Preprint: International Conference on Intelligent Robots and Systems (IROS 2015)

Learning product set models of fault triggers in high-dimensional software interfaces

Paul Vernaza pvernaza@nec-labs.com
David Guttendorf dguttendorf@nrec.ri.cmu.edu
Michael Wagner mwagner@cmu.edu
Philip Koopman koopman@cmu.edu

***Abstract*— We propose a method for generating interpretable descriptions of inputs that cause faults in high-dimensional software interfaces. Our method models the set of fault-triggering inputs as a Cartesian product and identifies this set by actively querying the system under test. The active sampling scheme is very efficient in the common case that few fields in the interface are relevant to causing the fault. This scheme also solves the problem of efficiently finding sufficient examples to model rare faults, which is problematic for other learning-based methods. Compared to other techniques, ours requires no parameter turning or post-processing in order to produce useful results. We analyze the method qualitatively, theoretically, and empirically. An experimental evaluation demonstrates superior performance and reliability compared to a basic decision tree approach. We also briefly discuss how the method has assisted in debugging a commercial autonomous ground vehicle system.**

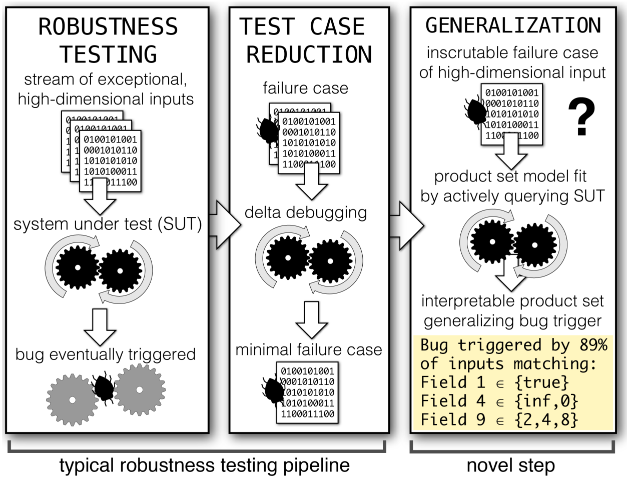

ROBUSTNESS TESTING
stream of exceptional, high-dimensional inputs
system under test (SUT)
bug eventually triggered
TEST CASE REDUCTION
failure case
delta debugging
minimal failure case
GENERALIZATION
inscrutable failure case of high-dimensional input
?
product set model fit by actively querying SUT
interpretable product set generalizing bug trigger
Bug triggered by 89% of inputs matching:
Field 1 ∈ {true}
Field 4 ∈ {inf,0}
Field 9 ∈ {2,4,8}
typical robustness testing pipeline
novel step

Fig. 1: Illustration of typical robustness testing pipeline, augmented with the novel generalization step proposed here.

I. Introduction

Recent advances in robotics have cleared many technical hurdles towards fielding complex, ubiquitous autonomous systems such as robotic cars. Our work addresses a remaining area that is critical and yet often overlooked: testing and debugging these systems to ensure their robustness and safety. Particularly, we aim to help a developer diagnose the cause of a failure by producing a human-interpretable description of the inputs that trigger the failure. This is especially important when diagnosing bugs in complex autonomous systems with high-dimensional software interfaces.

Figure 1 illustrates our general approach to testing such a system, which is based on the *robustness testing* paradigm [1]. Given an exposed interface to a system under test (SUT), the interface is bombarded with a mixed stream of nominal and exceptional inputs until a fault (software bug) is triggered, resulting in a system failure. This approach is highly effective at finding "edge cases" that cause system failures due to non-robust handling of exceptional inputs such as null pointers or NaN values. The stream of inputs can then be reduced via a test case reduction step using *delta debugging* [2] to isolate a minimal test case that reproduces the failure. For low-dimensional input spaces it is often easy to determine the trigger for a system failure by inspection [3]. But fielded robotic systems often have high-dimensional input spaces in which even a minimal test case may contain hundreds of input parameter values—far too many for a human to readily identify the trigger for the system failure.

