A Language-Based Causal Model for Safety

Marcello Bonsangue[1], Georgiana Caltais[2(✉)], Hui Feng[1], and Hünkar Can Tunç[3]

[1] Leiden University, Leiden, Netherlands
{m.m.bonsangue,h.feng}@liacs.leidenuniv.nl
[2] University of Twente, Enschede, Twente, Netherlands
g.g.c.caltais@utwente.nl
[3] Aarhus University, Aarhus, Denmark
tunc@cs.au.dk

Abstract. Inspired by the seminal works on causal analysis by Halpern and Pearl, in this paper we introduce a causal model based on counterfactuals, adapted to finite automata models and with safety properties defined by regular expressions. The latter encode undesired execution traces. We devise a framework that computes actual causes, or minimal traces that lead to states enabling hazardous behaviours. Furthermore, our framework exploits counterfactual information and identifies modalites to steer causal executions towards alternative safe ones. This can provide systems engineers with valuable data for actual debugging and fixing erroneous behaviours. Our framework employs standard algorithms from automata theory, thus paving the way to further generalizations from finite automata to richer structures like probabilistic or KAT automata.

Keywords: Causal models · Counterfactuals · Regular languages · Automata · Safety

1 Introduction

Causal models and associated causal inference machineries are precious tools for the interpretation and explanation of systems failures. Current testing and verification frameworks such as equivalence checking, for instance, assess whether or not systems comply to their specifications, and at most will produce a counterexample in case the system fails. Causal analysis, instead, plays an important role in explaining complex phenomena that are actual sources of hazards by adding, for example, additional information to counterexamples on how to avoid the hazard.

A notion of causality often embraced and adopted by computer scientists was introduced by Halpern and Pearl in their seminal works [9,10]. Their causal model encodes complex logical structures of multiple events that contribute to undesired effects, or hazards. In essence, the model is based on the the so-called alternative worlds, originally proposed by Lewis [17]. In short, Lewis assumes

Y. Aït-Ameur and F. Crăciun (Eds.): TASE 2022, LNCS 13299, pp. 290–307, 2022.
https://doi.org/10.1007/978-3-031-10363-6_20

the existence of worlds satisfying a sufficiency condition, where both the cause and the effect occur, and other worlds satisfying a necessity condition, in which neither the cause nor the effect occur. This enables formulating the counterfactual argument, which defines a first condition to be satisfied by a cause, namely: when the presumed cause does not occur, the effect will not occur either. More complex aspects such as redundancy and preemption are also captured by the causal model in [9,10]. For an intuition, redundancy refers to simultaneous events that play the same role in enabling an undesired effect. Orthogonally, preemption refers to subsequent events that have the same power to enable the effect. In both cases, the counterfactual test alone cannot determine the actual cause. Last, but not least, causes in the spirit of [9] comply to a minimality requirement which guarantees that only the relevant set of causal events is identified.

Related Work. Along time, several notions of causality have been proposed, each of which tailored to the type of the system under analysis, and associated correctness specifications. Of particular interest for this paper are the works in [4,5,15]. The aforementioned results propose trace-based adoptions of causality á la Halpern and Pearl, applicable to automata models. These, in combination with model checking-based methodologies, enabled computing causes for the violation of safety and liveness properties in Kripke structures and labelled transition systems, for instance.

Our work is closely related to the contribution in [5]. Given an automaton model, the naive goal is to identify the shortest sequence of actions that enable the effect, i.e., that can bring the system into a hazardous state. These are called “causal traces”. Note that, in contrast with the often tedious counterexamples identified by model-checkers, the minimality of causal traces implies concise descriptions of systems faults. Thus, causal traces encode essential information for systems engineers, for instance, and they can serve as a debugging aid. As previously stated, in the spirit of Halpern and Pearl, our definition of causality imposes a sufficiency condition: namely, whenever a causal trace is executed, the effect is reached as well. However, important information on how to actually avoid/fix hazardous behaviours can be extracted based on the aforementioned set of alternative worlds (or traces in our model), that do not lead to an undesired effect. Hence, we designed our causal model in the spirit of the counterfactual criterion of Lewis and identified modalities to avoid hazardous scenarios. Similarly to [4,5,15], we call these escape options – “events causal by their non-occurrence”. This information can be exploited in order to steer an execution towards an alternative safe one, with immediate applicability in synthesizing schedulers, for instance.

A rich body of work successfully exploited the counterfactual argument for fault analysis and debugging techniques. Examples related to counterexample explanation in model checking are the works in [7,8,21], for instance. In [7] the authors propose a framework for understanding errors in ANSI C programs, based on distance metrics for program executions. In [8] the cause describing the error includes the identification of source code fragments crucial to distinguishing success from failure, and differences in invariants between failing and non-failing runs. Distance criteria have also been exploited in [21], in combination with the so-called nearest neighbor queries to perform fault localization. The why-because-analysis in [14] was used to reason about aviation accidents, in a framework where Lamport's Temporal Logic of Actions (TLA) described both the behavior of a system, the (history of) hazards and the sequence of the states leading to an accident. The work in [23] provides a comprehensive approach to systematic debugging including, among others, delta debugging – a technique for isolating minimal input to reproduce an error.

For finer notions of causal dependencies that distinguish between interleaving and true concurrency, for instance, we refer to event structures [2,20]. Nevertheless, in our work, we adhere to the approaches in [4,5,15], and do not take into consideration the order of events along execution traces.

Our Contributions. We propose a shifting bisimulation setting presented in [5] to a trace based setting in the context of regular languages and automata theory. The benefits are multifold. For instance, the paradigm change facilitates the application of more standard algorithms from automata theory, in contrast with the rather ad-hoc procedures in [4,5,15]. Furthermore, the current framework enables using an expressive logic for defining safety properties in terms of regular expressions (or automata), instead of the ordinary Hennessy-Milner logic. The language based approach to causality enables representing both hazards and causal explanations in terms of automata – a format better accepted by engineers. In addition, in this paper, we use regular languages (or full regular expressions including Kleene-star) to encode non-occurrence of events. Previous related works such as [4,5,15] can only provide finite sets of runs steering an execution towards an alternative safe one. Orthogonal to the aforementioned results, the current approach entails a "may" semantics of causality, instead of "must"; nevertheless, we believe that the approach can be easily modified to cater for the "must" version. Besides, in contrast with the results in [5], steering executions are guaranteed not to jump over hazardous states by simply concatenating sequences causal by their non-occurrence and the causal trace. The ultimate goal of the current work is to generalize from finite automata to richer structures like probabilistic automata and NetKAT automata [1,6].

