Classifying Bugs with Interpolants

Andreas Podelski[1], Martin Schäf[2], and Thomas Wies[3]

[1] University of Freiburg
[2] SRI International
[3] New York University

Abstract. We present an approach to the classification of error messages in the context of static checking in the style of ESC/Java. The idea is to compute a semantics-based signature for each error message and then group together error messages with the same signature. The approach aims at exploiting modern verification techniques based on, e.g., Craig interpolation in order to generate small but significant signatures. We have implemented the approach and applied it to three benchmark sets (from Apache Ant, Apache Cassandra, and our own tool). Our experiments indicate an interesting practical potential. More than half of the considered error messages (for procedures with more than just one error message) can be grouped together with another error message.

1 Introduction

The classification of error messages, bug reports, exception warnings, etc. is an active research topic [1, 3, 5, 13, 16, 25, 30]. The underlying motivation is that grouping related error messages together will help with their analysis. The problem of classification is to infer what error messages are related (and, in what sense).

In this paper, we address the problem of classification in the context of static checking of sequential procedural programs in the style of ESC/Java, as in [4, 15, 21]. Although in this context error messages may refer to an error in the specification rather than the code, the same motivation applies. The error messages may come in batches of, say, thousands, and they have to be analyzed, if only to debug the specification.

In the context of static checking, it seems natural to explore whether concepts and techniques from semantics and verification can be put to use for the classification of error messages.

In this paper, we present an approach to *semantics-based* classification of error messages which come in the form of a sequence of statements along with a witness; a witness here is an initial state from which the execution of the sequence of statements leads to the violation of a specified assertion. As in verification, semantics here is used to abstract away from syntactical details. For example, we can abstract a statement (or a sequence of statements) by its *summary* in the form of a pre- and postcondition pair.

The idea behind the approach is to compute a semantics-based *signature* for each of the error messages and then group together error messages with

the same signature. More concretely, we associate each error message with a new verification problem. We apply a verification engine to infer a proof in the form of Hoare triples. We remove the *invariant-type* Hoare triples (of the form $\{F\}$ st $\{F\}$, expressing that an assertion F is invariant under a statement st). We take the remaining *change-type* Hoare triples to construct the signature.

Intuitively, *the larger the number of invariant-type Hoare triples* (and the smaller the number of change-type Hoare triples), *the more error messages will be grouped together* under the resulting signature. The approach exploits the fact that modern verification engines can often be geared to produce proofs with a large number of invariant-type Hoare triples. (We here think of Craig interpolation, constraint solving, and static analysis [9, 24, 29].)

We have implemented the new approach to classification on top of our own extended static checker for Java. We have applied it to three benchmark sets (from Apache Ant, Apache Cassandra, and our own tool). Our experiments indicate an interesting practical potential of the approach. More than half of the considered error messages (for procedures with more than just one error message) can be grouped together with another error message.

The technical contribution of this paper is to introduce the approach and to present an experimental evaluation of its implementation. The conceptual contribution is the formal foundation of the approach which associates each error message with a verification problem and constructs a *small but significant* signature from a correctness proof.

Roadmap. The next section illustrates the approach on an example. Section 3 fixes the notation and terminology of standard concepts. Section 4 introduces the approach together with its formal foundation. Section 5 presents the experimental evaluation and Section 6 discusses the related work.

2 Overview

We motivate our approach to classifying bugs using interpolation with the illustrative example in Figure 1. For simplicity of exposition, the example is constructed to be of reasonable size. However, real Java programs such as the ones used in our experiments show similar patterns in larger methods.

Figure 1 shows a method `m` that takes two objects `a` and `b` of type `A`, and one integer `x` as input. We analyze this method with a static checker such as ESC/Java [15] to obtain *error messages* that indicate uncaught exceptions[4]. In this paper, we consider an error message to consist of a specific initial state and an error trace whose execution from the initial state leads to a state that violates an assertion guarding an uncaught run-time exception.

[4] The fact that we use a static checker is not crucial for our discussion. The error messages could also be generated using a bounded model checker such as [14, 20], or a testing tool such as Randoop [27]

```
1  void m(A a, A b, int x) {
2    if (x>0) {
3      A obj = null;
4      try {
5        obj = b.clone();
6      } catch (Exception e) {
7        e.printStackTrace();
8      }
9      obj.bar();
10     a.bar();
11   }
12   a.bar();
13 }
```

Fig. 1. Example procedure `m`.

For the method `m`, the error messages that are produced by the static checker can be classified according to the line in the method where the run-time error occurs as follows:

1. If `x` ≤ 0, and `a` is `null` the execution of `m` leads to a `NullPointerException` on line 12.
2. If `x` > 0 and `a==null`, a `NullPointerException` is thrown on line 10.
3. If `x` > 0 and `b==null`, a `NullPointerException` is thrown on line 9

Figure 2 shows an example of an error message for each of these three types. Each error message starts from a given initial state, which is followed by the sequence of statements executed on the corresponding error traces, and ends in the (implicit) assertion that is violated when starting execution from the initial state. For convenience, the initial states of the error messages are described symbolically by an assume statement at the beginning of each trace. Note that if we used a random testing tool such as Randoop instead of a static checker, then several error messages of each type may be reported. For example, a testing tool might generate multiple test cases that invoke `m` with `a==null` and different values for `x` that satisfy `x` > 0.