Our technique was developed in the course of robustness testing a prototype commercial autonomous vehicle system, which we will refer to as `SystemX`. The drive-by-wire subsystem of `SystemX` can be modeled as as a function taking as input a vector of over 1000 fields; these fields convey histories of information such as the state of the brakes and the speed of the vehicle at various instants in time. Suppose now that we find a fault-triggering input. Standard practices would entail having the developer manually diagnose the cause of the failure from this input. Unfortunately, the high input dimensionality in our case makes this approach infeasible, since the developer can only guess as to which of the thousands of fields are really relevant to the failure. For example, perhaps the failure is triggered if and only if we activate the brakes at time step 2 and then activate the emergency stop at time step 3. Although every failure-triggering input must possess these features, it is impossible to identify these features as necessary and sufficient for the failure given just a single input; how do we know the fault is not triggered by activating the headlights at step 2 and deactivating the windshield wipers at step 10, if these are

This work was performed while Paul Vernaza was affiliated with the National Robotics Engineering Center (NREC), Carnegie Mellon University, 10 40th Street, Pittsburgh, PA 15201. David Guttendorf and Michael Wagner are affiliated with the NREC. Philip Koopman is affiliated with the Electrical and Computer Engineering Department, Carnegie Mellon University.

This work was supported by the Test Resource Management Center (TRMC) Test and Evaluation / Science Technology (TE/ST) Program through the U.S. Army Program Executive Office for Simulation, Training and Instrumentation (PEO STRI) under Contract No. W900KK-11-C-0025, Stress Testing for Autonomy Architectures (ASTAA).

also features of the observed fault-triggering input?

Fig. 1 illustrates our solution to this problem, which consists of *generalizing* a single fault-triggering input. This produces a set of fault-triggering inputs characterized by a few, simple, human-interpretable rules that implicate specific field-value assignments in triggering the bug. Even if these rules are not perfect, they can serve as hints to lead a developer towards the area of code that contains the bug.

The key idea of this paper is to represent the fault-triggering inputs as a Cartesian product set. Product sets provide a simple, interpretable model of independence that allows us to efficiently sift through a very high-dimensional input space to identify candidate trigger values for specific fields. We achieve this via a novel algorithm that exactly identifies an unknown product set from oracle queries, using a number of queries scaling only logarithmically in the input dimensionality and linearly in the number of relevant fields The algorithm achieves this by singling out the rare queries that are informative for fitting our model. Compared to a naive approach based on decision trees, we will also see that our approach produces reliable and repeatable results without requiring any sort of parameter tuning or pruning heuristics. Our approach is also efficient and easy to implement.

This work provides three main contributions. First, we propose and justify the idea of interpretably generalizing fault triggers via product set modeling. Next, we propose and analyze a novel active learning algorithm to efficiently identify product sets when the number of relevant fields is small. Finally, we provide anecdotal and quantitative evidence that our method is useful in the context of debugging a fielded, complex, autonomous system.

II. Method

Before describing our method in detail, we now introduce some key assumptions and notation. First, we assume access to an oracle indicating whether a given input caused a specific failure. In our case, the oracle may be obtained with black-box techniques—such as runtime verification [4] or crash detection—or white-box techniques, such as assertions inside the SUT [5]. Each input is composed of a fixed set of N fields, each of which takes on values in a finite set $\mathcal{X}$. Our rationale for assuming $\mathcal{X}$ is finite (even though the field may admit continuous values) comes from robustness testing, which makes the key observation that robustness vulnerabilities can often be found by probing with a small number of data-type-specific values. Supposing there are N fields, each input is a vector $\in \mathcal{X}^N$, and the oracle is a function $\mathcal{X}^N \to \{0, 1\}$, such that the output is equal to one iff. the input triggers a fault. Although it is straightforward to extend the methods below to heterogeneous inputs of the form $\Pi_{i=1}^N \mathcal{X}_i$, we assume homogeneous input types for notational simplicity. The assumption of determinism is straightforward to relax, but we omit details for the sake of brevity. We will denote by $\tau \subset \mathcal{X}^N$ the subset of fault-triggering inputs; we will sometimes refer to inputs $\in \tau$ as *positive examples* and inputs $\notin \tau$ as *negative examples*. When τ is assumed to be a product set, we will assume $\tau = \Pi_{f=1}^N \tau_f$ for some *component* sets $\tau_f \subset \mathcal{X}$. We will generally use subscripts to index elements of Cartesian products.

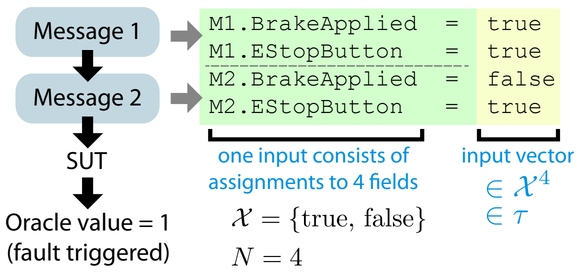


Fig. 2: Caricature illustrating basic assumptions and notation.