Structure of Paper. In Sect. 2 we provide an overview of regular languages and associated automata theory aspects. A running example is introduced in Sect. 3. Section 4 defines the language-based model of causality, whereas in Sect. 5 we show how to compute actual causes and safe computations. In Sect. 6 we provide an experimental evaluation of our method and in Sect. 7 we discuss how our model can be extended with tests and assignments. Section 8 concludes our work.

2 Preliminaries

In this section we recall few basic facts about regular languages, finite automata, and regular expressions [18].

Let A be a finite set of actions that we refer to as an *alphabet*. A *word* or *string* over A is a finite sequence $a_1 \dots a_n$ of elements from A. We denote by ε

the empty word, i.e. the sequence of length 0, and write A^* to denote the set of (possibly empty) words over A. A *language* L is just a subset of words, that is $L \subseteq A^*$. We call a word w' to be a *prefix* of a word w whenever $w = w'w''$. A word w' is said to be a *sub-word* of a word w, if w' is obtained by deleting one or more elements of A at some not necessarily adjacent positions in w. We denote by $sub(w)$ the set of all sub-words of w. Note that $sub(\varepsilon) = \emptyset$. Also, $\varepsilon \in sub(w)$ but $w \notin sub(w)$ for every non empty word w.

A *finite automaton* (FA) is a 5-tuple $M = (S, A, i, \longrightarrow, F)$, where S is a finite set of states, $i \in S$ is the initial state, $F \subseteq S$ is the set of accepting states and $\longrightarrow \subseteq S \times A \times S$ is the transition relation. For simplicity, we write $s \xrightarrow{a} t$ whenever $(s, a, t) \in \longrightarrow$. A transition relation is called deterministic if for all $s \in S$ and $a \in A$ if $s \xrightarrow{a} t_1$ and $s \xrightarrow{a} t_2$ then $t_1 = t_2$.

A string $w \in A^*$ is accepted by an automaton M from a state s if either (1) $w = \varepsilon$ and $s \in F$, or (2) $w = aw'$ and there exists $s \xrightarrow{a} t$ such that w' is accepted by M from the state t. The *language accepted* by a FA M is the set $L(M) = \{w \in A^* \mid M \text{ accepts } w \text{ from } i\}$. Since for every FA M we can build a FA N with a deterministic transition relation such that $L(M) = L(N)$, without loss of generality we will consider only finite automata with a deterministic transition relation.

A language L over the alphabet A is said to be *regular* if there exists a finite automaton M accepting it, that is $L(M) = L$. The class of all regular languages is closed under union, intersection, concatenation, complement and Kleene star. Here language union and intersections are the usual set theoretic operations, whereas concatenation of two languages L_1 and L_2 is given by the set $L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$. Finally, for a language L, its Kleene star closure is defined by $L^* = \bigcup_{n \in \mathbb{N}} L^n$ where $L^0 = \{\varepsilon\}$ and $L^{n+1} = L \cdot L^n$ for all $n \in \mathbb{N}$, thus denoting the concatenation of a language with itself a finite number of time.

In this paper, we are interested in system communicating by message passing, and thus we will always assume that the alphabet A is partitioned in three disjoint subsets A_I, A_O, and A_P of input, output and private actions, respectively. Notationally, for $a \in A$ we write $a?$ if a is an input action in A_I and $a!$ if a is an output action in A_O, and use no markings for private actions in A_P. We use σ to denote an action that can be either input, output or private.

Let A and B be two alphabets with disjoint private actions, and assume the set P is disjoint from Q. Given two finite automata $M = (P, A, i, \longrightarrow_M, E)$ and $N = (Q, B, j, \longrightarrow_N, F)$ their *parallel composition* is defined by the finite automaton $M \mid\mid N = (P \times Q, \Sigma, < i, j >, \longrightarrow, E \times F)$ where $\Sigma_I = (A_I \backslash B_O) \cup (B_I \backslash A_O)$, $\Sigma_O = (A_O \backslash B_I) \cup (B_O \backslash A_I)$, $\Sigma_P = (A_P \cup B_P) \cup (A_I \cap B_O) \cup (A_O \cap B_I)$, and $\longrightarrow$ is the least transition relation such that

$$\frac{p \xrightarrow{\sigma}_M p' \quad \sigma \notin B}{\langle p, q\rangle \xrightarrow{\sigma} \langle p', q\rangle} \qquad \frac{q \xrightarrow{\sigma}_N q' \quad \sigma \notin A}{\langle p, q\rangle \xrightarrow{\sigma} \langle p, q'\rangle}$$

$$\frac{p \xrightarrow{a?}_M p' \quad q \xrightarrow{a!}_N q'}{\langle p, q\rangle \xrightarrow{a} \langle p', q'\rangle} \quad \frac{p \xrightarrow{a!}_M p' \quad q \xrightarrow{a?}_N q'}{\langle p, q\rangle \xrightarrow{a} \langle p', q'\rangle}$$

The topmost rules are about either private actions that are not affected by the other automaton or communication actions that do not involve the other automaton. The two rules at the bottom are about complementary communication actions $a!$ and $a?$ that are synchronized resulting in the private action a. Note that when $A = B$ with $A_I = B_O$, $A_O = B_I$ and $A_P = B_P = \emptyset$ then parallel composition reduces to the product automata where all actions synchronize. In the case A is completely disjoint from B then parallel composition results in the so-called shuffle product. Other variation of synchronization product could be defined in a similar way, including multi-process synchronization, hiding of successful communication, value passing synchronization (for a finite value domain) and synchronization parameterized by a finite subset of actions.

