Three Views of Log Trace Triaging

Raphaël Khoury, Sébastien Gaboury, and Sylvain Hallé

Laboratoire d'informatique formelle
Département d'informatique et de mathématique
Université du Québec à Chicoutimi, Canada
Email: raphael.khoury@uqac.ca, s1gabour@uqac.ca, shalle@acm.org

Abstract. This paper extends previous work on execution trace triaging. We examine the problem of trace triaging along three of the four views used in the study of temporal properties, namely the automata-theoretic view, the temporal logic view and the set-theoretic view. For each case, we propose several partitions of universe of possible traces into equivalence classes, which follow naturally from the chosen view and form the basis for trace triaging.

1 Introduction

The problem of log trace triaging consists of partitioning a set of traces into meaningful equivalence classes, with respect to the evaluation of a sequential property over these traces. Stated more formally, triaging can be seen as a generalization of trace verification, replacing its two-valued (pass/fail) result with membership in one of potentially many equivalence classes —thus providing a more detailed feedback about the status of compliance or violation of the trace. As a simple example, consider the property "eventually, either a or b will hold", and the two traces cca and ccb. Checking the property in the classical way would return the same verdict ("true") for both; however, one could imagine multiple ways of creating a finer classification: separating traces that fulfill the property because a holds from those where b holds, or according to the length of their longest non-compliant prefix, and so on.

So far, log trace triaging has been the subject of scarce literature; as we shall see, the closest line of works concentrates on bug classification. However most techniques either require human intervention, lack a formal definition, or perform a form of partitioning that does not satisfy the definition of an equivalence class. There is therefore a need for an automated and formally-grounded partitioning methodology. In previous work, we introduced the basic concepts of such a trace classification, as approached from the angle of Linear Temporal Logic [27]. In particular, we introduced the concept of *trace hologram*, a tree structure resulting from the evaluation of an LTL formula on a specific trace. When interpreted as equivalence classes, manipulations on these holograms cluster event traces into various natural categories, many of which correspond to intuitive ways of grouping them.

However, temporal logic is but one of four equivalent *views* in which formal properties about traces can be stated and verified. These views, as defined in a paper by Chang et al., also include automata, language theory and topology [10]. Therefore, in this paper, we build on previous work by approaching the problem of trace triaging also from

the automata-theoretic and language-theoretic viewpoints. We shall show that the concepts introduced earlier, namely of creating equivalence classes based on some criterion computed from the evaluation of a temporal formula, have equivalents in finite-state automata and language theory. We will describe multiple ways of classifying traces with respect to a property; in some cases, we also demonstrate that some of these classifications subsume others.

Such a partitioning of event traces into equivalence classes can have several applications in the understanding, debugging and maintenance of complex software systems. For example, in test case generation, it may be desirable to select from the possibly infinite set of possible program behaviors, a finite subset of test cases that covers every possible type of behavior of interest. One possible way of doing so could be by picking one trace in each equivalence class defined by some triaging rule. When debugging, one can use triaging to narrow down the analysis on a subset of recorded traces for which the execution has violated a property in a particular manner. Triaging can also help system administrators minimize the overhead of record-keeping, since for many applications, records of a single trace in each equivalence class can be sufficient, if we can determine that the set of classes ranges over every possible behavior of interest. Finally, equivalence classes also provide an easy way to perform trace reduction (also called trace abstraction): abstracting a trace simply amounts to replacing it with the shortest trace that belongs to the same equivalence class. For all these reasons, the study of property-based partitioning of event traces introduced in this paper shall prove to be a stepping stone towards the achievement of these goals.

In this paper, we present a theoretical framework that guides the triaging of traces into meaningful subclasses, and permits new classifications schemes to be defined, and compared with existing ones. We show how the problem of trace triaging can be examined from each of three different views of temporal specifications defined in the literature, namely temporal logic, automata and language theoretic, and that each view gives rises to different classifications. Indeed, a classification that can be easily stated and performed when operating in a given view can be difficult of impossible to express under a different view.

The remaining of this paper proceeds as follows: Section 2 introduces preliminary notions related to traces and formalism. In section 3, we review our previous work on trace triaging, which approached the problem from the perspective pf temporal logic view. Section 4 and Section 5 propose triaging based on two additional views, namely automata theoretic and language theoretic. Section 6 surveys related works. Concluding remarks are given in Section 7.

2 Preliminaries

Let Σ be a finite or countably infinite set of events, each of which is assumed to be a set of name-value pairs. An execution trace is a finite sequence of events, and captures the behaviour of a system at runtime. We let σ, τ range over sequences. Let Π be a set of attribute names, and V be a set of values; event i in a trace σ is a partial function $\sigma_i : \Pi \to V$ that assigns a value to attributes. Σ^* denotes the set of all traces and $acts(\sigma)$

is the set of events occurring in trace σ. We write σ_* for the ultimate event of sequence σ. The concatenation of sequences σ and σ' is given as $\sigma;\sigma'$.

2.1 Log Trace Triaging and Temporal Specifications

Trace triaging is the problem of sorting traces into meaningful categories. In other words, it is the task of devising a triaging function $\kappa : \Sigma^* \to C$, with $C \subseteq \mathbb{P}(\Sigma^*)$, the selector, that maps each trace to a class $c \in C$. For a given triaging function κ we write $[\![c]\!]_\kappa$ for the set of traces $S \subseteq \Sigma^*$ such that $\sigma \in S \Leftrightarrow \kappa(\sigma) = c$. The subscript κ is omitted when clear from context.

