Explaining Inconsistent Code

Martin Schäf
United Nations University, IIST
P.O. Box 3058
Macau S.A.R
schaef@iist.unu.edu

Daniel Schwartz-Narbonne
New York University
251 Mercer Street
New York, NY 10012, USA
dsn@cs.nyu.edu

Thomas Wies
New York University
251 Mercer Street
New York, NY 10012, USA
wies@cs.nyu.edu

ABSTRACT

A code fragment is inconsistent if it is not part of any normally terminating execution. Examples of such inconsistencies include code that is unreachable, code that always fails due to a run-time error, and code that makes conflicting assumptions about the program state. In this paper, we consider the problem of automatically explaining inconsistent code. This problem is difficult because traditional fault localization techniques do not apply. Our solution relies on a novel algorithm that takes an infeasible code fragment as input and generates a so-called error invariant automaton. The error invariant automaton is an abstraction of the input code fragment that only mentions program statements and facts that are relevant for understanding the cause of the inconsistency. We conducted a preliminary usability study which demonstrated that error invariant automata can help programmers better understand inconsistencies in code taken from real-world programs. In particular, access to an error invariant automata tripled the speed at which programmers could diagnose the cause of a code inconsistency.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Symbolic execution

General Terms

Algorithms, Performance, Theory, Verification

Keywords

Inconsistent code, error detection, static analysis, fault localization, Craig interpolation

1. INTRODUCTION

A recent study conducted at the University of Cambridge [3] estimates that, on average, programmers spend 50% of their work time on debugging. Often the most tedious part of debugging is the task of *fault localization*. Once undesired behavior is spotted in a program, the relevant program fragments that cause this behavior have to be identified before a fix can be developed. Fault localization becomes more and more challenging as the size of the program and the number of control-flow paths increases. To assist programmers in this task and reduce the manual effort involved, a number of automated fault localization techniques have been developed [15–17, 27, 31, 32, 35–37].

In this paper, we consider the related problem of automatically explaining *inconsistent code*. We call a code fragment inconsistent if it is not part of any normally terminating execution. An example of inconsistent code is a program fragment that is guaranteed to fail, e.g., because a run-time assertion such as an array bounds check is always violated. Other examples include unreachable code such as an else branch of a conditional statement that is never taken because the branching condition always evaluates to true. Although inconsistent code is not necessarily in itself a bug, it reveals significant programmer confusion, and is often correlated with serious bugs, including kernel errors in both Linux and OpenBSD [13]. In addition, code inconsistencies are an interesting class of program anomalies because they can be effectively detected using symbolic program execution. In fact, a number of existing approaches reduce the problem of detecting infeasible code to proving unsatisfiability of logical formulas, which is then automated using SMT solvers [2, 12, 21, 34]. These approaches are designed to have no false positives, and hence only report a code inconsistency if they can prove that it occurs.

While inconsistent code can be detected more easily than general program errors, the problem of localizing and explaining inconsistencies is more difficult. To localize the cause of a run-time error such as a null-pointer exception, it is usually sufficient to investigate a single trace in which the error is observed. On the other hand, to explain a code inconsistency such as the fact that a particular line in the program is not reachable, it is not sufficient to look at a single trace. Instead, one has to consider all traces of the program–in general, infinitely many.

In this paper, we propose the first algorithm to localize inconsistent code automatically. Given an inconsistent program as input, our algorithm produces a so-called *error invariant automaton* that explains the inconsistency. The error invariant automaton is an abstraction of the input program. It consists of the statements of the input program that are the cause of the inconsistency as well as formulas over the program variables that summarize the remaining

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08...$15.00
http://dx.doi.org/10.1145/2491411.2491448

irrelevant statements, which have been eliminated. We refer to these formulas as *error invariants*. Intuitively, an error invariant is a formula describing the reachable states at a given program location and it captures the reason why the program does not terminate normally if execution is continued from these states. Our algorithm builds on our previous work on fault localization for single error traces [8, 15] in which we use Craig interpolation [10] to extract error invariants automatically from a symbolic encoding of the error trace.

We conducted a usability study to evaluate the usefulness of error invariant automata for understanding inconsistent code. For this purpose, we computed error invariant automata for inconsistent code fragments found in real-world programs. We then presented these code fragments to users, with and without additional visual aid that we generated from the error invariant automata. On average, the additional information extracted from the automata helped to reduce the time that the users needed to understand the reason for inconsistency by a factor of three.

Using our algorithm we were able to compute the error invariant automata for the example programs in a few seconds. This suggests that it is feasible to implement our technique as a plug-in for an integrated development environment that detects and explains infeasible code on-the-fly while the programmer is typing in the program.

2. EXAMPLES

We illustrate how our technique explains the causes of inconsistent code using two examples.

2.1 Example 1

Figure 1 shows inconsistent code in the open-source tool Rachota[1] (for space reasons, we have summerized some code lines as "..."). When we ran this code through Joogie [2], it reported a problem on line 27. The question is why?

On line 6, the programmer calls `task.getDescription()`. If `task` is `null`, the program will crash, so line 6 implies a requirement that `task` is not `null`. On the other hand, line 27 includes the test `task == null`. This implies that the programmer believes that `task` might be `null`, contradicting the invariant implied by line 6.

How should this anomaly be reported to the programmer? Most existing fault localization tools attempt to locate the cause of a particular erroneous execution through the program, whereas an inconsistency is a property of *every* feasible execution through the inconstant points.

Dynamic approaches such as [36] will not work for this case, because an execution where `task` is `null` will cause an exception at line 6 and therefore never reach the contradicting assumption that `task` can be `null` at line 27. Dynamic techniques can witness a potential error in line 6, but never the contradiction between line 6 and 27.

Static fault localization tools such as Bug-Assist [26], and our approach based on *error invariants* [8, 15], do fault localization for individual paths. In our example, we have two paths that connect line 6 and line 27 due to the conditional choice in line 16. An explanation for the inconsistency must take into account both of these possible paths. In general, inconsistent code might be witnessed by possibly infinitely many executions, and they might be inconsistent for different reasons. Hence we need a localization technique that can reason about an arbitrary number of inconsistent executions.

[1] `http://rachota.sourceforge.net/`

```
/* org.cesilko.rachota.gui.TaskDialog */
 1: public TaskDialog(Task task, Day day,
        boolean readOnly) {
 ~: ...
 6: txtDescription.setText(task.getDescription());
 ~: ...
16: if (notification) {
    ...
   }
 ~: ...
27: chbRegular.setEnabled(task == null);
 ~: ...
}
```

Figure 1: Infeasible code found in Rachota. Elided code does not modify `task`.

Error invariant automata. In this paper, we introduce *error invariant automata* which allow us to generate parsimonious explanations of the causes of code inconsistencies. At a high level, an error invariant automaton replaces code which does not contribute to the inconsistency with a suitably chosen invariant.