Grouping error messages *syntactically* based on the line in the program where a run-time error occurs may seem appropriate at first. However, this strategy does not yield a meaningful classification of bugs on its own. Two error messages that fail at the same program location may do so for different reasons and should therefore not be grouped together. Conversely, two error messages that fail at different program locations may do so for the same reason and should therefore be grouped together. Specifically, in our example the error messages of type 1 and 2 capture cases in which an error occurs because `m` has been called with the value `null` passed to parameter `a`. That is, the parameter `a` will be dereferenced with a `NullPointerException` regardless of the value of `x`. It therefore seems more appropriate to take into account only the error-relevant condition `a==null` that

```
assume(a == null
    && b == null
    && x == 0);
assume(x <= 0);
assert(a!=null);
```

Type 1

```
assume(a == null
    && b == A@15ce
    && x == 1);
assume(x > 0);
A obj = null;
obj = b.clone();
obj.bar();
assert(a!=null);
```

Type 2

```
assume(a == null
    && b == null
    && x == 1);
assume(x > 0);
A obj = null;
e.printStackTrace();
assert(obj!=null);
```

Type 3

Fig. 2. Different syntactic types of error messages for the method `m` in Figure 1

is common to the error messages of type 1 and 2 and group them together during classification. On the other hand, type 3 error messages should still be grouped separately. Our approach aims to infer such a semantics-based characterization of what is essential for the reason why the assertion in an error message fails.

The approach groups error messages by computing an *error signature* for each individual error message as follows. First, we replace the failing assert statement at the end of the error message by an assume statement with the same condition. The resulting trace will not have any feasible execution because the final condition is always violated. That is, if τ is the trace resulting from this transformation, then $\{\top\}\ \tau\ \{\bot\}$ is a valid Hoare triple, where $\top$ stands for the assertion *true* and $\bot$ for *false*. We can thus use an interpolating theorem prover to generate a Hoare proof for the validity of this triple. For instance, the generated Hoare proof for the trace obtained from the error message of type 2 may look as follows:

```
{⊤}
  assume(a == null && b == A@15ce && x == 1);
{a == null}
  assume(x > 0);
{a == null}
  obj = null;
{a == null}
  obj = b.clone();
{a == null}
  obj.bar();
{a == null}
  assert(a != null);
{⊥}
```

Observe that the intermediate assertion `a == null` is maintained throughout the trace after initialization. It captures the reason why the trace described by the original error message fails.

The next step is to extract the sequence of intermediate assertions from the Hoare proof and replace all consecutive occurrences of the same assertion by

just one copy of that assertion. We refer to the resulting condensed sequence of intermediate assertions as the error signature of the original error message. For our error messages of type 2, the error signature computed from the given Hoare proof only consists of the assertion `a == null`.

We group error messages that have identical error signatures together into equivalence classes. We refer to these equivalence classes as *buckets*. For example, the error signature for the error messages of type 1 is also `a == null`. Hence, all type 1 and type 2 error messages will be grouped together in the same bucket. On the other hand, the error signature of the type 3 error messages consists of the two assertions $\top$, `obj == null`. Type 3 error messages will therefore end up in a separate bucket.

In general, an error signature consists of a non-trivial sequence of assertions that captures how the error condition is propagated through the trace of the error message. Intuitively, an error signature abstracts away from the specific values of the initial state of an error message and the syntax of the statements in its error trace, including the specific location of the failing assertion. Error signatures only maintain the error-relevant semantic conditions that hold along the trace of the error message. For example, an error message has the error signature $\top$, `obj == null` if its initial state satisfies $\top$ (i.e., the initial state can be arbitrary). Moreover, its error trace contains one statement that establishes the postcondition `obj == null` from a state that satisfies $\top$, and if it ends with an assert statement whose execution fails if `obj == null` holds; it can contain an arbitrary number of additional statements as long as they leave the corresponding assertion (which is $\top$ or `obj == null` according to the position within the error trace) invariant. That is, an error message with the error signature $\top$, `obj ==null` consists of:

- an initial state that satisfies $\top$, which is the case for any initial state.
- a (possibly empty) sequence of statements *st* for which $\top$ is invariant, which is the case for every statement *st* (the Hoare triple $\{\top\}$ *st* $\{\top\}$ holds trivially),
- a statement *st* that establishes the postcondition `obj == null` (i.e., the Hoare triple $\{\top\}$ *st* {`obj == null`} holds),
- a (possibly empty) sequence of statements *st* for which the assertion `obj == null` is invariant (i.e., the Hoare triple {`obj == null`} *st* {`obj == null`} holds), and finally
- an statement `assert(`F`)` that fails when executed in a state where the assertion `obj == null` is true (i.e., the Hoare triple {`obj == null`} `assume`(F) $\{\bot\}$ holds).

Note that the first assertion of an error signature can generalize the initial state of an error message. This is needed in order to group together error messages with different initial states.

We have found that error signatures provide a useful classification mechanism in the context of static checking of Java programs if the classification is restricted to the error messages that belong to the same method, i.e., if it is combined with a coarse syntactic classification mechanism based on method affiliation.

3 Preliminaries

The purpose of this section is to fix the notation and terminology for existing, standard concepts.