Let $p \in \Pi$ be some attribute, $\sigma, \sigma' \in \Sigma^*$ two traces that are identical, except that at their i-th event, $\sigma_i(p) \neq \sigma'_i(p)$; where $a(p)$ indicates the valuation of attribute p in event a. These two traces are said to be (p,i)-different. A formula φ is said to be p-invariant if, for any pair of (p,i)-different traces σ, σ', $\sigma \models \varphi \Leftrightarrow \sigma' \models \varphi$.

This concept is best illustrated through an example. Let $\varphi \equiv \mathbf{G}\,(\mathrm{p} = 0)$, and the following two traces composed of a single event: $\sigma = \{(p,0),(q,0)\}$, $\sigma' = \{(p,0)\}$. One can see that σ and σ' are $(\mathrm{q},0)$-different, since $\sigma(\mathrm{q}) = \{0\}$ and $\sigma'(\mathrm{q}) = \emptyset$. Intuitively, it is clear that the truth value of φ is the same for any two (q,i)-different traces, and hence that φ is q-invariant.

Let φ be a specification of property of interest in the context in which the traces are generated and triaged. Any useful selector κ should respect the two following properties (from [27]).

Definition 1 (Coherence) *A selector κ is called* coherent *if and only if:*

$$\forall c \in C : \forall \sigma, \sigma' \in [\![c]\!]_\kappa : \sigma \models \varphi = \sigma' \models \varphi.$$

Informally, this first property states that a category cannot contain both compliant and non-compliant traces. The second property states that that two traces that are not meaningfully different with respect to the property of interest should be placed in the same category:

Definition 2 (Consistency) *A selector κ is called* consistent *if and only if: for each $p \in \Pi$, if φ is p-invariant and σ,σ' are two (p,i)-different traces, then $\kappa(\sigma) = \kappa(\sigma')$.*

We say that a triaging function is *reasonable* when it is both coherent and consistent. If we let $\kappa : \Sigma^* \to C$ and $\kappa' : \Sigma^* \to C'$ be two selectors, we say that κ is finer than κ' (written $\kappa \preceq \kappa'$) if $\forall \sigma \in \Sigma^* : \kappa(\sigma) \subseteq \kappa(\sigma')$.

2.2 The Four Views of Temporal Specifications

According to Chang et al. [10], a given property φ can be visualized along any one of four distinct views: temporal logic, automata, language theory and topology. In this paper, we approach the problem of trace triaging from the first three of these four views. We argue that since each view gives rise to a different property representation and a different verification mechanism, each view should also give rise to a different trace triaging paradigm.This intuition is illustrated in Figure 1.

Varvaressos et al. [27] introduced trace holograms, as a mechanisms to generate and represent triaging functions. An hologram is an abstraction of the tree generated when verifying the satisfaction of an LTL formula on a trace. Several triaging schemes occur naturally when specific information (e.g. nodes or node labels) is deleted from the tree. Two distinct traces can thus be considered equivalent and classified into the same category. In an analogous manner, if the desired property is stated as an automaton, the verification process generates a path over the states of the automaton, which can also be abstracted to produce a trace categorization. A similar reasoning can be applied to the language theoretic representation of properties.

The ability to examine the same property along multiple alternative view has several advantages, as a given property may be more concise, or more readily understandable or checkable, given the chosen representation. Alternative representations also give rise to alternative verification algorithms, and thus to different tools. Likewise, one of the main advantages of a multi-paradigm view of trace triaging, is that a given partitioning of Σ^* might be stated in a natural and intuitive manner in a given view of temporal properties, but the same classification would be difficult to achieve if a different view of temporal properties were used.

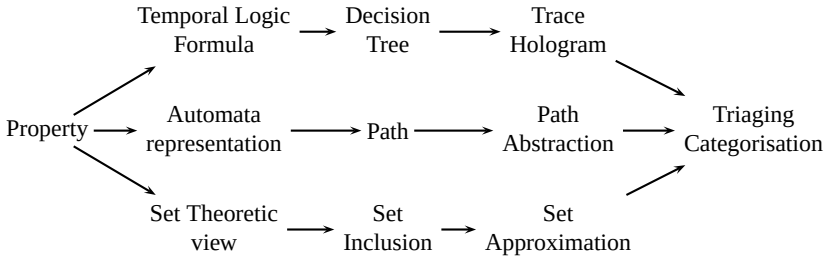

Fig. 1: The alternative views of log trace triaging

3 Temporal Logic View

In previous work [27], it was shown how the use of a formal specification of a system's expected behaviour, expressed as formulæ of Linear Temporal Logic (LTL), can be used as the basis for a classification of execution traces. By repeatedly applying the recursive rules defining the semantics of LTL operators, the evaluation of a LTL formula φ on a trace σ induces a tree. Figure 2 shows such a tree for the formula $\mathbf{G}(a \rightarrow \mathbf{X}b)$, evaluated on the trace *cab*; the top-level operator of that formula, **G**, corresponds to the top-level node of the tree. According to the semantics of LTL, $\mathbf{G}\varphi$ is true if and only if φ is true for every suffix of the current trace. The tree hence spawns three child nodes, corresponding to the evaluation of $a \rightarrow \mathbf{X}b$ for traces *cab*, *ab* and *b*, respectively. Taking the first such child node, the the top-level operator now becomes $\rightarrow$; this operator evaluates to $\top$ when, on the current trace, either a evaluates to $\bot$ or $\mathbf{X}b$ evaluates

to $\top$. This, in turn, spawns to child nodes corresponding to each condition, and so on. Provided that n-ary operators are evaluated in a fixed order, this structure is uniquely defined for a given formula and a given trace.