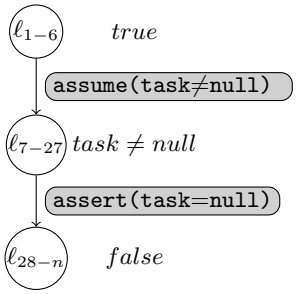
To see how this works in practice, notice that the function in Figure 1 is equivalent to the following simplified program:

- **Lines 1–5:** These lines do not affect the existence of the inconsistency.
- **Line 6:** `assert(task != null);`
- **Lines 7–26:** Any arbitrary code that does not affect whether `task == null`.
- **Line 27:** An assumption that `task` might be `null`.
- **Lines 28...end:** These lines do not affect the existence of the inconsistency.

We can represent this program graphically as a finite state machine where nodes represent predicates on the program state, and edges represent program statements that modify the value of the predicates. We refer to such a state machine as an *error invariant automaton* if it over-approximates the reachable states of the original program, and has no valid consistent terminating execution. Clearly, the full program is one such automaton (with invariants representing the reachable states at each program point); the simplified program presented above is another. The challenge is to create a *minimal* automaton that is sufficient to explain the program inconsistency.

	Assertion	Error Invariant
	$\ldots$	$\leftarrow true$
ℓ_6	$\wedge\, task \neq \texttt{null} \wedge \ldots$	$\leftarrow task \neq \texttt{null}$
	$\wedge \ldots$	$\leftarrow task \neq \texttt{null}$
ℓ_{16}	$\wedge[((\texttt{notif.} \wedge (\ldots))$	
	$\vee(\neg\texttt{notif.} \wedge (\ldots))]$	$\leftarrow task \neq \texttt{null}$
	$\wedge \ldots$	$\leftarrow task \neq \texttt{null}$
ℓ_{27}	$\wedge(task = \texttt{null}) \wedge \ldots$	$\leftarrow false$

Figure 2: (a) First-order logical formula for Figure 1



(b) Error Invariant Automaton for Figure 1.

Construction of an error invariant automaton. The construction of such an automaton is based on an insight from finite state machines: if a path through the machine visits the same state twice, then any states visited between the repeated state are unnecessary to prove the existence of an accepting run. If a machine has an accepting run labeled with states $A \cdot B \cdot C \cdot B \cdot D$, it must also have an accepting run $A \cdot B \cdot D$. This means that we can merge sequences of states in an error invariant automaton that start and end in a state labeled with the same predicate.

There are two equivalent ways to create a minimal error invariant automaton. One is to start with the full program, and to merge states until a fixed point is reached (this is the algorithm we present in § 4). The other, which we describe here, is to start with an empty program and add abstract states until the desired property is provable. Note that this procedure generates a *minimal* automaton but not necessarily a *minimum* automaton, since which states can be merged depends on which invariants are chosen for the state labels.

We assume that we have a static analysis tool such as Joogie that proves the existence of inconsistent code [2]. In this example, Joogie reports an error on line 27. We use this information to extract a program fragment which asserts the inconsistency. In this case, we assert that the program reaches line 27, and that `task` is `null` at line 27. From this program fragment, we construct a first-order logic formula which is unsatisfiable because of the inconsistency in the original program. This encoding is similar to the extended path formulas shown in [8], however our approach encodes multiple paths into one formula.

We can now use the generated first-order formula to identify the cause of the inconsistency. Our procedure proceeds recursively. First, we select a (sub)program and a candidate invariant which holds at the beginning of that sub-program (this invariant could be supplied by the user, or automatically generated by an interpolating theorem prover). If the candidate invariant also holds at the exit of the sub-program, then it is an inductive invariant, and we can replace that sub-program with the invariant. Otherwise, we split the program into two sub-programs. We then calculate a new candidate invariant for each sub-program, and repeat. At the end, we have a simple automaton which concisely represents the cause of the inconsistency.

The result for Example 1 can be seen in Figure 2. We begin by formulating the program as a first-order logical formula (the left column in Figure 2(a)). The right-hand column shows invariants which hold at each program location, and which are sufficient to prove the inconsistency of the code. Figure 2(b) shows the error invariant automaton generated using our procedure.

We construct our error invariant automaton starting from a single node which is labeled with the first candidate error invariant `true`. This invariant holds up to, but not past, line 6. We therefore split the program in two, with one node representing lines 1–6 and annotated `true`, and the other representing the rest of the program. We then repeat for lines 6–end. The predicate `task != null` is a valid error invariant for lines 7–27, so we split the program again. In particular, this invariant is inductive for the conditional choice in line 16, so neither of both branches has an effect that is relevant for the proof. Finally, `false` is a valid error invariant for the remainder of the program, so we are done.

A programmer attempting to locate the cause of the inconsistency by analyzing the original program would need to analyze a 28+ line procedure containing conditionals. A programmer using our tool would only need to analyze a straightforward procedure with two statements linked by one non-trivial invariant.

2.2 Example 2

This example demonstrates how we deal with non-trivial branches. In the previous example, we were able to replace the conditional with an invariant. In the procedure `toyExample` (Figure 3), the location of the error depends on the value `b`. If `b` is true, `toyExample` attempts to dereference a `null` pointer and fails in line 4; if `b` is false, it fails on line 6. Note that the branches fail at different locations.

```
/* A constructed example */
1: public void toyExample(Boolean b) {
2: MyObject x=null;
3: if (b) {
4:   x.foo();
5: }
6: x.bar();
7:}
```

Figure 3: Branch-dependent inconsistency in code

Figure 5 shows (a simplified version of) the unsatisfiable formula created for the program in Figure 3. Multiple as-

