A Novel Approach to Tracing Safety Requirements and State-Based Design Models

Mounifah Alenazi[*]
University of Cincinnati
Cincinnati, Ohio
alenazmh@mail.uc.edu

Nan Niu
University of Cincinnati
Cincinnati, Ohio
nan.niu@uc.edu

Juha Savolainen
Danfoss Drives A/S
Gråsten, Denmark
juha.savolainen@danfoss.com

ABSTRACT

Traceability plays an essential role in assuring that software and systems are safe to use. Automated requirements traceability faces the low precision challenge due to a large number of false positives being returned and mingled with the true links. To overcome this challenge, we present a mutation-driven method built on the novel idea of proactively creating many seemingly correct tracing targets (i.e., mutants of a state machine diagram), and then exploiting model checking within process mining to automatically verify whether the safety requirement's properties hold in the mutants. A mutant is killed if its model checking fails; otherwise, it is survived. We leverage the underlying killed-survived distinction, and develop a correlation analysis procedure to identify the traceability links. Experimental evaluation results on two automotive systems with 27 safety requirements show considerable precision improvements compared with the state-of-the-art.

CCS CONCEPTS

• **Software and its engineering → Requirements analysis**; **Traceability**; *System modeling languages.*

KEYWORDS

Traceability, mutation analysis, process mining, requirements engineering, Systems Modeling Language (SysML)

ACM Reference Format:
Mounifah Alenazi, Nan Niu, and Juha Savolainen. 2020. A Novel Approach to Tracing Safety Requirements and State-Based Design Models. In *42nd International Conference on Software Engineering (ICSE'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380332

1 INTRODUCTION

When engineering safety-critical systems like medical devices, it is of vital importance to ensure the system design meets the safety requirements. For example, one such requirement for a therapeutic robotic arm [25] concerns: "Automatic stoppage of the robotic arm if arm velocity sensors disagree on current velocity by more than x mps [40]". Design reviews, sometimes carried out by independent inspectors, are one of the main methods for ascertaining the satisfaction of safety requirements. In fact, one of the most widely used industrial standards for embedded systems—IEC 61508 [31]—rates design reviews as *Highly Recommended* (the highest importance rating) for all systems at all criticality levels.

The design of a software-intensive system often involves modeling. Modeling techniques have received wide industry acceptance, especially in critical domains like avionics and telecommunications, where state-based models are prevalent for describing system behaviors. These models consist of a finite set of states including the start state(s), a set of events (or "inputs"), and a transition function that determines the next state based on the current state and event [9]. Many variants exist, e.g., Statecharts specifying "Remote Identification" and other features at AT&T [45], RSML (requirements state machine language) adopted by the FAA to regulate collision avoidance installed on commercial aircrafts [36], etc.

With the use of model-driven engineering being on the rise, state-based models become larger and more complex. This presents a significant challenge for design reviews, where the inspector may have to browse through the models and manually analyze large numbers of links between safety requirements and design models [11]. Automated requirements traceability [18, 30] can alleviate this challenge, e.g., information retrieval algorithms rely on the textual information of requirements and that of model elements to establish plausible traceability [10].

In model-rich but not necessarily text-rich situations, researchers have developed slicing techniques to automatically identify those model elements related to a given interest (e.g., an event or a changing requirement) [9]. For instance, the seminal work by Korel *et al.* [35] analyzed data and control dependencies for backward slicing, and more recently, Nejati *et al.* [44] used reachability analysis to perform forward slicing. While the resulting slice (trace) typically bears a recall value close to 100%, the precision level is very low. For example, forward slicing combined with natural language processing achieved the best performance in tracing 16 requirements changes to the design models, and even this best performance had an average precision of only 29.4% [44], meaning that a large number of false positives were generated.

In this paper, we present a novel approach to tackling the false positives that have plagued automated traceability research for decades [10, 18, 30, 49, 67]. The idea is to intentionally generate many "false positives" from a state-based model (i.e., the tracing target) and then to check whether a safety requirement (i.e., the tracing source) is met in them in order to find the actual slice-trace. Our key insight is that false positives are "close" to the model

[*]Mounifah Alenazi is also affiliated with University of Hafr Albatin, Saudi Arabia.

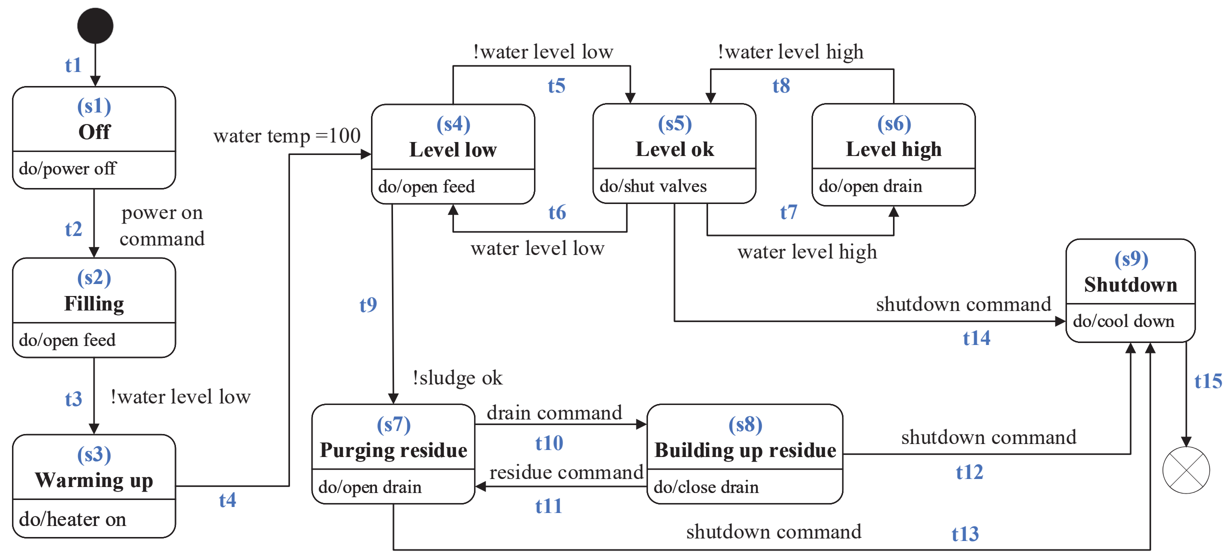


Figure 1: State machine diagram (SMD) of the water distiller example (*adapted from [24]*).

elements in the real trace, but that "closeness" turns out to be quite faulty *after* some tracing is done, e.g., measuring textual similarity or performing dependence analysis. This *faulty closeness* is the main reason causing false positives to be mingled with real elements, hurting precision. If we can exploit faulty closeness *before* tracing, then this proactive approach will provide new capabilities of addressing the low precision challenge of automated traceability.

To investigate the notation of "faulty closeness", our work leverages mutation analysis. Mutation analysis is commonly used as a fault-based software testing technique [32]. Given a program, mutants are created by simple changes that are intended to represent the mistakes often made by programmers. In our work, we define the mutants of a state-based design model based on the common modeling mistakes surveyed in the literature. Each mutant thus encapsulates the "faulty closeness" in some form. We then trace the safety requirement by analyzing how the mutants satisfy it. This step is carried out by manually formulating temporal logic formulas to capture the safety properties and employing an LTL model checker for automated verification. A mutant is killed if its model checking fails; otherwise, it is survived. We determine the final trace by examining the co-occurrence patterns of model elements (i.e., states and their transitions) in the killed mutants versus their patterns in the survived ones.

The contributions of this paper are threefold: (1) an innovative approach to exposing a great number of model mutants in support of safety requirements tracing, (2) an automated implementation of our approach based on model checking within process mining, and (3) an experimental evaluation of two subject systems with 27 requirements showing considerable precision improvements. The rest of the paper is structured as follows. We introduce the background of our work via a running example in Section 2. Section 3 describes our traceability information model. Section 4 details our mutation-inspired approach and process-mining-based implementation. We present the experimental results in Section 5, discuss related work in Section 6, and conclude the paper in Section 7.

2 BACKGROUND

This section uses a water distiller example adapted from [24] to illustrate the functional safety requirements that are to be traced. We also describe the syntax of the tracing target, which is rooted in the state machine diagram (SMD) of Systems Modeling Language (SysML) [53]. Finally, we show a couple of state-of-the-art methods [35, 44] in establishing the candidate traceability links of the running example, motivating our new tracing approach.

Consider a water distiller intended for use in remote, undeveloped regions where water is generally available but seldom safe to drink, possibly because of viral and bacterial contamination. A distiller unit purifies water via heating; however, an actual solution must consider broad issues like environmental protection, energy conservation, installation cost, and functional safety [24].

IEC 61508 [31] defines *functional safety* as part of the overall safety relating to the equipment under control (e.g., the water distiller). The goal is to ensure that any safety-related system must work correctly or fail in a predictable, safe way. In our running example, a fault of heated water running low may cause the hazard of leakage or explosion, and a functional safety requirement mitigating the fault can implement the proper safeguards to prevent the water level from staying low.

Tracing functional safety requirements supports critical needs such as inspections, assurance, and certification [55]. For the tracing target, we concentrate on the SMD modeled in SysML. SysML represents a significant and increasing segment of industrial support for building critical systems [59, 60, 68]. SMD is one of SysML's behavioral models, and the SMD considered in our work follows the syntax of an extended finite state machine (EFSM) [9]. Specifically, an EFSM consists of states (including an initial state and an exit state) and transitions between states. A transition is triggered when an event occurs and the guard (condition predicate) associated with the transition is evaluated to be true. During a transition, some action (input/output operation, variable manipulation, etc.) may be performed.

Figure 1 shows a SysML SMD design depicting the intended behavior of the water distiller. In this particular design with 9 states and 15 transitions, the real trace of the requirement, "preventing the water level from staying low," consists of only three states (i.e., $s4$, $s5$, and $s7$) and two of their transitions: $t5$ and $t9$. In other words, an inspector who is tasked with assuring the requirement would want to focus on these model elements because they provide the measures in the design, $s4 \xrightarrow{t5} s5$ and $s4 \xrightarrow{t9} s7$, to safely guard against the distiller's low water level.

