Automated Error Diagnosis Using Abductive Inference [*]

Isil Dillig
Department of Computer Science
College of William & Mary
idillig@cs.wm.edu

Thomas Dillig
Department of Computer Science
College of William & Mary
tdillig@cs.wm.edu

Alex Aiken
Department of Computer Science
Stanford University
aiken@cs.stanford.edu

Abstract

When program verification tools fail to verify a program, either the program is buggy or the report is a false alarm. In this situation, the burden is on the user to manually classify the report, but this task is time-consuming, error-prone, and does not utilize facts already proven by the analysis. We present a new technique for assisting users in classifying error reports. Our technique computes small, relevant queries presented to a user that capture exactly the information the analysis is missing to either discharge or validate the error. Our insight is that identifying these missing facts is an instance of the *abductive inference problem* in logic, and we present a new algorithm for computing the smallest and most general abductions in this setting. We perform the first user study to rigorously evaluate the accuracy and effort involved in manual classification of error reports. Our study demonstrates that our new technique is very useful for improving both the speed and accuracy of error report classification. Specifically, our approach improves classification accuracy from 33% to 90% and reduces the time programmers take to classify error reports from approximately 5 minutes to under 1 minute.

Categories and Subject Descriptors F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Verification, Algorithms, Experimentation

Keywords Error diagnosis, abductive inference, static analysis

1. Introduction

Automated software verification systems perform sophisticated reasoning to prove the correctness of program properties, such as validity of assertions, memory safety, and lack of run-time exceptions. If the tool concludes that the program satisfies the property, then all is well. On the other hand, if the tool claims that the program may violate the property, there are two possibilities:

- either the program is indeed buggy, or
- the report generated by the tool is a false alarm

Since program verification is, in general, undecidable, false alarms are inevitable no matter how sophisticated the reasoning performed by the analysis tool. Thus, when a static analysis fails to verify the correctness of a program, the burden is on the user to decide whether the report generated by the tool is a genuine bug or a false alarm. This situation is undesirable for multiple reasons:

- Manual report classification is a very time-consuming task that requires expertise, patience, and understanding of the program.
- Even if the cause of the report is very simple (e.g., the analysis could not establish a single, local fact), there is no bound on the amount of work the user must do to discover this cause. Effectively, the user must repeat all of the successful reasoning the tool performed just to discover where it became stuck.
- Manual classification of error reports is error-prone: As corroborated by our experimental results, users often misclassify false alarms as genuine bugs, or much more problematically, genuine bugs as false alarms.

We believe that the difficulty of manual report classification is a major remaining impediment to the wide-spread adoption of even the most precise and state-of-the-art static analysis tools. As reported in a recent article by Bessey et al., the overwhelming experience in the field is that users who do not understand error reports become frustrated and ignore those reports [1].

In this paper, we propose a new technique for assisting users in classifying error reports when automated static analyses fail to verify a program. Our technique allows verification tools to interact with users by computing small, relevant queries that capture exactly the facts that the analysis is missing to either verify the program or prove the existence of a real error. These queries are then presented to a user who decides whether the answer to the query is yes or no.

The first kind of query we compute is a *proof obligation query*, which asks the user whether a property P is a program invariant. These queries have the key characteristic that, if P is indeed an invariant, then the program is error-free. Furthermore, proof obligation queries are as simple and as general as possible: they capture exactly, and only, the information the analysis is missing to verify the program. Our analysis also computes a second kind of query, called a *failure witness query*, which asks the user whether a property P can arise in some execution. The key feature of a witness query is that, if P can indeed occur in some execution, the program must have an error. Furthermore, our technique computes simplest and most general failure witness queries: they capture precisely the facts the analysis is missing to be certain that the program is buggy.

Our approach to error report classification is semi-automatic: It computes simple and relevant queries to resolve the error report, but trusts a user to correctly answer them. Since this technique is only useful when an automated analysis has been unable to verify the program, it is inevitable to ask the user for some help. In fact, the current best practice asks the user to do all the work,

[*] This work was supported by NSF grants CCF-0915766 and CCF-0702681.

as manual report classification is 100% trusted user information. Our approach minimizes -in a precise sense- the amount of trusted information the user must supply, both simplifying and speeding up her job while also increasing the accuracy of the classification.

There are four salient features underlying our approach:

- We describe a static analysis algorithm that makes explicit all possible sources of incompleteness in static analyzers and uses facts inferred by any other program analysis technique.
- This representation allows the analysis to not only prove the absence of errors but also to prove their presence, i.e., in some cases, the analysis can decide that the program must be buggy.
- When the analysis can neither discharge nor validate a potential error, it computes small, relevant queries to be presented to the user, called proof obligations and failure witnesses, that ultimately allow the analysis to discharge or validate an error.
- If the user's answer to a proof obligation query is "yes", the analysis can discharge the potential error. Similarly, if the user's answer to a failure witness query is "yes", the analysis can prove the existence of an error. Otherwise, our technique computes new proof obligation and failure witness queries, until the error can either be discharged or validated. Queries are posed to the user in order of increasing cost, meaning that the user is asked the questions that should be the easiest to answer first. Since state-of-the-art static analyzers typically lose information in few places, it is easier for the user to answer simple queries about a handful of missing analysis facts than to manually inspect and understand an entire program.