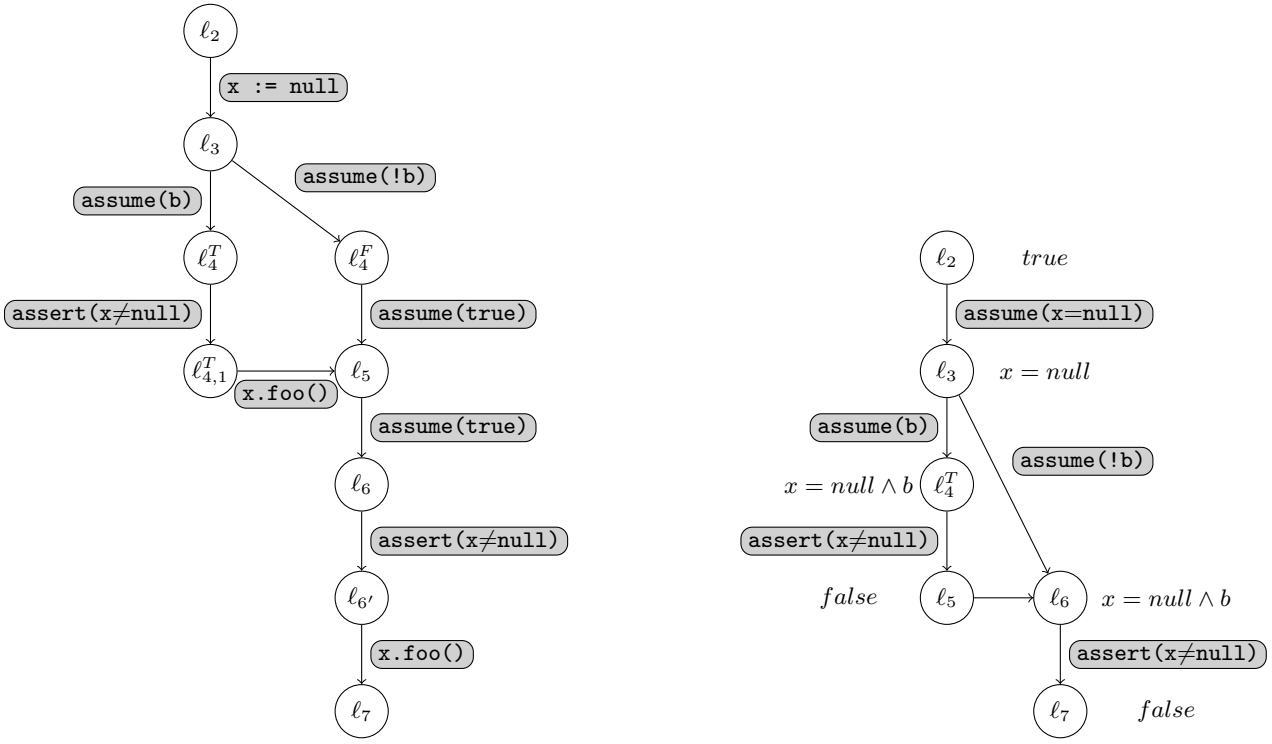


Figure 4: (a) Program automaton for code from Figure 3 (b) Error invariant automaton for Figure 3

	Assertion	Error Invariant
ℓ_2	$x^0 = \texttt{null}$	$\leftarrow true$
ℓ_3	$\wedge[\quad (\texttt{b} \wedge (x^0 = \texttt{null} \implies exit))$	$\leftarrow x = \texttt{null}$
	$\vee[(\neg\texttt{b} \wedge (x^2 = x^0))]$	
$\sim$	$\wedge[\quad ((\texttt{b} \wedge (x^3 = x^1))$	$\leftarrow (x = \texttt{null} \wedge \neg exit) \vee \neg b$
	$\vee(\neg\texttt{b} \wedge (x^3 = x^2)))]$	
	$\wedge \quad (\neg exit)$	$\leftarrow x = \texttt{null} \vee exit$
ℓ_6	$\wedge \quad (x^3 \neq \texttt{null}) \wedge \ldots$	$\leftarrow x = \texttt{null}$
		$\leftarrow false$

Figure 5: First-order logical formula for Fig. 3

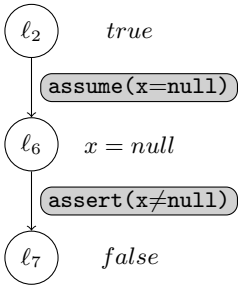


Figure 6: Alternative error invariant automaton for Fig. 3.

signments to the same variable are handled by representing the program in SSA (Single Static Assignment) form, with superscripts used to distinguish between different assignments. We handle early termination by introducing an auxiliary variable `exit`, which is set to true whenever the program exits early. All code subsequent to a potential exit is effectively guarded by the test (`!exit && code`).

Figure 4(b) shows a case where we were unable to find an inductive invariant for the conditional choice. We handle this by splitting the automaton. One branch represents the case where the conditional is taken, and the other represents the branch where it is not. We can then recursively apply the procedure on each branch, as before. Figure 6 shows the result of making an alternative choice for our inductive invariant. The predicate `x = null` is an inductive error invariant which holds across the conditional on lines 3–5, and hence allows us to collapse the size of the error invariant automaton.

This example shows the crucial effect of the quality of the invariants on the minimality of the derived automaton. It is encouraging to note that the more effective invariant (the one in Figure 6) is the one derived from an interpolant.

3. PRELIMINARIES

3.1 Program Automata

We present programs and their control flow structure using program automata [20]. Program automata provide a simple model of programs that abstracts from the syntactic constructs and semantics of specific programming languages. In section § 5, we describe how common language constructs can be described in terms of program automata.

A program automaton is a finite automaton. A state in a program automaton corresponds to a program location, and transitions between two states are labeled with program statements. That is, a program automaton accepts a regular language of finite words over statements. Each word in this language corresponds to one control flow path. Note that not every path of a program automaton gives rise to feasible executions.

Formally, let Σ be a fixed set of program statements. A *program automaton* $\mathcal{A}$ is a tuple $(Loc, \delta, \ell_0, \ell_e)$ where

- Loc is a finite set of locations,
- $\delta_{\mathcal{P}} \subseteq Loc \times \Sigma \times Loc$ is a finite transition relation,
- ℓ_0 is an *initial location*, and
- ℓ_e is an *exit location*.

We interpret $\mathcal{A}$ as a finite automaton over alphabet Σ with initial state ℓ_0 and unique final state ℓ_e. A *run* ρ of $\mathcal{A}$ is a finite sequence of locations and statements $\ell_0 st_0 \ell_1 \dots st_{n-1}\ell_n$, such that for all $i \in [0, n)$, $(\ell_i, st_i, \ell_{i+1}) \in \delta$. We call $path(\rho) = st_0 \dots st_{n-1}$ the *path* associated with ρ. A run ρ is accepting if its final state is ℓ_e. We call a word $\pi \in \Sigma^*$ a path of $\mathcal{A}$ if $\pi = path(\rho)$ for some accepting run ρ of $\mathcal{A}$.

Figure 4(a) shows the program automaton representing the program in Figure 3.

3.2 Semantics

We present the semantics of program automata using formulas in first-order logic. We assume standard syntax and semantics of such formulas. Let X be a fixed set of program variables. A *program state* (hereafter, simply referred to as *state*) is a valuation of the variables from X. A *state formula* F is a first-order constraint over free variables from X, i.e., F represents the set of all states s that satisfy F.

For a variable $x \in X$ and $i \in \mathbb{N}$, we denote by $x^{\langle i \rangle}$ the variable which models the value of x in a state that is shifted i time steps into the future. We extend this shift function from variables to sets of variables, as expected, and we denote by X' the set of variables $X^{\langle 1 \rangle}$. For a formula F with free variables from Y, we write $F^{\langle i \rangle}$ for the formula obtained by replacing each occurrence of a variable $y \in Y$ in F with the variable $y^{\langle i \rangle}$. A *transition formula* T is a first-order constraint over free variables from $X \cup X'$, where the variables X' denote the values of the variables from X in the next state. That is, a transition formula T represents a binary relation on states. We assume that every statement $st \in \Sigma$ has an associated transition formula $\mathbf{TF}(\sigma)$ that provides the semantics of st. For example the transition formula of an assume statement can be defined as $\mathbf{TF}(\texttt{assume(F)}) \equiv F \wedge X = X'$, where $X = X'$ stands for the conjunction of equalities $x = x'$ for all $x \in X$.