For the characterization of the parallel composition of two languages we need first to introduce the *projection* function. Given two alphabets A_1 and A_2 we define the projection $\pi_i : (A_1 \cup A_2)^* \to A_i^*$ by $\pi_i(\varepsilon) = \varepsilon$, and $\pi_i(\sigma \cdot w) = \sigma \cdot \pi_i(x)$ if $\sigma \in A_i$, and $\pi_i(w)$ otherwise. Because projections are surjective functions they have inverse π_i^{-1} returning the set of strings that are projected into a given one. More precisely, we define the *inverse projection* by $\pi_i^{-1}(w) = \{x \in (A_1 \cup A_2)^* \mid \pi_i(x) = w\}$ for every $w \in A_i^*$. Projections and their inverses can extended to languages by applying them to all the strings in the language. In general we have that $\pi_i(\pi_i^{-1}(L)) = L$ but for the converse it only holds that $L \subseteq \pi_i^{-1}(\pi_i(L))$. Note that if two alphabets A_1 and A_2 have disjoint private actions and we partition $A_1 \cup A_2$ as in the alphabet of the parallel composition of two automata, then projections will assign private actions of $A_1 \cup A_2$ to either private, input or output actions in A_i unambiguously. Similarly, inverse projections assign private actions to private actions, but may assign input and output actions to private ones.

The *parallel composition* of two languages $L_1 \subseteq A_1^*$ and $L_2 \subseteq A_2^*$ is the language $L_1 \mid\mid L_2$ on the alphabet $A_1 \cup A_2$ defined as $\pi_1^{-1}(L_1) \cap \pi_2^{-1}(L_2)$. Basically, the intersection takes care that dual communication actions will be synchronized, and that disjoint private events will be shuffled with the others. As expected, we have that $L(M_1 \mid\mid M_2) = L(M_1) \mid\mid L(M_2)$, implying that regular languages are closed under parallel composition [22].

We conclude this section by introducing extended regular expressions, that we may use as alternative syntax to FAs in order to reason about causality in complex systems composed of several components potentially communicating with each other.

Given an alphabet A including communication actions, *extended regular expressions* are given by the following grammar:

$$e ::= 0 \mid 1 \mid a \mid a? \mid a! \mid e\,;e \mid e + e \mid e \mid\mid e \mid e^*, \tag{1}$$

where $a \in A_P$, $a?$ implies $a \in A_I$, and $a!$ implies $a \in A_O$. In process theoretic terms, 0 denotes no behavior, 1 denotes a terminating process. The further building blocks of processes are (communication) actions. Processes can be composed sequentially, non-deterministically, in parallel, or can loop a finite number of times. Communication between process terms is performed based on synchronizations between opposite communication actions, that play thus a sender, respectively, receiver role. In the sequel we often use A as shorthand for the regular expression obtained by the finite set of every action in A, and $\neg a$ as a shorthand for the set of every action in A except a. Note that in general we could extend negation to all regular expression, as regular languages are closed under complement.

Ordinary regular expressions are expressions without any parallel composition. Except for the parallel composition we assume that an action cannot be used as input and output in the same 'sequential' expression, i.e., regular expression with no occurrence of the $||$ operator. With this mild restriction we can associate to each regular expressions e a language $L(e)$ inductively as follows:

$$\begin{array}{lll} L(0) = \emptyset & L(e_1 \,;\, e_2) = L(e_1) \cdot L(e_2) & \\ L(1) = \{\varepsilon\} & L(e_1 + e_2) = L(e_1) \cup L(e_2) & L(e^*) = L(e)^* \\ L(a) = \{a\} & L(e_1 \,||\, e_2) = L(e_1) \,||\, L(e_2) & \end{array}$$

It is well known [11] that the language of an ordinary regular expression is regular. The same holds for our extended regular expressions, as we have seen that regular languages are closed under parallel composition. This implies that for every (extended) regular expression e there exists an automaton M such that $L(e) = L(M)$. We will not describe the construction here as it is outside the scope of this paper.

3 A Railway Crossing Example

In this section we recall the railway crossing example from [5], and adapt it to our present setting. The example consists of a car, a train, and a gate of a crossing that communicates with the train. The gate can communicate its the status of being closed ($Gc!$) or open ($Go!$). The status changes to closed only after the gate receives a message from the train that is approaching the crossing ($Ta?$), and it can change to open only after it receives the message that the train leaves the crossing ($Tl?$). The behavior of the gate is described by the following regular expression:

$$G = (Go!^* \,;\, (1 + Ta? \,;\, Gc!^* \,;\, Tl?))^*.$$

When a train is approaching the crossing, it sends a message ($Ta!$). After that, it will actually enter the crossing (Tc) and then send a message informing its departure from the crossing ($Tl!$). This behavior is described by the following regular expression:

$$T = Ta! \,;\, Tc \,;\, Tl!.$$

Finally, a car can approach the crossing (Ca), wait as long as the gate is closed ($Gc?$), eventually observe the gate being open ($Go?$), and only then it may enter the crossing (Cc) and leave the crossing afterwards (Cl). The regular expression encoding this is given by:

$$C = Ca \,;\, Gc?^* \,;\, Go? \,;\, Cc \,;\, Cl.$$

The FAs corresponding to the above three regular expression are illustrated in Fig. 1. Note that the car can enter the crossing only after the gate is open, whereas the gate enters the state of being open only after a train signals its departure.

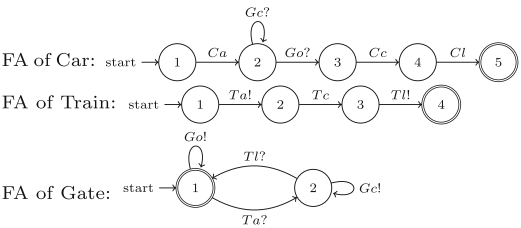

Fig. 1. The car, train and gate as FAs

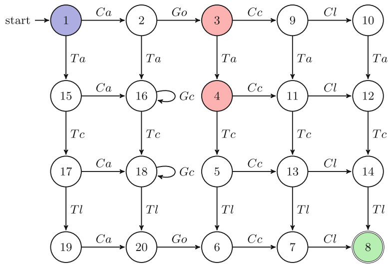

Fig. 2. The railway system as a FA

In Fig. 2 we see the automaton describing the railway system that results by the parallel composition of the three regular expressions: $C \mid\mid T \mid\mid G$ where, for simplicity, we renamed the states. For example, the initial state ① corresponds to the state $\langle 1, 1, 1\rangle$ and the only accepting state is ⑧ corresponding to $\langle 5, 4, 1\rangle$. The red states ③ and ④ will be used in the next section as examples of states leading to a hazard situation: a car entering the crossing and not leaving it before the train enters the crossing too.