Figure 2 illustrates our notation for a trivial example. Here, we assume the SUT accepts a stream of network messages as input; specifically, we assume the SUT is a deterministic function of the last two messages sent to it over a network. Each of these messages consists of a set of assignments to a fixed set of variables or fields; in this case, these fields are named *BrakeApplied* and *EStopButton*. In order to convert this sequence of messages to a fixed-length input vector, we simply concatenate the field values. The figure indicates that the input vector {true, true, false, true} triggers a fault.

A. Product sets for fault diagnosis

A description of τ in the form of a product set is especially useful for fault diagnosis in that it provides an intuitive means for expressing which fields are relevant to triggering the fault, resulting in system failure. If τ is a product set, then any field f such that $\tau_f = \mathcal{X}$ is *irrelevant*, since the value of f for a particular input does not affect whether the fault is triggered. We therefore identify all the components τ_f, discard those such that $\tau_f = \mathcal{X}$, and communicate the rest to the developer. This allows the developer to quickly focus on the fields and values that are relevant to causing the failure, no matter how many irrelevant fields may present in an input. Taking this approach implies a model of independence that has proven to be useful in practice for the systems we have studied.

Figure 4 shows examples of product sets generated for a example failure in `SystemX`. The particular values included are from the dictionary of valid and exceptional inputs used for robustness testing. Normal robustness testing followed by delta debugging identifies a minimal length set of function calls that triggers a fault. Our algorithm then mutates the baseline set of inputs to determine which fields actually affect whether the fault is triggered. In this case, 10 relevant fields were identified in an input composed of 1176 fields spanning several function calls, along with the particular values necessary to trigger the fault. This information was sufficient to diagnose and fix the bug.

B. Identifying a product set with queries

The problem of identifying the product set is readily treated as an *active learning* problem [6], in which the learning algorithm has the ability to construct and label arbitrary examples using a labeling oracle. Minimizing the

```
Function GetFieldTriggers($\mathcal{F}$, $s$) =
    Input: a set of fields $\mathcal{F}$, a seed $s \in \tau$
    Retrieve all values $x$ such that the fault is triggered
    when we substitute $s_f = x, \forall f \in \mathcal{F}$:
    $X := \{x \in \mathcal{X} \mid s^{x \to \mathcal{F}} \in \tau\}$
    if $|\mathcal{F}| = 1$ or $X = \mathcal{X}$ then
        return $\{(f, X) \mid f \in \mathcal{F}\}$
    end
    else
        $\mathcal{F}_0$, $\mathcal{F}_1$ = EvenlySplitFields ($\mathcal{F}$)
        return GetFieldTriggers ($\mathcal{F}_0$, $s$)
        $\cup$ GetFieldTriggers ($\mathcal{F}_1$, $s$)
    end
```
Algorithm 1: Hierarchical Product Set Learning (HPSL)

number of oracle queries is very desirable in our case because evaluating the oracle involves running a complex system.

The algorithm is a search for fields and values that are relevant to triggering the system failure. Assuming only k fields are relevant and given a seed input $s \in \tau$, Algorithm 1 (Hierarchical Product Set Learning, or HPSL) identifies the product set using only $O(k|\mathcal{X}|\log N)$ samples. Given a set of fields $\mathcal{F}$ and the baseline seed set of test inputs s, GetFieldTriggers returns the elements of the component sets $\tau_f, \forall f \in \mathcal{F}$. If all fields in the input set $\mathcal{F}$ are irrelevant, it correctly returns $\mathcal{X}, \forall f \in \mathcal{F}$ after $|\mathcal{X}|$ oracle queries. Each query consists of setting s_f to some $x \in \mathcal{X}$, $\forall f \in \mathcal{F}$, denoted by $s^{x \to \mathcal{F}}$, and checking whether the resulting input triggers the fault (i.e., whether $s^{x \to \mathcal{F}} \in \tau$). If at least one input field is relevant, GetFieldTriggers detects this when $s^{x \to \mathcal{F}}$ fails to trigger the fault for some $x \in \mathcal{X}$. It then splits the input set in half, and recurses on each half. When the set of input fields is a singleton $\{f\}$, s_f is mutated while fixing all other fields in order to identify τ_f. A sample execution of the algorithm is depicted in Fig. 3.