We assume a simple imperative language whose basic statements st consist of assignments as well as assume and assert statements:

$$\begin{array}{ll} x \in \mathcal{X} & \text{program variables} \\ e \in \mathcal{E} & \text{expressions} \\ F \in \mathcal{F} & \text{formulas} \\ st ::= \texttt{assume}(F) \mid \texttt{assert}(F) \mid x := e & \text{(basic) statements} \end{array}$$

We do not define the syntax of expressions $e \in \mathcal{E}$ and formulas $F \in \mathcal{F}$. We only require that they fall into quantifier-free first-order logic for a signature that is defined by a suitable theory $\mathcal{T}$ (e.g., linear integer arithmetic). Moreover, we require that the variables appearing in e and F are drawn from the set $\mathcal{X}$. We assume standard syntax and semantics of first-order logic and use $\top$ and $\bot$ to denote the Boolean constants for *true* and *false*, respectively.

A *state* $s = (M, \beta)$ consists of a model M of the theory $\mathcal{T}$ and an assignment β of the variables in $\mathcal{X}$ to values drawn from the universe of M. The model M may be fixed for all states if M is the *canonical model* of the theory $\mathcal{T}$ (e.g., the integer numbers in the case of linear integer arithmetic). For an expression e, we denote by $s(e)$ the value obtained by interpreting e in s and we use similar notation for formulas. We write $s \models F$ to say that s satisfies F, i.e., $s(F) = \top$. A formula is valid if $s \models F$ for all states s and it is called unsatisfiable if $\neg F$ is valid.

Following the presentation in [11, 26], we define the semantics of statements using the weakest precondition transformer $\texttt{wp}$, which maps a pair of a statement st and a formula F to another formula:

$$\begin{aligned} \texttt{wp}(\texttt{assume}(G), F) &= G \Rightarrow F \\ \texttt{wp}(\texttt{assert}(G), F) &= G \wedge F \\ \texttt{wp}(x := e, F) &= F[e/x] \end{aligned}$$

The Hoare triple $\{F\}\ st\ \{F'\}$ stands for the formula $F \Rightarrow \texttt{wp}(st, F')$.

For example, the Hoare triple $\{x = 0\}\ \texttt{assume}(x \neq 0)\ \{\bot\}$ is valid, whereas the Hoare triple $\{x = 0\}\ \texttt{assert}(x \neq 0)\ \{\bot\}$ is not valid.

A *trace* τ is a finite sequence of basic statements $\tau = st_1; \ldots; st_n$. We extend both $\texttt{wp}$ and Hoare triples from statements st to traces τ in the expected way.

A sequence of formulas and statements $F_1, st_1, F_2, \ldots, st_n, F_{n+1}$ is called a *Hoare sequence* if for all $i \in [1, n]$, $\{F_i\}\ st_i\ \{F_{i+1}\}$ is valid. Intuitively, the Hoare sequence corresponds to an annotation or proof outline to prove the Hoare triple $\{F_1\}\ \tau\ \{F_{n+1}\}$ for the trace $\tau = st_1, \ldots, st_n$.

A trace τ is called *infeasible* if $\{\top\}\ \tau\ \{\bot\}$ is valid. Intuitively, the execution of the sequence of statements of an infeasible trace always (i.e., for every starting state) blocks on some assume statement in the sequence.

4 Classifying Error Traces through Error Signatures

The purpose of this section is to introduce the formal foundation for an approach to the classification of error traces based on concepts and techniques from verification.

Definition 1. *A trace τ is called* error trace *if $\neg(\{\top\}\ \tau\ \{\top\})$ is satisfiable. An* error message *is a pair $\epsilon = (\tau, s_0)$ of an error trace τ and a state s_0 such that $s_0 \models \neg(\{\top\}\ \tau\ \{\top\})$.*

Intuitively, an error trace has at least one execution that violates an assert statement in the trace. Every state s_0 such that $s_0 \models \neg(\{\top\}\ \tau\ \{\top\})$ is the initial state s_0 of such a faulty execution of τ. The initial state s_0 may be obtained from the satisfiable formula $\neg(\{\top\}\ \tau\ \{\top\})$ by using a model-generating theorem prover. It could also be obtained directly (together with τ) from a failed test or a bug report.

An error trace has at least one execution that does not block on any assume statement (namely, the faulty execution). This fact may help to avoid the confusion with the terminology of error trace in software model checking as in [17].

The definition implies that an error trace must contain at least one assert statement. To simplify the discussion, we will restrict ourselves to error traces that contain exactly one assert statement and assume that this statement is the last statement in the trace.

4.1 From Error Messages to Proofs

The notion of error trace is not directly amenable to the use of verification technology and to the concept of proof. Recall that an error trace has some execution that violates an assert statement, which also means that it may still have normally terminating executions. The notion of an error trace is thus incompatible with the notion of an infeasible trace. We know that the infeasiblity of a trace can be tied to a proof. In order to make the connection from error traces to proofs, we transform error traces to infeasible traces. Intuitively, the transformation of an error trace eliminates all normally terminating executions from them. Given an error message (τ, s_0), the first step of the transformation is to encode the given initial state s_0 of a faulty execution of τ into an assume statement that is prepended to τ. The resulting trace is still feasible.[5] In fact, the trace has exactly one execution. The execution must start in the state s_0 (otherwise the newly added assume statement would immediately block the execution). The execution fails the assert statement in the trace. The second step of the transformation is to replace the assert statement in the trace is by an assume statement. The resulting trace is infeasible.