4 A Language-Based Causal Model

In this section we introduce a notion of causality with respect to a so-called *hazard*, or *effect* expressed in terms of regular expressions. The current causal framework is inspired from the model introduced in [5] and massaged into the setting of FAs with the goal to use trace semantics instead of bisimulation, and define different system properties in terms of regular expressions (such as reachability) instead of the ordinary Hennessy-Milner logic.

In short, a hazard is a regular language specified by a regular expression e (or the corresponding automaton). It is said to occur in a FA M representing our model whenever there is a finite (and possibly empty) string $c = a_0 \dots a_n$ in M such that after c we may observe the hazard, that is, $L(c\,;\, e) \cap L(M) \neq \emptyset$. In this case, we say that c may enable the hazard e in M. Additional conditions that have to be satisfied by c, such as minimality and non-occurrence of events, are formalized in Definition 1.

For an intuition, consider the railway crossing example of the previous section. A hazardous situation can happen whenever both the train and the car enter the crossing, and none of them leaves the crossing before the other one enters it. The regular expression encoding this hazard is:

$$e = (Cc\,;\, (\neg Cl)^*\,;\, Tc + Tc\,;\, (\neg Tl)^*\,;\, Cc)\,;\, A^* \tag{2}$$

Note that the hazard situation can terminate with any string in A^*. This is to guarantee that after a trace c enables e, their concatenation will contain behaviors accepted by the automaton, and thus the hazard is observed. It is straightforward to see that in the FA in Fig. 1 it is possible to reach the above hazard with the string $c_1 = Ca\,Go$ leading to the state ③, but also with the string $c_2 = Ca\,Go\,Ta$ leading to the state ④. In fact, the intersection of the language of the hazard e with that of the automaton M starting from either state ③ or ④ instead of ① is non-empty. Furthermore, state ③ and ④ are both reachable from the initial state ①.

We may say that c_1 does a better job at describing the relevant sequence of actions that, if triggered, lead to a hazard because it is a *minimal* sequence enabling it. Moreover, we see that it is possible to avoid the hazard by "decorating" the string c_1 with the strings Ta, $Tc\,Tl$ and, respectively, $Cc\,Cl$. This can result, for instance, in the string $w = Ta\,\underline{Ca}\,Tc\,Tl\,\underline{Go}\,Cc\,Cl$ which does not lead to a hazard. Sequences such as Ta, $Tc\,Tl$ and $Cc\,Cl$ are called *causal by non-occurrence* in works such as [4,5]. Non-occurrence is essential for describing how certain dangerous situations, if controllable, can be avoided within a system. This concept plays an important role in our definition of causality.

As formalized in Definition 1, non-occurrence of events is captured in terms of the so-called *computations* [5]. The latter are strings in a regular language, typically denoted by π, built on top of a string $c = a_0 \dots a_n$, and "decorated" with strings $d_0^i, \dots, d_{n+1}^i$, with $i \in I$, where I is a finite set of integers, such that:

$$w \in \pi \quad \Rightarrow \quad w = d_0^i a_0 d_1^i \cdots a_n d_{n+1}^i.$$

Intuitively, given a trace c that enables a hazard, strings in π describe all the alternative runs (such as w above) that execute all actions in c and avoid the hazard. The only requirement is that all strings specified by π are observable executions of M; i.e., for a given FA M, $\pi \subseteq L(M)$. Notice that π being a regular language means that it can be expressed as a regular expression r, and because all strings in π contain c as subword, we have $r = \Sigma_{j,k} r_k^j$ with $r_k^j = r_0^j;\, a_0;\, r_1^j;\, \dots a_k;\, r_{k+1}^j$ for some finite indexes j and k and regular expressions r_{k+1}^j. For simplicity, we sometimes write r in lieu of π.

The next definition formally introduces *decorated causes* for a FA M with respect to a hazard e.

Definition 1 (Causality for FAs). *Let $M = (S, A, s_0, \rightarrow, F)$ be a FA, e be a regular expression over A, denoting a hazard, and $c \in A^*$. We say that the computation π built on top of c, with $\pi \subseteq L(M)$, is a* decorated cause *of the hazard e if*

AAC1: *The string c may enable e* – $L(c\,;\, e) \cap L(M) \neq \emptyset$
AAC2.1: *If the effect e is not observed then it has not been caused by c* –
$\forall w \in L(M) \backslash L(A^*\,;\, e) \,:\, (L(w\,;\, e) \cap L(M) = \emptyset) \Rightarrow (c \notin sub(w) \vee w \in \pi)$.
AAC2.2: *Strings of π are safe, i.e., they do not cause the effect e* –
$\forall w \in \pi : w \notin L(A^*\,;\, e) \wedge (L(w\,;\, e) \cap L(M) = \emptyset)$

AAC3: *Minimality –*
for all $c' \in sub(c)$ there is no computation π' built on top of c' with $\pi' \subseteq L(M)$, that satisfies **AAC1–AAC2.2** *with respect to the string c' and the hazard e.*

We call c as above a causal trace*, and sometimes write $Cause_c(e, M)$ to denote the corresponding decorated cause π. We let $Causes(e, M)$ be the union of all $Cause_c(e, M)$.*

Intuitively, **AAC1** identifies a scenario where the string c enables the hazard e in M. Note that **AAC1** entails a "may" semantics of causality, instead of "must", as c does not always have to lead to e. Catering for the "must" version requires modifying **AAC1** to $L(c\,;\,e) \subseteq L(M)$. **AAC2.1** is a necessity condition according to which, if a word w cannot enable e, then either w does not contain the causal trace c (meaning it is an execution bringing not to the hazard), or it has been decorated with events that eliminate the possibility of executing the hazard. Note that **AAC2.1** can be equivalently expressed (by modus tollens) as a sufficiency condition stating that a string w enables the hazard e whenever the causal trace is contained in w but it is not decorated with elements causal by their non-occurrence that would avoid the execution of the hazard:

$$\forall w \in L(M)\backslash L(A^*\,;\,e)\,:\,(c \in sub(w) \wedge w \notin \pi) \Rightarrow (L(w\,;\,e) \cap L(M) \neq \emptyset)$$

AAC2.2 requires causal traces decorated with events causal by their non-occurrence to avoid the hazard. Furthermore, note that c itself cannot be a safe computation in π, because otherwise **AAC2.2** would contradict **AAC1**. Observe that **AAC2.2** is reminiscent of the traditional counterfactual criterion of Lewis, as it allows to test the dependence of e on c under certain contingencies encoded, in our case, in terms of non-occurrence of events. We refer to [10] for more insight on the so-called *structural contingencies*. **AAC3** is the minimality condition that requires to consider decorated causes entailed by the shortest causal traces c satisfying **AAC1–AAC2.2**.