1.1 Informal Overview

We illustrate key features of our approach with a simple example:

```
void foo(int flag, unsigned int n)
{
1:   int k = 1;
2:   if(flag) k = n*n;

3:   int i = 0, j = 0;
4:   while(i <= n) {
5:      i++;
6:      j+=i;
7:   }
8:  int z = k + i + j;
9:  assert(z > 2*n);
}
```

This code snippet contains an assertion at line 9 which is guaranteed to hold [1]. Assume that a static analysis fails to verify this program and reports a potential assertion failure for line 9. Of course, just because the tool reports a false alarm does not mean it has inferred no useful information about the program. For instance, the analysis may have inferred that `k`'s value after line 2 is at least 0 (as `n*n` is always non-negative), and that the value of `i` after the loop is at least 0 and greater than `n`. Our goal is to utilize these facts inferred by the analysis to assist the user in classifying this report.

The first key idea underlying our technique is to model potential sources of incompleteness in static analyses using *abstraction variables*. In this example, since the exact values of `i` and `j` are unknown after the loop, our technique introduces abstraction variables α_i and α_j to model the unknown values of `i` and `j` at line 7. Similarly, assuming the analysis does not reason precisely about non-linear arithmetic, our technique introduces abstraction variable α_{n*n} to model the unknown result of the multiplication at line 2.

Next, our technique utilizes invariants obtained by any static analysis to infer restrictions on abstraction variables. For example, since an existing static analyzer has inferred that `n*n` is always non-negative, we have the side condition:

$$\alpha_{n*n} \geq 0$$

Similarly, since we know that the value of `i` after the loop is greater than `n` and at least 0, the abstraction variable α_i must satisfy:

$$\alpha_i \geq 0 \ \land \alpha_i > n$$

Thus, for line 9, our technique obtains the following invariant $\mathcal{I}$:

$$\mathcal{I} = (\alpha_{n*n} \geq 0 \ \land \ \alpha_i \geq 0 \ \land \ \alpha_i > n \ \land \ n \geq 0)$$

Here, the first three predicates in $\mathcal{I}$ express invariants obtained from an existing static analysis, and the last predicate $n \geq 0$ expresses that unsigned variables are non-negative.

Now that we can explicitly name each potential source of imprecision, our technique computes an exact symbolic value set for each expression in the program. For instance, the value of variable `k` at line 2 is represented by the symbolic value set:

$$\{(1, \neg\text{flag}), (\alpha_{n*n}, \text{flag})\}$$

meaning that `k` has value 1 under constraint ¬flag and the unknown value represented by α_{n*n} under condition flag. Similarly, the values of variables `i` and `j` at line 7 after the loop are represented by the singleton value sets $\{(\alpha_i, \textit{true})\}$ and $\{(\alpha_j, \textit{true})\}$. Since `z` is obtained by adding `k`, `i`, and `j`, we can use symbolic value sets of `k`, `i`, and `j` to compute the symbolic value of `z`, given by:

$$\{(1 + \alpha_i + \alpha_j, \neg\text{flag}), (\alpha_{n*n} + \alpha_i + \alpha_j, \text{flag})\}$$

Since the assertion at line 9 succeeds if the value of `z` is greater than `2*n`, we can use the symbolic value set for `z` to compute the condition ϕ under which the assertion holds:

$$\phi \ = \ (1{+}\alpha_i{+}\alpha_j > 2{*}n \land \neg\text{flag}) \lor (\alpha_{n*n}{+}\alpha_i{+}\alpha_j > 2{*}n \land \text{flag})$$

Observe that the assertion is verified if the invariants $\mathcal{I}$ entail the success condition ϕ of the assertion, i.e.,:

$$\mathcal{I} \models \phi$$

Similarly, we know that the assertion is guaranteed to fail if:

$$\mathcal{I} \models \neg\phi$$

In this example, since neither $\mathcal{I} \models \phi$ nor $\mathcal{I} \models \neg\phi$, the analysis cannot discharge or validate the potential assertion failure.

Thus, our goal is to identify the possible facts the analysis is missing to either discharge or validate the error and ask the user whether these facts indeed hold. Our insight is that this problem is an instance of the *abductive inference* problem in logic, where the goal is to find an explanatory hypothesis for a desired outcome. Formally, given known facts F and a desired outcome O, an abductive inference problem is to find an explanation E such that:

$$F \land E \models O \text{ and } \text{SAT}(F \land E)$$

In other words, the abduction E is consistent with known facts F, and together with F, is sufficient to explain O.