C. Precision of approximate product sets

When τ is not a product set, Algorithm 1 can still be used to obtain a product set $\hat{\tau}$ that approximates τ. In this case, it is important to communicate to the developer that not all inputs in $\hat{\tau}$ will actually trigger the fault. We therefore estimate and report the classification *precision* along with a product set. Specifically, we define the precision of a set $\hat{\tau}$ to be the fraction of inputs in $\hat{\tau}$ that actually trigger the fault (i.e., $|\hat{\tau} \cap \tau|/|\hat{\tau}|$). A high precision estimate gives us confidence that our assumptions were valid, implying that the product set probably contains most of the information relevant to debugging the defect. The lower the precision, the more likely it is that the product set lacks some important bit of information: for example, the defect may actually depend on some field that was not identified as relevant. Figure 4 (discussed in detail later) shows an example of this phenomenon: the product set in Fig. 4a is missing two relevant fields, which manifests as reduced precision compared to a product set that contains the missing fields.

Precision is easily estimated by sampling the product set. We can also compute and report a confidence interval for the true precision. Given T samples and a confidence level of $1 - p_{\text{err}}$, the Hoeffding bound yields a confidence interval of size $\epsilon = \sqrt{-(\log p_{\text{err}})/2T}$.

By contrast, the classification *recall* is not as important for our purposes, as debugging generally favors the ability to reliably reproduce a bug over finding *most* ways to trigger the bug. Furthermore, once identified, a bug can be corrected, and testing re-run to see if there are any residual bugs that weren't identified by the previous testing session.

III. Discussion

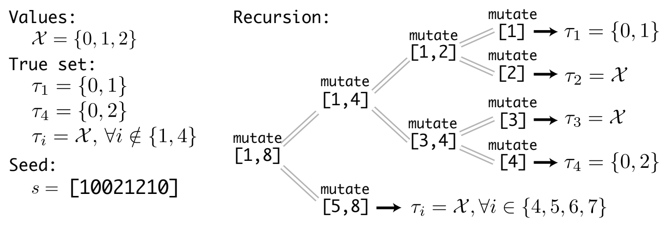
In this section, we justify our method for our particular application and describe its limitations.

1) Generative models for high-dimensional faults: Although we have proposed an essentially generative approach, it is natural to consider potential discriminative alternatives that are also capable of producing interpretable rules, such as decision trees. The discriminative approach is problematic for several reasons. First, there is no natural distribution over inputs with respect to which we might judge the performance of a discriminative classifier; e.g., training a classifier on a uniform distribution over inputs with only a very few system failures favors labeling no input faulty, but label-balancing the distribution is also difficult and unmotivated. Devising heuristics to locate additional positive examples in the vicinity of an identified system failure is a path that leads back to our proposed approach.

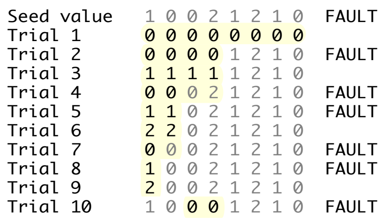
2) Simplicity: As previously mentioned, decision trees are an obvious alternative to our method, with the limitation that they are only practical when system failures are dense enough in the input space for random sampling to find a sufficient number of examples to use as training data. In our context, a decision tree can produce an arbitrary union of product sets, and is therefore more expressive. However, it can only be trained in a myopic way that is prone to high variability in results. We prefer low variability, and are willing to accept the cost of increased bias by adopting the model of a single product set.

3) Lack of tuning parameters: A key advantage of our method in practice is that it has no significant parameters to tune. The only degree of freedom in our algorithm is in the way the set of input fields is split upon recursion. In practice, we have observed that choosing different splits has had no effect on the results. By contrast, training a decision tree is dependent on many parameters that have major effects on the result, such as thresholds for various pruning heuristics.

4) Limitations of product set models: The ability of our method to assist in fault diagnosis is largely determined by how well the product set model fits the true structure of the inputs and faults of a system. For example, suppose that the inputs are interpreted by the SUT as the elements of a covariance matrix, and a fault is triggered when the matrix is not positive definite. Since the set of non-positive-definite matrices cannot be expressed as a product set in the matrix elements, our method would probably not be very useful in diagnosing such a fault beyond possibly ruling out parameters other than the matrix as fault triggers (which may