Notation $\overline{\tau}$: For a trace τ, we denote by $\overline{\tau}$ the trace obtained from τ by replacing every assert statement of the form $\texttt{assert}(F)$ in τ by $\texttt{assume}(F)$.

[5] Note that we use weakest preconditions, as opposed to weakest liberal preconditions; see Section 4. For example, the trace $\texttt{assume}(x = 0); \texttt{assert}(x \neq 0)$ is not infeasible since $\texttt{wp}(\texttt{assume}(x = 0); \texttt{assert}(x \neq 0), \bot) \ = \ (x = 0 \Rightarrow (x \neq 0 \wedge \bot)) \ = \ (x \neq 0)$.

Definition 2 (Infeasible Extension of Error Messages). *Let $\epsilon = (\tau, s_0)$ be an error message and let $\{x_1, \ldots, x_k\}$ be the (finite) set of variables occuring in the statements of τ. Let further $e_1, \ldots, e_k$ be expressions that define the values of $x_1, \ldots, x_k$ in the state s_0.*[6] *Then the trace τ' of the form*

$$\tau' = \texttt{assume}(x_1 = e_1 \wedge \cdots \wedge x_k = e_k);\, \overline{\tau}$$

is the infeasible extension *of the error message ϵ.*

Remark 3 (Infeasibility of infeasible extension of error message). If ϵ is an error message and τ' its infeasible extension, then τ' is infeasible.

Note that, in the formal setting as introduced in Section 3, all three kinds of statements are deterministic. In the presence of a non-deterministic statement such as `havoc(x)`, we would need to add an assume statement to encode the non-deterministically chosen value for `x` in the faulty execution of an error trace τ. Definition 2 would accommodate this in the setting where each non-deterministic assignment statement in a trace is of the form $x := x^{(i)}$ with each $x^{(i)}$ a fresh renaming of x.

4.2 Error Signatures

Let ϵ be an error message and τ its infeasible extension. An error signature σ for ϵ is a sequence of formulas that can be extended to form a Hoare sequence with τ by allowing each formula in σ to be repeated for some (possibly empty) subtrace of τ. That is, each formula in σ is invariant for some subtrace of τ and each consecutive pair of formulas in σ is inductive for some statement in τ that connects the respective invariant subtraces. The intuition behind this definition is that the error signature abstracts the irrelevant statements in the trace (those contained in the invariant subtraces) while keeping the statements that are relevant for understanding the error (those connecting the invariant subtraces). The following definition makes this notion formally precise.

Definition 4 (Error Signatures). *Let $\tau = st_1; \ldots; st_n$ be an infeasible extension of an error message ϵ. A sequence of formulas $\sigma = F_1, \ldots, F_{m-1}$ with $m \leq n$ is an* error signature *of ϵ if there exists a strictly monotone function $h : [1, m] \to [1, n]$ such that:*

- *the sequence*

$$\top, st_{h(1)}, F_1, st_{h(2)}, F_2, \ldots, st_{h(m-1)}, F_{m-1} st_{h(m)}, \bot$$

  is a Hoare sequence,

[6] In the general case, we may not be able to describe s_0 using simple equalities and instead must consider its *diagram* [6]. For the sake of the clarity of presentation, we skim over these technicalities.

– *every F_i is invariant on the subtrace from $st_{h(i)}$ to the last statement before $st_{h(i+1)}$, i.e., for every $i \in [1, m-1]$, the sequence*

$$F_i, st_{h(i)}, F_i, st_{h(i)+1}, F_i, \ldots, st_{h(i+1)-1}, F_i$$

is a Hoare sequence.

We call the trace $st_{h(1)}, \ldots, st_{h(m)}$ the abstract slice *of ϵ induced by σ and h.*

Remark 5. Let $\epsilon = (\tau, s_0)$ be an error message and σ an error signature of ϵ. Then the formulas in σ are all different from $\bot$. This means that σ corresponds to a proof that the execution of τ that starts in s_0 is non-blocking and fails the final assert statement in τ.

Note that error signatures always exist. In particular, for the infeasible extension $\tau = st_1; \ldots; st_n$ of an error message ϵ, the sequence of formulas

$$\sigma = \mathsf{wp}(st_2; \ldots; st_n, \bot), \ldots, \mathsf{wp}(st_n, \bot)$$

is an error signature of ϵ. Evidently this error signature is not very informative, as the abstract slice of ϵ induced by σ is identical to τ. We will discuss below how to compute error signatures that yield proper abstract slices.

4.3 Classifying Error Messages

In the following, let sig be a function that maps error messages to error signatures. Then sig defines an equivalence relation $=_{\mathsf{sig}}$ on error messages. Two error messages ϵ_1 and ϵ_2 are equivalent with respect to sig if sig maps them to the same error signature:

$$\epsilon_1 =_{\mathsf{sig}} \epsilon_2 \;\Leftrightarrow\; \mathsf{sig}(\epsilon_1) = \mathsf{sig}(\epsilon_2).$$

Definition 6 (Buckets). *Given a set of error message E and a function* sig *mapping the elements of E to error signatures, we refer to the equivalence classes in the quotient $E/ =_{\mathsf{sig}}$ as* buckets.