In our setting, the desired outcome is to either prove the absence of an error or to validate its existence. Thus, we solve two abductive inference problems, one to infer the missing information necessary to verify the program, and one to find the missing facts to validate the presence of an error. Specifically, to prove the absence of an error, we compute a *proof obligation* Γ by solving the following abductive inference problem:

$$\mathcal{I} \land \Gamma \models \phi \ \text{ and } \ \text{SAT}(\mathcal{I} \land \Gamma)$$

In other words, a proof obligation Γ is consistent with program invariants $\mathcal{I}$ and, along with $\mathcal{I}$, is sufficient to discharge the error. Furthermore, we are not interested in just any solution to the abductive inference problem; what we want is a *weakest minimum proof*

[1] In this example, we assume there are no integer overflows.

obligation so that the queries presented to the user are as small and as general as possible. In this example, using the techniques of Section 4, we compute a weakest minimum proof obligation as:

$$\alpha_j \geq n$$

Thus, if the user can show that `j >= n` always holds at line 7, the analysis can discharge the potential error.

Dually, to prove the presence of an error, we compute a *failure witness* Υ by solving a second abductive inference problem:

$$\mathcal{I} \land \Upsilon \models \neg\phi \text{ and } \mathrm{SAT}(\mathcal{I} \land \Upsilon)$$

In other words, a failure witness is also consistent with program invariants, and if Υ holds in *some* execution, we know that the program must have an error. Again, we are not interested in any solution to the abductive inference problem; instead, we want to find a *weakest minimum failure witness* to ensure that the queries we compute are as small and as general as possible. For our running example, techniques described in Section 4 yield the following weakest minimum failure witness:

$$\neg\text{flag} \land \alpha_i + \alpha_j < 0$$

Thus, if the user can show that `i+j < 0` is possible at line 7 in an execution where `!flag` holds, the analysis can validate the error.

After computing weakest minimum proof obligations and failure witnesses, our technique then decides whether it is more promising to try to discharge the error or to validate it by comparing the costs of the proof obligation and failure witness. In this example, our technique decides that it is more promising to try to discharge the error and therefore queries the user whether `j>=n` is a program invariant at line 7.

Since it is easy to show that `j >= n` always holds at line 7, the analysis can immediately discharge the error. Observe that, although the assertion condition in this example requires reasoning about values of multiple variables `i`, `j`, `k`, and `z`, our technique can take advantage of facts already established by the analysis to compute a simple and intuitive query involving only variable `j`.

1.2 Organization and Contributions

The rest of this paper is organized as follows: Section 2 defines a simple language in which we formalize our technique. Section 3 describes a static analysis that makes explicit potential sources of imprecision and performs symbolic value propagation. Section 4 defines weakest minimum proof obligations and failure witnesses, describes a technique for computing them, and presents an iterative algorithm for validating or discharging error reports. Section 5 describes our implementation; Section 6 presents experimental results. Finally, Section 7 surveys related work, and Section 8 concludes. In summary, this paper makes the following contributions:

- We present a new technique for semi-automatic report classification when static analyzers are unable to verify the program.
- We define weakest minimum proof obligations and failure witnesses as a technical characterization of simple, relevant facts useful for resolving error reports.
- We present the problem of computing proof obligations and failure witnesses as an abductive inference problem and give a new algorithm for computing abductions in this setting.
- We show how proof obligations and failure witnesses can be used to interact with users until a potential error is resolved.
- We perform a user study to evaluate our technique. Our results show that the new technique is very useful both for improving the time required to classify error reports as well as for dramatically improving the accuracy of report classification. Specifically, our approach improves classification accuracy from 33% to 90% and reduces the time programmers take to classify error reports from approximately 5 minutes to under 1 minute.

$$\frac{}{S \vdash v : S(v)} \qquad \frac{}{S \vdash c : c} \qquad \frac{\begin{array}{l}\oplus \in \{+,-,*\}\\ S \vdash e_1 : c_1 \;\; S \vdash e_2 : c_2\end{array}}{S \vdash e_1 \oplus e_2 : c_1 \oplus c_2}$$

$$\frac{\begin{array}{l} S \vdash e_1 : c_1 \;\; S \vdash e_2 : c_2 \\ b = \begin{cases} \text{true} & \text{if } c_1 \oslash c_2 \\ \text{false} & \text{otherwise} \end{cases}\end{array}}{S \vdash e_1 \oslash e_2 : b} \qquad \frac{\begin{array}{l}\text{lop} \in \{\land, \lor\}\\ S \vdash p_1 : b_1 \;\; S \vdash p_2 : b_2\end{array}}{S \vdash p_1 \;\text{lop}\; p_2 : b_1 \;\text{lop}\; b_2}$$

$$\frac{S \vdash p : b}{S \vdash \neg p : \neg b} \qquad \frac{S \vdash e : c}{S \vdash v = e : S[c/v]} \qquad \frac{}{S \vdash \texttt{skip} : S}$$