Values:
$\mathcal{X} = \{0,1,2\}$
True set:
$\tau_1 = \{0,1\}$
$\tau_4 = \{0,2\}$
$\tau_i = \mathcal{X}, \forall i \notin \{1,4\}$
Seed:
$s =$ [10021210]
Recursion:
mutate [1,8]
mutate [1,4]
mutate [1,2]
mutate [1] $\rightarrow \tau_1 = \{0,1\}$
mutate [2] $\rightarrow \tau_2 = \mathcal{X}$
mutate [3,4]
mutate [3] $\rightarrow \tau_3 = \mathcal{X}$
mutate [4] $\rightarrow \tau_4 = \{0,2\}$
mutate [5,8] $\rightarrow \tau_i = \mathcal{X}, \forall i \in \{4,5,6,7\}$

(a) Recursive structure of algorithm



Seed value	1	0	0	2	1	2	1	0	FAULT
Trial 1	0	0	0	0	0	0	0	0	
Trial 2	0	0	0	0	1	2	1	0	FAULT
Trial 3	1	1	1	1	1	2	1	0	
Trial 4	0	0	0	2	1	2	1	0	FAULT
Trial 5	1	1	0	2	1	2	1	0	FAULT
Trial 6	2	2	0	2	1	2	1	0	
Trial 7	0	0	0	2	1	2	1	0	FAULT
Trial 8	1	0	0	2	1	2	1	0	FAULT
Trial 9	2	0	0	2	1	2	1	0	
Trial 10	1	0	0	0	1	2	1	0	FAULT

(b) First 10 queries made by algorithm

Fig. 3: Execution of the algorithm for a sample input. Fig. 3a illustrates the recursive structure of the algorithm. Each node indicates the components of the seed mutated in that call of the algorithm. Leaf nodes return the subset of identified component sets corresponding to the mutated fields. Fig. 3b shows the actual query values chosen by the algorithm.

still be useful). This observation also applies to any case in which the fault-triggering-inputs constitute a submanifold of the input space. Another simple example of the limitation of using a single product set is the case where a fault is triggered if two fields are not equal to each other or have some other dependency. In this case, our method may fail to identify these fields as being relevant to triggering the fault. One could envision considering pairs or triples of parameters as potential candidates for chunking into tuples that are treated as single fields for the purposes of the algorithm, but that is beyond the scope of this paper. In general, our method is most useful when applied to interfaces which have a large number of relatively independent inputs.

IV. Analysis

First, we prove that Algorithm 1 correctly identifies the underlying product set, if the set of failure-inducing inputs τ is indeed a product set.

Theorem 1: Suppose that there exist sets $\tau_1, \ldots, \tau_N$, $\tau_f \subset \mathcal{X}, \forall f \in \{1, \ldots, N\}$, such that $\tau = \Pi_{f=1}^{N} \tau_f$. Then for all $s \in \tau$, $\mathcal{F} \subset \{1, \ldots, N\}$, $f \in \mathcal{F}$, and $X \subset \mathcal{X}$ such that $(f, X) \in$ `GetFieldTriggers`$(\mathcal{F}, s)$, $X = \tau_f$.

Proof: We proceed by induction on $|\mathcal{F}|$. When $|\mathcal{F}| = 1$,

$$\begin{aligned} X &= \{x \mid s^{x \to \mathcal{F}} \in \tau\} && (1) \\ &= \{x \mid s^{x \to \mathcal{F}} \in \Pi_{f=1}^{N} \tau_f\} && (2) \\ &= \{x \mid x \in \tau_f, s_g \in \tau_g, \forall g \in \{1, \ldots, N\} \setminus \{f\}\} && (3) \\ &= \{x \mid x \in \tau_f\} && (4) \\ &= \tau_f. && (5) \end{aligned}$$

Note that we used the assumption $s \in \tau$ to arrive at (4). We now assume the conclusion $\forall |\mathcal{F}| < n$ and prove it for $|\mathcal{F}| = n$. We consider two cases. If $X = \mathcal{X}$, then by the same reasoning as above, we have $\mathcal{X} = X = \{x \mid x \in \tau_f, \forall f \in \mathcal{F}\} \subset \tau_f, \forall f \in \mathcal{F}$. Since $\tau_f \subset \mathcal{X} \subset \tau_f$, we conclude $\mathcal{X} = \tau_f = X, \forall f \in \mathcal{F}$. Otherwise, if $X \neq \mathcal{X}$, the result trivially follows from the inductive hypothesis, since both $|\mathcal{F}_0|$ and $|\mathcal{F}_1| < n$. ■

We now give a bound on the sample complexity of `GetFieldTriggers` showing that it is particularly efficient when the number of relevant fields is much smaller than the total number of fields, which we have observed is a likely case for autonomous vehicle software.