For a path $\pi \in \Sigma^*$ with $\pi = st_0 st_1 \dots st_n$, we define its *path formula* $\mathbf{PF}(\pi)$ as the conjunction of the shifted transition formulas of the statements in π

$$\mathbf{PF}(\pi) = \mathbf{TF}(st_0) \wedge \mathbf{TF}(st_1)^{\langle 1 \rangle} \wedge \dots \wedge \mathbf{TF}(st_n)^{\langle n \rangle} \ .$$

A path π is called *inconsistent* if its path formula is unsatisfiable. A program automaton $\mathcal{A}$ is inconsistent if all its paths are inconsistent.

3.3 Reducible Program Automata

Given a program automaton $\mathcal{A} = (Loc, \delta, \ell_0, \ell_e)$, we call a location $\ell \in Loc$ a *fork* if it has more than one outgoing edge, and we call it a *join* if it has more than one incoming edge. We say that location ℓ *dominates* location ℓ', written $\ell \gg_{\mathcal{A}} \ell'$, if every run of $\mathcal{A}$ from ℓ_0 to ℓ' includes ℓ. Analogously, ℓ' *post-dominates* ℓ, written $\ell' \ll_{\mathcal{A}} \ell$, if every run of $\mathcal{A}$ that starts in ℓ and ends in ℓ_e goes through ℓ'.

We consider *structured* programs, i.e., we only consider program automata that are reducible. If $\mathcal{A}$ is reducible, then for every fork ℓ there exists a unique join ℓ' such that ℓ dominates ℓ' and every other location that dominates ℓ' also dominates ℓ. We denote ℓ' by $join(\ell)$. Intuitively, a fork ℓ in a reducible program automaton is the starting location of a control flow construct such as a conditional or a loop, and $join(\ell)$ is the unique exit location of the corresponding construct. For example, in the program automaton shown in Figure 4(a), the location ℓ_3 is a fork with $join(\ell_3) = \ell_5$.

For a fork ℓ and a (direct) successor ℓ_s of ℓ, we denote by $\mathcal{A}(\ell, \ell_s)$ the *branch automaton* that corresponds to the branch going through ℓ_s of the control flow construct starting in ℓ. Formally, the branch automaton $\mathcal{A}(\ell, \ell_s)$ is the program automaton $(Loc_s, \delta_s, \ell, join(\ell))$ where $Loc_s = \{\ell\} \cup \{\, \ell' \in Loc \mid \ell_s \gg_{\mathcal{A}} \ell' \wedge join(\ell) \ll_{\mathcal{A}} \ell' \,\}$ and $\delta_s = \delta \cap Loc_s \times \Sigma \times Loc_s$. For example, in the program automaton shown in

Figure 4(a), the fork ℓ_3 has two branch automata $\mathcal{A}(\ell_3, \ell_4^T)$ and $\mathcal{A}(\ell_3, \ell_4^F)$ corresponding to the two paths encoding the conditional in the program of Figure 3.

4. ERROR INVARIANT AUTOMATA

In our previous work [15], we introduced the notion of error invariants to localize faults in error traces. In the following, we generalize this technique from single traces to entire program automata, which in general represent infinitely many traces.

4.1 Error Invariants

We can represent an error trace of $\mathcal{A}$ as a sequence of statements $\pi = st_0 \dots st_n$ of $\mathcal{A}$, together with state formulas *Pre* and *Post*. The formula *Pre* describes the initial state(s) of the error trace, and *Post* is the assertion that is violated by the state obtained after executing the statements in π, starting in a state that is described by *Pre*. Note that this means that the formula $Pre \wedge \mathbf{PF}(\pi) \wedge Post^{\langle n \rangle}$ is unsatisfiable.

An *error invariant* for position $i \in [0, n+1]$ in the error trace $(Pre, \pi, Post)$ is a state formula I such that the following two conditions hold:

1. $\mathbf{PF}(st_0 \dots st_{i-1})$ implies $I^{\langle i \rangle}$, and
2. $I \wedge \mathbf{PF}(st_i \dots st_n)$ is unsatisfiable.

That is, the error invariant explains why, after reaching position i, the trace will fail if execution of the trace is continued from that position.

We say that an error invariant I is *inductive* for positions $i < j$, if I is an error invariant for both i and j. Intuitively, an inductive error invariant tells us that the statements between positions i and j in π are irrelevant for the error trace.

In [15], we presented an algorithm ErrInv that, given an error trace $(Pre, \pi, Post)$ computes an alternating sequence

$$I_o st_{i_1} I_1 st_{i_1} \dots st_{i_k} I_k$$

such that: (1) $st_{i_1} \dots st_{i_k}$ is a subsequence of π; and (2) for all $j \in [0, k]$, I_j is an inductive invariant for positions i_j and i_{j+1} in π (where $i_0 = 0$ and $i_{k+1} = n + 1$). That is, $\mathsf{ErrInv}(Pre, \pi, Post)$ is the subsequence of error-relevant statements in π together with error invariants that provide summaries of the sliced irrelevant statements in π. The algorithm ErrInv relies on Craig interpolation [10] to automatically obtain the inductive error invariants from the proof of unsatisfiability of the formula $Pre \wedge \mathbf{PF}(\pi) \wedge Post^{\langle n \rangle}$.

In the following, we generalize the notion of error invariants from single traces to program automata. We then present an algorithm that uses our previous algorithm ErrInv to compute an explanation for the inconsistency of a given program automaton.

4.2 Error Invariant Automata

Let $\mathcal{A} = (Loc, \delta, \ell_0, \ell_e)$ be an inconsistent program automaton. A state formula I is called error invariant for a location ℓ of $\mathcal{A}$, if for all accepting runs $\rho = \ell_0 st_0 \dots st_n \ell_{n+1}$ of $\mathcal{A}$ with $\ell = \ell_i$ for some $i \in [0, n+1]$, I is an error invariant for position i of $(true, path(\rho), true)$. We adapt the notion of inductiveness accordingly: for two location $\ell, \ell' \in Loc$ such that ℓ dominates ℓ', we call a state formula I an *inductive* error invariant for ℓ and ℓ' if I is an error invariant for both ℓ and ℓ'.

An *error invariant automaton* $\mathcal{A}_\mathcal{I}$ is an inconsistent program automaton together with a mapping $\mathcal{I}$ from locations of $\mathcal{A}_\mathcal{I}$ to state formulas such that for all locations ℓ, $\mathcal{I}(\ell)$ is an error invariant for ℓ. We say that an error invariant automaton $\mathcal{A}_\mathcal{I} = (Loc', \delta', \ell_0', \ell_e')$ *explains* an inconsistent program automaton $\mathcal{A} = (Loc, \delta, \ell_0, \ell_e)$ if there exists a surjective mapping $h : Loc \to Loc'$ such that the following two conditions hold:

1. for all $\ell_1, \ell_2 \in Loc$, $st \in \Sigma$, $(h(\ell_1), st, h(\ell_2)) \in \delta'$ if and only if $(\ell_1, st, \ell_2) \in \delta$ and $h(\ell_1) \neq h(\ell_2)$.
2. for all $\ell_1, \ell_2 \in Loc$, if $h(\ell_1) = h(\ell_2) = \ell'$, then there exists $\ell_1', \ell_2' \in h^{-1}(\ell')$ such that $\mathcal{I}(\ell')$ is an inductive error invariant for ℓ_1' and ℓ_2' in $\mathcal{A}$, ℓ_1' dominates both ℓ_1 and ℓ_2, and ℓ_2' post-dominates both ℓ_1 and ℓ_2.