$$\frac{S \vdash p : \text{true} \quad S \vdash s_1 : S_1}{S \vdash \texttt{if}(p)\;\texttt{then}\;s_1\;\texttt{else}\;s_2 : S_1} \qquad \frac{S \vdash p : \text{false} \quad S \vdash s_2 : S_2}{S \vdash \texttt{if}(p)\;\texttt{then}\;s_1\;\texttt{else}\;s_2 : S_2}$$

$$\frac{S \vdash s_1 : S_1 \quad S_1 \vdash s_2 : S_2}{S \vdash s_1; s_2 : S_2} \qquad \frac{\begin{array}{l} S \vdash p : \text{true} \quad S \vdash s : S' \\ S' \vdash \text{loop}^{\rho}(p)\{s\} : S'' \end{array}}{S \vdash \text{loop}^{\rho}(p)\{s\} : S''}$$

$$\frac{S \vdash \text{loop}^{\rho}(p)\{s\} : S' \quad S' \vdash p' : \text{true}}{S \vdash \texttt{while}^{\rho}(p)\{s\}[@p'] : S'} \qquad \frac{S \vdash p : \text{false}}{S \vdash \text{loop}^{\rho}(p)\{s\} : S}$$

$$\frac{\begin{array}{l} S = [c_1/a_1, \ldots, c_k/a_k][0/v_1, \ldots, 0/v_n] \\ S \vdash s : S' \quad S' \vdash p : b \end{array}}{\vdash \lambda\vec{a}.(\texttt{let}\;\vec{v}\;\texttt{in}\;(s; \texttt{check}(p)))(c_1, \ldots c_k) : b}$$

Figure 1. Operational semantics of the language from Section 2

2. Language

In this section, we present a simple programming language that we use to formalize our technique:

$$\begin{array}{lll}
\text{Program } P & := & \lambda\vec{a}.\;(\texttt{let}\;\vec{v}\;\texttt{in}\;(s; \texttt{check}(p))) \\
\text{Statement } s & := & v = e \mid \texttt{skip} \mid s_1; s_2 \\
 & & \mid \texttt{if}(p)\;\texttt{then}\;s_1\;\texttt{else}\;s_2 \\
 & & \mid \texttt{while}^{\rho}(p)\{s\}[@p']? \\
\text{Expression } e & := & v \mid c \mid c * e \mid e_1 \oplus e_2 \;\; (\oplus \in \{+,-\}) \\
\text{Predicate } p & := & e_1 \oslash e_2 \;\; (\oslash \in \{<,>,=\}) \\
 & & \mid p_1 \land p_2 \mid p_1 \lor p_2 \mid \neg p
\end{array}$$

In this language, a program with inputs $\vec{a}$ and local variables $\vec{v}$ consists of a statement s and a $\texttt{check}(p)$ statement, which checks whether predicate p evaluates to *true*. The program evaluates to *true* if predicate p holds, and to *false* otherwise. We say that an execution of a program P is *error-free* if P evaluates to *true* in this execution, and *buggy* otherwise. Similarly, we say that program P is error-free if P evaluates to *true* in *all* possible executions, and buggy if P evaluates to *false* in some execution.

Statements in this language consist of assignments ($v = e$, where v is bound in the `let` statement) , `skip` statements, sequencing ($s_1; s_2$), `if` statements, and `while` loops labeled with unique identifiers ρ. Observe that `while` loops can be optionally tagged with annotations of the form $@p'$, where predicate p' corresponds to invariants that hold after the loop. For the purposes of this paper, these invariants may be obtained from any automatic sound static analysis technique, such as abstract interpretation or predicate abstraction. We say that a program P is *analyzed* if all while loops are annotated with sound post-conditions using a static analyzer.

Expressions include integer variables v, integer constants c, and arithmetic operations. This language has an expressive family of predicates which include comparisons between expressions, as well as conjunction, disjunction, and negation. The operational semantics is given in Figure 1. We omit function calls from this

$$\frac{\begin{array}{c}\theta_1 = \{(\pi_1,\phi_1),\ldots,(\pi_k,\phi_k)\}\\ \theta_2 = \{(\pi'_1,\phi'_1),\ldots,(\pi'_n,\phi'_n)\}\\ \theta = \bigcup_{ij}((\pi_i \oplus \pi'_j),(\phi_i \wedge \phi'_j))\end{array}}{\vdash \theta_1 \oplus \theta_2 : \theta}$$

$$\frac{\begin{array}{c}\theta_1 = \{(\pi_1,\phi_1),\ldots,(\pi_k,\phi_k)\}\\ \theta_2 = \{(\pi'_1,\phi'_1),\ldots,(\pi'_n,\phi'_n)\}\\ \phi = \bigvee_{ij}((\pi_i \oslash \pi'_j) \wedge \phi_i \wedge \phi'_j)\end{array}}{\vdash \theta_1 \oslash \theta_2 : \phi}$$

$$\frac{\theta' = \bigcup_{(\pi_i,\phi_i)\in\theta}(\pi_i,(\phi_i \wedge \phi))}{\vdash \theta \wedge \phi : \theta'}$$

