report generation [20], change impact and traceability support [3], and automated analysis of critical subsystems as described in this work).

RAAML. The *Risk Analysis and Assessment Modeling Language* (RAAML) is an extension of SysML for documenting system hazards and other safety information [1]. It provides profiles and libraries to support risk / safety analysis. These include a library of core elements and method-specific extensions for different hazard analysis techniques including STPA. RAAML support—which includes editing and rendering support, but no automated analysis—is installable as a plugin in some modeling tools, such as Dassault Systèmes' Cameo Enterprise Architecture (CEA) [9]. We implemented our approach using CEA and RAAML in order to more deeply integrate it into existing model-based tooling that practitioners are already using.

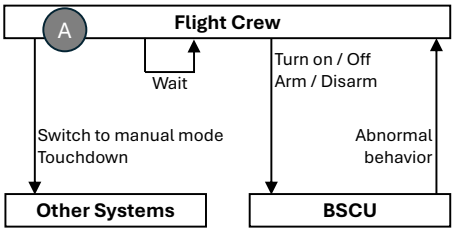
2.3 Robustness Analysis

Robustness is a quality attribute that describes the ability of a system to handle external disturbances or deviations in the environment. In a recent line of work, Zhang et al. [25] have proposed a formal definition of robustness for computer systems and a model-based analysis tool, called *Fortis* [24], that can be used to analyze the robustness of a system. In particular, Fortis takes as input: (1) a model of a computer system (M), specified as a transition system, (2) a model of the normative environment (E), also specified as a transition system, and (3) a desired safety property (P); the tool then computes *unsafe environmental deviations* that can lead to a violation of the safety property. In this setting, a *deviation* is a *trace* (i.e., a sequence of events in E) that shows how the environment may behave differently from its normative behavior. For example, M may describe the design of an avionics braking system; E the expected behavior of a human operator; and P a safety property stating that the brake is activated as the plane lands. In this example, an unsafe deviation would represent an erroneous sequence of actions by the operator (e.g., skipping or performing critical actions out-of-order) that could cause the plane to land without proper braking.

Later in Section 4.2, we describe how the problem of finding UCAs can be formulated as generating unsafe deviations and given automated analysis support using a tool like Fortis.

3 Case Study: BSCU Analysis with FASR

We selected a simplified avionics Braking System Control Unit (BSCU) system to help illustrate and evaluate our work. The system is derived from a well-studied



B	Provided	Not Provided
Arm BSCU	*Not unsafe for this hazard*	Crew turns on the BSCU but does not arm it before landing
Disarm BSCU	Crew turns on and arms the BSCU but then (inadvertently or mistakenly) disarms it	*Not unsafe for this hazard*

Fig. 1. (A) Control structure and sample (B) unsafe control actions for the Braking System Control Unit (BSCU) case study

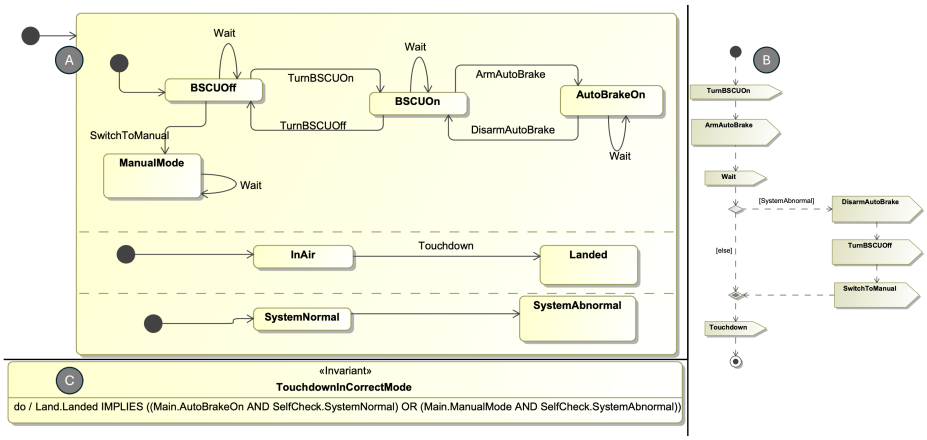


Fig. 2. A screenshot of Cameo Enterprise Architecture (CEA) showing the BSCU case study modeled using the SysML profile created for this effort. (A) is a state machine diagram of the BSCU, (B) is an activity diagram of the crew instructions, (C) is the hazard we want to avoid, expressed as an invariant.