4.3 Computing Error Invariant Automata

We next present our algorithm that takes an inconsistent program automaton $\mathcal{A}$ and computes an error invariant automaton $\mathcal{A}_\mathcal{I}$ that explains $\mathcal{A}$. We start with an algorithm that assumes $\mathcal{A}$ to be loop-free. In the next section, we then explain how to extend this algorithm to handle loops and other language constructs.

Our algorithm is shown in Figure 7. We give a high-level description. The algorithm takes a loop-free program automaton and two state formulas *Pre* and *Post* as input. It assumes that $\mathcal{A}$ is inconsistent subject to *Pre* and *Post*. That is, for every path π of $\mathcal{A}$ that is of some length i, $Pre \wedge \mathbf{PF}(\pi) \wedge Post^{\langle i \rangle}$ is assumed to be unsatisfiable. The algorithm returns an error invariant automaton that explains the inconsistency of $\mathcal{A}$. To compute the error invariant automaton for the initial $\mathcal{A}$, we apply the algorithm with $Pre = Post = true$.

The first step of the algorithm is to translate $\mathcal{A}$ into a single path $\pi_\mathcal{A} = st(\ell_0) \dots st(\ell_n)$. The statements in this path constitute the top-level basic block of the program that is represented by $\mathcal{A}$. That is, each statement $st(\ell_i)$ may represent a complex control flow construct that is composed of many atomic statements in Σ. The path $\pi_\mathcal{A}$ is still inconsistent, hence we can view it as an error trace $(Pre, \pi_\mathcal{A}, Post)$ to which we apply the algorithm ErrInv. The resulting sequence $I_0 st(\ell_{i_1}) \dots st(\ell_{i_k}) I_k$ consists of (1) the subsequence of (composite) statements in $\pi_\mathcal{A}$ that are relevant for explaining the inconsistency of $\mathcal{A}$; and (2) error invariants for the locations at which these statements start. The locations and statements not appearing in the returned sequence (i.e., those that are covered by an inductive error invariant) can be collapsed to a single location in the error invariant automaton that explains $\mathcal{A}$. This is implemented by the operation *collapse*. For each remaining non-atomic statement $st(\ell_{i_j})$, we compute the branch automata for the control flow construct represented by $st(\ell_{i_j})$. We then apply the algorithm recursively to all of these smaller automata. Here,

proc Explain :
input
Pre : precondition state formula
$\mathcal{A}$: program automaton
$Post$: postcondition state formula
output
$\mathcal{A}_\mathcal{I}$: error invariant automaton
requires
$\mathcal{A}$ is inconsistent subject to Pre and $Post$
ensures
$\mathcal{A}_\mathcal{I}$ explains $\mathcal{A}$
begin
var $\mathcal{A}_\mathcal{I} = \mathcal{A}$ with $\mathcal{I} = (\lambda \ell.\, false)$
var $\ell_0, \ldots, \ell_{n+1} = toploc(\mathcal{A})$
var $\pi_\mathcal{A} = st(\ell_0) \ldots st(\ell_n)$
var $I_o, st(\ell_{i_1}), I_1, \ldots, st(\ell_{i_k}), I_k = \mathsf{ErrInv}(Pre, \pi_\mathcal{A}, Post)$
var $i_0 = 0$
var $i_{k+1} = n + 1$
for each $j \in [0, k]$ **do**
$\mathcal{I}[\ell_{i_j}] := I_j$
$\mathcal{A}_\mathcal{I} := collapse(\mathcal{A}_\mathcal{I}, \ell_{i_j}, \ell_{i_{j+1}})$
for each $j \in [1, k]$ **such that** ℓ_{i_j} is a fork **do**
for each direct successor ℓ_s of ℓ_{i_j} in $\mathcal{A}$ **do**
var $\mathcal{A}_s = \mathcal{A}(\ell_{i_j}, \ell_s)$
var $\mathcal{A}_{\mathcal{I}s} = \mathsf{Explain}(I_{j-1}, \mathcal{A}_s, \neg I_j)$
$\mathcal{A}_\mathcal{I}$:= replace $\mathcal{A}_s$ in $\mathcal{A}_\mathcal{I}$ with $\mathcal{A}_{\mathcal{I}s}$
return $\mathcal{A}_\mathcal{I}$
end

Figure 7: Algorithm for explaining an inconsistent program automaton.

we exploit the properties of the computed error invariants I_j, which ensure that each such automaton is inconsistent subject to $Pre = I_{j-1}$ and $Post = \neg I_j$.

It remains to define formally the statements in the path $\pi_\mathcal{A}$ that constitute the top-level basic block of $\mathcal{A}$. For this purpose, let $toploc(\mathcal{A})$ be the ordered sequence of locations of $\mathcal{A}$ that are the starting points of statements in the top-level basic block. That is, $toploc(\mathcal{A})$ is the maximal sequence of distinct locations $\ell_0, \ell_1, \ldots, \ell_{n+1}$ such that for all i, j, such that $0 \leq i \leq j \leq n+1$, we have $\ell_i \gg_\mathcal{A} \ell_j$. In particular, we have $\ell_{n+1} = \ell_e$. For every location $\ell \in toploc(\mathcal{A})$, we denote by $lp(\ell)$ the length of the longest path from ℓ_0 to ℓ in $\mathcal{A}$ and we define $lp(\mathcal{A}) = lp(\ell_e)$.

Now, for every location ℓ_i, with $i \in [0, n]$, let $st(\ell_i)$ be a fresh statement not in Σ and let Y_i be a fresh copy of the program variables in X. For a formula F we denote by $F[Y_i/X]$ the formula that is obtained from F by substituting all occurrences of (primed) variables in X by their (primed) versions in Y_i. The transition formula $\mathbf{TF}(st(\ell_i))$ of the new statement $st(\ell_i)$ is defined as follows. If ℓ_i is a fork, we must have $join(\ell_i) = \ell_{i+1}$. Then define $\mathbf{TF}(st_i)$ as the disjunctions of the path formulas of the automata for the branches between ℓ_i and ℓ_{i+1}. That is, let $\ell_{i,0}, \ldots, \ell_{i,k}$ be the immediate successors of ℓ_i in $\mathcal{A}$. Further, for all $j \in [0, k]$, let $\mathcal{A}_j = \mathcal{A}(\ell_i, \ell_{i,j})$, $m_j = lp(\mathcal{A}_j)$, and define $m = \max\{\, m_j \mid 0 \leq j \leq k \,\}$. Finally, we define

$$\mathbf{TF}(st(\ell_i)) = \begin{cases} X = Y_i \wedge \\ X' = Y_i^{\langle m \rangle} \wedge \\ \bigvee_{0 \leq j \leq k} \mathbf{PF}(\pi_{\mathcal{A}_j})[Y_i/X] \wedge Y_i^{\langle m_j \rangle} = Y_i^{\langle m \rangle} \end{cases}$$

Otherwise, if ℓ_i is not a fork, then ℓ_{i+1} is the unique direct successor of ℓ_i for some statement $st_i \in \Sigma$. In this case, simply define $\mathbf{TF}(st(\ell_i)) = \mathbf{TF}(st_i)$. It is easy to prove by induction that $Pre \wedge \mathbf{PF}(\pi_\mathcal{A}) \wedge Post^{\langle n \rangle}$ is unsatisfiable iff $\mathcal{A}$ is inconsistent subject to Pre and $Post$.