The model elements in the real trace (also known as the *answer set*[1]) is typically determined manually and shall be driven by the safety requirements rather than by the design. For example, an independent inspector not involved in the construction of the SysML models would be interested solely in "water level being low or not" without concerning (or knowing) whether any check on sludge is performed or if there is a shutdown state in the SMD design. In the running example, therefore, we designate "water level" to be the only point of interest and explain how backward slicing [35] and forward slicing [44] work based on this point of interest (slicing criterion).

- **Backward slicing** (BS) identifies those model elements that affect "water level" by analyzing the define-use dependencies of this interested data variable. Beginning with the exit state of Figure 1, the BS algorithm [35] traverses the SMD backward and selects $\{s6, s5, s4, s3, s2\}$ to be the model slice.
- **Forward slicing** (FS) identifies those model elements that are being affected by "water level" via reachability analysis. Beginning with the initial state, the FS algorithm [44] traverses the SMD in a forward manner and returns the reachable subset of $\{s3, s4, s5, s6, s7, s8, s9\}$ as the model slice.

It is important to point out that, given the safety requirement of "preventing the water level from staying low", we follow the essences of the state-of-the-art [35, 44] to generate the candidate traceability links. With $\{s4, s5, s7\}$ serving as the answer set of the running example, BS achieves the recall=67% and precision=40%, whereas FS's recall=100% and precision=43%. Applying our approach to the running example returns $\{s4, s5, s6, s7\}$, resulting in the recall=100% and precision=75%. While the details of our approach will be presented in Section 4, we next define the context and scope of the traceable artifact types and their relations.

3 TRACEABILITY INFORMATION MODEL

Strategically, defining a traceability information model (TIM) is key to the development of safety-critical systems [40]. A TIM explicitly records what artifacts are important and what others are not under

[1]We define the answer set to be the set of relevant states by excluding the transitions, e.g., $\{s4, s5, s7\}$—rather than $\{t5, t9\}$ or $\{s4, s5, s7, t5, t9\}$—is the answer set of our running example shown in Figure 1 with respect to the safety requirement: "preventing the water level from staying low." We discuss the threats to construct validity of this choice in Section 5.

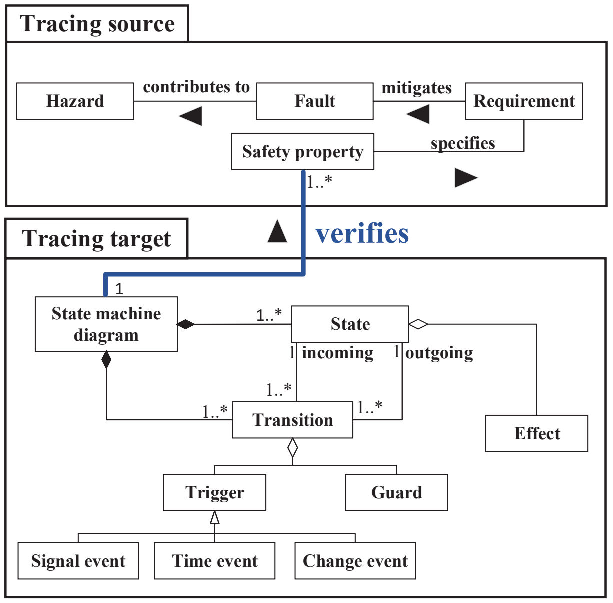


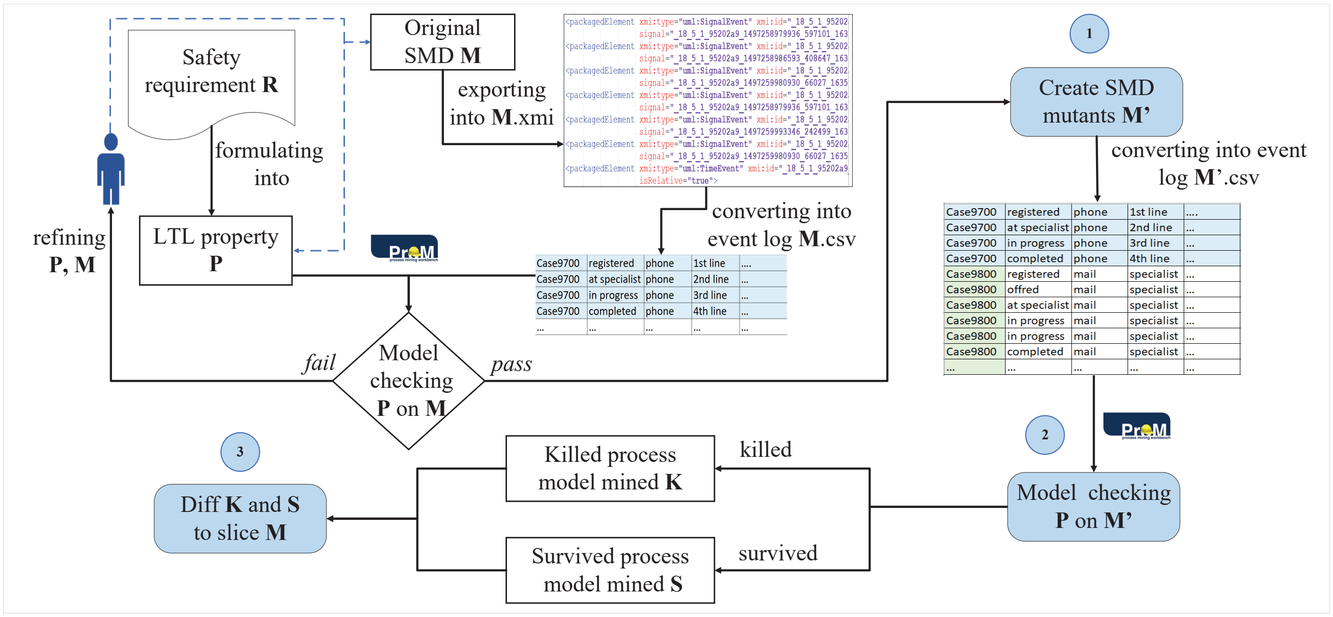
Figure 2: Traceability information contextualizing the artifacts and relations relevant to our approach.

the current traceability consideration. In addition, the traceable artifacts' relations are expressed in the TIM. In practice, depicting the planned, permitted trace paths in a TIM offers at least two benefits [39]:

- As tracing is a complex task, a TIM provides a guideline to ease its set up and allows for the validation of changes; and
- As traceability is also used by people who did not create it, these people need to know how it has been defined and what to expect from it.

Figure 2 presents the TIM underlying our work. While "requirement" is a central artifact type, the "tracing source" of Figure 2 shows it is the mitigation of "hazard"-contributing "fault" that gives rise to this specific type of functional safety "requirement". Referring to the example mentioned earlier, "automatic robotic arm stoppage" (requirement) is needed to mitigate the "velocity sensor failure" (fault), which in turn contributes to the danger of "moving the patient's arm at an excessive velocity" (hazard) [25, 40]. This shows the human-centric nature surrounding the "tracing source" of Figure 2, as the causal chain of reasoning involved in this therapeutic robotic arm case requires domain knowledge and relevant expertise. Methods like fault tree analysis (FTA) [58] and failure modes and effects analysis (FMEA) [64] can facilitate but cannot replace the manual work in safety requirements engineering.

Since requirements engineering must span the gap between the informal world of stakeholder needs and the formal world of software systems behavior, the key question over the use of formal methods is not *whether* to formalize, but *when* to formalize [52]. SysML's SMD, practiced in the context of model-driven engineering, embraces the formalization of systems behavior in the design. We thus use one such formal method (namely, model checking [15]) to



Safety requirement **R**
formulating into
LTL property **P**
refining **P, M**
Original SMD **M**
exporting into **M**.xmi
converting into event log **M**.csv
ProM
Model checking **P** on **M**
fail
pass
1
Create SMD mutants **M'**
converting into event log **M'**.csv
2
ProM
Model checking **P** on **M'**
killed
survived
Killed process model mined **K**
Survived process model mined **S**
3
Diff **K** and **S** to slice **M**

```
<packagedElement xmi:type="uml:SignalEvent" xmi:id="_18_5_1_95202
                 signal="_18_5_1_95202a9_1497258979936_597101_163
<packagedElement xmi:type="uml:SignalEvent" xmi:id="_18_5_1_95202
                 signal="_18_5_1_95202a9_1497258986593_408647_163
<packagedElement xmi:type="uml:SignalEvent" xmi:id="_18_5_1_95202
                 signal="_18_5_1_95202a9_1497259980930_66027_1635
<packagedElement xmi:type="uml:SignalEvent" xmi:id="_18_5_1_95202
                 signal="_18_5_1_95202a9_1497258979936_597101_163
<packagedElement xmi:type="uml:SignalEvent" xmi:id="_18_5_1_95202
                 signal="_18_5_1_95202a9_1497259993346_242499_163
<packagedElement xmi:type="uml:SignalEvent" xmi:id="_18_5_1_95202
                 signal="_18_5_1_95202a9_1497259980930_66027_1635
<packagedElement xmi:type="uml:TimeEvent" xmi:id="_18_5_1_95202a9
                 isRelative="true">
```

Case9700	registered	phone	1st line	....
Case9700	at specialist	phone	2nd line	...
Case9700	in progress	phone	3rd line	...
Case9700	completed	phone	4th line	...
...	...	...	...	...

Case9700	registered	phone	1st line	....
Case9700	at specialist	phone	2nd line	...
Case9700	in progress	phone	3rd line	...
Case9700	completed	phone	4th line	...
Case9800	registered	mail	specialist	...
Case9800	offred	mail	specialist	...
Case9800	at specialist	mail	specialist	...
Case9800	in progress	mail	specialist	...
Case9800	in progress	mail	specialist	...
Case9800	completed	mail	specialist	...
...	...	...	...	...