example system described in, among other publications, the STPA Handbook [16]. This case study captures a scenario in which a flight crew operates the BSCU of an airplane during a landing. The flight crew is expected to follow the operating instructions that appear in the manual for the BSCU. Correctly operating the BSCU is crucial for avoiding the key hazard in this case study: the airplane overshooting the runway during the landing due to improper braking.

STPA. In this case study, the flight crew represents the controller for the BSCU. We show a diagram of the control structure in Figure 1(A), including actions that allow the flight crew to turn the BSCU on or off, arm or disarm the BSCU, and detect abnormalities in the baking unit. In this case study, we will primarily focus on UCA identification, i.e. identifying control actions that lead to the airplane overshooting the runway. We show a sample of the UCAs for this case study in Figure 1(B). For example, the upper right-hand box in Figure

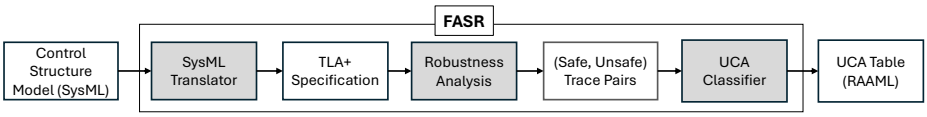


Fig. 3. An overview of the FASR tool.

1(B) identifies that a hazard may occur if the flight crew fails to arm the BSCU before landing, which is required by the BSCU instruction manual.

UCA Identification with FASR. Ordinarily, an STPA analyst must manually identify the UCAs in the third step. In this paper, we advocate assisting UCA identification with an automated tool such as FASR. A key advantage to this approach is that FASR is *guaranteed to identify all UCAs* with respect to user-provided models. This completeness guarantee is important because the control structure may be complex, difficult to reason about, and possibly tough for a human to identify all UCAs.

As input, FASR requires a model of the control structure in SysML. We show the model for the BSCU in Figure 2(A), the correct operating behavior of the flight crew (as specified in the BSCU manual) in Figure 2(B), and the key safety requirement—an invariant that implies the airplane has not overshot the runway—in Figure 2(C). FASR then *automatically* identifies all UCAs for the given model, which a human can then use to create a table similar to the one shown in Figure 1(B).

We emphasize that FASR's completeness guarantee is contingent on the correctness of the models of the control structure that are given as input. In essence, our approach shifts an engineer's concern from completeness of UCAs to properly constructing a control structure model; this type of shift is common in MBE.

4 FASR Tool

The high-level overview of the FASR tool is illustrated in Figure 3. The tool takes as input a model of the control structure written in SysML and outputs a list of UCAs that may cause the system to enter a hazardous state in RAAML. The rest of this section describes the main components of the FASR tool (highlighted in gray in Figure 3).

4.1 SysML to TLA+ Translation

The input SysML model consists of three different parts:

- *Controlled process*, specified as a state machine. The states of the machine are represented using a set of state variables, and each transition in the machine corresponds to an event that modifies one or more of these variables.
- *Controller*, specified as an activity diagram that describes normative sequences of control actions.

– *Invariant*, specified as a Boolean condition over state variables, indicating the absence of a hazard to be avoided. The invariant is assumed to hold under the normative control sequences as specified in the controller model.

The SysML model is then translated into a specification in the TLA+ language [15]. TLA+ is a formal specification language that has been widely used for modeling and verifying concurrent and distributed computer systems. We omit the detailed translation scheme from SysML to TLA+ here but briefly, both the controller and the controlled process in SysML are translated into transition systems in TLA+, and invariants are translated into safety properties (based on temporal logic [19]) in TLA+.