Figure 2. Operations on symbolic value sets

$$\frac{}{\mathbb{S} \vdash v : \mathbb{S}(v)} \qquad \frac{}{\mathbb{S} \vdash c : (c, \mathit{true})} \qquad \frac{\begin{array}{c}\oplus \in \{+,-,*\}\\ \mathbb{S} \vdash e_1 : \theta_1 \quad \mathbb{S} \vdash e_2 : \theta_2\end{array}}{\mathbb{S} \vdash e_1 \oplus e_2 : \theta_1 \oplus \theta_2}$$

Figure 3. Symbolic evaluation rules for expressions

language, as the issues raised by function calls are orthogonal and not necessary for understanding our technique. However, our implementation is an interprocedural analysis (see Section 5).

3. Analysis

In this section, we describe a static analysis that is performed after a verification tool already analyzed the program, inferring the $@p'$ annotations on `while` loops and reporting a potential error. Our analysis, which is a prerequisite for computing relevant queries to classify the error report, has the following key characteristics:

- Values of program variables are represented by symbolic expressions consisting of constants and *analysis variables*.
- There are two kinds of analysis variables: *input variables* ν represent unknown values of program inputs, and *abstraction variables* α model unknown values of variables due to an imprecision in the analysis. For instance, abstraction variables represent values that may be unknown after loops.
- The analysis uses facts inferred by other analyzers, which are annotated using the $@p'$ construct on loops. These invariants are used to constrain values of abstraction variables.
- The only source of imprecision in this analysis is loops; it performs exact symbolic value propagation on loop-free code.

Our static analysis is described in Figures 2, 3, 4, and 5. Values of program variables are represented as symbolic expressions π:

$$\pi := \nu \mid \alpha \mid c \mid \pi_1 + \pi_2 \mid \pi_1 - \pi_2 \mid c * \pi$$

Besides input variables ν and abstraction variables α, symbolic expressions are integer constants c, addition or subtraction of symbolic expressions, and linear multiplication. Since the only imprecision of the static analysis for the simple language from Section 2 is due to loops, abstraction variables are only used to model the (potentially) unknown values of program variables after loops.

In the analysis, environment $\mathbb{S}$ maps program variables to *symbolic value sets* θ:

$$\theta := 2^{(\pi,\phi)}$$

where π is a symbolic expression and ϕ is a constraint. Since variables may have different symbolic values on different program paths, the constraint ϕ allows the analysis to keep values on different paths separate. For concreteness, constraints in this paper are in the theory of of linear arithmetic over integers.

$$\frac{\begin{array}{c}\mathbb{S} \vdash e_1 : \theta_1\\ \mathbb{S} \vdash e_2 : \theta_2\end{array}}{\mathbb{S} \vdash e_1 \oslash e_2 : \theta_1 \oslash \theta_2} \qquad \frac{\begin{array}{c}\text{lop} \in \{\wedge,\vee\}\\ \mathbb{S} \vdash p_1 : \phi_1\\ \mathbb{S} \vdash p_2 : \phi_2\end{array}}{\mathbb{S} \vdash p_1 \ \text{lop}\ p_2 : \phi_1 \ \text{lop}\ \phi_2} \qquad \frac{\mathbb{S} \vdash p : \phi}{\mathbb{S} \vdash \neg p : \neg\phi}$$

Figure 4. Symbolic evaluation rules for predicates

$$\frac{\begin{array}{c}\mathbb{S} \vdash e : \theta\\ \mathbb{S}' = \mathbb{S}[\theta/v]\end{array}}{\mathbb{S},\mathcal{I} \vdash v = e : \mathbb{S}',\mathcal{I}} \qquad \frac{}{\mathbb{S},\mathcal{I} \vdash \texttt{skip} : \mathbb{S},\mathcal{I}} \qquad \frac{\begin{array}{c}\mathbb{S},\mathcal{I} \vdash s_1 : \mathbb{S}_1,\mathcal{I}_1\\ \mathbb{S}_1,\mathcal{I}_1 \vdash s_2 : \mathbb{S}_2,\mathcal{I}_2\end{array}}{\mathbb{S},\mathcal{I} \vdash s_1; s_2 : \mathbb{S}_2,\mathcal{I}_2}$$

$$\frac{\begin{array}{c}\mathbb{S} \vdash p : \phi\\ \mathbb{S},\mathcal{I} \vdash s_1 : \mathbb{S}_1,\mathcal{I}_1 \quad \mathbb{S},\mathcal{I} \vdash s_2 : \mathbb{S}_2,\mathcal{I}_2\\ \mathbb{S}' = (\mathbb{S}_1 \wedge \phi) \sqcup (\mathbb{S}_2 \wedge \neg\phi)\\ \mathcal{I}' = ((\phi \Rightarrow \mathcal{I}_1) \wedge (\neg\phi \Rightarrow \mathcal{I}_2))\end{array}}{\mathbb{S},\mathcal{I} \vdash \texttt{if}(p)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 : \mathbb{S}',\mathcal{I}'}$$