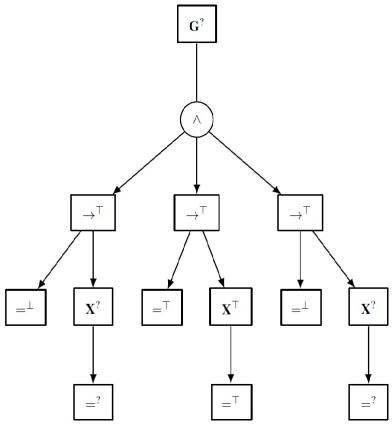

Fig. 2: Evaluating an LTL formula on a trace induces a tree.

As a first classification, we take κ to be the function that associates each trace to its hologram. We have demonstrated in earlier work that such a classification is reasonable [27]. However, different traces are likely to have different holograms. We introduce a number of systematic rules by which pieces of an hologram can be taken off. Applying these rules has for effect that traces with originally different holograms may now belong to the same category, thereby merging trace categories.

3.1 Fail-fast Deletion

The first deletion pattern is the fail-fast deletion. It consists of deleting all children of a temporal operator node that no longer have an influence on its truth value. Figure 3 shows the procedure for the **G** operator; φ is an arbitrary subformula, and the symbols $\star_i$ represent its truth value for each event, with the additional condition that $\star_i \neq \bot$ for $1 \leq i < n$. The box φ_n hence represents the first child node that evaluates to $\bot$. One can see in Figure 3b that all subtrees following φ_n are deleted. Intuitively, this represents the fact that, once the n-th event has φ evaluate to $\bot$, then $\mathbf{G}\,\varphi$ itself evaluates to $\bot$, no matter how φ evaluates on the subsequent events.

Consider for example the formula $\mathbf{G}\,(a \rightarrow \mathbf{X}\,b)$, and the two traces *caaa* and *caac*. Originally, these two traces have different holograms; however, the application of fail-fast deletion on both holograms results in the same output, and the two traces become members of the same category. This corresponds to the intuitive notion that one does not care what follows in a trace once a violation has occurred. Note however that the trace *cacc* is still considered distinct, since the reason for the failure is different (a "*c*",

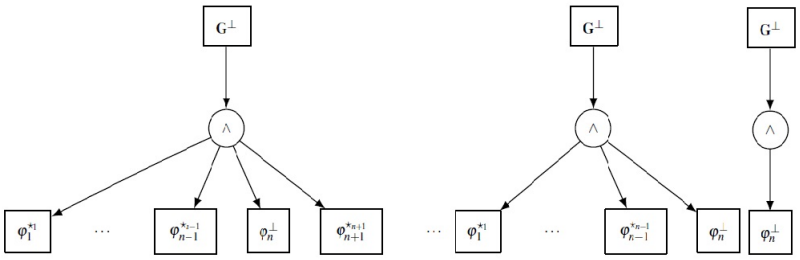

Fig. 3: Two deletion patterns for the **G** operator. (a) Original hologram (b) After fail-fast deletion (c) After polarity deletion

rather than an "*a*", occurred instead of the expected "*b*"). A dual rule for the **F** operator can also be devised, by swapping the roles of $\top$ and $\bot$.

3.2 Polarity Deletion

Fail-fast deletion applies only to temporal operators. As an extension of that rule, one may only keep nodes that are sufficient to decide on the value of an expression. For example, if the expression $\varphi \wedge \psi$ evaluates to $\bot$ because φ evaluates to $\bot$, then it is not necessary to conserve ψ, since its truth value has no effect on the result (and dually for the $\vee$ operator). Similarly, if the formula $\mathbf{G}\,\varphi$ evaluates to $\bot$ because the n-th event of a trace $\overline{\sigma}$ does not satisfy φ, it is not necessary to conserve nodes describing how φ evaluates to $\top$ on the $n-1$ previous events: the knowledge that $\sigma^n \not\models \varphi$ is sufficient to decide on the value of $\mathbf{G}\,\varphi$. More generally, it is not necessary to keep nodes of a hologram whose *polarity* (i.e. their truth value) does not contribute to the final result of the global formula. Figure 3c shows the result of polarity deletion on the hologram of Figure 3a. Again, a dual reasoning can be made for the **F** temporal operator.

When applied to temporal operators, this deletion rule expresses the fact that two traces where the same violating sequence of events occurs are considered the same, even if this sequence is preceded by a varying number of events irrelevant to the violation. For example, using polarity deletion, the traces *bcabbbd* and *cabbd* get similar holograms for the formula $\mathbf{G}\,(a \rightarrow \mathbf{X}\,(b\,\mathbf{U}\,c))$. The problem with both traces is that, after the occurrence of an "*a*", the sequence of "*b*" is broken off by a "*d*" instead of the expected "*c*". The position of the "*a*", and the actual number of "*b*" seen before the offending "*d*" are both abstracted away.

3.3 Truncation

A simple deletion rule consists of trimming from the hologram all nodes beyond a certain depth n. An extreme case is $n = 1$, which deletes all but the root of the hologram. In such a situation, κ classifies traces only according to the global truth value of the

specification. For other values of n, truncation is such that one does not distinguish traces up to a certain level of abstraction. For example, truncating the holograms for $\mathbf{G}((a \rightarrow \mathbf{X}b) \wedge (\mathbf{F}b \wedge \mathbf{F}c))$ at a depth of $n = 2$ indicates that one is interested in knowing which of the two eventualities caused the failure, but not the actual contents of the event that caused it.