4.2 Unsafe Trace Generation through Robustness Analysis

In the next step, FASR leverages robustness analysis (implemented in the Fortis tool [24]) to generate traces that show how UCAs in the controller can lead the controlled process to a hazardous state. The key idea in this step is to treat the controller as an entity that may sometimes deviate from its normative behavior, and UCAs as those deviations that lead to a violation of the safety invariant.

Recall from Section 2.3 that Fortis accepts models of the system (M) and the environment (E) and a safety property (P) as input, and produces a set of unsafe environmental deviations. In the context of FASR, the controlled process is treated as the system (M), the controller as the environment (E) and the invariant as the safety property (P). In particular, the transition system for E in TLA+ encodes traces that correspond to normative control sequences; then, an unsafe deviation, generated by Fortis, is a trace that results in a violation of the safety invariant.

Some readers might find it counter-intuitive at first that the controller, not the controlled process, is treated as the environment. Conceptually, in this setting, robustness analysis is being used to analyze how robust or brittle the controlled process (e.g., the BSCU) is against possible errors or mishaps in the controller (e.g., the flight crew deviating from ideal instructions).

In its output, Fortis accompanies each unsafe deviation with a safe trace that avoids the deviation. Therefore, Fortis outputs trace *pairs* of the form (`safe`, `unsafe`). For example, consider the trace: $\langle$`TurnBSCUOn`, `Wait`, `Touchdown`$\rangle$. This trace is unsafe because the flight crew fails to arm the BSCU before the airplane touches down, which may result in a hazard (the airplane overshooting the runway). In the robustness setting, the trace is viewed as a deviation from the normative environment, which expects the flight crew to correctly arm the BSCU before touchdown as described in the instruction manual. In contrast, the following trace is safe and expected from the normative environment: $\langle$`TurnBSCUOn`, `ArmAutoBrake`, `Wait`, `Touchdown`$\rangle$. Fortis would accompany this safe trace with the prior deviated trace in its output. In the following section, we will describe how these pairs can be used to classify each UCA by an STPA guideword.

Edit	Removed	Guideword	Control Action
Deletion	`Wait`	Too Early	Early Action
Deletion	`Any`	Not Providing	Removed Action

Edit	Added	Guideword	Control Action
Addition	`Wait`	Too Late	Late Action
Addition	`Any`	Providing	Additional Action

Edit	Correct	Incorrect	Guideword	Control Action
Substitution	`Any`	`Any`	Providing	Provided Action
Substitution	`Any`	`Wait`	Not Providing	Removed Action
Substitution	`Wait`	`Any`	Providing	Additional Action
Transposition	`Any`	`Any`	Out of Order	Incorrect Action
Substitution	`Any`	`Wait`	Too Late	Late Action
Substitution	`Wait`	`Any`	Too Early	Early Action

Table 1. Mapping from atomic string edits in the Damerau-Levenshtein's edit distance algorithm [8] to STPA's Step 3 Guidewords. The `Any` action signifies any action other than `Wait` or the expected action from the safe trace (in the Removed, Added, or Correct columns) or unsafe trace (in the Incorrect column).

4.3 UCA Classification

The next and final step in FASR is to classify unsafe control sequences generated from the preceding robustness analysis into categories of UCAs that correspond to guidewords in STPA's third step. For each pair of safe and unsafe traces generated by Fortis, the UCA classifier selects a guideword that best explains the difference between the two traces. To do so, the classifier uses a modified version of the *Damerau-Levenshtein* [8] algorithm. The unmodified algorithm computes the edit distance between two strings, i.e., the minimum number of atomic operations (i.e., addition, deletion, substitution of a character or transposition of two adjacent characters) required to transform one string into the other; our modifications were to make the algorithm operate on traces (rather than strings) and to map the atomic string edits to STPA's guidewords using the mapping in Table 1.