Figure 3: Overview of our mutation-driven traceability approach where mutants are created by modifying the tracing target in small ways to mimic typical modeling errors (①); mutants are then model checked (②) to identify the slice-trace (③).

link the "tracing source" and "tracing target" of Figure 2. In our TIM, each SMD design "verifies" one or more "safety properties". These properties are derived from the functional safety requirements and are formulated into temporal logic formulas amenable to model checking. Our TIM places the derivation and formulation of the safety properties inside "tracing source", emphasizing the human-centric nature of these activities.

On the "tracing target" side of Figure 2 is the SMD, which is typically used in SysML to model the behavior of critical components, such as hardware, software, data, personnel, procedures, and facilities [53]. When inspecting functional safety does not require an entire SMD design, it becomes valuable to identify the specific model elements (i.e., the subset of states and their transitions). This subset represents a *model slice* of a whole SMD, and in this paper, we refer to the subset as the *trace* and the elements of the subset as the *traceability links* with respect to a given safety requirement.

Although tracing is aimed at identifying the specific states and transitions, Figure 2 shows that the SMD semantics are defined also by "trigger", "guard", and "effect". In our running example of Figure 1, water temp = 100 is a "change event" and shutdown command is a "signal event", both triggering the water distiller to alter its behavior. A "time event", for instance, could trigger the scheduled maintenance at noon, September 1, 2019 (not shown in Figure 1), and in many occasions, "guard" like ! sludge ok specifies the condition that must be true for a transition to happen. Finally, "effect" in Figure 2 represents an action invoked directly on the object that owns the state machine as a result of transitioning into a state [53], e.g., open drain or shut valves.

The TIM of Figure 2 delineates the focuses of our work presented in this paper. For example, it is beyond our current scope to trace a single safety property over multiple SMDs or over different types of SysML models like internal block and activity diagrams. With manual effort in specifying safety properties, our objective is to automatically and accurately slice the SMD design to find the traceability links for the critical requirements.

4 MUTATION-DRIVEN TRACEABILITY

We tackle the low precision challenge faced by contemporary tracing algorithms from a new angle: Rather than striving for defining an accurate tracing mechanism which often ends up with many imperfect links, our core idea is to create many imperfect tracing targets and then take full advantage of them to discover the links. These imperfect tracing targets are mutants of the SMD design, and our entire approach shown in Figure 3 is driven by them.

An important check is performed before the mutants are generated. This is represented by the decision node (diamond) in Figure 3. For a safety property **P**, making sure that the to-be-traced SMD **M** satisfies **P** is of great practical value. In our running example of the water distiller, if the analyst writes the following LTL formula:

$$[]\,((\text{state} == \text{“Level low”} \rightarrow\, !(<>\ (\text{state} == \text{“Level low”})))) \quad (1)$$

trying to express that, "it is always ([]) the case once the water level is low it will eventually (<>) not be low," then the SMD of Figure 1 fails to satisfy this property. A counterexample, $\ldots\ s4 \rightarrow s5 \rightarrow s4 \ldots$, shows the looping structure in the SMD design, and thus formula (1) fails the LTL model checking. In these situations, the human analyst shall refine **P** or **M**, and here, a new property is written:

$$\begin{aligned}[]\,((\text{state} == \text{“Level low”} \rightarrow\ (<>\ (\text{state} != \text{“Level low”})\ /\backslash\\ (\text{state} == \text{“Level low”})\ \cup\ (\text{state} != \text{“Level low”})))) \quad (2)\end{aligned}$$

Table 1: State Machine Diagram (SMD) Mutation Operators

Category	ID	Mistake Description	Sample Mutation Operation on the SMD of Figure 1
state	mo_1	a state is subsumed by another state	adding an "ensuring off" state ($s0$) right after the initial state causes $s0$ to be subsumed by $s1$
	mo_2	a state that should be modeled is missing	removing $s5$
	mo_3	a state has incorrect transition(s)	adding a self-looping transition to $s1$
transition	mo_4	a transition comes from or goes into a wrong place	changing the direction of $t5$, i.e., flipping $t5$
	mo_5	a transition that should be modeled is missing	removing $t9$
	mo_6	a transition is subsumed by another transition	adding a "!water level low" transition ($t16$) from $s6$ to $s5$ causes $t16$ to be subsumed by $t8$
	mo_7	a transition is modeled without trigger	removing "shutdown command" on $t13$
guard	mo_8	a guard has incorrect condition	changing "!sludge ok" to "sludge ok" on $t9$
	mo_9	a guard refers to an undefined variable	adding "humidity ok" to $t9$ with "humidity" undefined in SysML's block definition diagram
trigger	mo_{10}	expression is incorrect	changing "water temp = 100" to "water temp < 100" on $t4$
	mo_{11}	time event is incorrect	changing scheduled maintenance from "noon, September 1, 2019" to "every 30 minutes" (not shown in Figure 1)
	mo_{12}	signal event is incorrect	changing "power on command" to "power off command" on $t2$
	mo_{13}	change event is incorrect	changing "water temp = 100" to "water temp != 100" on $t4$
effect	mo_{14}	effect refers to an undefined variable	adding "open humidifier" to $s6$ with "humidifier" undefined in SysML's block definition diagram
	mo_{15}	state invariant, do, entry and/or exit are incorrect	changing "open drain" to "close drain" on $s7$

to assert: "once the water level is low, it always becomes not low eventually after being low for some time." This property is now met by the SMD of Figure 1, leading our approach to the creation of SMD mutants. The main reason of checking **P** and **M** is that, if **M** fails **P**, then **M** already does not implement the requirement, so no tracing should be performed.

Our automated implementation is built with the help of the ProM tool [56], especially its LTL model checker operated on the logged events in a .csv file. In Figure 3, model checking **P** on both **M** and **M'** is therefore performed with ProM. Our own implementations include a Python script to mutate **M** in its xmi form [3] and a diff procedure to generate the candidate traceability links. Next, we discuss in more detail the three major steps shown in Figure 3.

4.1 Creating SMD Model Mutants

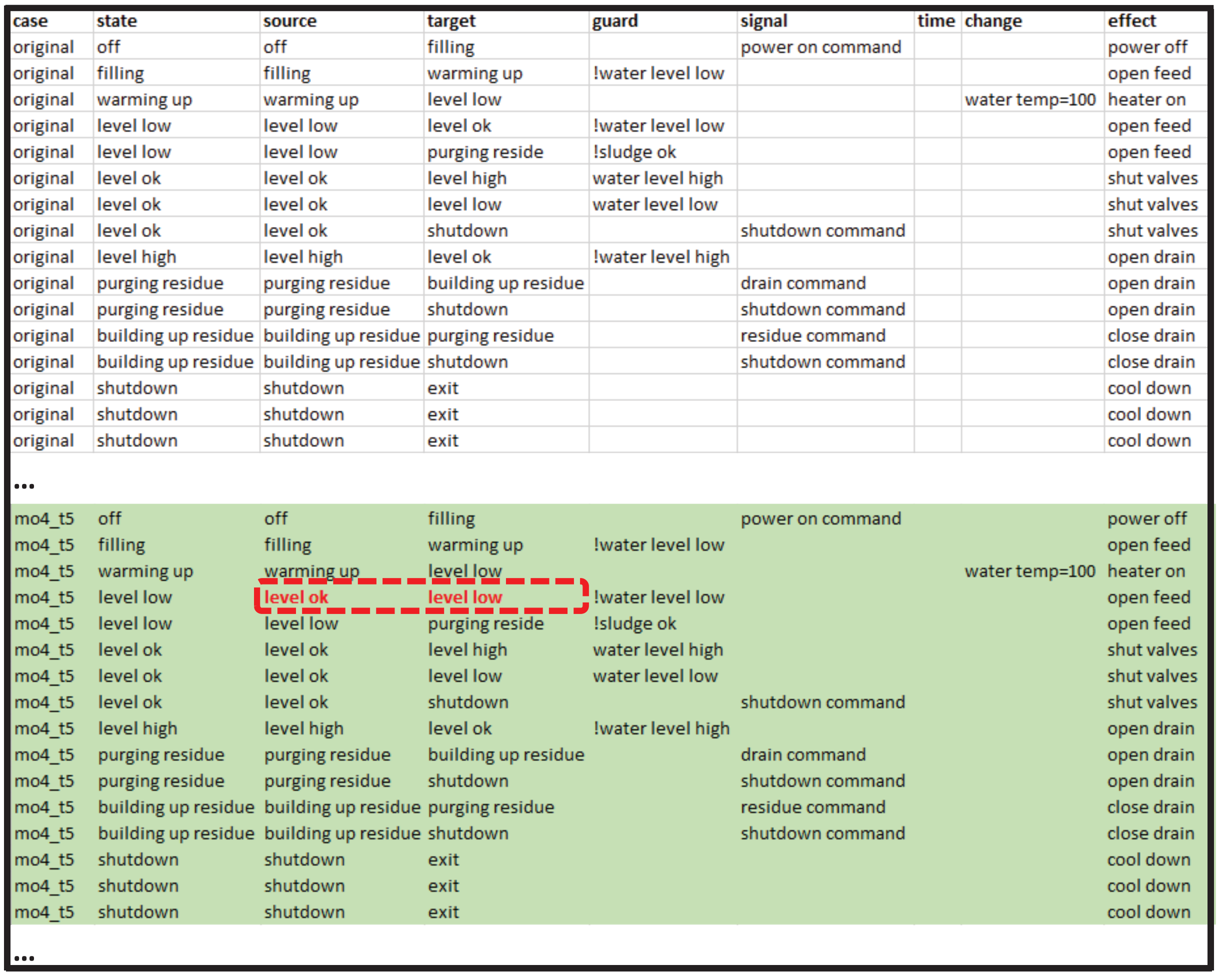
In software testing, mutants are results of deliberately seeding faults into the original program. The mutants can then be used to assess the quality of a test set: the more faults detected (or the more mutants killed), the more effective the test set. In mutation testing, only faults constructed from several simple syntactic changes are applied. A key tenet here is that [23]: "Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors." One of the first set of mutation operators was implemented in the Mothra system [34] and contained 22 operators, ranging from logical connector replacement to statement deletion. At any rate, mutants of a program are created based on a few simple faults representing the mistakes that programmers often make [32].