4 Automata Theoretic View

We shall now adopt a different point of view, and consider properties on traces expressed as finite automata. A finite deterministic automaton $\mathscr{A}$ over the alphabet Σ is a tuple $\langle Q, q_0, \delta, S \rangle$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma \rightarrow Q$ is a transition function and $S \subseteq Q$ is a set of accepting states.

For any automaton there exists a minimal unique (up to isomorphism) canonical automaton. In what follows, we only consider canonical automata. A path π is a sequence of states $\langle q_1, q_2, q_3, ..., q_n \rangle$, such that such that there exists a finite sequence of symbols $a_1, a_2, a_3, ..., a_n$ called the label of π such that $\delta(q_i, a_i) = q_{i+1}$ for all $i \leq n$. In fact, a path is a sequence of states consisting of a possible run of the automaton, and the label of this path is the input sequence that generates this run. A run is initial if it begins on the start initial state q_0. A run is accepting if it ends on an accepting state $s \in S$.

A sequence σ satisfies the property φ if the associated run on the property automaton is initial and accepting.

The properties of coherence and consistency stated above can be stated quite straightforwardly in a automata theoretic formalism. A selector κ is coherent iff it does not include both accepting and non-accepting paths. A selector κ is consistent if it does not distinguish between two traces that exhibit the same path over the property automaton. The converse does not hold, as two sequences can meaningfully differ, and yet exhibit the same path over the canonical automaton of a property. This means that it may not be possible to state the finest classification when examining a property in the automata-theoretic view. Instead, the finest classification expressible in automata view is one that places in the same category every sequence that exhibits the same path over the property automaton.

Proposition 1 *Let $\kappa : \Sigma^* \rightarrow C$ and let φ be a property. If selector κ is consistent then $\forall \sigma, \sigma' \in \Sigma^* : \pi(\sigma) = \pi(\sigma') \Rightarrow \kappa(\sigma) = \kappa(\sigma')$.*

Proposition 2 *Let $\kappa : \Sigma^* \rightarrow C$ and let φ be a property. Selector κ is coherent iff $\forall \sigma, \sigma' \in \Sigma^* : \pi(\sigma)$ is an accepting run $\Leftrightarrow \pi(\sigma')$ is an accepting run $\Rightarrow \kappa(\sigma) = \kappa(\sigma')$.*

The automata-theoretic view presents two interesting advantages that distinguishes it from the other representations of properties. First, it is easy to generate a "typical", or generic trace in each category C by computing the the shortest run over the property automata that visits the states needed for the trace to be included in C. Secondly, for a given program, it is possible to count how many different executions of length less than n are present in each category, by adapting the algorithm proposed by Bang et al. [4]. These functionalities have multiple applications, notably for estimating the coverage

of an explicit-state model checking algorithm. This counting measurement can also be used as a complexity metric, providing an alternative to cyclomatic complexity.

In the following, We propose three path-based triaging schemes that are potentially useful for log trace triaging. All are abstractions of the notion of paths, thus analogous to trace holograms.

4.1 Shallow history based triaging

The shallow history (i.e. the same unordered set of visited states) was first introduced by Fong [13], who showed that it is a sufficiently fine approximation to serve in the enforcement of most real-life access-control policies, and other safety properties. This notion can be used for triaging, with two sequences being considered equal iff they share the same shallow history. Formally, $\kappa_{sh} : \Sigma* \rightarrow \mathscr{P}(\Sigma) : \forall \sigma, \sigma' \in \Sigma^* : \kappa_{sh}(\sigma) = \kappa_{sh}(\sigma') \Leftrightarrow acts(\sigma) = acts(\sigma')$.

For example, the Chinese Wall Policy [7] is used to avoid conflicts of interests arising from the unrestricted flow of information. In this model, a user which accesses a data object o is forbidden to simultaneously accessing certain other data objects that are identified as being in conflict with o. A violation of the policy occurs if the user accesses conflicting data objects. Since the application of the property is not sensitive to the order or number of occurrences of each data access, it makes senses to classify log traces according to their shallow history. Other access control policies [13] behave similarly.

When abstracted to its shallow history, a trace of data access events will retain only a list of the objects accessed by each user, regardless of the number of times each object has been accessed. Since this information is sufficient to detect a violation of the security policy or to generate a useful counterexample, it is not necessary to distinguish between multiple traces that vary only with respect to other, extraneous, information.

Theorem 1. *Let property φ be a safety property or a guarantee property. κ_{sh} is a reasonable selector.*

Proof. An automata representing a safety property possesses a single distinguished end state s, with no outgoing transitions [3]. Every invalid sequence passes through this state. Since every trace σ for which $s \notin \kappa_{sh}(\sigma)$ is valid, and every trace for which $s \in \kappa_{sh}(\sigma)$ is invalid, this classification is coherent. Conversely, an automata representing a guarantee property has a single valid accepting state s with no outgoing transitions, and only those sequence that exhibits s are valid. That κ_{sh} is consistent follows immediately from definition 1.

For non-safety and non-guarantee properties, the κ_{sh} is not necessarily reasonable.