5. EXTENSIONS

Next, we discuss how we can use our basic algorithm from the previous section to handle common features found in actual programming languages.

5.1 Loops and Procedure Calls

To handle loops and procedure calls in inconsistent code, we rely on existing techniques. For example, in our previous work on inconsistent code detection [22], we presented an approach that we named *abstract unrolling*. Abstract unrolling over-approximate the behavior of a program with loops by one without loops. The technique unrolls the first and last iteration of a loop and abstracts all intermediate iterations by a single transition that assigns non-deterministic values to the modified variables in the loop. We have found that this technique scales well because it is a simple syntactic transformation of the program, yet preserves code inconsistencies in practice. In particular, using this technique one can still detect common code inconsistencies in loops such as off-by-one errors. Since the abstraction over-approximates the behavior of the original program we guarantee that the input program is inconsistent if the abstraction is inconsistent. Abstract unrolling can be generalized to handle procedure calls by inlining called procedures in the analyzed code fragment, but abstracting subsequent calls inside the inlined procedure bodies.

It is also possible to combine the above techniques with more heavy-weight analyses that increase the detection rate but are more expensive. Note that the problem of detecting inconsistent code can be reduced to verifying a safety property, namely that the exit location of the program is unreachable. We can therefore use existing static analysis techniques for inferring loop invariants and procedure summaries to increase the precision of abstract unrolling (respectively, abstract inlining). Techniques that are based on interpolation [1, 14, 30] are particularly well-suited because our localization algorithm already uses interpolation procedures. Using the computed invariants one can then obtain more precise transformations into loop-free programs.

In summary, the problem of how to deal with loops and procedure calls must already have been addressed in the detection of code inconsistencies. In fact, a (Hoare) proof of inconsistency of a program always yields a syntactic transformation into a loop-free program that is inconsistent.

5.2 Nonstructured Control Flow

In Section 4, we assumed that the input program automaton has structured control flow and our algorithm for explaining inconsistent code relies on this property to encode the automaton effectively into a formula. Despite this restriction, we can still support common forms of unstructured control flow that can be found in many programming languages such as return, break, and continue statements, and exception mechanisms. All these mechanisms have in common that control does not jump arbitrarily. Instead, control is transferred immediately to some program location that is reachable by following the regular control flow of the program. We can therefore encode these mechanism by introducing auxiliary variables.

For example, to model a return statement, we introduce an auxiliary Boolean variable *returned*. Initially, this variable is set to *false* and it is set to *true* if a return statement is executed. All the transition formulas $\mathbf{TF}(st)$ of the program are then guarded by this variable, i.e., they are of the form $\neg returned \Rightarrow F(X, X')$, where $F(X, X')$ is the actual transition formula that provides the semantics of statement st. Hence, if a return statement is executed, control follows the normal flow of the program but all statements along the path are skipped. A location along the path is then reachable in the original program if it is reachable in the new program in a state in which *returned* is *false*.

Other mechanisms for non-structured control flow can be modeled in a similar manner, including assert statements that check for the occurrence of run-time errors such as null-pointer dereferences. By using different auxiliary variables for encoding these mechanisms, we can also classify code inconsistencies, e.g., to distinguish between inconsistencies that are caused by guaranteed errors, and inconsistencies such as code that is unreachable because a preceding return statement is always executed.

6. EVALUATION

6.1 Construction of Error Invariant Automata

We evaluated our approach using six real-world examples of inconsistent code found in open-source projects. Three examples were taken from the mind mapping tool FreeMind, one example (the one from Figure 1) is taken from Rachota, and the remaining two are taken from device drivers in the Linux kernel discussed by Engler et al. [13].

For each of these examples, we constructed an error invariant automaton following the algorithm discussed in § 4. Procedure calls were abstracted as calling `havoc` on their modset, which was sufficient to prove the inconsistency in all examples. None of the examples contained loops, so we did not have to use loop abstraction techniques in the program automaton. Since we were able to prove inconsistency even given this very weak approximation, all generated error invariant automata represent real code inconsistencies, with no false alarms. The generated path formulas for the initial automata ranged from 70–142 lines of SMT-LIB2 [5] code (including comments), with a median of 89 lines. The translation was performed manually, but was fairly mechanistic and would not be difficult to automate.

We generated candidate error invariants using the interpolation procedures implemented in the SMT solver MATHSAT [9]. Using repeated calls to the SMT solver we then identified the code fragments for which they are inductive. In some cases we split conjuncts by adding auxiliary variables, in order to allow precise placement of the interpolation points.

Results. Running time to prove `unsat` and generate the interpolants ranged from 0.008 seconds (experiment 4) to 0.019 seconds (experiment 6), which suggests that this technique is practical for use in real-time tools such as code editors.

6.2 Usability Testing

We conducted an experiment to evaluate whether error invariant automata can be used to provide visual assistance which allows a programmer to more quickly understand the causes of code inconsistencies. We recruited 11 programmers and computer scientists for this study, 5 at the United Nations University in Macau, and 7 at New York University. We gave a 5 minute introduction to each candidate where we explained the concept of inconsistent code, the purpose of the experiment, and some samples of inconsistent code.

Participants were told that they would be presented with a series of functions which contained inconsistent code, and that their job was to identify the cause of the inconsistency as soon as possible. Half of the examples they would be shown would contain the entire body of the relevant function, with the line where the inconsistency manifested itself underlined in red. The other half of the examples used the error invariant automaton to provide visual assistance as follows: all statements of the function that do not have a corresponding edge in the error invariant automaton are hidden behind solid blue boxes. The boxes are labeled with the invariant associated with the node in the error invariant automaton that summarizes the hidden statements under it. Figure 8 gives an example of a function without (left) and with visual assistance (right). For each candidate we alternated the snippets for which we provided the visual assistance. For each example, half of the participants (chosen randomly) were shown the full function; the other half were shown the error invariant automaton.

As soon as a code snippet (with or without visual assistance) was on the screen, we started a stopwatch and told the candidate to say "stop", once (s)he is sure what the cause of inconsistency is. If the explanation was wrong, we continued the stopwatch. If no correct answer was given within 150 seconds, we stopped the watch and explained the solution. The set of slides used in our experiments is available on Dropbox[2].