We now have everything in place to give the classification algorithm, which is shown in Algorithm 1. The algorithm takes as input a set E of error messages. The output of the algorithm is the map *Buckets* whose domain is a set of error signatures (such that every error message in E is covered by some error signature in the domain). Each error signature σ in the domain is mapped to the corresponding *bucket*, i.e., a set of error messages which all have the error signature σ.

The algorithm computes a function sig mapping error messages to error signatures, as follows. For every error message $\epsilon = (\tau, s_0)$ in E, we first compute its infeasible extension τ' using the helper function `InfeasibleExtension`. Suppose τ' is of length n. Then we compute the formulas $F_0, \ldots, F_{n+1}$ for a Hoare sequence of τ' by applying an interpolating theorem prover to the path formula

Algorithm 1: Classification of error messages.

Input: E: set of error messages
Output: *Buckets*: map from error signatures to buckets of error messages from E

```
1  begin
2      for ϵ ∈ E do
3          (τ, s_0) ← ϵ ;
4          τ' ← InfeasibleExtension(τ, s_0) ;
5          F_0, ..., F_{n+1} ← Interpolate(τ') ;
6          // remove successive duplicates in F_1, ..., F_n ;
7          curr ← 1 ;
8          σ ← F_1 ;
9          for i from 1 to n do
10             if F_i ≠ F_curr then
11                 σ ← σ, F_i ;
12                 curr ← i ;
13             end if
14         end for
15         if σ ∉ dom(Buckets) then
16             Buckets[σ] ← ∅;
17         end if
18         Buckets[σ] ← Buckets[σ] ∪ {ϵ};
19     end for
20 end
```

constructed from τ'. This step is implemented by the function `Interpolate`. Note that the resulting interpolant sequence always satisfies $F_0 = \top$ and $F_{n+1} = \bot$. Moreover, the subsequence $F_1, \ldots, F_n$ is guaranteed to be an error signature for ϵ. However, it is not yet abstracting any statements in τ'. To obtain a proper error signature, we exploit the observation that interpolating theorem provers often produce interpolant sequences that consecutively repeat the same interpolant. Thus, we simply iterate over the formulas $F_1, \ldots, F_n$ and remove consecutive duplicates of formulas F_i to obtain the actual error signature σ for ϵ. The obtained error signature is then used to insert the current error message into its bucket.

5 Evaluation

Our approach to categorize error messages is embodied in a tool called Bucketeer. More precisely, the tool implements Algorithm 1 where the helper procedure `Interpolate` is implemented using the interpolation procedure of Princess [28]. The tool is available online, together with the benchmarks discussed in this paper.[7]

We have implemented the tool on top of a (prototype of a) static checker for Java [2]. The static checker is similar to OpenJML [8]. It uses Princess [28] to test

[7] `http://www.csl.sri.com/~schaef/experiments.zip`

Benchmarks	Ant	Cassandra	Bucketeer
Lines of code	271k	299k	15k
# of methods	7847	9373	331
Time for static checking (min)	87.35	55.65	6.00
# of methods with error traces	2470	2190	203
# of methods with multiple error traces	820	937	102
Sum of error traces in methods with multiple traces	2715	3243	376
Time for categorization (min)	25.23	54.78	6.06
Number of Buckets	1595	2041	258

Table 1. Raw data of the experimental evaluation.

an SMT formula for satisfiability and to compute a model if possible. It checks for null pointer dereferences, out-of-bound access to arrays, and division by zero errors. The checks are realized by inserting assertions into the code. Assertion violations are detected by translating the transition relation of each method into an SMT formula; a model of the SMT formula corresponds to an execution that violates an assertion (during the construction of the SMT formula, the checker unwinds loops (twice), and it replaces method calls by the specified (possibly trivial) contracts). The corresponding error message, i.e., the error trace for the failing execution together with the computed model, is fed to Bucketeer. For performance reasons, the static checker ensures that no two error messages share the same sequence of statements in the error trace (otherwise, we might find an infinite number of error messages). Bucketeer, however, does not require that error messages exercise different paths. Bucketeer categories error messages which belong to the same method (which makes sense only if more than one error message belongs to the method).

For the test of equality between formulas used in Line 10 of Algorithm 1 ("$F_i \neq F_{curr}$"), we use syntactic equality. In our experiments, using the more costly test of logical equivalence instead of syntactic equality does not change the outcome of the tests. The reason lies in the fact that the generation of the formulas by the interpolation procedure is optimized towards using the same formula whenever possible. This means in particular that the generation of a syntactically new but logically equivalent formula is unlikely.

Experimental Setup. To evaluate our approach we conducted two experimental analyses: a quantitative analysis to evaluate if the number of buckets that have to be investigated by the user is significantly lower than the number of original error traces, and a qualitative analysis where we analyze the buckets for one application in-depth to assess if the error traces that are grouped in one bucket actually share properties that make it easier to fix them together.

For the quantitative analysis, we evaluate Bucketeer approach on three open-source Java applications: the build system Apache Ant, the database Cassandra, and on our own tool, Bucketeer. Table 1 shows an overview of the benchmarks and a summary of some of the raw data of our evaluation.

Applied to Ant, the static checker finds 2470 methods with error traces, out of which 820 methods have more than one error trace. Bucketeer is applied to the in total 2715 error traces of those 820 methods. Applied to Cassandra, the static checker finds 2190 methods with error traces, out of which 937 methods have more than one error trace. Bucketeer is applied to the in total 3243 error traces of those 937 methods. Applied to Bucketeer itself, the static checker finds 203 methods with error traces, out of which 102 methods have more than one error trace. Bucketeer is applied to the in total 376 error traces of those 102 methods. Summarizing over all three benchmark sets, Bucketeer is applied to 6333 error traces of 1859 methods.