$$\frac{\mathbb{S}' = \mathbb{S}[(\alpha_1^\rho, \mathit{true})/v_1,\ldots,(\alpha_k^\rho, \mathit{true})/v_k](\vec{v}\ \text{modified in}\ s)}{\mathbb{S},\mathcal{I} \vdash \text{loop}^\rho(p)\{s\} : \mathbb{S}',\mathcal{I}}$$

$$\frac{\mathbb{S},\mathcal{I} \vdash \text{loop}^\rho(p)\{s\} : \mathbb{S}',\mathcal{I} \quad \mathbb{S}' \vdash p' : \phi}{\mathbb{S},\mathcal{I} \vdash \texttt{while}^\rho(p)\{s\}[@p'] : \mathbb{S}',\mathcal{I} \wedge \phi}$$

$$\frac{\begin{array}{c}\mathbb{S} = [(\nu_1, \mathit{true})/a_1,\ldots,(\nu_k, \mathit{true})/a_k]\\ \mathbb{S}' = \mathbb{S}[(0, \mathit{true})/v_1,\ldots,(0, \mathit{true})/v_n]\\ \mathbb{S}', \mathit{true} \vdash s : \mathbb{S}'',\mathcal{I} \quad \mathbb{S}'' \vdash p : \phi\end{array}}{\vdash \lambda\vec{a}.(\texttt{let}\ \vec{v}\ \texttt{in}\ (s; \texttt{check}(p))) : \mathcal{I},\phi}$$

Figure 5. Transformers for the static analysis

Figure 2 defines some useful operations on symbolic value sets. The first rule $\theta_1 \oplus \theta_2$ describes how to perform arithmetic operations on symbolic value sets, where $\oplus$ ranges over $+,-,*$ and where θ is the symbolic value set representing the result of the arithmetic operation. The second rule $\theta_1 \oslash \theta_2$ (where $\oslash$ is $<,>$, or $=$) describes how to compare value sets θ_1 and θ_2. The result is a constraint ϕ, which describes the condition under which θ_1 is less than, greater than, or equal to θ_2. Finally, the last rule in this figure defines what it means to conjoin a constraint ϕ with a value set θ.

Figures 3 and 4 describe symbolic evaluation of expressions and predicates, and are direct analogues of the corresponding operational semantics rules in Figure 1, with integer constants replaced by symbolic value sets and boolean constants with constraints.

The first six rules in Figure 5 describe the transformers for statements and derive judgements of the form:

$$\mathbb{S},\mathcal{I} \vdash s : \mathbb{S}',\mathcal{I}'$$

Since statements may modify values of program variables, each statement may modify $\mathbb{S}$ and produce a new symbolic store $\mathbb{S}'$. The constraints $\mathcal{I}$ and $\mathcal{I}'$ describe invariants about abstraction variables obtained from annotations on `while` loops.

The first three rules in Figure 5 are self-explanatory and are straightforward analogues of their concrete counterparts from Figure 1. In the rule for `if` statements, facts that are obtained by analyzing the `then` branch s_1 (resp. `else` branch s_2) only hold under the conditional p (resp. $\neg p$). Therefore, we first compute the symbolic evaluation of conditional p as ϕ and conjoin ϕ to all facts obtained in the `then` branch and $\neg\phi$ to facts obtained in the `else` branch. In this rule, conjunction on symbolic stores is defined as:

$$\forall v \in \mathit{dom}(\mathbb{S}).\ \ (\mathbb{S} \wedge \phi)(v) = \{(\pi_j, \phi_j \wedge \phi) \mid (\pi_j,\phi_j) \in \mathbb{S}(v)\}$$

This rule also uses an (exact) join operation $\sqcup$ on symbolic stores, defined as:

$$\begin{array}{l}(\pi,\phi) \in \mathbb{S}_1(v) \wedge (\pi,\phi') \in \mathbb{S}_2(v) \Rightarrow (\pi, \phi \vee \phi') \in (\mathbb{S}_1 \sqcup \mathbb{S}_2)(v)\\ (\pi,\phi) \in \mathbb{S}_i(v) \wedge (\pi,\_) \notin \mathbb{S}_j(v) \Rightarrow (\pi,\phi) \in (\mathbb{S}_1 \sqcup \mathbb{S}_2)(v)\end{array}$$

Thus, in the rule for `if` statements, the symbolic store $\mathbb{S}'$ is obtained by conjoining ϕ to all bindings in $\mathbb{S}_1$, $\neg\phi$ to all bindings in $\mathbb{S}_2$, and then combining the resulting symbolic stores. Similarly, observe that a new invariant $\mathcal{I}'$ is obtained as:

$$(\phi \Rightarrow \mathcal{I}_1) \wedge (\neg\phi \Rightarrow \mathcal{I}_2)$$

because invariants obtained from the `then` branch only hold under ϕ, while invariants obtained in the `else` branch hold under $\neg\phi$.