For example, consider the following (`safe`, `unsafe`) trace pair:

$$(\langle \texttt{TurnBSCUOn}, \texttt{ArmAutoBrake}, \texttt{Wait}, \texttt{Touchdown} \rangle, \langle \texttt{TurnBSCUOn}, \texttt{Wait}, \texttt{Touchdown} \rangle)$$

The operation needed to transform the safe trace into its unsafe counterpart is the deletion of `ArmAutoBrake` action; the classifier then selects the STPA guideword *not providing* for this UCA. In other words, this UCA describes a scenario where the flight crew fails to arm the BCSU before landing.

The guidewords *provided too early* and *too late* describe an incorrect timing of a control action being performed. Since SysML models or TLA+ specifications

do not have a built-in notion of time, FASR designates a special action called `Wait` to denote passing of a unit time. The number of `Wait` actions in a trace is then used to determine whether a certain UCA is performed too early or too late. As another example, consider the following (`safe`, `unsafe`) trace pair:

$$(\langle \texttt{ArmAutoBrake}, \texttt{Wait}, \texttt{Touchdown} \rangle, \langle \texttt{ArmAutoBrake}, \texttt{Wait}, \texttt{Wait}, \texttt{Touchdown} \rangle)$$

Since the unsafe trace contains additional `Wait`, signifying that the flight crew waits longer than expected before performing touchdown, this UCA is classified under the guideword *too late* (for the `Touchdown` action).

Note that the guidewords *Applied too Long* and *Stopped too Soon* are calculated using a separate mechanism: rather than using the modified Damerau-Levenshtein algorithm, users can specify actions in SysML which signify the start or end of ongoing processes. If the unsafe trace has more or fewer `Wait` actions between these start and stop actions than the safe trace, we classify the UCA as *Applied too Long* and *Stopped too Soon*, respectively.

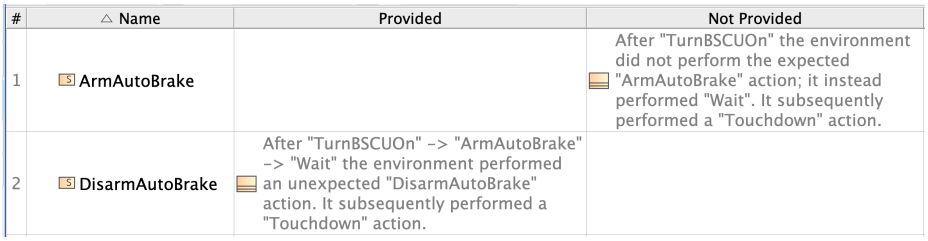
While other automated UCA identification approaches exist (see, e.g., [23, 18] and others in Section 6), to our knowledge, the mapping of the problem to that of computing string edits and the use of a modified edit-distance calculation algorithm is novel.

4.4 FASR Analysis of BSCU

We ran the FASR toolchain on the BSCU example using an Apple M3 Max running macOS 26.3.1; some performance statistics are shown in Table 2. Though performance wasn't a primary goal of this effort, the runtimes are likely to be acceptable: roughly 4-5 seconds when abnormal BSCU behavior was not considered and 6-8 seconds when it was. The state space explored by Fortis was significant: thousands of states were found when abnormal behavior was disregarded, and tens of thousands were found when it was included. Table 2 also shows the significant number of near-duplicate UCAs being discovered: more than half of Fortis-generated UCAs were filtered out by the FASR front-end, which performs only rudimentary de-duplication by, e.g., removing UCAs that are identical except in the number of consecutive wait actions. A screenshot of the generated UCA table (formatted using RAAML and displayed in Cameo Enterprise Architecture) is shown in Figure 4; these UCAs correspond to those shown in Figure 1.

5 Exploratory Study

In this section, we present a user study in which participants analyzed the BSCU example using STPA with the aid of FASR. The study was exploratory in nature; our goal was to gather qualitative feedback to understand whether our approach provides benefits as well as how the tooling might be improved in future iterations. In particular, we study the following two research questions:



#	Name	Provided	Not Provided
1	ArmAutoBrake		After "TurnBSCUOn" the environment did not perform the expected "ArmAutoBrake" action; it instead performed "Wait". It subsequently performed a "Touchdown" action.
2	DisarmAutoBrake	After "TurnBSCUOn" -> "ArmAutoBrake" -> "Wait" the environment performed an unexpected "DisarmAutoBrake" action. It subsequently performed a "Touchdown" action.	