We discuss the results of the quantitative analysis in Section 5.1. For the qualitative analysis, which we discuss in Section 5.2, we manually inspected all buckets produced by running Bucketeer on its own source code.

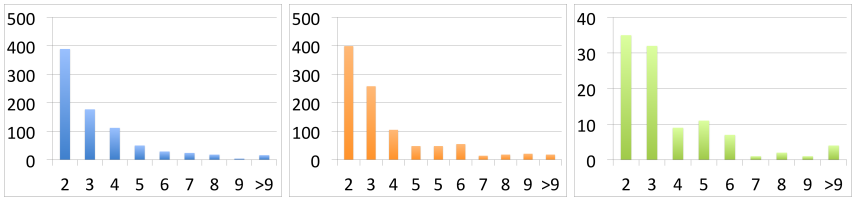
The experiments were run on a 2.7GHz i7 machine with 16GB memory and an initial size of 4GB for the Java virtual machine. We used an analysis timeout of 30 seconds per method. We experimented with larger timeouts up to five minutes per method but it had no significant effect on the number of methods that could be analyzed.

The time for the static checking and the time for categorization given in Table 1 does not account for the time spent on methods that time out or where the interpolant generation crashes.

Due to the timeout we were not able to analyze 1053 methods in Ant, 346 methods in Cassandra, and 132 methods in Bucketeer. These methods are not included in the numbers reported above.

The interpolating prover crashed for all methods where interpolation involved reasoning about the sub-typing relation used in our encoding of Java programs. We excluded these methods from our evaluation (1012 methods for Ant, 896 methods for Cassandra, and 80 methods for Bucketeer). These methods are not included in the numbers reported above.

5.1 Quantitative Analysis



(a) Ant (b) Cassandra (c) Bucketeer

Fig. 3. Number of methods (on y-axis) with n error traces (x-axis).

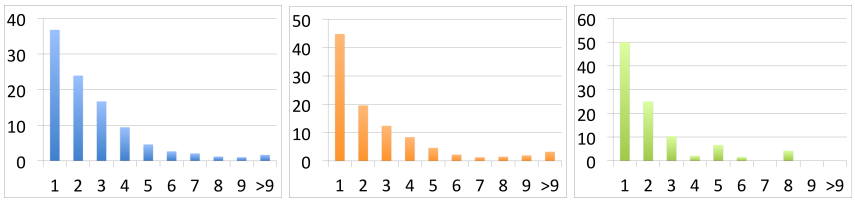
The distribution of the numbers n of error traces across the methods of a benchmark program is needed in order to interpret the performance of a classification tool on the benchmark program (in principle, the lower the number of error traces, the lower are the chances that some of them can be grouped together). Figure 3 shows, for $n = 2, 3, \ldots$, how many methods contain n error traces (the number of methods with $n = 1$ error traces (which is available in Table 1) is not present here because the classification tool is not applied to such methods). As expected, the number of methods decreases with increasing n.

We observe that the number of methods containing four or more error traces almost adds up to the number of methods containing only two error traces. This indicates that a user of a static checker will encounter methods with more than four error traces relatively frequently.

We can see that the distribution of error traces across methods is similar for Ant and Cassandra, while for Bucketeer there are more methods with three or more error traces. The difference may stem from the fact that the code of Bucketeer implements more involved algorithms.

As shown in Table 1, the overall time cost may almost double when one adds classification to static checking. The cost for classification lies in the interpolant generation, which is a relatively new technique in SMT solving, with a high potential for optimization. In any case, the cost for classification seems acceptable.

We next evaluate how many error traces can be categorized in buckets of a given size. For Ant, 1595 buckets are generated, which means that, on average, each bucket contains 1.7 error traces. For Cassandra, 2041 buckets are generated, which means an average of 1.6 error traces per bucket. For Bucketeer, 258 buckets are generated, which means an average of 1.45 error traces per bucket.

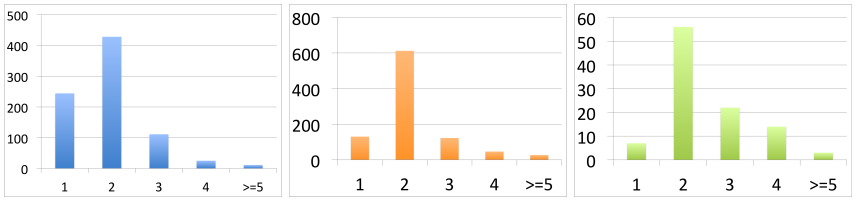


(a) Ant (b) Cassandra (c) Bucketeer

Fig. 4. Percentage of trace (y-axis) grouped in a bucket of size n (x-axis).

One way to evaluate the effectiveness of our tool for classification is to measure its behavior in view of the two extreme cases of unsatisfactory behavior. The two extreme cases are the scenario (a) where each trace ends up in a separate bucket (no trace is grouped together with another one), and the scenario (b) where all traces of a method are grouped together into one single bucket. To compare against scenario (a), we count how many traces are grouped in buckets

of a given size n, for $n = 1, 2, \ldots$. Figure 4 shows the percentage of error traces that are grouped in buckets of size n. More than half of the error traces (from 50% to 65%) are categorized in a bucket of size $n \geq 2$, i.e., more than half of the error traces are grouped together with at least one other error trace. This means that we are rather far away from the scenario (a).