Results. All candidates in total took 1 hour and 6 minutes to identify the problems in all code snippets. For the code snippets without explanation they took a total of 51 minutes, and for the code snippets with explanation they took 17 minutes, which roughly is a speed up by a factor of 3.

[2] `http://goo.gl/FF9an`

```
public void deRegister() {
    controller.deregisterNodeSelectionListener(mNotesManager);
    controller.deregisterNodeLifetimeListener(mNotesManager);
    noteViewerComponent.getActionMap().remove("jumpToMapAction");

    if (noteViewerComponent != null && shouldUseSplitPane()) {
        hideNotesPanel();
        noteViewerComponent = null;
    }
    logger.fine("Deregistration of note undo handler.");
    controller.getActionFactory().deregisterActor(
    getDoActionClass());
}
```

```
public void deRegister() {
    noteViewerComponent.getActionMap().remove("jumpToMapAction");
    if (noteViewerComponent != null && shouldUseSplitPane()) {
```

noteViewerComponent must not be null

Figure 8: Example of the code snippets used in the experiment with, and without visual assistance created from the error invariant automaton.

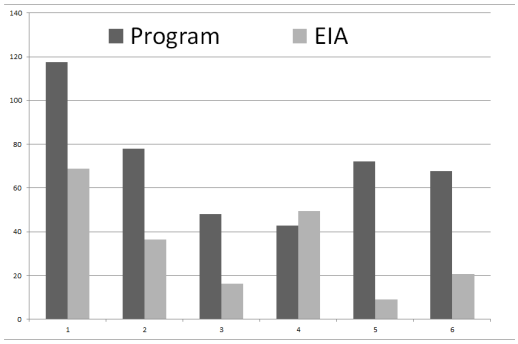


Figure 9: Average time per candidate in seconds to spot the problem in our 6 code snippets. The left bars in darker color refer to the average time without visual assistance, the right bars in brighter color show the average time with assistance.

Figure 9 shows the average time our candidates took per question with and without visual assistance. In general, our participants performed significantly better when given visual assistance than when they were not. The one exception is Experiment 4. In this experiment, we showed a procedure from Freemind, where a local variable is initialized to `null`, and then it is checked at three different locations if this variable is `null`, causing all three else blocks to be unreachable. This was the only experiment where we highlighted multiple lines in the same program, which caused confusion.

For all other experiments, our visual assistance based on error invariant automata helped the candidates to spot the problem more quickly. We observed that the candidates got faster from one experiment to another, as they got used to the patterns of code inconsistencies. This corresponds with the feedback that we got from our candidates that they are not used to looking for inconsistencies, but after they understood the problem found it easier to find the relevant statements. Hence, for our future experiments, we plan to do several training rounds with the candidates.

6.3 Threats to Validity

There are several threats to validity in this study. The first is that the participants may not be representative programmers, as they were selected based on their availability rather than on statistically meaningful criteria. This, combined with the small sample size used, makes it difficult to make any statistically rigorous claims based on our data. However, the results are extremely promising and we will continue to further evaluate with larger sample sizes as tools to detect inconsistent code become more mature.

Another threat to validity is the selection of the inconsistent code to test. Our code snippets are taken from real world programs, but we cannot say that they represent relevant cases of inconsistent code. So far, there are no studies about the different classes of inconsistent code that occur in real world programs and their frequency of occurrence. Hence, with improving tools to detect inconsistent code that will find more and different instances of inconsistencies, we will have to re-evaluate our approach. It is possible that larger code bases may be more difficult to analyze. However, experience with inconsistent code detection tools such as Joogie [2] suggest that function local reasoning is sufficient to detect such errors, which provides a relatively small bound on the size of code we need to analyze per instance. Modern SMT solvers such as CVC4 [4], Z3 [11], and MathSAT [9], scale to large programs [29], so this is unlikely to pose a significant problem in practise.

The effectiveness of our reduction algorithm depends on the quality of the candidate error invariants generated by the decision procedure. In our experiments, we generated candidate invariants using interpolation. This requirement for effective interpolation may limit which SMT theories we can use, although this was not a problem with the set of programs we tested.

Finally, another threat to validity is the way we provide visual assistance to the programmer. In our experiments we use the approach shown in Figure 8. However, experimenting with other visualization techniques like animations might lead to very different results.

7. RELATED WORK

The problem of detecting inconsistent code has been studied quite extensively in the literature [6, 12, 13, 19, 21, 24, 24, 34]. Note that many of these papers use a slightly different terminology such as deviant code [13], doomed code [21], infeasible code [2], and fatal code [34]. However, all of these notions are identical to or subsumed by our notion of inconsist code. Several of these techniques have been implemented in actual tools, including Joogie [2], which is co-developed by one of the authors. These tools can identify inconsistent code in

programs. The code fragments that are extracted by these tools then serve as input to the algorithm presented in this paper.

Our localization algorithm works well in combination with static techniques for detecting inconsistent code because they ensure that an actual proof of inconsistency can be constructed automatically. There also exist various techniques that trade soundness for speed. For example, in [13] Engler et al. use dynamic pattern matching to identify contradicting believes about the value of program variables. A similar approach is implemented in Findbugs [23], which uses a fast but imprecise static analysis to detect bug patterns, including inconsistent assumptions. By looking for patterns instead of actually tracking the values of variables, these analysis become extremely fast and scalable but on the other hand may produce false warnings. Since our algorithm uses a sound static analysis technique to prove the existence of a code inconsistency, it can be used as an additional filter to eliminate such false warnings.

A common approach to fault localization is to compare failing with successful executions (e.g., [16,17,25,31,32,35–37]). However, these dynamic approaches are not suitable for explaining inconsistent code. Inconsistency is a property of all executions and not a single execution of a program. One therefore needs to apply static techniques that can reason symbolically about all executions.

One prominent static approach to fault localization in single error traces is implemented in Bug-Assist [26, 27]. Bug-Assist takes an error trace as input, translates it into a formula in first-order logic, and then computes a maximal satisfiable core to exclude statements from the trace that are not needed to reproduce the error. Similar to the approach presented in this paper, the Bug-Assist approach analyzes the proof of unsatisfiability to remove non-relevant facts from the program.

In our previous work on error invariants [8, 15], we localize errors by computing Craig interpolants from symbolic representations of error traces. If the computed interpolants are inductive for a portion of the error trace, then that part of the trace is irrelevant for the error and can be abstracted. In this work, we generalize this technique to consider not just a single path through the program, but arbitrarily many paths. This generalization then enables us to explain code inconsistencies.

Although the applications are quite different, our use of Craig interpolation for localization is inspired by invariant generation techniques in software verification (e.g., [1, 7, 14, 30]). The problem of detecting inconsistent code can be phrased as a verification problem. Invariant generation techniques can therefore also be used to increase the detection rate of code inconsistencies.