4.2 Duplicate Deletion

The second automata-based trace classification scheme is based upon the deletion of some visited states from the path. This classification scheme is useful if we are interested in examining which behaviors are present or absent in the trace, but not in how

many times each present behavior occurs. Two traces abstract into the same class iff they share the same ordered list of visited states. For example, sequences $a;a;a;b;b;c$ and $a;b;b;b;c$ and $a;b;c$ are in the same category, but $c;b;a$ is not. We consider an erasing function $\delta : \Sigma^* \to \Sigma^*$ defined as follows:

$$\delta(\sigma;a) = \begin{cases} \sigma;a, & \text{if } \nexists i \in \mathbb{N} : \sigma_i = a; \\ \sigma, & \text{otherwise.} \end{cases}$$

The selector is defined with respect to this function as $\kappa_{dd} : \Sigma^* \to \mathscr{P}(\Sigma) : \forall \sigma, \sigma' \in \Sigma^* : \kappa_{dd}(\sigma) = \kappa_{dd}(\sigma') \Leftrightarrow \delta(\sigma) = \delta(\sigma')$.

The ordered list of states captures meaningful information about a program's behavior in many cases, as it gives a concise summery of the different behaviors present in the trace. For example, system call trace can contain several repetitions of the same system calls, representing copying a file to memory or sending data through a network. While the actual system call numbers carry meaningful information (i.e. identifies the higher-level behavior present in the trace), the number of occurrences of each system call is not particularly relevant, and may vary between runs of the same program.

As was the case with the shallow history based classification, duplicate deletion is only reasonable for safety and guarantee properties. For liveness properties, a sequence path may alternate between two states, and the entire execution will be considered valid or invalid depending on the final state present in the path — an information that is not recorded by the duplicate deletion abstraction.

Theorem 2. *Let property φ be a safety property or a guarantee property. κ_{dd} is a reasonable classification.*

Proof. Proceeds identically as that of theorem 1.

Theorem 3. *κ_{dd} is a finer selector than κ_{dd}.*

Proof. Follows immediately from the fact that κ_{dd} distinguishes between path that visit the same states with a different ordering, while κ_{sh} does not.

4.3 Stuttering Insensitivity

As a final automata-based classification, we consider stuttering insensitivity [14]. Stuttering is the repetition of more than one consecutive occurrence of the same token in a sequence. This notion has multiple applications, notably optimizations in model-checking [17]. Stuttering Insensitivity differs from Duplicate Deletion in that the former preserves multiple occurrences of same token in a path, as long as they are not consecutive, while the latter erases all but one occurrence of each trace-event. When using Stuttering Insensitivity as a classifier, $a;a;a;b;b;a;$ and $a;a;b;a;a$ are in the same category but $a;a;a;b;b$ is not. In this paper, we consider stuttering in the path over the automaton validating the property (stuttering in the predicates of the automata's input sequences themselves would not be a reasonable selector in the sense of definitions 1 and 2, except in the particular case of stuttering-insensitive languages). As was the case

with duplicate deletion, stuttering insensitivity is defined with respect to an erasing function $\gamma : \Sigma^* \to \Sigma^*$:

$$\gamma(\sigma;a) = \begin{cases} \sigma, & \text{if } \sigma_* = a; \\ \sigma;a, & \text{otherwise.} \end{cases}$$

The selector is defined as $\kappa_{si} : \Sigma^* \to \mathscr{P}(\Sigma) : \forall \sigma, \sigma' \in \Sigma^* : \kappa_{dd}(\sigma) = \kappa_{dd}(\sigma') \Leftrightarrow \gamma(\sigma) = \gamma(\sigma')$.

Theorem 4. *Let property φ be a safety or a guarantee property. κ_{dd} is a reasonable classification.*

Proof. Proceeds identically as that of theorem 1.

We can now show that κ_{si} is a finer selector than κ_{dd} and κ_{sh}.

Theorem 5. $\kappa_{si} \preceq \kappa_{dd} \preceq \kappa_{sh}$.

Proof. That follows immediately from the fact that κ_{dd} distinguishes between paths that visits the same states multiples times in a non-stuttering manner, while κ_{dd} does not.

5 Language Theoretic View

The third and final view of properties which we will consider is the language theoretic-view. Language theory is a convenient representation in which properties are directly represented as sets of sequences. This allows theorems and proofs to be formulated with ease. First off, the desirable properties of consistency and coherence can be formalized in a Language Theoretic manner using the notion of *residual language*. The residual language of a sequence σ, with respect to a property φ is the set of sequences τ such that $\sigma;\tau \in \varphi$. Formally $res(\sigma_\varphi) \equiv \{\tau | \sigma;\tau \in \varphi\}$. Let φ be a property, and let $\sigma\sigma' \in \Sigma^*$ be two sequences that are identical except for the value of their ultimate event, which differ in a single path value q . φ is q-invariant iff $res(\sigma_\varphi) = res(\sigma'_\varphi)$.

Proposition 3 (Consistency) *Let $\kappa : \Sigma^* \to C$ and let φ be a property. κ is consistent iff $\forall p \in \Pi : \forall \sigma\Sigma^* : \sigma, \sigma'$ are identical except that at their ith event: $\sigma_i;(p) = \sigma_i;(p) : res(\sigma_\varphi) = res(\sigma'_\varphi) \Rightarrow \kappa(\sigma;a) \wedge \kappa(\sigma;a')$.*

Proposition 4 (Coherence) *Let $\kappa : \Sigma^* \to C$ and let φ be a property. If selector κ is consistent then $\forall \sigma \in \Sigma^* : \kappa(\sigma) \subseteq \varphi \vee \kappa(\sigma) \cap \varphi = \emptyset$.*

5.1 Edit distance based classification

The trace correction distance [28] is the minimal number of insertions, deletions and substitutions needed to transform a given sequence σ into a new sequence σ' such that $\sigma' \models \varphi$. It is a generalization of the Levenshtein distance. Observe that for sequences that already satisfy the property, the correction distance is 0.