Fig. 4. An excerpt of the UCA table generated by FASR for the BSCU example.

	Time (sec)		Generated UCAs		Fortis	
	Cold	Warm	Fortis	Filtered	States	Trans.
BSCU, No Abnormal Behavior	4.7	4.1	31	14	241	3100
BSCU, With Abnormal Behavior	7.5	5.8	55	23	1249	50024

Table 2. Runtime (with cold and warm starts), no. of UCAs (generated by Fortis and shown to users), and no. of Fortis-explored states and transitions for BSCU.

RQ1: Do study participants perceive any benefits from using FASR to identify UCAs?
RQ2: What downsides and improvements to the tool do study participants identify?

Cohort Demographics We invited ten participants from Carnegie Mellon University's Software Engineering Institute to participate in the study. Nine participants completed the study, of which six were given access to the tooling and three asked to perform STPA manually. Participants were asked to rate their familiarity with STPA, MBE, and formal methods; while most participants were familiar with MBE, there was a wider range of knowledge in STPA and formal methods.

5.1 User Study Description

In order to provide a baseline level of knowledge in the relevant techniques, we developed a one-hour training course which introduced STPA, the BSCU system as described in Section 3, and the tool itself (including the modeling environment it is a plugin for, Cameo Enterprise Architecture). The slides and a recording of the training were provided to participants afterwards for review.

We then met individually with each participant for an hour and had them examine the system models and perform the third step of STPA on the BSCU, i.e., identify which control actions in the scenario could be unsafe. The manual cohort filled out an UCA table directly. The tool cohort was allowed to use the FASR software to generate candidate UCAs, but was asked to evaluate their quality (to determine if they were legitimate or spurious) and rephrase or rewrite

them as much as necessary so they would be clear to a third party reviewer. The tool cohort was also encouraged to record additional UCAs they thought of that the tool did not generate.

Third, we interviewed each participant individually to collect qualitative data on the FASR tool. We asked participants a series of demographic and open-ended questions about their overall experience using it to evaluate the BSCU, the tool's strengths and weaknesses, as well as any potential improvements.

5.2 Results and Discussion

RQ1: In response to the question "Would you use this tool for identifying hazards in safety-critical systems?" every participant answered in the affirmative, though some more strongly than others. One participant, who had previously worked with different tool support for STPA, was particularly enthusiastic, responding: "This is absolutely the sort of tool I would use [if I were still working in this area]. I really like what I'm seeing here, [the tool] generated more information than the tools we used to have that were less connected to formalisms." Another participant, however, responded to the same question with "Yeah, [although] this iteration, maybe not yet... This would be super helpful if you could clean it up a bit."

By giving them a starting point to work from, rather than a blank page, the tool particularly helped users who were less experienced with STPA find more UCAs. The tool also gave participants a higher confidence than their manual counterparts that they found all relevant UCAs, though several noted that doing so essentially relied on having suitably complete system models to use as input. Our results therefore suggest that the study participants do perceive a benefit from our approach.

RQ2: Our interview data suggest that the way FASR displays UCAs has downsides and can be improved. Like all formal methods tools, Fortis can produce a larger amount of output than is easily comprehended by users. FASR currently includes basic filtering, based on the number of UCAs per guideword; however, our data suggests that more sophisticated filtering (e.g., based on a notion of semantic diversity in UCAs) could have positive impact on usability in future iterations of the tool.

Additionally, participants noted that expanding our input language to include additional SysML diagram types (e.g., sequence or block diagrams) could provide more information to Fortis and enhance the specificity of the analysis. One member of the tool cohort explained that, while generating UCAs for STPA's third step is important, the main timesink / challenge is its fourth step: identifying loss scenarios, i.e., the ways the UCAs could occur given the system's implementation. The models we use as input do not contain the level of information necessary to automate this step, so extending the tool in this way would be non-trivial.