Extending mutation analysis, we survey the literature to identify the mistakes commonly made in SMD modeling [4]. Our survey focuses on practices over sizable models relevant to critical domains. We also favor the mistakes reported in common by different studies. We define 15 mutation operators and list them in Table 1. These operators are drawn from the SMD modeling mistakes discussed in five papers. Table 2 maps the sources with the operators.

Table 2: Mutation Operators of Table 1 Grounded in the Literature of SMD Modeling

Source	Domain (model size)	Mutation Operators
Aichernig *et al.* [1]	Automotive (19 states, 39 transitions)	mo_8, mo_{10}, mo_{11}, mo_{15}
Ali *et al.* [7]	Video Conferencing (13 states, 18 transitions) & Elevator Control (10 states, 14 transitions)	mo_1—mo_6, mo_8, mo_{10}, mo_{12}, mo_{13}, mo_{15}
Briand *et al.* [11]	Production Cell System (8 states, 10 transitions)	mo_2—mo_5, mo_7, mo_8, mo_{11}—mo_{13}, mo_{15}
Choppy and Reggio [14]	Library System (6 states, 9 transitions)	mo_1, mo_2, mo_7, mo_9, mo_{10}, mo_{15}
Mi and Ben [42]	Control System (32 states, 45 transitions)	mo_1, mo_3—mo_6, mo_8, mo_{13}, mo_{15}

We group the 15 mutation operators into five categories according to the "tracing target" of our TIM in Figure 2. For "state", "transition", "guard", "trigger", and "effect", there exist three, four, two, four, and two operators respectively. These categories show *where* to mutate, whereas the "mistake description" column of Table 1 explains *how* to mutate. The rightmost column of Table 1 illustrates



case	state	source	target	guard	signal	time	change	effect
original	off	off	filling		power on command			power off
original	filling	filling	warming up	!water level low				open feed
original	warming up	warming up	level low				water temp=100	heater on
original	level low	level low	level ok	!water level low				open feed
original	level low	level low	purging reside	!sludge ok				open feed
original	level ok	level ok	level high	water level high				shut valves
original	level ok	level ok	level low	water level low				shut valves
original	level ok	level ok	shutdown		shutdown command			shut valves
original	level high	level high	level ok	!water level high				open drain
original	purging residue	purging residue	building up residue		drain command			open drain
original	purging residue	purging residue	shutdown		shutdown command			open drain
original	building up residue	building up residue	purging residue		residue command			close drain
original	building up residue	building up residue	shutdown		shutdown command			close drain
original	shutdown	shutdown	exit					cool down
original	shutdown	shutdown	exit					cool down
original	shutdown	shutdown	exit					cool down
...								
mo4_t5	off	off	filling		power on command			power off
mo4_t5	filling	filling	warming up	!water level low				open feed
mo4_t5	warming up	warming up	level low				water temp=100	heater on
mo4_t5	level low	level ok	level low	!water level low				open feed
mo4_t5	level low	level low	purging reside	!sludge ok				open feed
mo4_t5	level ok	level ok	level high	water level high				shut valves
mo4_t5	level ok	level ok	level low	water level low				shut valves
mo4_t5	level ok	level ok	shutdown		shutdown command			shut valves
mo4_t5	level high	level high	level ok	!water level high				open drain
mo4_t5	purging residue	purging residue	building up residue		drain command			open drain
mo4_t5	purging residue	purging residue	shutdown		shutdown command			open drain
mo4_t5	building up residue	building up residue	purging residue		residue command			close drain
mo4_t5	building up residue	building up residue	shutdown		shutdown command			close drain
mo4_t5	shutdown	shutdown	exit					cool down
mo4_t5	shutdown	shutdown	exit					cool down
mo4_t5	shutdown	shutdown	exit					cool down
...								

Figure 4: Event log snippet showing: (1) the SMD of Figure 1 (top records whose case ID="original"), (2) the mutant resulted from flipping *t*5 (shaded records whose case ID="mo4_t5"), and (3) the syntactic change of *t*5 flipping (dotted box).

each mutation operator with a sample operation performed on the SMD design of the water distiller running example.

Similar to mutating a program, the SMD mutation operators of Table 1 are syntactic modifications of insertion (adding), replacement (changing), or deletion (removing). Different from mutating a program that is textual, we automatically mutate the graphical SMD by first exporting the model into an xmi file. We perform this step in the Cameo MagicDraw tool [51].

We developed a Python script to modify **M**.xmi: Removing an existing model element (mo$_2$, mo$_5$, or mo$_7$), changing its syntactic property (mo$_4$, mo$_8$, mo$_{10}$, mo$_{11}$, mo$_{12}$, mo$_{13}$, or mo$_{15}$), and adding an incorrect one (mo$_3$, mo$_9$, or mo$_{14}$). Due to the rather complex subsumption relations involved in mo$_1$ and mo$_6$, they are not implemented in our current Python script. Each resulting **M'**.xmi corresponds to one single syntactic change in one location (i.e., applying only one mutation operator), though the same operator at one location may generate more than one mutant, e.g., mo$_{10}$ applied to *t*4 ("water temp = 100") of Figure 1 outputs five mutants by replacing "=" with "<", "≤", "!=", "≥", and ">". Our current SMD mutation implementation is trying to be comprehensive as our goal is to use **M'** for tracing; selective mutation is investigated experimentally in Section 5.

4.2 Verifying Model Mutants

Once the SMD mutants are created, they undergo model checking so as to automatically verify the safety property **P**. An innovative aspect of our implementation is to leverage LTL model checking within process mining (i.e., the ProM tool [56, 66]). Process mining employs data mining algorithms to extract operational knowledge from event logs [65]. These event logs record instances (or "cases") of some underlying process (e.g., that of granting sabbatical), but automatically extracting that process is difficult when there is a lot of flexibility [66]. Our model mutants are a good fit to process mining in that flexibility of each mutant is restricted to a single, simple, and syntactic change over the original SMD **M**.

Figure 4 shows a sample event log recording our running example's SMD, and for comparison purposes, one mutant's records are also displayed (namely, mo$_4$ applied to *t*5). Figure 4 highlights the

```
Input: original SMD M, process model of killed mutants K,
       process model of survived mutants S
Output: set of candidate traceability links L

Procedure
1.  L ← all the states of M
2.  For each pair of states <$s_i$, $s_j$> ∈ M with
    correlation_K(<$s_i$, $s_j$>) > 0
3.      If correlation_K(<$s_i$, $s_j$>) > threshold_K AND
        correlation_K(<$s_i$, $s_j$>) − correlation_S(<$s_i$, $s_j$>) > 1
4.          Mark both states <$s_i$, $s_j$> with "remove"
5.      Else
6.          Mark both states <$s_i$, $s_j$> with "do-not-remove"
7.  L ← L \ states being marked AND being marked
        only with "remove"
8.  Return L
```

Figure 5: Identifying model elements from the SMD design to be the candidate traceability links.

way that our implementation uses "case ID" to group all the activities belonging to the same SMD. With this "case ID" mechanism, we convert **M'**.xmi into an event log file **M'**.csv without including the original SMD model **M** in it.

Although process mining techniques such as the alpha algorithm can extract one model directly from **M'**.csv in forms like a Petri net [65], we are interested in obtaining two models based on ProM's LTL model checking of **P** on **M'**.csv [66]: one underlying all the killed mutants (**K**) and the other for all the survived ones (**S**). We mark a mutant is killed if its model checking fails (i.e., the injected fault causes **P** to no longer be satisfied[2]); otherwise, the mutant is survived.

One of the killed mutants in the running example is "flipping $t5$" shown in the bottom of Figure 4. Compared to **M** that allows the water level to be low for some time before not being low eventually, the "flipping $t5$" mutant fails to meet the safety property **P** expressed in formula (2). For instance, a traversal containing "$\ldots s4 \rightarrow s5 \rightarrow s4 \rightarrow s7 \ldots$", which is permitted in **M**, is no longer valid due to the injected fault. Thus, model checking **P** on **M'** effectively distinguishes the faults directly violating the safety property from the remaining faults whose negative effects are not observed via automated verification.

4.3 Identifying Slice-Trace

Recognizing killed versus survived mutants allows for our approach's final step to identify the candidate traceability links. For a given requirements specification expressed in LTL, we refer to its links as the set of corresponding states in the SMD. Figure 5 presents our algorithm to slice **M** by contrasting **K** and **S**. We rely on ProM's correlation analysis over a mined process model[3]: For a pair of states <s_i, s_j>, a correlation score between −1 and 1 is

[2]Recall that the original SMD **M** satisfies **P** with the same ProM-based LTL model checking mechanism; otherwise, no mutant will be created. Such a control is elaborated by the decision node of Figure 3.

[3]ProM's correlation calculation automatically decides four relations between each state pair <s_i, s_j>: (i) s_j directly follows s_i, (ii) s_j sometimes follows s_i but never the other way around, (iii) s_j sometimes follows s_i and sometimes the other way around, and (iv) s_i and s_j do not follow each other [65].