The correction distance is useful because it gives users an intuitive measure of how invalid a sequence is, allowing to distinguish gradations between violations of the property. In the context of security policy enforcement [20], it provides an approximation of the amount of modifications needed to recover from a violation. Trace correction can also be used for triaging, with two sequences being considered equal iff they share the same edit distance to a valid sequence. Let $correct : \Sigma^* \to \mathbb{N}$ be a function that calculates the minimal edit distance from any sequence in Σ^* to a sequence in the property of interest, using the algorithm presented in [28]. Selector κ_{ed} is formally defined as:

$$\kappa_{ed} : \Sigma* \to \mathscr{P}(\Sigma) : \forall \sigma, \sigma' \in \Sigma^* : \kappa_{ed}(\sigma) = \kappa_{ed}(\sigma') \Leftrightarrow correct(\sigma) = correct(\sigma')$$

Theorem 6. *κ_{ed} is a reasonable classification.*

Proof. That κ_{ed} is coherent holds trivially from the fact that every sequence σ such that $\sigma \models \varphi$ has correction 0, and no invalid sequence does. Let σ, σ' be two φ-invariant sequences and let n be the correction distance for σ. Since σ and σ' are φ-invariant, they differ with respect to only one path event e. Since φ is invariant with respect to e, the correction of the distance necessarily implies either modifying this event for both traces, or for neither. Since the two traces do not differ with respect to any other event, their respective edit distance are the same.

5.2 Classifications based on subwords

As a final classification strategy, we consider two schemes based upon the presence or absence of subwords of a given length k. The subwords (or factors) of length k of a sequence σ are the sequences of length k that occur inside a word. For example the sequence $a;b;a;b;a;b;a$ contains the factors of length 2 $a;b$ and $b;a$ and the factors of length 3 $a;b;a$ and $b;a;b$. Subwords are frequently used as an abstraction for the behavior of complex systems.

Let $sub_k(\sigma)$ be the set of subwords of length k present in a sequence σ. We define κ_{s_k} as follows:

$$sub_k(\sigma) : \Sigma* \to \mathscr{P}(\Sigma) : \forall \sigma, \sigma' \in \Sigma^* : \kappa_{s_k}(\sigma) = \kappa_{s_k}(\sigma') \Leftrightarrow sub_k(\sigma) = sub_k(\sigma')$$

Theorem 7. *$\forall j,k \in \mathbb{N} : j < k \Rightarrow \kappa_{s_k}$ is a finer classifier than κ_{s_j}.*

While subword are frequently used in trace analysis, they are only a coherent classification in the case of *locally testable properties* [6]. Membership of a word in a locally testable languages are defined by a set of factors of bounded length k of that word, irrespective of the order of occurrences or their frequency. As shown in [26] these include a number of security-relevant properties. The classification is not generally consistent, unless care is taken to merge classes that differ only with respect to to the presence of two p-different subwords for a p-invariant property.

Theorem 8. *Let property φ be a locally testable property. κ_{s_k} is a coherent classification.*

Proof. That κ_{s_k} is coherent follows immediately from the definition of locally testable properties, which can be defined by the inclusion or exclusion of finite length subwords.

5.3 Residual Language

We can use the residual language of a sequence as the basis for classification, with the intuition that two sequence are equivalent iff the same set of continuations will lead to a valid sequence:

$$\kappa_{res} : \Sigma* \rightarrow \mathscr{P}(\Sigma) : \forall \sigma, \sigma' \in \Sigma^* : \kappa_{res}(\sigma) = \kappa_{res}(\sigma') \Leftrightarrow res(\sigma) = res(\sigma')$$

The residual language is a useful notion, notably with respect to runtime enforcement [25]. Observe that the set of possible residual languages correspond to the states of a deterministic finite automata that accept the words of the language accepted by this automata, with each state defining a different residual.

Theorem 9. *κ_{res} is a reasonable classification.*

Proof. The classifier κ_{res} is coherent, each possible residual corresponds to a state of the DFA that accepts φ, and any state is either accepting or not accepting. Likewise, the classifier κ_{res} is consistent, since any p-different sequence σ, σ' exhibit the same path over the DFA that accepts a p-invariant property, and thus exhibit the same residual language.

We can now show that κ_{res} is a coarser classification than κ_{dd} and κ_{si}.

Theorem 10. $\kappa_{si} \preceq \kappa_{dd} \preceq \kappa_{res}$

Proof. Follows immediately from the fact that the residual language of a sequence σ is uniquely defined by the final state of the path of σ over the DFA that accepts sequences in the property of interest. Let $[\![c]\!]$ be an equivalence class of sequences, classified using κ_{res}. For every sequence σ in $[\![c]\!]$, the path $\pi(\sigma)$ in the DFA of the corresponding property end in the same state d. It is easy to see that any two traces that share the same classification according to κ_{dd}, will also have the same final state. Conversely, two sequence may exhibit the same final state, but differ in other parts of the sequence and thus be classified differently according to κ_{dd}. The selector κ_{dd} is thus finer than κ_{res}.

Interestingly, the comparison between κ_{res} and the automata-based classifications is the only case in which we were able to compare selectors that originate with different views of the target property. In other cases we were only able to show that a selector was finer than another selector of the same type. This motivates the use of multiple representations of the property of interest. Likewise, we also showed how certain views were more adequate to classify certain types of properties, such as safety and guarantee properties for the automata-based view, or locally testable languages for κ_s.