(a) Ant (b) Cassandra (c) Bucketeer

Fig. 5. Number of methods that (y-axis) contain n buckets (x-axis).

To compare against the scenario (b), we count the number of methods containing n buckets, for $n = 1, 2, \ldots$. Figure 5 shows that a very large portion of the methods have two or more buckets. In other words, we are rather far away from the scenario (b).

5.2 Qualitative Analysis

The goal of our qualitative analysis was to evaluate if the error messages grouped in one bucket have a common root cause and, thus, the grouping helps to reduce redundant work for the user. To this end, we manually investigated the buckets that Bucketeer produced when we applied it to its own source code. Evaluating the tool on the code written by us introduces some confirmation bias. On the other hand, since we are familiar with the code it also increases our confidence in judging if error messages in a bucket have a common root cause.

We inspected all 65 buckets generated for Bucketeer that contained at least two error messages. These buckets can be grouped into two categories. The first category consists of buckets that contain error messages that fail because the initial state of the method sets one of the method parameters to `null` which is dereferenced later in the method body. 73% of the buckets in Bucketeer are of that form. These buckets contain between two and seven error messages (the average is 3 error messages per bucket). All these error messages have in common that the initial state sets a particular method parameter to `null`, which triggers a run-time exception somewhere in the method body. Often the actual run-time error occurs at different points in the method body. However, the important observation is that these error messages share the initialization statement of the specific input parameter and they all can be fixed by enforcing that this parameter is not `null`. That is, instead of inspecting each error message in the bucket,

a user of Bucketeer can pick any error message in one of these buckets, fix it by adding an adequate precondition, and thus eliminate all other error messages in the bucket without further inspection. The grouping of error messages provided by our approach can therefore reduce the user's workload substantially for these types of buckets.

The remaining 27% of the buckets that we inspected contained error messages where the initial state assumes a field to be `null` which is dereferenced later on the trace. Again, each bucket contains between two and seven error messages with an average of 3 error messages per bucket. The error messages in each bucket share that the initial state sets a field of an object to `null` which is dereferenced later in the method body. All error messages in one bucket share the initial state that sets the field to `null`. Some error messages also share the statement that raises the run-time exception (but take different paths to get there). Others raise run-time exceptions at different statements but because of the same field. That is, all error messages in one bucket can be fixed by adding a precondition that ensures that the given field is not `null` (or, alternatively, by guarding all dereferencing expressions with an appropriate check if such a precondition cannot be established). Again, the grouping of the error messages helps the user of Bucketeer by reducing the number of error messages that she has to inspect.

Thus, for all the buckets that we inspected, the contained error messages had a common root cause that could be fixed after inspecting only one error message in the bucket. In summary, the qualitative analysis shows that the grouping of error error messages done by Bucketeer is useful.

6 Related Work

The classification and bucketing of error messages is an active research topic. We will discuss what seems the most relevant work in our context. In summary, no existing approach to classification and bucketing addresses the question whether the comparison between error messages can be based on criteria other than syntactic or statistical criteria (as opposed to criteria based on the semantics of statements as in our work).

The original motivation for our work stems from the work in [3] which addresses the error traces generated by a software model checker (somewhat confusingly, the existing notions of error trace are subtly but substantially different from each other). The classification of error traces in [3] is based on common statements that have been identified as a possible *root cause*. The software model checker then only reports one error trace per root cause. The identification of the root cause works by comparing error traces to non-error traces which are obtained from correct executions of the program (in this it is similar to dynamic fault localization techniques such as delta debugging [31]). Our approach does not require any successful executions of a program to compare against.

A static approach to cluster static analysis errors is presented in [23]. They introduce the notion of sound dependency of alarms which is based on the trace

partitioning abstract domain. An alarm depends on another alarm if it spuriousness implies the spuriousness of the other alarm. This is different from our error signatures which can, in general, group arbitrary traces, even if they do not share control locations.

Other approaches, such as [19] or [22] cluster static analysis alarms (not necessarily only error traces) using unsound techniques. That is, their approaches may suppress alarms related to genuine errors, or highlight alarms that are actually false positives. Our approach just groups error traces. It does not suppress or highlight particular error traces, and genuine errors and false alarms may be grouped in the same bucket if they share the same error signature.

Most industrial static analyzers such as Coverity, HP Fortify, Facebook Infer, or Red Lizards Goanna have complex systems to categorize error messages, group them into buckets, and eliminate potential false alarms. These approaches usually combine statistical analysis, feasibility checks, and data-flow analysis and are heavily customized. While our approach is similar in spirit, we try to obtain a more semantic categorization of error traces with our error signatures. Existing approaches tend to group traces that violate the same property, while the error signatures capture that traces perform similar computations. That is, using error signatures is conceptually different from existing approaches.

A related problem to classification of error messages is duplicate analysis of bug reports in bug tracking systems [1, 30]. Existing techniques for automating the analysis of bug reports focus on the verbal description of the bug that is provided by the bug reporter. The information in the bug report that describes the actual error trace is typically incomplete and not amenable to automated analysis. For example, in a bug report for a program crash, one will at most find a stack trace of the program state when the crash occurred but no further information about the actual execution leading to that state. Recently, techniques have been explored to automatically reconstruct the actual error trace from a field failure by using symbolic execution [18].