Threats to Validity We mitigate the threats to validity in our study by focusing on exploratory research questions that are well-scoped, yet informative

for the discussions above. For example, RQ1 is in terms of *perceived* benefits; the data to answer this question is readily available in the interview transcripts, mitigating internal validity. On the other hand, RQ2 is in terms of the downsides and improvements that participants *identify*, which is also readily available in the interview transcripts and mitigates external validity (i.e., generalization to other FASR users).

6 Related Work

The use of formal methods for STPA has been explored in prior works. Datkwa and Villani propose an approach that uses the UPPAAL model checker [5] to check whether certain control actions could result in hazards over a formal model of a system [7]; the identification of UCAs in their approach is performed manually by a human analyst, while our tool generates such UCAs automatically. Integration of STPA with Event-B is another line of work in this direction [21, 6], although their goal is to develop a safe-by-construction system design through refinement rather than automating parts of STPA. Abdulkhaleq et al. proposes an approach in which a software implementation is formally verified against safety requirements derived using STPA [2]. In addition, alternative formal semantics for STPA have been proposed [23, 4], although their utility is demonstrated mainly by manual application to case studies.

Jung et al. describe a tool-supported process for extracting the *process model* used by STPA's second step from a specification of a system's output variables and control actions. Those can then be converted to UCAs, though Fault Tree Analysis (FTA) is used to filter the control actions as the approach can produce too many UCAs for analysts without that additional step [14]. Petzold et al. describe `HotPASTA`, which is a tool that presents a number of improvements to STPA, including some amount of permutation-based UCA generation [18].

STPA has been embedded in various MBSA tools and processes. These include SysML [22, 3], AADL [12, 20] and the Capella tool set [13]. In many cases, the actual work of performing STPA is done manually because, e.g., the tool's primary goal is traceability [3], or report generation [12, 20], or supporting the use of system models when performing a manual STPA [13]. de Souza et al.'s work incorporates formal methods, though only for simulation after the analysis is complete [22].

7 Future Work

FASR, like other automated MBSA tools, partially transforms an analytical problem (identifying hazards) into a modeling problem (creating sufficiently precise models). While manual STPA also relies on system models, they are typically much more abstract than the models used as input for FASR, and building these high-fidelity models of system behavior can be challenging and time consuming. We did not include model creation / editing tasks in our study due to the time

required and to avoid further narrowing our participant pool, but we would like to analyze the time required for specifying these models in a future study.

Participants in our user study also identified several issues with the FASR tool itself; these also point towards potential next steps. We would like to explore clustering the outputs from Fortis in a way that is meaningful to users, and provides a clearer understanding of the underlying safety issue. We also plan on exploring alternative input formats; though whether to include additional SysML diagrams or switch to an alternate language like SysMLv2 is still under discussion. Finally, integrating our tool with a larger tool suite and / or tool-supported process, in particular one that addresses traceability, is something we are exploring.

Acknowledgments

This material is based upon work supported by the Department of War under Air Force Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

References

1. Risk Analysis and Assessment Modeling Language (RAAML) Libraries and Profiles Version 1.1 Beta 2. Tech. Rep. ptc/24-03-02, OMG Standards Development Organization (March 2024)
2. Abdulkhaleq, A., Wagner, S., Leveson, N.: A comprehensive safety engineering approach for software-intensive systems based on stpa. Procedia Engineering **128**, 2–11 (2015), proceedings of the 3rd European STAMP Workshop
3. Ahlbrecht, A., Zaeske, W., Durak, U.: Model-Based STPA: Towards Agile Safety-Guided Design with Formalization. In: 2022 IEEE International Symposium on Systems Engineering (ISSE). pp. 1–8 (Oct 2022), iSSN: 2687-8828
4. Asare, P., Lach, J., Stankovic, J.A.: Fstpa-i: A formal approach to hazard identification via system theoretic process analysis. In: 2013 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS). pp. 150–159 (2013)
5. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Formal Methods for the Design of Real-Time Systems. LNCS, vol. 3185, pp. 200–236. Springer, Berlin, Heidelberg (2004)
6. Colley, J., Butler, M.: A formal, systematic approach to stpa using event-b refinement and proof. In: Proceedings of the 21st Safety-Critical Systems Symposium. Safety-Critical Systems Club, Bristol, UK (February 2013)
7. Dakwat, A.L., Villani, E.: System safety assessment based on stpa and model checking. Safety Science **109**, 130–143 (2018)
8. Damerau, F.J.: A technique for computer detection and correction of spelling errors. Commun. ACM **7**(3), 171–176 (Mar 1964)
9. Dassault Systèmes: Cameo Enterprise Architecture (January 2026), `https://www.3ds.com/products/catia/no-magic/cameo-enterprise-architecture`, 2024x