Further related to this work is the detection and explanation of vacuously true specifications in finite-state model checking (e.g., [18,28,33]). However, the techniques being used in this context are very different from the work in our paper. For example, in [18], multi-valued logic is used to find witnesses that explain a correctness proof in model checking. These witnesses are then used to reveal potential errors in a specification that is expressed in a temporal logic.

8. FUTURE WORK

We see two major themes of future work. First, we will investigate different ways of generating visual assistance from error invariant automata. Currently, we only hide statements that are not represented by edges in the automaton and show the invariants that hold in between. However, hiding code in an IDE might be confusing to some programmers. Further, in our examples the invariants have been very simple and thus it is easy to present them. We are still looking for examples where the prover generates more complex invariants and we will investigate to what extent they have to be processed to be useful to a programmer.

The other aspect of our future work is the problem of how to classify code inconsistencies. We expect that most programmers would find the detection of unreachable code and guaranteed errors useful. For other classes of inconsistent code this is less clear. For example, Joogie, which detects code inconsistencies in Java programs, does not operate directly on the source code, but on the byte code that is generated by the compiler. The tool can therefore report inconsistencies that are not immediately obvious to the programmer, e.g., the compiler may translate a boolean expression in the source code into a branch in the byte code. If the expression evaluates to the same constant for all executions, then one branch will be inconsistent. While such inconsistencies are interesting for compiler optimizations, they should probably not be reported to the programmer. In fact, such a redundant expression may be intensional to document the code implicitly.

9. CONCLUSION

We presented error invariant automata, as a model to explain inconsistent code. Our experiments indicate that error invariant automata can be used to provide useful visual assistance for programmers to spot the cause of inconsistencies in code. We have provided an algorithm that automatically computes such automata for a given inconsistent program. This work can be seen as a generalization of the fault localization technique that we presented in [8]. In particular, error invariant automata can also be used for fault localization on a single error trace and thus provide a general tool to assist programmers in debugging.

10. REFERENCES

[1] A. Albarghouthi, A. Gurfinkel, and M. Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In *VMCAI*, volume 7148 of *LNCS*, pages 39–55. Springer, 2012.

[2] S. Arlt and M. Schäf. Joogie: Infeasible code detection for java. In *CAV'12*, volume 7358 of *LNCS*, pages 767–773. Springer, 2012.

[3] J. Baker. Experts battle £192bn loss to computer bugs. `http://www.cambridge-news.co.uk/Education/Universities/Experts-battle-192bn-loss-to-computer-bugs-18122012.htm`, December 2012. Accessed 2013-06-29.

[4] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV'11*, LNCS, pages 171–177. Springer, 2011.

[5] C. Barrett, A. Stump, C. Tinelli, S. Boehme, D. Cok, D. Deharbe, B. Dutertre, P. Fontaine, V. Ganesh, A. Griggio, J. Grundy, P. Jackson, A. Oliveras, S. Krstić, M. Moskal, L. D. Moura, R. Sebastiani, T. D. Cok, and J. Hoenicke. The SMT-LIB Standard: Version 2.0, 2010.

[6] C. Bertolini, M. Schäf, and P. Schweitzer. Infeasible code detection. In *VSTTE'12*, pages 310–325. Springer, 2012.

[7] I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim. Slicing abstractions. *Fundam. Inf.*, 89(4):369–392, Dec. 2008.

[8] J. Christ, E. Ermis, M. Schäf, and T. Wies. Flow-sensitive fault localization. In *VMCAI'13*, volume 7737 of *LNCS*, pages 189–208. Springer, 2013.

[9] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT 5 SMT solver. In *TACAS'13*, LNCS. Springer, 2013.

[10] W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, pages 269–285, 1957.

[11] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08*, pages 337–340. Springer, 2008.

[12] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *PLDI'07*, pages 435–445. ACM, 2007.

[13] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP '01*, pages 57–72. ACM, 2001.

[14] E. Ermis, J. Hoenicke, and A. Podelski. Splitting via interpolants. In *VMCAI*, pages 186–201, 2012.

[15] E. Ermis, M. Schäf, and T. Wies. Error Invariants. In *FM'12*, LNCS, pages 338–353. Springer, 2012.

[16] A. Groce. Error explanation with distance metrics. In *TACAS'04*, LNCS, pages 108–122. Springer, 2004.

[17] A. Groce and D. Kroening. Making the Most of BMC Counterexamples. *Electr. Notes Theor. Comput. Sci.*, 119(2):67–81, 2005.

[18] A. Gurfinkel and M. Chechik. How vacuous is vacuous? In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *LNCS*, pages 451–466. Springer, 2004.

[19] I. J. Hayes, C. J. Fidge, and K. Lermer. Semantic characterisation of dead control-flow paths. *IEE Proceedings—Software*, 148(6):175–186, 2001.

[20] M. Heizmann, J. Hoenicke, and A. Podelski. Refinement of trace abstraction. In *SAS'09*, pages 69–85, 2009.

[21] J. Hoenicke, K. R. Leino, A. Podelski, M. Schäf, and T. Wies. It's doomed; we can prove it. In *FM'09*, pages 338–353. Springer, 2009.

[22] J. Hoenicke, K. R. Leino, A. Podelski, M. Schäf, and T. Wies. Doomed program points. *Formal Methods in System Design*, 37(2), 2010.

[23] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *PASTE'07*, pages 9–14. ACM, 2007.

[24] M. Janota, R. Grigore, and M. Moskal. Reachability analysis for annotated code. In *SAVCBS'07*, pages 23–30. ACM, 2007.

[25] W. Jin and A. Orso. Bugredux: reproducing field failures for in-house debugging. In *ICSE'12*, pages 474–484. IEEE Press, 2012.

[26] M. Jose and R. Majumdar. Bug-Assist: Assisting Fault Localization in ANSI-C Programs. In *CAV'11*, LNCS, pages 504–509. Springer, 2011.

[27] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *PLDI '11*, pages 437–446. ACM, 2011.

[28] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. In *CHARME'99*, pages 82–96. Springer, 1999.

[29] A. Lal, S. Qadeer, and S. Lahiri. A solver for reachability modulo theories. In P. Madhusudan and S. Seshia, editors, *CAV'12*, volume 7358 of *LNCS*, pages 427–443. Springer, 2012.

[30] K. L. McMillan. Lazy abstraction with interpolants. In *CAV'06*, LNCS, pages 123–136. Springer, 2006.

[31] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: an approach for debugging evolving programs. In *ESEC/FSE'09*, pages 33–42. ACM, 2009.

[32] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39, 2003.

[33] M. Samer and H. Veith. On the notion of vacuous truth. In *LPAR'07*, pages 2–14. Springer, 2007.

[34] A. Tomb and C. Flanagan. Detecting inconsistencies via universal reachability analysis. In *ISSTA'12*, pages 287–297. ACM, 2012.

[35] T. Wang and A. Roychoudhury. Automated path generation for software fault localization. In *ASE*, pages 347–351. ACM, 2005.

[36] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE'02*, pages 1–10. ACM, 2002.

[37] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE*, pages 272–281. ACM, 2006.