The rule for `while` loops does not infer any loop invariants, but instead uses loop postconditions already inferred by other analyses annotated using the $@p'$ construct. For this reason, we first bind the values of all variables $v_1, \ldots, v_k$ modified in the loop body to fresh abstraction variables $\alpha_1^\rho, \ldots, \alpha_k^\rho$ in the $\text{loop}^\rho(p)\{s\}$ helper rule. Now, to utilize any known invariants on the values of $v_1, \ldots, v_k$, we symbolically evaluate the annotated invariant $@p'$ as constraint ϕ and obtain a new invariant $\mathcal{I} \wedge \phi$ after analyzing the `while` loop. Observe that since the annotation $@p'$ is symbolically evaluated under store $\mathbb{S}'$ and since loop modified variables are bound to their corresponding abstraction variables α in $\mathbb{S}'$, the new invariant ϕ constrains values of abstraction variables introduced in this loop.

The last rule in Figure 5 describes the analysis of the whole program. The initial store is obtained by binding the arguments $a_1, \ldots, a_k$ to analysis variables $\nu_1, \ldots, \nu_k$ and initializing local variables to 0, as specified by the concrete semantics. The result of the analysis is a pair of constraints $\mathcal{I}, \phi$ where $\mathcal{I}$ represents all known invariants on the abstraction variables and ϕ represents the condition under which the program evaluates to *true*.

LEMMA 1. *Let P be any program such that $\vdash P : \mathcal{I}, \phi$. If $\mathcal{I} \models \phi$, then P is error-free (i.e., evaluates to* true *in all executions).*

PROOF 1. *Given in the extended version of this paper [2].*

LEMMA 2. *Let P be any program such that $\vdash P : \mathcal{I}, \phi$. If $\mathcal{I} \models \neg\phi$, then P must be buggy.*

PROOF 2. *Given in the extended version of this paper [2].*

At first glance, the condition in Lemma 2 may seem too strong as the satisfiability of the formula $\mathcal{I} \wedge \neg\phi$ is sufficient to establish that the program *may* be buggy. However, here, we are interested in proving that the program *must* be buggy, which we can only guarantee if $\mathcal{I} \models \neg\phi$.

EXAMPLE 1. *Consider the following program with annotated invariants obtained from a static analysis:*

```
λa₁, a₂.(let k, i, j, z in (
  if(a₂ > 0) then k = a₂ else k = 1;
  while¹(i < a₂ + 1){
    i = i + 1;
    j = j + i;
  }@[i > −1 ∧ i > a₂]
  if(a₁ > 0) then z = k + i + j else z = 2 ∗ a₂ + 1;
  check(z > 2 ∗ a₂)
))
```

Applying the analysis rules described in this section, we obtain:

$$\mathcal{I} = \alpha_i^1 \geq 0 \wedge \alpha_i^1 > \nu_2$$

and success condition ϕ:

$$\begin{array}{ll} & (\nu_2 + \alpha_i^1 + \alpha_j^1 > 2\nu_2 \wedge \nu_2 > 0 \wedge \nu_1 > 0) \\ \vee & (1 + \alpha_i^1 + \alpha_j^1 > 2\nu_2 \wedge \nu_2 \leq 0 \wedge \nu_1 > 0) \\ \vee & (2\nu_2 + 1 > 2\nu_2 \wedge \nu_1 \leq 0) \end{array}$$

Since neither $\mathcal{I} \models \phi$ nor $\mathcal{I} \models \neg\phi$, the potential error can neither be discharged nor validated.

4. Query Guided Error Diagnosis

As Lemmas 1 and 2 from Section 3 show, our analysis can sometimes prove that a program is error-free or definitely buggy. Unfortunately, in many cases, the outcome of the analysis is not definite: It can neither discharge the potential error nor prove that the program is buggy. In this case, the error report must be inspected by a human who classifies the report as a genuine bug or false alarm. As mentioned in Section 1, this situation is problematic because manual classification is time-consuming, difficult, and error-prone.

In this section, we describe a novel technique for computing small, relevant queries that systematically allow definite classification of error reports. More technically, our goal is to identify relevant facts such that, along with these facts, either Lemma 1 applies (in which case, the report is definitely a false alarm) or Lemma 2 holds (in which case the report is confirmed to be a genuine bug).