Matrix	s1	s2	s3	s4	s5	s6	s7	s8	s9
s1	-0.52	0.75	-0.1	-0.1	-0.1	-0.1	-0.1	-0.1	-0.12
s2	-0.1	-0.52	1	-0.1	-0.1	-0.1	-0.1	-0.83	-0.1
s3	-0.11	-0.1	-0.52	0.85	-0.52	-0.1	-0.1	■	-0.1
s4	-0.1	-0.1	-0.1	■	0.3	0.5	-0.52	-0.1	-0.1
s5	0.61	-0.1	-0.1	-0.83	■	0.1	-0.1	-0.1	-0.1
s6	-0.08	-0.1	-0.1	-0.52	0.1	-0.1	0.5	-0.1	-0.02
s7	-0.1	-0.1	-0.32	-0.1	-0.52	-0.1	■	0.77	-0.19
s8	-0.1	-0.01	-0.1	-0.1	0.71	-0.52	-0.1	■	0.77
s9	-0.01	-0.09	-0.1	-0.1	-0.1	-0.1	-0.1	-0.1	■

(a) Killed mutants K

Matrix	s1	s2	s3	s4	s5	s6	s7	s8	s9
s1	-0.1	-0.85	-0.1	-0.1	-0.52	0.31	-0.1	-0.1	-0.1
s2	-0.1	-0.1	-0.85	0.31	-0.1	-0.1	-0.52	-0.52	-0.53
s3	-0.1	-0.53	-0.1	-0.5	0.02	-0.6	-0.1	-0.1	-0.1
s4	0.09	-0.1	-0.1	■	-0.5	-0.1	-0.1	0.27	-0.52
s5	-0.53	-0.1	-0.53	0.1	0.61	-0.2	-0.1	-0.1	-0.1
s6	0.03	-0.1	-0.1	-0.13	-0.1	0.12	-0.1	-0.1	-0.1
s7	0.03	-0.73	-0.1	-0.1	-0.1	-0.1	■	-0.41	0.18
s8	-0.1	■	-0.1	-0.52	-0.52	-0.1	-0.1	■	-0.5
s9	0.11	-0.1	0.53	-0.1	-0.1	-0.1	0.04	-0.1	■

(b) Survived mutants S

Figure 6: Correlation analysis of the running example's SMD mutants (black cell shows the correlation is unknown).

produced, and if the score is greater than 0, less than 0, or equal to 0, then it indicates the co-occurrence of s_i and s_j is strong, weak, or unknown respectively. Figure 6 visualizes the correlation analysis results over our running example's **K** and **S**.

We illustrate our slice-trace identification with the correlation analysis results of Figure 6. To maintain a high recall value, our tracing algorithm initializes **L** with all the states of **M**, i.e., after line #1 of Figure 5, **L**={s_1, s_2, …, s_9}. We then check the pair of states having only positive correlation in the killed mutants (**K**) and ignore all the other pairs. The rationale is to focus only on the commonly occurred state-pair causing the property to fail. For Figure 6a, line #2 selects the following pairs to examine: <s_1, s_2>, <s_2, s_3>, <s_3, s_4>, <s_4, s_5>, <s_4, s_6>, <s_5, s_1>, <s_5, s_6>, <s_6, s_5>, <s_6, s_7>, <s_7, s_8>, <s_8, s_5>, and <s_8, s_9>.

For each of the above selected state pairs, we mark both states with "remove" if the two conditions shown in line #3 of Figure 5 are met:

- The correlation score of <s_i, s_j> in **K** is greater than a positive degree (i.e., threshold_**K**), implying that s_i and s_j tend to co-occur in the killed mutants; and
- Such co-occurrence is significantly weaker among the survived mutants **S**.

We operate the latter condition by checking if the difference between the correlation score of <s_i, s_j> in **K** and that in **S** is larger than 1, as this difference is sufficient to reverse the correlation strength in the $[-1, 1]$ scale. Note that if <s_i, s_j> has a correlation score of 0 (unknown co-occurrence) in either **K** or **S**, then line #3 of Figure 5 is evaluated to be false, i.e., $\{s_i, s_j\}$ are marked with "do-not-remove". For Figure 6, the results of adopting threshold_**K**=0.5 as in ProM are:

- s_1 is marked with "remove" from <s_1, s_2> and with "remove" from <s_5, s_1>;
- s_2 is marked with "remove" from <s_1, s_2> and with "remove" from <s_2, s_3>;
- s_3 is marked with "remove" from <s_2, s_3> and with "remove" from <s_3, s_4>;
- s_4 is marked with "remove" from <s_3, s_4>, with "do-not-remove" from <s_4, s_5>, and with "do-not-remove" from <s_4, s_6>;
- s_5 is marked with "do-not-remove" from <s_4, s_5>, with "remove" from <s_1, s_5>, and with "do-not-remove" from <s_5, s_6>;
- s_6 is marked with "do-not-remove" from <s_4, s_6>, with "do-not-remove" from <s_5, s_6>, with "do-not-remove" from <s_6, s_5>, and with "do-not-remove" from <s_6, s_7>;
- s_7 is marked with "do-not-remove" from <s_6, s_7> and with "remove" from <s_7, s_8>;
- s_8 is marked with "remove" from <s_7, s_8>, with "remove" from <s_8, s_5>, and with "remove" from <s_8, s_9>; and
- s_9 is marked with "remove" from <s_8, s_9>.

Having made the marks on the examined states, we remove from **L** those states received marks and only "remove" marks. Line #7 of Figure 5 therefore removes $\{s_1, s_2, s_3, s_8, s_9\}$ and line #8 returns **L**=$\{s_4, s_5, s_6, s_7\}$. The candidate traceability links returned by our algorithm lead to recall=100% and precision=75%; however, this performance is achieved only for tracing the property of formula (2) to the SMD design of Figure 1. The next section evaluates our approach quantitatively.

5 EXPERIMENTAL EVALUATION

5.1 Research Questions

We set out to answer three research questions.

RQ$_1$: How accurate is our proposed mutation-driven traceability approach?

While overcoming the low precision challenge is our primary goal, we do not want our approach to hurt recall. Our measures for **RQ$_1$** also include F$_1$ which is the harmonic mean of recall and precision defined as $F_1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$. We use the state-of-the-art BS [35] and FS [44] algorithms introduced in Section 2 as baselines for accuracy comparisons.

RQ$_2$: How to best operate our approach in practical settings?

When no answer set is available, fully automated solution is preferred. As our automated implementation is built upon the ProM tool, we investigate here the influence of threshold_**K** value on tracing accuracy. In our tracing algorithm presented in Figure 5, threshold_**K** is a key condition to identify potentially removable states. Discovering an optimal threshold range can readily transfer our approach into ProM tooling, especially in terms of delivering a new traceability service with a calibrated correlation analysis.

RQ$_3$: How can selective mutation be instrumented in our approach?

Because mutation's cost is not negligible, reducing effort is of practical value. **RQ$_3$** thus examines one way toward selective mutation informed by feature ablation [26]: by removing the mutation operators in a specific category (e.g., "state"), we are interested in how the tracing accuracy changes.

5.2 Subject Systems

Our experiments are carried out in the context of two subject systems from the automotive domain. We choose these systems due to the relevant discussions of the safety requirements, the availability of the SMD design models, and our recent experience of using them in the research work [3–6].

- Adaptive Cruise Control (ACC) [43] of a vehicle consists of several components that interact in real time. A critical component of the ACC is the speed controller whose function is to take over the task of maintaining a constant speed at the driver's request. Once the speed controller is adjusted, the ACC is activated and supports the throttle control. After the ACC is activated, it can be suspended and restarted by the driver through pressing the suspend/resume button, or the brake pedal. While suspended, the system must memorize the desired speed. In ACC, the user data and the sensor data are read at the input. These data are used to set the new value of the desired speed, which will be compared to the current speed. The result of this comparison is used to define the adjustment value of actuator output.
- Power System (PS) [61] of a vehicle consists of mechanical and electronic components. Among the critical parts under PS's control are the gearbox and the switch between gears based on input from multiple sensors as well as data provided by the control module of the engine. The control module of the PS then processes these inputs to calculate how and when to shift gears in the transmission and generates the signals that drive actuators to perform this shift.

The traceability-related characteristics of the two subject systems are shown in Table 3. On the "tracing source" side, we manually performed a FMEA analysis and identified a few hazards and several contributing faults of those hazards. We further derived functional requirements to mitigate the faults and formulated LTL properties based on the safety requirements. In ACC, for instance, []!(speed>160 $\wedge$ state=="set speed") assures the cruise value cannot be set while the vehicle's speed is high, which mitigates the risk of accident.

The "tracing target" parts of Table 3 show that the average size of the SMD designs of ACC and PS is about 14 states and 17–22 transitions. Compared to the SMD sizes listed in Table 2, our subject systems' models are similar to the models studied in Ali *et al.* [7], and notably the video conferencing models of [7] were successfully applied at Cisco for the purpose of robustness testing. One of the

Table 3: Subject System Characteristics (integers represent *total* numbers whereas decimal numbers represent the *averages*)

Subject System	Tracing Source					Tracing Target				states per answer set
	hazards	faults	req.s	properties per req.	variables per property	SMD	states per SMD	transitions per SMD	properties per SMD	
Adaptive Cruise Control (ACC)	3	6	10	2.90	2.09	11	13.63	21.81	2.72	4.40
Power System (PS)	5	11	17	1.88	2.28	15	14.25	17.33	2.07	4.85