Theorem 2: Suppose that τ is a product set with k relevant fields. Then `GetFieldTriggers`$(\{1, \ldots, N\}, \cdot)$ performs no more than than $2k(2 + \log_2 N)|\mathcal{X}|$ oracle queries.

Proof: The result is obtained by bounding the total number of recursive invocations of `GetFieldTriggers` and multiplying this by the number of queries directly performed by each invocation. First, note that each invocation performs exactly $|\mathcal{X}|$ oracle queries. The total number of recursive invocations can be bounded by the number of invocations at each recursion depth multiplied by the maximum recursion depth. Assuming $||\mathcal{F}_0| - |\mathcal{F}_1|| \leq 1$, the maximum recursion depth is bounded by $2 + \log_2 N$ (proof omitted).

Finally, we show that the number of invocations at each recursion depth is bounded by $2k$. This is trivially true at depth 0. Now consider an arbitrary depth $n > 0$. The number of invocations at this depth is equal to twice the number of invocations at depth $n - 1$ that resulted in recursion. Of all the invocations at depth $n - 1$, only those called with input sets $\mathcal{F}$ containing relevant fields result in recursion, since the proof of Theorem 1 implies that the recursion terminates otherwise (this is the case $X = \mathcal{X}$). Therefore, since no more than k relevant fields exist, no more than k invocations at depth $n - 1$ contain relevant fields, and no more than k invocations at depth $n - 1$ result in recursion, producing no more than $2k$ invocations at depth n. ■

V. Related work

As previously discussed, our work assumes that some initial exploration of the test input space has been done, and thus is a way to augment rather than replace techniques such as combinational testing [5], [3], [7]. Most comparable work on automated fault diagnosis and debugging is discriminative in nature: fault causes are associated with attributes that differ between faulty conditions and normal conditions. Works in this vein include statistical debugging [8], [9], [10], [11] and spectrum-based fault localization [12], [13]. We have already argued against such a discriminative approach for our application in the Discussion section. Furthermore, these methods are designed for a slightly different scenario, in which fault causes are identified in terms of features of heavily instrumented code, rather than exceptional inputs. The aforementioned delta debugging algorithm has also been used previously in order to identify relevant differences between faulty and normal conditions [14].

From an abstract, algorithmic perspective, our method can be thought of as a generalization of the delta debugging method. Delta debugging might be interpreted as identifying a binary product set, whereas we identify a general product set. The algorithms therefore also share the same general structure. However, the generalization to arbitrary product sets allows us to make what we consider a crucial leap from discriminative to generative models of fault triggers.

The product-set-learning algorithm presented here was inspired by the classic algorithm for PAC-learning rectangles in Euclidean space [15]. We are not directly aware of previous work on the problem addressed here of efficiently learning product sets via membership queries when few attributes are relevant and an initial seed is given. However, similar problems have been studied many times in computational learning theory. Common variations consider equivalence queries in addition to membership queries, unions of discretized boxes in Euclidean space, and online problems [16], [17], [18].

VI. Experiments

We implemented HPSL and evaluated its effectiveness in diagnosing the causes of actual bugs found in SystemX. For comparison, we also implemented decision trees based on the C4.5 algorithm [19], with features corresponding to field values. The set of inputs positively labeled by such a decision tree is equivalent to a union of product sets. In order to generate sufficient positive training examples for the decision tree, we mutated the same minimal length test pattern used in our method, randomly choosing 10% of the fields to mutate for each example. Two different pruning techniques were evaluated. The first consisted of pruning the tree at a depth of 10 in addition to pruning branches matching 10 or fewer examples in each class. We also implemented a pruning technique similar to reduced error pruning [20], but altered to improve precision instead of reducing error on a validation set; we will refer to this as *improved precision pruning* or IPP. We trained many decision trees on varying sets of 5000 examples each; for IPP, the input set of 5000 examples was evenly split into a training set and a validation set. For IPP, the trees were pre-pruned to depth 30.