6 Related Work

6.1 Logic-Based Approaches

From a logical point of view, the notion of hologram bears resemblance to the concept of Henkin witness [16], of which it can be seen as a generalization. Parallels can also

be drawn with multi-valued logics, such as LTL_3 [19] and RV-LTL [5], which provide truth values in addition to the classical $\top$ and $\bot$. For example, in the case of LTL_3, a trace σ evaluates to $\top$ with respect to a property φ if all extensions σ' are such that $\sigma;\sigma' \models \varphi$ (and dually for $\bot$). A trace is associated with a third truth value, ?, when there exist extensions σ' and σ'' such that $\sigma;\sigma' \models \varphi$ and $\sigma;\sigma'' \not\models \varphi$.

These truth values can be seen as one possible partition of the set of traces with respect to a property. The present paper generalizes this idea, and introduced many more ways of creating equivalence classes with respect to a property, which are not related to the concept of possible extensions.

6.2 Bug Classification

A second line of work relates to the classification of software bugs. The most common way of categorizing bugs is is based on their assessed severity [8]. This approach makes sense from a business standpoint, since it allows project managers to easily prioritize the resolution of bugs. However, severity is generally distributed across a handful of qualitative levels, such as "catastrophic", "essential" and "cosmetic". Some approaches rather suggest to classify bugs by ease of reproduction [15] and by type (e.g. system bugs, code bugs, etc.) [30]. Other categorizations reported include HP's three-dimensional scheme (origin, type, mode) and IBM's Orthogonal Defect Classification's six-dimensional scheme (type, source, impact, trigger, phase found and severity) [29]. The IEEE also defines a standard for the classification of software anomalies using 18 attributes [1]. All these categories, however, require human intervention (apart from basic fields such as date), and are based on a qualitative evaluation of the reported bugs.

Other approaches attempt to classify bugs through automated means. Most of these works use clustering techniques borrowed from data mining, mostly based on textual data [2, 31], which can be mined either to directly separate bugs from non-bugs [22], or to generate labels (categories) from the most frequent terms [23].

Closer to the goals introduced earlier are approaches applying data mining to execution traces for classification [21]. A number of patterns (i.e. orderings of atomic events) are first defined based on the problem domain, and the most frequent patterns occurring in a set of traces are used as the basis of a feature vector that is then fed to a clustering algorithm. However, contrarily to the approach we will present, this technique requires a set of traces to perform computation; the category to which a trace belongs is not intrinsic, and rather depends on the set of other traces on which the algorithm was applied. Moreover, since this approach, as with all data mining techniques, is based on statistical computations, the clusterings obtained do not necessarily correspond to intuitive ways of grouping traces.

6.3 Trace Abstraction

Our proposed approach also relates to *trace abstraction*, which is widely used in program maintenance and other tasks that require a solid comprehension of complex programs. Cornelissen et al. survey four of these techniques [12]. The first is subsequence summarization, [18], which assigns consecutive events that have equal or increasing nesting levels (in terms of method calls) to the same group; when a level decrease is

encountered and the difference exceeds a certain threshold, called the gap size, a new group is initiated. The second, stack depth limitation [11, 24], removes events from a trace that exceed some maximum level of nesting in method calls. Language-based filtering removes events based on their characteristics: for example, getters and setters, or private method calls, are taken out from the trace. Finally, sampling techniques simply keep every n-th event of a trace [9].

The main difference between these methods and the one we propose is that in our case, the classification has solid theoretical foundation, and more importantly are based on some property of the trace. This allows us to reason about the equivalence classes; moreover, the property is preserved despite the abstraction process. In contrast, Cornelissen's survey rather compares these techniques with respect to informal criteria, such as the proportion of a trace that is taken off, or the processing time required to compute each filter.

7 Conclusion

In this paper, we have shown how techniques borrowed from runtime verification can be adapted to the classification of event traces. First, we introduced the concept of a partition of the set of event traces, and in particular the case of *coherent* and *consistent* classification functions. For an arbitrary trace property φ, we then presented different classification functions based on φ, depending on whether it is expressed as a temporal logic formula, a finite-state automaton, or a first-order expression over languages.

The approach itself lends itself to a number of extensions and refinements. At the moment, all violations of a specification are considered equally. From a bug triaging perspective, it would be interesting to assign a weight to various parts of a specification; hence a violation of $\varphi \wedge \psi$ when φ is false could be given more weight than when ψ is false. Similarly, the number of times a property is violated could be integrated in the computation: a trace failing $\mathbf{G}\,\varphi$ ten times in a row could be given a higher value than a trace where φ is false only once. Similarly for a finite-state machine, a numerical score could be assigned to each non-accepting state, so that violating the property bears a different cost depending on the final state that is reached. In turn, these numerical values could be used to infer the equivalent of a severity metric, providing a systematic and automated alternative to the qualitative and manual assessment currently in use.

The set of all possible combinations of deletion rules forms a lattice over the inclusion relation $\sqsubseteq$ on categories. This is one of the first conditions for one to be able to apply basic data mining on Σ^*, where $\sqsubseteq$ can act as a generality relation. It would be possible, for example, to search for the tightest set of deletion rules that still puts all observed buggy traces into a single category, thereby providing a "common point" to all the bug instances found and hinting at a possible repair.