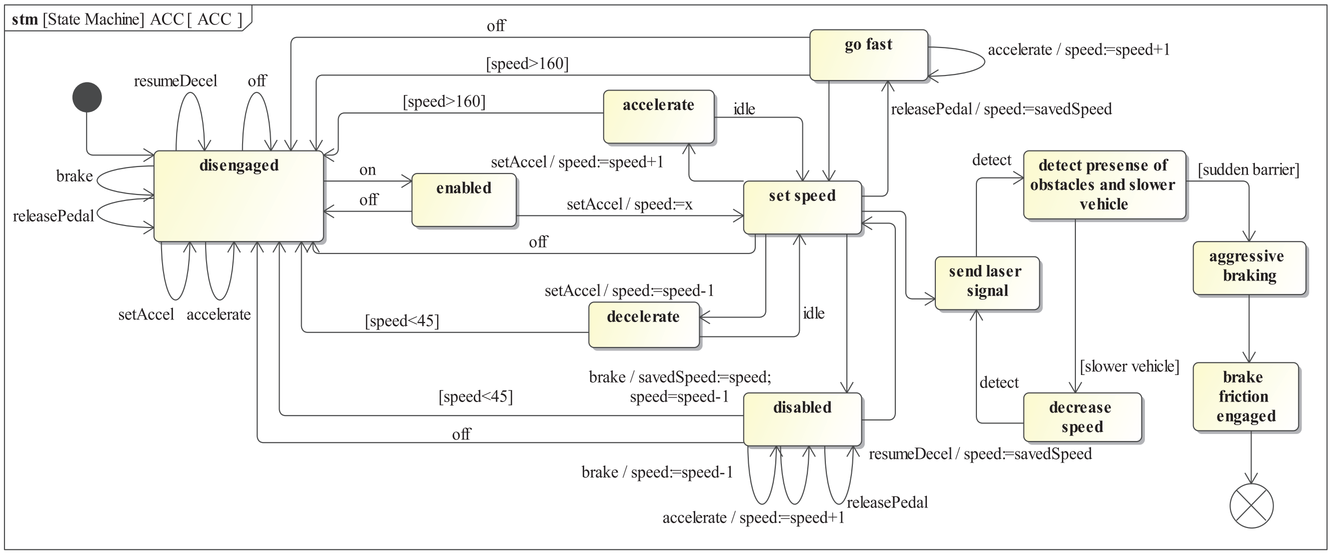


Figure 7: One SMD design of the adaptive cruise control (ACC) under our study.

representative SMD designs of ACC is shown in Figure 7. For each SMD in our study, Table 3 shows that around 2–3 safety properties are verified via model checking.

Finally, a software and systems expert working in the safety engineering domain who has more than 15 years of industrial experience manually constructed the answer set for each of the checked properties. This was done not as a vetting task [22, 41, 47] but as an independent design review task without input from any automated traceability tools. The rightmost column of Table 3 suggests that, averagely speaking, only about one third of the states are the real traces. The main objective of automated tracing methods is therefore to identify all the real traces and only the real traces.

5.3 Results

The answers to **RQ**$_1$ on tracing accuracy are summarized in Table 4 where the average recall, precision, and F$_1$ values are reported. The BS algorithm [35] misses a few true links in ACC and quite some in PS. One reason is that, when a state has multiple outgoing transitions, slicing backward from the exit state often fails to cover those branches that require forward tracing. For example, in PS's tracing of a liveness property ensuring braking is applied when RPM is close to 4000, BS is unable to reach the braking paths of the SMD and covers only the paths by following RPM's define-use dependencies. In contrast, the FS algorithm [44] achieves 100% recall based on its reachability analysis; however, its results are noisy by not halting the forward propagation early enough. Our approach does not rely on structural dependencies and therefore does not face the challenges of which direction to slice and when to stop slicing. As shown by the comparisons with FS in Table 4, using model checking to dynamically verify the SMD and its mutants improves the average precision by over 10% without compromising the tracing coverage at recall=100%.

Table 4: Tracing Accuracy (BS refers to backward slicing [35], FS refers to forward slicing [44], and MD refers to our mutation-driven approach)

	ACC			PS		
	recall	precision	F$_1$	recall	precision	F$_1$
BS	92.3%	39.1%	54.90%	74.5%	45.8%	56.70%
FS	100%	37.6%	54.65%	100%	34.1%	50.85%
MD	100%	48.9%	65.68%	100%	50.6%	65.68%

The key question **RQ**$_2$ addresses is which value threshold_**K** of Figure 5 should be in practical settings when no answer set is available. To this end, we calibrate threshold_**K** and measure how it impacts the tracing accuracy. As recall is maintained at the 100% level throughout the calibration, Figure 8 plots the impacts only on precision. Using ProM's default value of 0.5 to identify the positively correlated states from the killed mutants **K**

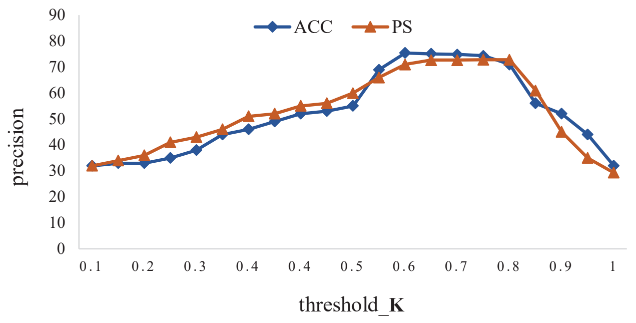


Figure 8: Calibrating threshold_K of our approach.

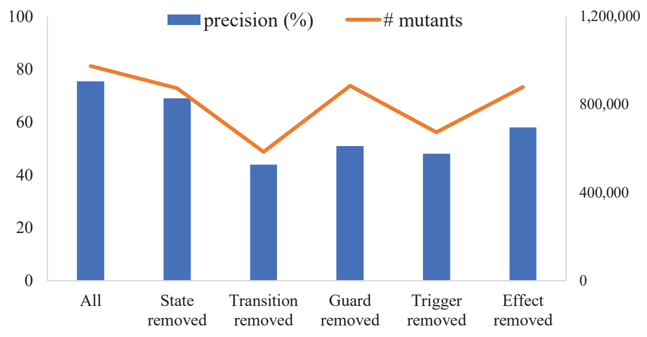


Figure 9: Ablation results of removing one and only one category of mutation operators.

turns out not to be threshold_**K**'s optimal value. Figure 8 suggests threshold_**K**'s range of [0.6, 0.8] further improves the average precision to 75.4% for ACC and 72.8% for PS. An explanation is that higher values of threshold_**K** facilitates the condition, correlation_**K**(<s_i, s_j>)−correlation_**S**(<s_i, s_j>)>1, to be met; however, too high of a value will cause few states to satisfy: correlation_**K**(<s_i, s_j>)>threshold_**K**. Therefore, to fully capitalize on our mutation-driven traceability approach, one can try to set threshold_**K**∈[0.6, 0.8], especially practiced to a high degree of automation in the ProM tool chain.

RQ$_3$ is aimed at exploring ways to reduce the cost of our approach. As our current implementation favors comprehensiveness, close to one million mutants are generated for the 26 SMD models in our study. In our experiments, creating a million mutants with our Python script took about 9 days, and model checking them within ProM took roughly 2 hours. If less numbers of mutants could deliver comparable tracing accuracies, then the findings would have both theoretical and practical implications. To investigate selective mutation empirically, we adopt the idea of feature ablation. In machine learning, feature ablation is designed to assess the informativeness of a feature group by quantifying the change in predictive power when comparing the performance of an approach trained with all the feature groups versus the performance without a particular feature group [26].

Following the SMD mutation operators defined in Table 1, we remove the category (feature group) one at a time and the ablation results are shown in Figure 9. When all the operators are applied in ACC and PS, the average precision is at the 74% level with the optimal threshold_**K** for each subject system. Using the same optimal values of threshold_**K**, Figure 9 depicts the precision drop as well as the drop in the number of mutants. Unsurprisingly, these two drops are proportional: the less number of mutants generated, the lower precision the tracing results. This reinforces the cost-benefit tradeoff: the more savings in cost, the worse the performance becomes. Surprisingly, even though the tracing performance in our study is measured according to the subset of *states*, removing "state" operators results in the least precision decrease. This shows the importance of tracing context, i.e., in order to find the model elements of interest (e.g., subset of the states), it could be more cost-effective to mutate other elements (especially "transition" and/or "trigger" according to Figure 9) which provide the semantics of the interested ones ("states").

5.4 Threats to Validity

We discuss some of the most important factors that must be considered when interpreting our experimental results. A threat to construct validity is our choice of measuring the tracing accuracy based only on the relevant states; in particular, we exclude the transitions in the answer set definition. The states and their transitions are clearly related, and our rationale is not to over-penalize an automated and scalable method for missing correct model elements or returning incorrect ones. The feedback from the expert devising the answer set recommends using transitions as anchoring constructs because "*they inject the state machine with life stories.*"

A threat to internal validity concerns the quality of the safety properties expressed in LTL formulas. As we manually performed this task, we relied on the decision node of "model checking **P** on **M**" in Figure 3 to make sure the SMD design models [43, 61] met the LTL properties. A confounding factor is the number of variables (e.g., "speed", "RPM", etc.) expressed in the properties, as the variables play a key role in our operations of the BS [35] and FS [44] algorithms. Table 3 shows that each LTL property contains an average of two variables; however, the tracing accuracy of BS, FS, and our own approach is likely to be influenced by this number and our results must be interpreted with this in mind.

As far as the external validity is concerned, our experimental subjects are both drawn from the automotive domain, and therefore applying our approach to systems in other safety-critical domains will be valuable. Another threat here is the specific class of state-based design models being traced, namely EFSMs. This is one of the simplest classes in that EFSMs do not support some of the advanced behavioral modeling features, such as concurrency and hierarchy. It is future work to expand our mutation-driven approach for tracing safety requirements to Statecharts [45], RSML models [36], or other behavioral models in SysML like activity diagrams.

6 RELATED WORK

To position our work in the existing literature, we review three strands of research: tracing safety-critical requirements, slicing state-based design models, and mutation analysis.

Traceability is essential for assuring software and systems are safe to use. However, gaps exist between what is prescribed by the safety regulations/guidelines and how traceability is implemented in practice [57]. To reduce manual effort, machine learning methods have been applied to automatically trace regulatory codes [17], and more recently, Guo and her colleagues [29] created a deep learning tracing architecture by leveraging Word Embedding and Recurrent Neural Network to better capture the requirements semantics in safety-critical domains. In these domains, the interviews by Goodrum *et al.* [27] with 14 experienced developers and by Chen *et al.* [13] with nine safety experts advanced our understandings about practitioners' views and needs of traceability. To address some of the needs, a family of reusable traceability queries was codified [19], a visual language hiding complex details in querying traceability was proposed [38], and new ways of managing safety stories in agile projects were developed [16, 20]. Mäder *et al.* [40] provided an actionable checklist to best practice requirements traceability in safety-critical projects. Among the checklist's ten items, our approach explicitly considers six by clearly defining the TIM, offering tool support, and generating traces as slices.