We conclude the section with a few examples intended to clarify certain aspects of the above definition and the differences with the work [5]. To begin with, we illustrate the role played by loops in the decorations of computations.

Example 1. Consider the automaton M_1 in Fig. 3 and let the hazard be expressed by the regular expression $e = c\,;\,A^*$, meaning that we have to avoid executing action c.

Clearly, the string $a\,b$ is a possible cause for the hazard. Hence, $Cause_{ab}(e, M_1)$ for this example can be encoded via the regular expression: $a\,;\,f\,;\,h^*\,;\,b\,;\,g$. Note that as a result of considering the decorations as regular expressions, all finite repetitions of the loop are conveniently represented with the Kleene star operator. The work in [5] handles loops in the decorations by unfolding the loop only a finite number of times specified a-priori, hence, only the string afh^nbg would be describing hazard avoidance, for all $n \leq k$ and some fixed k.

In the second example, we consider the case when there are no possible decorations to steer a causal trace away from its hazard.

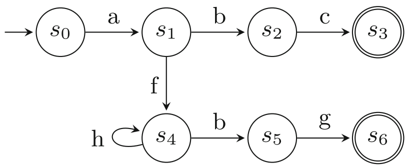

Fig. 3. Automaton M_1

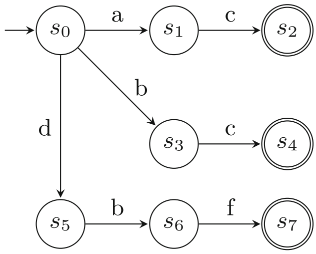

Fig. 4. Automaton M_2

Example 2. Consider the automaton M_2 in Fig. 4 and let the hazard be as before expressed by the regular expression $e = c\,;\, A^*$.

In this example, there are two possible causal traces, namely, a and b. There are no possible decorations for the causal trace a to make it avoid the hazard, whereas, there exists a decoration for the causal trace b with $Cause_b(e, M_2) = d\,;\, b\,;\, f$. Whenever there are no computations π satisfying Definition 1 for e in M w.r.t. a trace c, we say that the hazard e, if enabled by c, is *unavoidable* in M.

In the above two examples there was no actual difference if we would have used c as hazard instead of the regular expression $c\,;\, A^*$. In the next example we show a FA where the two expressions entail different decorated causes.

Example 3. Consider the automaton M_3 in Fig. 5 and the hazards $e = c\,;\, A^*$ and $e' = c$.

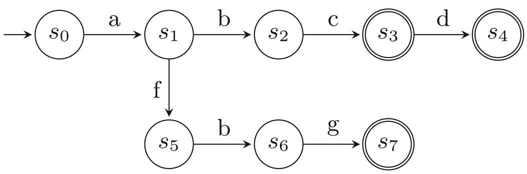

Fig. 5. Example 3

For both hazards, ab is the causal trace, but

$$\begin{aligned} Cause_{ab}(e, M_3) &= a\,;\, f\,;\, b\,;\, g \\ Cause_{ab}(e', M_3) &= a\,;\, f\,;\, b\,;\, g \;\;+\;\; a\,;\, b\,;\, c\,;\, d \end{aligned}$$

Observe that the string $abcd$ is considered safe (i.e., avoids the hazard) according to $Cause_{ab}(e', M_3)$ but is not considered safe in $Causes_{ab}(e, M_3)$, wheres the string $afbg$ is considered safe in both cases. This is different than the usual notion of safety (modeled as in e and thus forbidding any possible continuation after the hazard) as e' allow to overpass the hazard if the system does not stop there. In fact, the expression e' asserts that the trace cannot halt with the action c. Accordingly, both $abcd$ and $afbg$ are valid strings that satisfy this condition and thus avoid the hazard e'. On the other hand, the expression e asserts that the action c followed by any possible sequence of actions (i.e., in A^*) constitutes a violation, hence, the action c cannot be observed at any point in an execution. Therefore, only $afbg$ is a valid execution that will avoid the hazard e. It is essentially not possible to define properties similar to e with the approach in [5], as they allow jumping over a hazardous state while executing strings in π.

5 Computing Causes

Given a FA $M = (S, A, i, \longrightarrow, F)$ and an effect specified by a regular expression e on A, we show an algorithm for computing the set $Causes(e, M)$ using standard operations on automata and graphs. The algorithm first computes the set of loop-free traces that lead to the hazard e. Then, for each one of them, it determines the associated computation satisfying conditions **AAC2.1**–**AAC2.2** in Definition 1. The union of all such computations will give a first approximation of the set $Causes(e, M)$. We will then show below how to obtain precisely the set $Causes(e, M)$ by requiring the minimality condition **AAC3** in Definition 1.

Algorithm 1: Computing Causes

Input: A FA $M = (S, A, i, \longrightarrow, F)$, an effect e.
Output: The set of decorated causes $Causes(e,\ M)$.

(1) Compute the set of traces that lead to e by following the steps:
  (1.1) For all $s \in S$, construct the FA $P_s = (S,\ A,\ s,\ \longrightarrow,\ F)$ and compute the following intersection:
  $L(P'_s) = L(P_s) \cap L(e)$.
  (1.2) Construct the automaton $P = (S, A, i, \longrightarrow, F')$ where $F' = \{s \mid L(P'_s) \neq \emptyset\}$.
  (1.3) Compute all simple paths from the intial states i and a final state $f \in F$ in P.
  (1.4) Let $CausalTraces$ be the set of all strings in $L(P)$ labeling the paths computed in (1.3).
(2) For all $c = a_0 \dots a_n \in CausalTraces$, compute $Cause_c$(e, M) by :

$$(L(A^*; a_0; A^*; \dots; A^*; a_n; A^*) \setminus \{c\}) \cap (L(M) \setminus (L(A^*; e) \cup L(P)))$$

(3) Return the union of all the languages computed in step (2) as Causes(e, M).