Figure 4 shows the results obtained for a specific bug, which we will refer to as bug 116. The results generated by HPSL were sufficient to diagnose the fault, which was subsequently fixed in SystemX. The fault associated with bug 116 is triggered when the vehicle fails to execute a safety stop despite its speed exceeding a predefined threshold. Robustness testing was used to find an input triggering this fault, which was then generalized via our method and via decision trees. As shown in Table I, the inputs to the system in this case constituted a 1176-dimensional vector of fields of varying types. HPSL terminated after querying the system 3459 times, while we trained several decision trees on 5000 training examples generated via the method described above (varying the training set for each tree). Fig. 4a shows a typical decision tree converted to an equivalent product set (irrelevant fields are omitted). The result of HPSL was the same, except for two additional rules shown in Fig. 4b.

```
M1.BrakeApplied in set [ true ]
M1.CurrentGear in set [ 0 ]
M1.IsStopped in set [ 0 ]
M1.MotorOn in set [ false ]
M1.SystemConnected in set [ true ]
M1.EStopButton in set [ false ]
M2.ActualSpeed in set [ -1.79769e+308,
  -2.71828, ...,  -65535, -inf ]
M2.EStopButton in set [ false ]
```

(a) Decision tree output expressed as a product set

```
M0.InitialMode in set [ 1, ..., 4294967295 ]
M2.SystemConnected in set [ true ]
```

(b) Additional rules found by HPSL

Fig. 4: Comparison between output generated by decision trees and HPSL for bug 116. The set of rules found by HPSL was equal to the set found by decision trees, augmented by two additional rules.

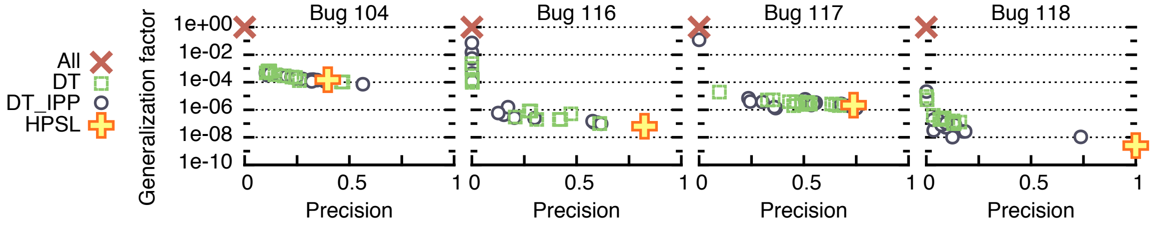
Bug id.	N	HPSL m	DT m
104	179	2035	5000
116	1176	3459	5000
117	1706	4506	5000
118	1176	3663	5000

TABLE I: Parameters for experiments of Fig. 5: N denotes the number of fields in an input and m denotes the number of training examples or queries used by each method.

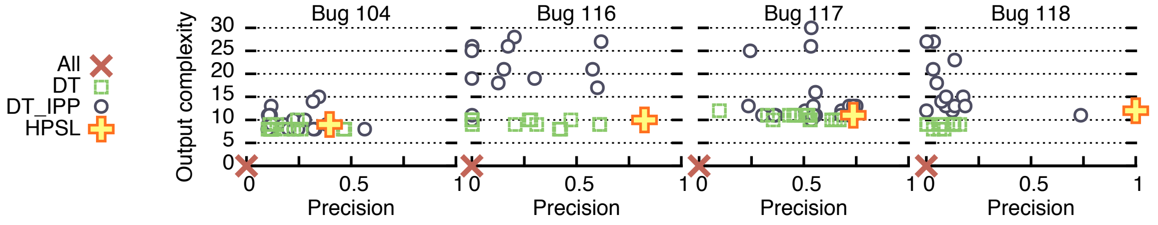
The results make it clear that the bug is triggered when the ActualSpeed input is negative: ultimately, the cause of the bug was traced to failing to take the absolute value of this field in the vehicle's speed limit check. The other identified fields are necessary to put the software into a state such that an emergency stop may need to be issued. We note that the quality of the decision tree results was highly dependent on the pruning strategy employed and the particular training set.

Quantitative results evaluating our method on other faults are shown in Fig. 5. We compared the two methods via three statistics: the fraction of all possible inputs labeled positive by the algorithm (converting decision trees to unions of product sets), the number of fields named in the algorithm's output, and the classification precision. The first statistic quantifies what we will refer to as the *generalization factor*; returning the seed input alone corresponds to a score very close to 0, whereas returning the set of all possible inputs achieves a score of 1. Although a high generalization factor is desirable, it must be accompanied by high precision to be useful. The number of fields named in the output is a rudimentary measure of complexity; the more complex the output, the less interpretable it is.