Slicing has been mostly applied to source code in order to locate the parts of the program that affect a user-specified point of interest (known as the slicing criterion) [63]. Since Weiser's seminal work on program slicing [70], researchers have investigated slicing in state-based models [9]. Of particular relevance to our work is applying model slicing to support safety inspections. Nejati and her colleagues [46] presented a comprehensive framework with concrete procedures to manage the traceability information between safety requirements and SysML models. A controlled experiment quantified the benefits of using design slices in terms of increasing the correctness of conformance decisions from 50% to 63% and reducing the effort of safety inspections by 27% [11]. To improve the automated traceability, Nejati *et al.* [44] used reachability analysis to perform forwarding slicing. This state-of-the-art serves as a baseline in our evaluation, though two important differences shall be noted. First, the TIM of [44] includes SysML's block definition diagrams and activity diagram, whereas our tracing target is focused on SMD models. Second, the slicing criterion of [44] consists of a specific requirements change, whereas our slicing criterion is given in an LTL formula capturing requirements safety. Although LTL formula and model checking have been used in model slicing [21, 54], to our best knowledge, none of the existing slicing approaches performs model checking on millions of mutants.

In mutation analysis, faults are automatically seeded into the software artifacts (in most cases, the source code), and the survey by Jia and Harman [32] provides evidence of applicability and maturity for the technique used in software testing. Two main applicabilities exist: measuring a test set's ability to detect faults and generating additional test cases. In both cases, the distinction between killed and survived mutants is important. If the result of testing a mutant is different from the result of testing the original program, then the mutant is classified as killed; otherwise, it is survived. Our approach, especially the decision on "modeling checking **P** on **M**", aligns with this distinction. A test set's effectiveness can then be scored on the proportion of the killed mutants, and additional test cases can be generated to kill the survived mutants. At model levels, researchers have also applied mutation analysis to reveal faults and to guide the generation of new test cases [2, 8, 28, 62]. Our work differs fundamentally from prior research in that our intention is not to kill as many mutants as possible, but to use the killed-survived distinction to automatically trace safety requirements.

7 CONCLUSIONS

We have presented a mutation-driven approach to tracing safety requirements and SMD in SysML modeling. Not only is the conceptual framework depicted, but an automated implementation based on model checking within process mining is developed. Experimental results show the precision improvements, and the practical support that can be delivered as ProM plug-ins (especially our SMD mutation script and calibrated correlation analysis).

The novelty of our work lies in our creation of many imperfect tracing targets, leading to new ways of establishing traceability links. Our future work includes expanding the TIM to handle more complex requirements with socio-technical interdependencies [12, 50], improving the usability of the model slices with visualization and executability [38, 48], and integrating the new generation of deep learning based refactoring methods for intelligently identifying mutation locations or mutant names [33, 37, 69] to further enhance tracing capabilities.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their constructive comments. We also thank Gary Duncanson and Nicole Gonzalez (No Magic, Inc.), Ana Rozinat (Fluxicon), and the ProM team—especially Eric Verbeek (Eindhoven University of Technology) and Wil van der Aalst (RWTH Aachen University)—for their research collaborations. Funding is provided in part by the U.S. National Science Foundation (Award CCF-1350487).

REFERENCES

[1] Bernhard K. Aichernig, Jakob Auer, Elisabeth Jöbstl, Robert Korosec, Willibald Krenn, Rupert Schlick, and Birgit Vera Schmidt. 2014. Model-Based Mutation Testing of an Industrial Measurement Device. In *International Conference on Tests and Proofs (TAP)*. York, UK, 1–19.

[2] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. 2015. Killing Strategies for Model-Based Mutation Testing. *Software Testing, Verification & Reliability* 25, 8 (December 2015), 716–748.

[3] Mounifah Alenazi, Nan Niu, and Juha Savolainen. 2019. A Process Mining Based Approach to Improving Defect Detection of SysML Models. https://doi.org/10.7945/6rna-m982. In *Late Breaking Results Track, International Conference on Automated Software Engineering (ASE)*. Last accessed: February 2020.

[4] Mounifah Alenazi, Nan Niu, and Juha Savolainen. 2019. SysML Modeling Mistakes and Their Impacts on Requirements. In *International Model-Driven Requirements Engineering Workshop (MoDRE)*. Jeju Island, South Korea, 14–23.

[5] Mounifah Alenazi, Nan Niu, Wentao Wang, and Juha Savolainen. 2018. Using Obstacle Analysis to Support SysML-Based Model Testing for Cyber Physical Systems. In *International Model-Driven Requirements Engineering Workshop (MoDRE)*. Banff, Canada, 46–55.

[6] Mounifah Alenazi, Deepak Reddy, and Nan Niu. 2018. Assuring Virtual PLC in the Context of SysML Models. In *International Conference on Software Reuse (ICSR)*. Madrid, Spain, 121–136.

[7] Shaukat Ali, Tao Yue, and Lionel C. Briand. 2014. Does Aspect-Oriented Modeling Help Improve the Readability of UML State Machines? *Software and System Modeling* 13, 3 (July 2014), 1189–1221.

[8] Paul E. Ammann, Paul E. Black, and William Majurski. 1998. Using Model Checking to Generate Tests from Specifications. In *International Conference on Formal Engineering Methods (ICFEM)*. Brisbane, Australia, 46–55.

[9] Kelly Androutsopoulos, David Clark, Mark Harman, Jens Krinke, and Laurence Tratt. 2013. State-Based Model Slicing: A Survey. *Comput. Surveys* 45, 4 (August 2013), 53:1–53:36.

[10] Markus Borg, Per Runeson, and Anders Ardö. 2014. Recovering from a Decade: A Systematic Mapping of Information Retrieval Approaches to Software Traceability. *Empirical Software Engineering* 19, 6 (December 2014), 1565–1616.

[11] Lionel C. Briand, Davide Falessi, Shiva Nejati, Mehrdad Sabetzadeh, and Tao Yue. 2014. Traceability and SysML Design Slices to Support Safety Inspections: A Controlled Experiment. *ACM Transactions on Software Engineering and Methodology* 23, 1 (February 2014), 9:1–9:43.

[12] Darius Cepulis and Nan Niu. 2018. Creating Socio-Technical Patches for Information Foraging: A Requirements Traceability Case Study. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Lisbon, Portugal, 17–21.

[13] Jinghui Chen, Micayla Goodrum, Ronald A. Metoyer, and Jane Cleland-Huang. 2018. How Do Practitioners Perceive Assurance Cases in Safety-Critical Software Systems?. In *International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. Gothenburg, Sweden, 57–60.

[14] Christine Choppy and Gianna Reggio. 2009. A Method for Developing UML State Machines. In *ACM Symposium on Applied Computing (SAC)*. Honolulu, HI, USA, 382–388.

[15] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 2001. *Model Checking*. MIT Press.

[16] Jane Cleland-Huang. 2017. Safety Stories in Agile Development. *IEEE Software* 34, 4 (July/August 2017), 16–19.

[17] Jane Cleland-Huang, Adam Czauderna, Marek Gibiec, and John Emenecker. 2010. A Machine Learning Approach for Tracing Regulatory Codes to Product Specific Requirements. In *International Conference on Software Engineering (ICSE)*. Cape Town, South Africa, 155–164.

[18] Jane Cleland-Huang, Orlena Gotel, Jane Huffman Hayes, Patrick Mäder, and Andrea Zisman. 2014. Software Traceability: Trends and Future Directions. In *Future of Software Engineering (FOSE)*. Hyderabad, India, 55–69.

[19] Jane Cleland-Huang, Mats Heimdahl, Jane Hayes, Robyn R. Lutz, and Patrick Mäder. 2012. Trace Queries for Safety Requirements in High Assurance Systems. In *International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ)*. Essen, Germany, 179–193.

[20] Jane Cleland-Huang and Michael Vierhauser. 2018. Discovering, Analyzing, and Managing Safety Stories in Agile Projects. In *International Requirements Engineering Conference (RE)*. Banff, Canada, 262–273.

[21] Daniela Colangelo, Daniele Compare, Paola Inverardi, and Patrizio Pelliccione. 2006. Reducing Software Architecture Models Complexity: A Slicing and Abstraction Approach. In *International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*. Paris, France, 243–258.

[22] David Cuddeback, Alex Dekhtyar, and Jane Huffman Hayes. 2010. Automated Requirements Traceability: the Study of Human Analysts. In *International Requirements Engineering Conference (RE)*. Sydney, Australia, 231–240.

[23] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer* 11, 4 (April 1978), 34–41.

[24] Sanford Friedenthal, Alan Moore, and Rick Steiner. 2012. Water Distiller Example Using Functional Analysis. In *A Practical Guide to SysML (Second Edition)*, Sanford Friedenthal, Alan Moore, and Rick Steiner (Eds.). The MK/OMG Press, 393–429.

[25] Antonio Frisoli, Luigi Borelli, Alberto Montagner, Simone Marcheschi, Caterina Procopio, Fabio Salsedo, Massimo Bergamasco, Maria C. Carboncini, Martina Tolaini, and Bruno Rossi. 2007. Arm Rehabilitation with a Robotic Exoskeleleton in Virtual Reality. In *International Conference on Rehabilitation Robotics (ICORR)*. Noordwijk, The Netherlands, 631–642.

[26] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Columbus, OH, USA, 580–587.

[27] Micayla Goodrum, Jane Cleland-Huang, Robyn R. Lutz, Jinghui Cheng, and Ronald A. Metoyer. 2017. What Requirements Knowledge Do Developers Need to Manage Change in Safety-Critical Systems?. In *International Requirements Engineering Conference (RE)*. Lisbon, Portugal, 90–99.