Next, we discuss the underlying ideas behind the certain steps of Algorithm 1 and then provide a proof of correctness for the algorithm. We first compute all traces that enable e by constructing in steps (1.1) and (1.2) the automaton P that accepts exactly all traces in M possibly causing the effect e. The only difference between the automata P and M is their set of final states. The procedure for constructing P first involves constructing a set of automata P_s, for all the states s of the automaton M, such that s is the initial state in P_s and accepts strings of the language of the hazard e. If the intersection of $L(P_s)$ with $L(e)$ is non-empty, then the corresponding state is considered as a final state in the automaton P (step (1.2)). As a result, the strings in $L(P)$ are exactly those strings bringing M to a state of where the hazard is activated. For our railway crossing example in Sect. 3 with the hazard given by the regular expression in (2), the automaton P would be the one in Fig. 2 with states ③ and ④ as the only final states.

In step (1.3) we compute $CausalTraces$ as the subset of strings accepted by P via a simple path starting from the initial state and ending in a final state. These paths correspond to the set of loop-free traces that lead to the hazard e. While this condition does not guarantee minimality (see discussion below) it already reduces the set of possibly causal traces to a finite set. In general, $L(P)$ will be infinite, if it involves a loop in the automaton.

For each of the above finitely many causal traces, in step (2), we compute the set of associated computations. For a given possibly causal trace c, this is done by subtracting all the traces which enable the effect (i.e., $L(P)$) and all the traces which observe the effect (i.e., $L(A^*; e)$) from $L(M)$ and then take the intersection of the resulting language with the language resulted from c decorated with non-occurrence in all possible ways. Note that the intersection computed in step (2) may be empty, meaning that the hazard e is unavoidable when executing the actions of c. For our running example in Sect. 3, the possibly causal traces computed by the algorithm are $CaGo$ and $CaGoTa$. Examples of strings in the associated computations are $CaGoTaCcClTcTl$ and $CaGoCcClTaTcTl$. Note that the first string avoids the hazard for both possibly causal traces, while the latter is a string that avoids the hazard for $CaGo$.

Finally, the union of the resulting languages in the step (2) of Algorithm 1 is returned as a first approximation of the set of all decorated causes of M for the hazard e. For this set, the following theorem guarantees that conditions **AAC1**–**AAC2.2** hold. However, condition **AAC3** may fail to hold.

Theorem 1. *The computations in $Causes(e, M)$ returned by Algorithm 1 satisfy conditions* **AAC1**–**AAC2.2** *by construction.*

Proof. The set $Causes(e, M)$ returned by Algorithm 1 is obtained as the union of all $Causes_c(e, M)$ for all $c \in CausalTraces$. Elements in this sets are obtained in step (1.4). These strings are computed based on the language that the automaton P (constructed in step (1.2)) recognizes. By construction, $x \in L(P)$ implies there is $y \in L(e)$ such that $xy \in L(M)$. Hence $L(x\,;\,e) \cap L(M) \neq \emptyset$. Since $CausalTraces \subseteq L(P)$, condition **AAC1** holds.

In order to show that **AAC2.1** holds for some $c \in CausalTraces$, take a string x accepted by M that is not in $L(A^*\,;\,e)$. Assume that $L(x\,;\,e) \cap L(M) = \emptyset$. Then $x \notin L(P)$ because otherwise, as we have just seen above, there would exist $y \in L(e)$ such that $xy \in L(M)$. Therefore, $x \in L(M) \backslash (L(A^*; e) \cup L(P))$. Because $CausalTraces \subseteq L(P)$, it follows that $x \neq c$ for any possibly causal trace c. We have now two cases: for every $c \in CausalTraces$ either $c \in sub(x)$ or not. In the latter case **AAC2.1** holds. In the other case $c \in sub(x)$ and thus $x \in L(A^*; a_0; A^*; \ldots; A^*; a_n; A^*)$, from which it follows based on step (2) that $x \in Causes_c(e, M)$, and thus **AAC2.1** holds.

It remains to show that **AAC2.2**. For some possibly causal trace $c \in CausalTrace$ let $x \in Cause_c(e, M)$. We must show that $x \notin A^*e$ and that $L(x; e) \cap L(M) = \emptyset$. The first part of the conjunction in **AAC2.2** holds because by the construction in step (2) $Cause_c(e, M)$ cannot contain strings from $L(A^*; e)$. Similarly, the second part of the conjunction holds because $L(P)$ is subtracted from $L(M)$ in the same step.

Condition **AAC3** does not necessarily hold for $Cause_c(e, M)$ used by the Algorithm 1. In fact, for possibly causal traces $x, y \in CausalTraces$, if $x \in sub(y)$ then any sub-string of x is also a sub-string of y. In other words, for $a_0 \cdots a_n = x \neq y = b_0 \cdots b_m$ we have

$$L(A^*; a_0; A^*; \ldots; A^*; a_n; A^*) \subseteq L(A^*; b_0; A^*; \ldots; A^*; b_m; A^*) \tag{3}$$

By step (2) of Algorithm 1 we thus have that $Causes_x(e, M) \subseteq Causes_y(e, M)$. Note that it must be the case that $m > n$ for $x \in sub(y)$. We can therefore easily compute the smallest sets of safe computations by removing from the set $CausalTraces$ all strings y that have another possibly causal trace $x \in CausalTraces$ of smaller length as sub-word. In our running example, the trace $CaGo$ is clearly a sub-word of the other one $CaGoTa$, and indeed, the computation for $CaGoTa$ is included in the computation for $CaGo$ as well. Hence, only the causal trace $CaGo$ satisfies the minimality condition **AAC3** .

6 Experimental Evaluation

In this section, we provide an experimental evaluation and assess the applicability of our method. We developed a tool prototype implementing our approach and evaluated the time performance by computing the decorated causes on randomly generated FAs with growing size. The implementation is based on Python and closely follows Algorithm 1. The inputs to our tool are a FA and a regular expression which describes the effect on the given FA. The output of our tool is an automaton which characterizes the set of all decorated causes with respect to the given inputs. In our implementation we utilized the BRICS automaton library [19] for performing standard automaton operations.