References

1. IEEE standard classification for software anomalies. Tech. Rep. 1044-2009, IEEE (2010)
2. Alenezi, M., Magel, K., Banitaan, S.: Efficient bug triaging using text mining. JSW 8(9), 2185–2190 (2013)

3. Alpern, B., Alpera, B., Schneider, F.B.: Recognizing safety and liveness. Distributed Computing 2, 117–126 (1986)
4. Bang, L., Aydin, A., Bultan, T.: Automatically computing path complexity of programs. In: Nitto, E.D., Harman, M., Heymans, P. (eds.) Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015. pp. 61–72. ACM (2015)
5. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. J. Log. Comput. 20(3), 651–674 (2010), http://dx.doi.org/10.1093/logcom/exn075
6. Beauquier, D., Pin, J.: Languages and scanners. Theor. Comput. Sci. 84(1), 3–21 (1991)
7. Brewer, D.F.C., Nash, M.J.: The chinese wall security policy. In: S&P. pp. 206–214. IEEE Computer Society (1989)
8. Carstensen, P.H., Sørensen, C., Tuikkar, T.: Let's talk about bugs!! Scandinavian Journal of Information Systems 7(6) (1995)
9. Chan, A., Holmes, R., Murphy, G.C., Ying, A.T.T.: Scaling an object-oriented system execution visualizer through sampling. In: 11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA. pp. 237–244. IEEE Computer Society (2003), http://dx.doi.org/10.1109/WPC.2003.1199207
10. Chang, E., Manna, Z., Pnueli, A.: The Safety-Progress Classification, NATO ASI Series, vol. 94, pp. 143–202. Springer (1993)
11. Cornelissen, B., van Deursen, A., Moonen, L., Zaidman, A.: Visualizing testsuites to aid in software understanding. In: Krikhaar, R.L., Verhoef, C., Lucca, G.A.D. (eds.) CSMR. pp. 213–222. IEEE Computer Society (2007), http://dx.doi.org/10.1109/CSMR.2007.54
12. Cornelissen, B., Moonen, L.: On large execution traces and trace abstraction techniques. Tech. rep., Delft University of Technology, Software Engineering Research Group (2008)
13. Fong, P.W.L.: Access control by tracking shallow execution history. In: S&P. pp. 43–55. IEEE Computer Society (2004)
14. Groote, J.F., Vaandrager, F.W.: An efficient algorithm for branching bisimulation and stuttering equivalence. In: Paterson, M. (ed.) ICALP. Lecture Notes in Computer Science, vol. 443, pp. 626–638. Springer (1990), http://dblp.uni-trier.de/db/conf/icalp/icalp90.html#GrooteV90
15. Grottke, M., Trivedie, K.S.: A classification of software faults. ISSRE pp. 4.19–4.20 (2005)
16. Henkin, L.: The completeness of the first-order functional calculus. Journal of Symbolic Logic 14(3), 159–166 (1949)
17. Holzmann, G.J., Peled, D.A.: An improvement in formal verification. In: Formal Description Techniques VII, Proceedings of the 7th International Conference on Formal Description Techniques, Berne, Switzerland, 1994. pp. 197–211 (1994)
18. Kuhn, A., Greevy, O.: Exploiting the analogy between traces and signal processing. In: 22nd IEEE International Conference on Software Maintenance (ICSM 2006), 24-27 September 2006, Philadelphia, Pennsylvania, USA. pp. 320–329. IEEE Computer Society (2006), http://dx.doi.org/10.1109/ICSM.2006.29
19. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. 78(5), 293–303 (2009)
20. Ligatti, J., Bauer, L., Walker, D.: Edit automata: Enforcement mechanisms for run-time security policies. Int. J. of Information Security 4, 2–16 (2005)
21. Lo, D., Cheng, H., Han, J., Khoo, S.C., Sun, C.: Classification of software behaviors for failure detection: a discriminative pattern mining approach. KDD pp. 557–566 (2009)
22. Moha, N., Guéhéneuc, Y.G., Leduc, P.: Automatic generation of detection algorithms for design defects. In: ASE. pp. 297–300. IEEE Computer Society (2006)
23. Nagwani, N.K., Verma, S.: CLUBAS: An algorithm and Java based tool for software bug classification using bug attributes similarities. Journal of Software Engineering and Applications 5(6), 436–447 (2012)

24. Rountev, A., Connell, B.H.: Object naming analysis for reverse-engineered sequence diagrams. In: Roman, G., Griswold, W.G., Nuseibeh, B. (eds.) ICSE. pp. 254–263. ACM (2005), http://doi.acm.org/10.1145/1062455.1062510
25. Sridhar, M.: Model-checking In-lined Reference Monitors. Ph.D. thesis, The University of Texas at Dallas, Richardson, Texas (August 2014)
26. Talhi, C., Tawbi, N., Debbabi, M.: Execution monitoring enforcement under memory-limitation constraints. Inf. Comput. 206(2-4), 158–184 (2008)
27. Varvaressos, S., Lavoie, K., Gaboury, S., Hallé, S.: A generalized monitor verdict for log trace triaging. In: PCODA. pp. 13–18. IEEE Computer Society (2015)
28. Wagner, R.A.: Order-n correction for regular languages. Commun. ACM 17(5), 265–268 (May 1974)
29. Wagner, S.: Defect classification and defect types revisited. DEFECTS '08 Proc. of the 2008 workshop on Defects in large software systems pp. 39–40 (2008)
30. Wiszniewski, H.K.B., Mork, H.: Classification of software defects in parallel programs. Tech. Rep. 2, Faculty of Electronics, Technical University of Gdansk, Poland (1994)
31. Xuan, J., Jiang, H., Ren, Z., Yan, J., Luo, Z.: Automatic bug triage using semi-supervised text classification. In: SEKE. pp. 209–214. Knowledge Systems Institute Graduate School (2010)