10. De Saqui-Sannes, P., Vingerhoeds, R.A., Garion, C., Thirioux, X.: A Taxonomy of MBSE Approaches by Languages, Tools and Methods. IEEE Access **10**, 120936–120950 (2022)
11. Ericson II, C.A.: Hazard Analysis Techniques for System Safety. John Wiley & Sons, Inc., Fredericksburg, Virginia, United States of America, second edn. (2016)
12. Galois, Inc: Camet tools (2025), `https://tools.galois.com/camet/overview/camet-tools`, Last accessed on 2026-03-03
13. Hetherington, D., Roques, P.: STPA Analysis of Automotive Safety Using Arcadia and Capella. In: ERTS 2022. pp. 191–200. HAL Open Science, Toulouse, France (Jun 2022)
14. Jung, S., Heo, Y., Yoo, J.: A formal approach to support the identification of unsafe control actions of STPA for nuclear protection systems. Nuclear Engineering and Technology **54**(5), 1635–1643 (May 2022)
15. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (6 2002)
16. Leveson, N., Thomas, J.: STPA Handbook. Tech. rep. (2018)
17. OMG® Systems Modeling: OMG Systems Modeling Language™(SysML®) version 1.7. Tech. Rep. formal/24-01-07, OMG Standards Development Organization (January 2024)
18. Petzold, J., von Hanxleden, R.: Hot PASTA: Improved Pragmatics for System-Theoretic Process Analysis. In: Gallina, B., Törngren, M., Bitsch, F. (eds.) Computer Safety, Reliability, and Security. pp. 160–174. Springer Nature Switzerland, Cham (2026)
19. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977). pp. 46–57 (1977)
20. Procter, S., Hatcliff, J.: An architecturally-integrated, systems-based hazard analysis for medical applications. In: 12th ACM/IEEE International Conference on Methods and Models for System Design, MEMOCODE 2014 (2014)
21. Salehi Fathabadi, A., Snook, C., Dghaym, D., Hoang, T.S., Alotaibi, F., Butler, M.: Designing critical systems using hierarchical stpa and event-b. In: Glässer, U., Creissac Campos, J., Méry, D., Palanque, P. (eds.) Rigorous State-Based Methods. pp. 220–237. Springer Nature Switzerland, Cham (2023)
22. de Souza, F.G.R., de Melo Bezerra, J., Hirata, C.M., de Saqui-Sannes, P., Apvrille, L.: Combining stpa with sysml modeling. In: 2020 IEEE International Systems Conference (SysCon). pp. 1–8 (2020)
23. Thomas, J., Leveson, N.: Generating formal model-based safety requirements for complex, software- and human-intensive systems. In: Proceedings of the Twenty-first Safety-Critical Systems Symposium. Safety-Critical Systems Club, Bristol, United Kingdom (2013)
24. Zhang, C., Dardik, I., Meira-Góes, R., Garlan, D., Kang, E.: Fortis: A tool for analysis and repair of robust software systems. In: Formal Methods in Computer-Aided Design (FMCAD). pp. 1–9. IEEE (2023)
25. Zhang, C., Garlan, D., Kang, E.: A behavioral notion of robustness for software systems. In: Proceedings of ESEC/FSE. p. 1–12 (2020)