We evaluated our tool in the following experimental setting: we generated random FAs by using the libalf [3] framework. In the process of generating FAs we fixed the size of the alphabet to 5. We then generated over 1000 FAs with increasing number of states and achieved a maximum of 300 states. Figure 6 shows an example of a FA with 5 states that was generated randomly by libalf. For each generated FA we also randomly computed an effect for which the decorated causes are determined. We fixed the size of the effect length to 3. All the experiments were conducted on a computer running Ubuntu 20.04.3 with 8 core 1.8GHz Intel i7-10510U processor and 16 GB RAM.

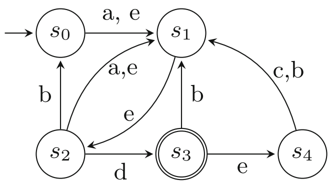

Fig. 6. Randomly generated FA with 5 states.

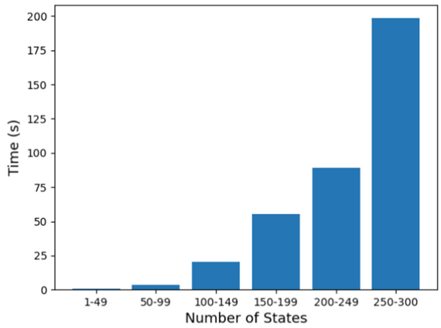

Fig. 7. Experimental results

The results of our experiments are displayed in Fig. 7. We group the randomly generated FAs by their number of states and report the average running times in each group. We only report the times of the experiments in which the decorated causes were not empty. The results indicate that for relatively small FAs with less than 100 states, a result is obtained within 10 s. For larger FAs with 250 to 300 states, a result is obtained in 3 min on average and within 15 min at maximum. We remark that these results are obtained without any attempts to tailor the standard automaton operations to our setting.

Table 1. Average size of obtained decorated causes.

	Number of states in the input FA					
	1–49	50–99	100–149	150–199	200–249	250–300
# States	71	185	266	422	484	560
# Transitions	236	654	997	1565	1862	2177
# Potential Causes	81	328	10476	21932	44750	73318
# (Minimal) Causes	3	8	10	18	10	22

In Table 1 we summarize some information on the automata that recognize the decorated causes returned by the algorithm. Depending on the number of states of the automata given as input, we report the average number of states and transitions of the returned automata, the average number of causes and the average number of minimal causes obtained. As to be expected, the size of the automata of the output increased linearly with that of the input. However, the number of potential causal traces computed increases exponentially. That is not the case for the number of minimal causal traces, as it increases only marginally when the size of the input increases. In fact, in the majority of the cases, the number of minimal causes is less than 5, regardless of the size of the given input automaton.

7 Extensions

To illustrate the generality of our causal model we briefly discuss possible extensions so to consider addition of tests and assignments.

Adding Tests: KAT. The set of regular expressions we considered in (1) can be extended with a set B of Boolean tests that we assume generated from a finite set At of atoms, meaning that every $b \in B$ is equivalent modulo the equations of the Boolean algebra to a finite disjunction of atoms in At. This way one can model basic programming constructs, like conditionals, loops, guarded actions, and assertions using tests in B and actions in A.

Kozen [12] showed that the above extensions of regular expressions, called KAT (Kleene algebra with tests) expressions, plays the same role with regular sets of guarded strings as ordinary regular languages play for regular expressions. Here a *guarded string* is an ordinary string over the alphabet $A \cup At$, such that the symbols in A alternate with the atoms At. Formally, a guarded language is a subset of $(At \times A)^* \times At$.

A *deterministic KAT automaton* recognizing guarded strings [13] is just a deterministic finite automaton $(S, \Sigma, i, \longrightarrow, F)$ with $\Sigma = At \times A$ and $F \subseteq S \times At$. The only differences are thus the transition relations that is now labeled by guarded actions (α, a), and the accepting states, that are now labeled with atoms marking the end of an accepted string. The idea is that an action a is executed only when its guard α (pre-condition) is true, and a string is accepted only in states where the post-condition holds. We say that a guarded string $w \in (At \times A)^* \times At$ is accepted by a KAT automaton M from a state s if either (1) $w = \alpha$ and $(s, \alpha) \in F$, or (2) $w = (\alpha, a)w'$ and there exists $s \xrightarrow{\alpha,a} t$ such that w' is accepted by M from the state t. The *language accepted* by a KAT automaton M is the set $L(M) = \{w \in (At \times A)^* \times At \mid M \text{ accepts } w \text{ from } i\}$.

Our causal model for automata extends naturally to KAT automata by considering hazards e as KAT-expressions and causes c as strings in $(At \times A)^*$. Safe computations in M for the hazard e with respect to c are non-empty strings of $L(M)$ satisfying **AAC1** as in Definition 1 but with respect to the alphabet $(At \times A)$ instead of A only. Also the algorithm for computing causes needs basically no adjustment, but for the way how operations on automata are computed.

Adding Assignments: NetKAT. NetKAT [1] is a network programming model, which is used for specifying and verifying the packet-processing behavior of software-defined networks. In a nutshell, it is a variation on KAT that considers actions not as abstract elements of an alphabet A but rather as state transformers, like assignments, that are executed when a precondition α is satisfied and modify it into a post-condition β.

For a given set of atoms At of a Boolean algebra B, a *deterministic NetKAT automaton* [6] is a deterministic FA $M = (S, \Sigma, i, \longrightarrow, F)$ such that $\sigma = At \times At$ and $F \subseteq S \times (At \times At)$. The transition relation $\longrightarrow$ is thus labeled by pairs of atoms (α, β) and so are the accepting states. The interpretation of these pair of atoms is that they represent pre-conditions and post-conditions of one-step executions.