The results show that neither pruning heuristic reliably produced decision trees that were competitive with HPSL. The decision trees varied significantly as we varied the training set, whereas HPSL produces deterministic results given a seed and a deterministic strategy for splitting the input fields (i.e., EvenlySplitFields). The generalization factor vs. precision plots show that HPSL achieved a superior trade-off of generalization and precision in the following sense: for any objective that is a positive linear combination of



(a) Generalization factor (fraction of inputs labeled positive) vs. precision



(b) Output complexity (number of fields named in output) vs. precision

Fig. 5: Plots showing measures of generalization factor and output complexity vs. classification precision for different methods. "HPSL" refers to our method, "DT" refers to decision trees pruned to depth 10, "DT_IPP" refers to decision trees pruned to improve precision on a validation set, and "All" corresponds to the trivial method of reporting all inputs as causing the fault. Each plotted point for DT corresponds to a different random training set. See text for details.

(log) generalization factor and precision, the decision tree result almost never optimized the objective. The same can be said for the complexity-precision tradeoff. This follows from simple linear programming arguments.

VII. Conclusions

We have presented a method that efficiently learns interpretable product set representations of fault triggers in high-dimensional software interfaces. The algorithm actively queries the SUT, requiring a number of queries that scales only logarithmically in the number of irrelevant fields. The method is suitable for diagnosing faults that are triggered by corner cases and rare situations in which combinatorial testing techniques are likely to find one or only a few system failures; such cases prove challenging for naive approaches due to the difficulty of obtaining sufficient informative training examples. Our experimental results have validated the method's ability to provide useful debugging information for commercial autonomous vehicle systems. Compared to a naive approach based on decision trees, our method provides results that are generally better, without any need for parameter tuning or pruning heuristics.

In future work, we would like to explore ways to make our results more informative when the product set model is not strictly accurate. In particular, we would like to automatically identify fields that exhibit interdependence, modifying the algorithm to jointly sample these fields. This or similar heuristics may be necessary in order to diagnose more subtle bugs that may appear in the wild.

References

[1] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek, "Automated robustness testing of off-the-shelf software components," in *FTCS*. IEEE, 1998.

[2] A. Zeller, "Yesterday, my program worked. today, it does not. why?" in *Software EngineeringESEC/FSE99*. Springer, 1999, pp. 253–267.

[3] J. Pan, P. Koopman, and D. Siewiorek, "A dimensionality model approach to testing and improving software robustness," in *AUTOTESTCON*. IEEE, 1999, pp. 493–501.

[4] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

[5] D. R. Kuhn, R. N. Kacker, and Y. Lei, "Practical combinatorial testing," NIST, Tech. Rep. 800-142, October 2010.

[6] B. Settles, "Active learning literature survey," University of Wisconsin-Madison, Tech. Rep. 1648, 2010.

[7] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating fuctional tests," *CACM*, vol. 31, no. 6, pp. 676–686, 1988.

[8] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," *ACM SIGPLAN Notices*, vol. 38, no. 5, pp. 141–154, 2003.

[9] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: simultaneous identification of multiple bugs," in *ICML*. ACM, 2006, pp. 1105–1112.

[10] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 10, pp. 831–848, 2006.

[11] A. X. Zheng, M. I. Jordan, B. Liblit, and A. Aiken, "Statistical debugging of sampled programs," in *NIPS*, 2003.

[12] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Automated Software Engineering*. IEEE, 2003, pp. 30–39.

[13] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *TAICPART*, 2007, pp. 89–98.

[14] A. Zeller, "Isolating cause-effect chains from computer programs," in *SIGSOFT Symp. on Foundations of Sw. Eng.* ACM, 2002, pp. 1–10.

[15] M. J. Kearns and U. V. Vazirani, *An introduction to computational learning theory*. MIT press, 1994.

[16] P. W. Goldberg, S. A. Goldman, and H. D. Mathias, "Learning unions of boxes with membership and equivalence queries," in *COLT*, 1994.

[17] N. H. Bshouty, P. W. Goldberg, S. A. Goldman, and H. D. Mathias, "Exact learning of discretized geometric concepts," *SIAM Journal on Computing*, vol. 28, no. 2, pp. 674–699, 1998.

[18] W. Maass and G. Turán, "Algorithms and lower bounds for on-line learning of geometrical concepts," *Machine Learning*, vol. 14, no. 3, pp. 251–269, 1994.

[19] J. R. Quinlan, *C4.5: programs for machine learning*. Morgan kaufmann, 1993, vol. 1.

[20] T. Elomaa and M. Kaariainen, "An analysis of reduced error pruning," *JAIR*, vol. 15, pp. 163–187, 2001.