[28] Maria Fernanda Granda, Nelly Condori-Fernández, Tanja E. J. Vos, and Oscar Pastor. 2016. Using Model Checking to Generate Tests from Specifications. In *International Conference on Information Systems Development (ISD)*. Katowice, Poland, 17–37.

[29] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. 2017. Semantically Enhanced Software Traceability Using Deep Learning Techniques. In *International Conference on Software Engineering (ICSE)*. Buenos Aires, Argentina, 3–14.

[30] Jane Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundaram. 2006. Advancing Candidate Link Generation for Requirements Tracing: the Study of Methods. *IEEE Transactions on Software Engineering* 32, 1 (January 2006), 4–19.

[31] International Electrotechnical Commission. 2010. Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related System (IEC 61508). https://www.iec.ch/functionalsafety/. Last accessed: February 2020.

[32] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (September/October 2011), 649–678.

[33] Lin Jiang, Hui Liu, and He Jiang. 2019. Machine Learning Based Recommendation of Method Names: How Far are We. In *International Conference on Automated Software Engineering (ASE)*. San Diego, CA, USA, 602–614.

[34] K. N. King and Jeff Offutt. 1991. A Fortran Language System for Mutation-Based Software Testing. *Software: Practice and Experience* 21, 7 (July 1991), 685–718.

[35] Bogdan Korel, Inderdeep Singh, Luay Ho Tahat, and Boris Vaysburg. 2003. Slicing of State-Based Models. In *International Conference on Software Maintenance (ICSM)*. Amsterdam, The Netherlands, 34–43.

[36] Nancy G. Leveson, Mats Heimdahl, Holly Hildreth, and Jon Damon Reese. 1994. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering* 20, 9 (September 1994), 684–707.

[37] Hui Liu, Zhifeng Xu, and Yanzhen Zou. 2018. Deep Learning Based Feature Envy Detection. In *International Conference on Automated Software Engineering (ASE)*. Montpellier, France, 385–396.

[38] Patrick Mäder and Jane Cleland-Huang. 2013. A Visual Language for Modeling and Executing Traceability Queries. *Software and System Modeling* 12, 3 (July 2013), 537–553.

[39] Patrick Mäder, Orlena Gotel, and Ilka Philippow. 2009. Getting Back to Basics: Promoting the Use of a Traceability Information Model in Practice. In *International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*. Vancouver, Canada, 21–25.

[40] Patrick Mäder, Paul L. Jones, Yi Zhang, and Jane Cleland-Huang. 2013. Strategic Traceability for Safety-Critical Projects. *IEEE Software* 30, 3 (May/June 2013), 58–66.

[41] Salome Maro, Jan-Philipp Steghöfer, Jane Hayes, Jane Cleland-Huang, and Miroslaw Staron. 2018. Vetting Automatically Generated Trace Links: What Information is Useful to Human Analysts?. In *International Requirements Engineering Conference (RE)*. Banff, Canada, 52–63.

[42] Lei Mi and Kerong Ben. 2011. A Method of Software Specification Mutation Testing Based on UML State Diagram for Consistency Checking. *Procedia Engineering* 15 (2011), 110–114.

[43] Valéry M. Monthe, Laurent Nana, Georges E. Kouamou, and Claude Tangha. 2016. A Decision Support Framework for the Choice of Languages and Methods for the Design of Real Time Embedded Systems. *Journal of Software Engineering and Applications* 9 (2016), 353–397.

[44] Shiva Nejati, Mehrdad Sabetzadeh, Chetan Arora, Lionel C. Briand, and Felix Mandoux. 2016. Automated Change Impact Analysis between SysML Models of Requirements and Design. In *International Symposium on Foundations of Software Engineering (FSE)*. Seattle, WA, USA, 242–253.

[45] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. 2007. Matching and Merging of Statecharts Specifications. In *International Conference on Software Engineering (ICSE)*. Minneapolis, MN, USA, 54–64.

[46] Shiva Nejati, Mehrdad Sabetzadeh, Davide Falessi, Lionel C. Briand, and Thierry Coq. 2012. A SysML-Based Approach to Traceability Management and Design Slicing in Support of Safety Certification: Framework, Tool Support, and Case Studies. *Information & Software Technology* 54, 6 (June 2012), 569–590.

[47] Nan Niu, Anas Mahmoud, Zhangji Chen, and Gary Bradshaw. 2013. Departures from Optimality: Understanding Human Analyst's Information Foraging in Assisted Requirements Tracing. In *International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA, 572–581.

[48] Nan Niu, Sandeep Reddivari, and Zhangji Chen. 2013. Keeping Requirements on Track via Visual Analytics. In *International Requirements Engineering Conference (RE)*. Rio de Janeiro, Brazil, 205–214.

[49] Nan Niu, Wentao Wang, and Arushi Gupta. 2016. Gray Links in the Use of Requirements Traceability. In *International Symposium on Foundations of Software Engineering (FSE)*. Seattle, WA, USA, 384–395.

[50] Nan Niu, Wentao Wang, Arushi Gupta, Mona Assarandarban, Li Da Xu, Juha Savolainen, and Jing-Ru C. Cheng. 2018. Requirements Socio-Technical Graphs for Managing Practitioners' Traceability Questions. *IEEE Transactions on Computational Social Systems* 5, 4 (December 2018), 1152–1162.

[51] No Magic, Inc. 2020. MagicDraw. https://www.nomagic.com/products/magicdraw. Last accessed: February 2020.

[52] Bashar Nuseibeh and Steve Easterbrook. 2000. Requirements Engineering: A Roadmap. In *Future of Software Engineering (FOSE)*. Limerick, Ireland, 35–46.

[53] Object Management Group. 2020. Systems Modeling Language (SysML). http://www.omgsysml.org. Last accessed: February 2020.

[54] Vesa Ojala. 2006. *A Slicer for UML State Machines.* Technical Report HUT-TCS-25. Laboratory for Theoretical Computer Science, Helsinki University of Technology, Espoo, Finland.

[55] Rajwinder Kaur Panesar-Walawege, Mehrdad Sabetzadeh, and Lionel C. Briand. 2011. Using Model-Driven Engineering for Managing Safety Evidence: Challenges, Vision and Experience. In *International Workshop on Software Certification (WoSoCER)*. Hiroshima, Japan, 7–12.

[56] Process Mining Group, Eindhoven University of Technology. 2016. ProM Tools. http://www.promtools.org. Last accessed: February 2020.

[57] Patrick Rempel, Patrick Mäder, Tobias Kuschke, and Jane Cleland-Huang. 2014. Mind the Gap: Assessing the Conformance of Software Traceability to Relevant Guidelines. In *International Conference on Software Engineering (ICSE)*. Hyderabad, India, 943–954.

[58] Enno Ruijters and Mariëlle Stoelinga. 2015. Fault Tree Analysis: A Survey of the State-of-the-Art in Modeling, Analysis and Tools. *Computer Science Review* 15-16, 3 (February-May 2015), 29–62.
[59] Mehrdad Sabetzadeh, Shiva Nejati, Lionel C. Briand, and Anne-Heidi Evensen Mills. 2011. Using SysML for Modeling of Safety-Critical Software-Hardware Interfaces: Guidelines and Industry Experience. In *International Symposium on High-Assurance Systems Engineering (HASE)*. Boca Raton, FL, USA, 193–201.
[60] Wilhelm Schäfer and Heike Wehrheim. 2007. The Challenges of Building Advanced Mechatronic Systems. In *Future of Software Engineering (FOSE)*. Minneapolis, MN, USA, 72–84.
[61] Zilvinas Strolia and Saulius Pavalkis. 2017. Building Executable SysML Model. https://blog.nomagic.com/building-executable-sysml-model-automatic-transmission-system-part-1/. Last accessed: February 2020.
[62] Haiying Sun, Mingsong Chen, Min Zhang, Jing Liu, and Ying Zhang. 2014. Improving Defect Detection Ability of Derived Test Cases Based on Mutated UML Activity Diagrams. In *IEEE Annual Computer Software and Applications Conference (COMPSAC)*. Atlanta, GA, USA, 275–280.
[63] Frank Tip. 1995. A Survey of Program Slicing Techniques. *Journal of Programming Languages* 3, 3 (1995), 121–189.
[64] U.S. Department of Defense. 1980. Procedures for Performing a Failure Mode Effect and Criticality Analysis (MIL-STD-1629A). http://www.fmea-fmeca.com/milstd1629.pdf. Last accessed: February 2020.
[65] Wil M.P. van der Aalst and Kees van Hee. 2004. *Workflow Management: Models, Methods, and Systems*. MIT Press.
[66] Wil M. P. van der Aalst, H. T. de Beer, and Boudewijn F. van Dongen. 2005. Process Mining and Verification of Properties: An Approach Based on Temporal Logic. In *International Conferences "On the Move to Meaningful Internet Systems" (OTM)*. Agia Napa, Cyprus, 130–147.
[67] Wentao Wang, Arushi Gupta, Nan Niu, Li Da Xu, Jing-Ru C. Cheng, and Zhendong Niu. 2018. Automatically Tracing Dependability Requirements via Term-Based Relevance Feedback. *IEEE Transactions on Industrial Informatics* 14, 1 (January 2018), 342–349.
[68] Wentao Wang, Nan Niu, Mounifah Alenazi, and Li Da Xu. 2019. In-Place Traceability for Automated Production Systems: A Survey of PLC and SysML Tools. *IEEE Transactions on Industrial Informatics* 15, 6 (June 2019), 3155–3162.
[69] Wentao Wang, Nan Niu, Hui Liu, and Zhendong Niu. 2018. Enhancing Automated Requirements Traceability by Resolving Polysemy. In *International Requirements Engineering Conference (RE)*. Banff, Canada, 40–51.
[70] Mark Weiser. 1979. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method.* Ph.D. Dissertation. University of Michigan, Ann Arbor, MI, USA.