A string $w \in At \times At \times At^*$ is accepted by M from a state s only when post-conditions match the subsequent pre-condition, meaning that either (1) $w = \alpha\beta$ and $(s, \alpha, \beta) \in F$, or (2) $w = (\alpha\beta)w'$ and there exists $s \xrightarrow{\alpha,\beta} t$ such that $\beta w'$ is accepted by M from the state t. Note that in the last condition is crucial that w' is not the empty string. The *language accepted* by a NetKAT automaton M is the set $L(M) = \{w \in At \times A \times At^* \mid M \text{ accepts } w \text{ from } i\}$.

As for KAT automata, our causal model for automata extends naturally to NetKAT automata too, with hazard represented by NetKat expressions [6], causes as strings in At^*, and safe computations as strings in $L(M)$ that can be projected into a cause by deleting some atoms and satisfying the rest of the conditions of Definition 1.

8 Conclusions

In this paper we moved the causal model proposed in [5] from labeled transition systems to finite automata in order to obtain a language-based causal model for safety. The model is in-line with the notion of causality described in a logical context in [9] in the sense that a hazard may be observed if and only if it has been caused. Analogously to the alternative worlds of Lewis [16], we also considered decorated causes as alternative to causes in the sense that they allow executing all actions of a cause interleaved with other actions that guarantee hazard avoidance.

We treated only the case when causes *may* enable a hazard while strings of the decorated causes must avoid it. While it can be interesting to consider a stronger notion of causes as strings c that bring the automaton M to states where the hazard e is inevitable for any of its possible extensions (i.e., by changing **AAC1** to $L(c\,;e) \subseteq L(M)$), such a change would imply that there would be no causes in our railway system example.

We have also presented an algorithm to compute decorated causes, relying only on basic automata theoretic operations. The algorithms could be improved, using model checking techniques for marking those states in which a hazard is enabled, and search techniques to find the decorated causes avoiding marked states. Also, it would be interesting to move from automata back to labeled transition systems but remaining into a trace setting, with hazard specified as LTL-properties.

Finally, we briefly discussed extensions of our work to KAT and NetKAT automata. Clearly, more work needs to be done here, both to precisely set the definitions and to show the applicability of the method to, for example, find causes of a hazard in a software defined network.

Acknowledgments. The work of Georgiana Caltais and Hünkar Can Tunç was supported by the DFG project “CRENKAT”, no. 398056821.

References

1. Anderson, C.J., et al.: NetKAT: semantic foundations for networks. In: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, 20–21 January 2014, pp. 113–126 (2014). https://doi.org/10.1145/2535838.2535862
2. Arbach, Y., Karcher, D.S., Peters, K., Nestmann, U.: Dynamic causality in event structures. Logical Methods Comput. Sci. **14**(1) (2018). https://doi.org/10.23638/LMCS-14(1:17)2018
3. Bollig, B., Katoen, J.-P., Kern, C., Leucker, M., Neider, D., Piegdon, D.R.: libalf: the automata learning framework. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 360–364. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_32
4. Caltais, G., Guetlein, S.L., Leue, S.: Causality for general LTL-definable Properties. In: Finkbeiner, B., Kleinberg, S. (eds.) Proceedings 3rd Workshop on formal reasoning about Causation, Responsibility, and Explanations in Science and Technology, CREST@ETAPS 018, Thessaloniki, Greece, 21 April 2018. EPTCS, vol. 286, pp. 1–15 (2018). https://doi.org/10.4204/EPTCS.286.1
5. Caltais, G., Mousavi, M.R., Singh, H.: Causal reasoning for safety in Hennessy Milner logic. Fund. Inform. **173**, 217–251 (2020). https://doi.org/10.3233/FI-2020-1922
6. Foster, N., Kozen, D., Milano, M., Silva, A., Thompson, L.: A coalgebraic decision procedure for NetKAT. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, 15–17 January 2015, pp. 343–355 (2015). https://doi.org/10.1145/2676726.2677011
7. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. Int. J. Softw. Tools Technol. Transf. (STTT) **8**(3), 229–247 (2006)
8. Groce, A., Visser, W.: What went wrong: explaining counterexamples. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 121–136. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44829-2_8
9. Halpern, J.Y.: A modification of the Halpern-Pearl definition of causality. In: Yang, Q., Wooldridge, M.J. (eds.) Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, 25–31 July 2015, pp. 3022–3033. AAAI Press (2015). http://ijcai.org/Abstract/15/427
10. Halpern, J.Y., Pearl, J.: Causes and explanations: a structural-model approach: part 1: causes. In: Breese, J.S., Koller, D. (eds.) UAI 2001: Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence, University of Washington, Seattle, Washington, USA, 2–5 August 2001, pp. 194–202. Morgan Kaufmann (2001). https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=100&proceeding_id=17
11. Kleene, S.C.: Representation of events in nerve nets and finite automata. Autom. Stud. **34**, 3–42 (1956)
12. Kozen, D.: Kleene algebra with tests. ACM Trans. Program. Lang. Syst. **19**(3), 427–443 (1997)
13. Kozen, D.: Automata on guarded strings and applications. Mat. Contemp. **24**, 117–139 (2003)
14. Ladkin, P., Loer, K.: Analysing aviation accidents using WB-analysis - an application of multimodal reasoning. In: AAAI Spring Symposium. AAAI (1998). https://www.aaai.org/Papers/Symposia/Spring/1998/SS-98-04/SS98-04-031.pdf

15. Leitner-Fischer, F., Leue, S.: Causality checking for complex system models. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 248–267. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35873-9_16
16. Lewis, D.: Causation. J. Philos. **70**, 556–567 (1973)
17. Lewis, D.: Counterfactuals. Blackwell Publishers, Hoboken (1973)
18. Martin, J.C.: Introduction to Languages and the Theory of Computation, vol. 4. McGraw-Hill, New York (1991)
19. Møller, A.: dk.brics.automaton – finite-state automata and regular expressions for Java (2021). http://www.brics.dk/automaton/
20. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, part I. Theor. Comput. Sci. **13**, 85–108 (1981). https://doi.org/10.1016/0304-3975(81)90112-2
21. Renieris, M., Reiss, S.: Fault localization with nearest neighbor queries. In: 18th International Conference on Automated Software Engineering (2003)
22. Shabana, H., Volkov, M.V.: Optimal synchronization of partial deterministic finite automata. CoRR abs/2002.01045 (2020). https://arxiv.org/abs/2002.01045
23. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Elsevier, Amsterdam (2009)