The focus of our classification approach is on error traces of sequential programs. This is different in the work on the classification of concurrency bugs to identify the type of concurrency violation (out of order violation, atomicity violation, deadlocks, etc.); see, e.g., [5, 16]. The approach in [16] is related to our tool in that it also uses an SMT solver to perform this type of classification.

The notion of error signature that we introduce in this paper is somewhat related to the notion of error invariants and abstract slices explored in [7, 13, 25]. There, the goal is to obtain an explanation of a bug in an individual error trace. In contrast, the work of this paper is about the classification of a set of error traces.

The work on tools to infer preconditions such as [10] and [12] is related to our work in the sense that it may be conceivable to classify error messages according to the same precondition. In comparison, error signatures are strictly more fine grained (i.e., error messages with different error signatures may still share the same precondition; e.g., the precondition can always be $\top$ if the initial state is irrelevant for reaching the error).

7 Conclusion

We have presented an approach that uses concepts and techniques from semantics and verification in order to classify error messages in the context of static checking. We have presented the formal foundation that allows us to associate each error message with a verification problem whose solution (i.e., the proof of validity of a certain correctness property for a program derived from the error message) can be used to construct a small but significant *error signature* (on which the classification is based). We have implemented the approach and applied it to three benchmark sets. Our experiments indicate an interesting practical potential.

While our motivation stems from the context of extended static checking, it may be interesting to explore how the approach can be used to complement existing approaches to classification in other contexts (abstract interpretation, bounded model checking, testing, ...).

A more fundamental question for future research concerns the existence of a metric for error signatures in order to define a *distance between error messages*.

Acknowledgement This work is funded in parts by AFRL contract No. FA8750-15-C-0010 and the National Science Foundation under grant CCF-1350574.

References

1. J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE*, pages 361–370. ACM, 2006.
2. S. Arlt, C. Rubio-González, P. Rümmer, M. Schäf, and N. Shankar. The gradual verifier. In *NASA Formal Methods*, pages 313–327. Springer, 2014.
3. T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. *SIGPLAN Not.*, pages 97–105, 2003.
4. M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011.
5. M. T. Befrouei, C. Wang, and G. Weissenbacher. Abstraction and mining of traces to explain concurrency bugs. In *RV*, pages 162–177. Springer, 2014.
6. C. Chang and H. J. Keisler. *Model theory*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1990.
7. J. Christ, E. Ermis, M. Schaef, and T. Wies. Flow-sensitive fault localization. In *VMCAI*, 2013.
8. D. R. Cok. OpenJML: JML for Java 7 by extending OpenJDK. In *NASA Formal Methods*, pages 472–479. Springer, 2011.
9. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282. ACM, 1979.
10. P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic inference of necessary preconditions. In *VMCAI*, pages 128–148, 2013.
11. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
12. I. Dillig, T. Dillig, and A. Aiken. Automated error diagnosis using abductive inference. In *PLDI*, pages 181–192, 2012.

13. E. Ermis, M. Schäf, and T. Wies. Error Invariants. In *FM*, LNCS, pages 338–353, Berlin, Heidelberg, 2012. Springer.
14. S. Falke, F. Merz, and C. Sinz. LLBMC: improved bounded model checking of C programs using LLVM - (competition contribution). In *TACAS*, pages 623–626. Springer, 2013.
15. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. *SIGPLAN Not.*, pages 234–245, 2002.
16. A. Gupta, T. A. Henzinger, A. Radhakrishna, R. Samanta, and T. Tarrach. Succinct representation of concurrent trace sets. In *POPL*, pages 433–444. ACM, 2015.
17. M. Heizmann, J. Hoenicke, and A. Podelski. Software model checking for people who love automata. In *CAV*, pages 36–52. Springer, 2013.
18. W. Jin and A. Orso. Bugredux: Reproducing field failures for in-house debugging. In *ICSE*, pages 474–484. IEEE, 2012.
19. T. Kremenek and D. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *SAS*, pages 295–315, 2003.
20. D. Kroening and M. Tautschnig. CBMC - C bounded model checker - (competition contribution). In *TACAS*, pages 389–391, 2014.
21. A. Lal and S. Qadeer. Powering the static driver verifier using corral. In *FSE*, pages 202–212. ACM, 2014.
22. W. Le and M. L. Soffa. Path-based fault correlations. In *FSE*, pages 307–316, 2010.
23. W. Lee, W. Lee, and K. Yi. Sound non-statistical clustering of static analysis alarms. In *VMCAI*, pages 299–314, 2012.
24. K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, pages 101–121, 2005.
25. V. Murali, N. Sinha, E. Torlak, and S. Chandra. A hybrid algorithm for error trace explanation. In *VSTTE*, 2014.
26. G. Nelson. A generalization of Dijkstra's calculus. *ACM Trans. Program. Lang. Syst.*, 11(4):517–561, 1989.
27. C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM.
28. P. Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *LPAR*, 2008.
29. S. Sankaranarayanan, H. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In *POPL*, pages 318–329. ACM, 2004.
30. X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *ICSE*, pages 461–470. ACM, 2008.
31. A. Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT FSE*, pages 1–10, 2002.