4.1 Weakest Minimum Queries to Discharge Error

Suppose that, for a given program P, the analysis from Section 3 derives the judgment

$$\vdash P : \mathcal{I}, \phi$$

but neither Lemma 1 nor Lemma 2 applies. To be able to discharge the error, what we need to do is to find a formula Γ such that, along with $\mathcal{I}$, Γ implies the success condition ϕ. Specifically, we need to solve the following abductive inference problem:

DEFINITION 1. **(Proof Obligation)** *Given known facts $\mathcal{I}$ and success condition ϕ, a* proof obligation *is a formula Γ such that:*

$$\Gamma \wedge \mathcal{I} \models \phi \quad \textit{and} \quad \textit{SAT}(\Gamma \wedge \mathcal{I})$$

Thus, a proof obligation Γ is consistent with known program invariants $\mathcal{I}$, and together with $\mathcal{I}$ allows us to prove the success condition ϕ. Observe that there is always a trivial proof obligation, namely $\Gamma = \phi$. In other words, a trivial way to discharge the error is simply to ask the user to show the success condition ϕ itself! Therefore, we are not interested in just any proof obligation, but simple, intuitive proof obligations that are local and easy for the user to decide. To make precise what we mean by simple proof obligations, it is necessary to assign a cost to each formula Γ. In this paper, we assign costs to proof obligations as follows:

DEFINITION 2. **(Cost of Proof Obligation)** *Let Γ be a proof obligation query for $\mathcal{I}, \phi$, and let Π_p be a mapping from variables to costs such that $\Pi_p(\alpha) = 1$ for abstraction variable α and $\Pi_p(\nu) = |\textit{Vars}(\phi) \cup \textit{Vars}(\mathcal{I})|$ for input variable ν. Then:*

$$\textit{Cost}(\Gamma) = \sum_{v \in \textit{Vars}(\Gamma)} \Pi_p(v)$$

The above cost function assigns cost 1 to each abstraction variable α and cost $|\text{Vars}(\phi) \cup \text{Vars}(\mathcal{I})|$ to each input variable ν. Of course, it is impossible to exactly capture the intuitiveness of a program fact with a mathematical formula, and any definition of query cost necessarily approximates the human effort required to decide that query. We choose to approximate the simplicity of a query as given by Definition 2 because it captures the following intuitions:

- Local facts are generally easier to decide for humans: Our cost function assigns a lower cost to formulas with fewer abstraction variables, because queries involving fewer variables require the user to reason about fewer sources of analysis imprecision.
- In general, it is undesirable to make assumptions about the program's execution environment (i.e., inputs in our language): For this reason, the cost function assigns a higher cost to formulas containing input variables ν. In fact, the intuition behind assigning cost $|\text{Vars}(\phi) \cup \text{Vars}(\mathcal{I})|$ to input variables ν is to ensure that it is more expensive to constrain a single input vs. all sources

of imprecision. In other words, if there is any way to discharge the error without constraining the program's execution environment, the analysis will ask those queries first.

While this cost definition could be refined, we found it to capture simplicity well in practice (see Section 6). Furthermore, the techniques we describe are agnostic to the definition of query cost; we only require that cost is a function of variables in the formula. Given a function that assigns costs to queries, we can now define a special kind of proof obligation that we are interested in computing:

DEFINITION 3. **(Weakest Minimum Proof Obligation)** *Given facts $\mathcal{I}$ and success condition ϕ, a* weakest minimum proof obligation *is a formula Γ such that:*

1. $\Gamma \wedge \mathcal{I} \models \phi$ *and* $SAT(\Gamma \wedge \mathcal{I})$
2. *For any other Γ' that satisfies (1), either $Cost(\Gamma) < Cost(\Gamma')$ or $Cost(\Gamma) = Cost(\Gamma') \wedge (\Gamma \not\Rightarrow \Gamma' \vee \Gamma \Leftrightarrow \Gamma')$*

Thus, a weakest minimum proof obligation Γ has cost less than or equal to any other proof obligation, and Γ is no stronger than other proof obligations with the same minimum cost as Γ. As argued earlier, the condition of minimality captures the intuitive notion of simplicity. Furthermore, the requirement that Γ is no stronger than other proof obligations with the same cost expresses generality. In particular, we do not want to ask the user about invariants that are stronger than necessary to discharge the error.

4.1.1 Computing Weakest Minimum Proof Obligations

We now turn to the problem of how to compute weakest minimum proof obligations. First, observe that we can rewrite $\Gamma \wedge \mathcal{I} \models \phi$ as:

$$\Gamma \models \mathcal{I} \Rightarrow \phi$$

Thus, Γ is a formula with minimum cost that entails $\mathcal{I} \Rightarrow \phi$. Our insight is that we can make use of a *minimum partial satisfying assignment* of $\mathcal{I} \Rightarrow \phi$ to compute the weakest Γ with minimum cost.

DEFINITION 4. **(Cost of Partial Assignment)** *Let σ be a partial assignment for a formula ϕ and let Π be a mapping from variables in ϕ to non-negative integers. The cost of partial assignment σ, written $Cost(\sigma)$ is $\sum_{v \in Vars(\sigma)} \Pi(v)$*

DEFINITION 5. **(Minimum Satisfying Assignment)** *Given mapping Π from variables to costs, a minimum satisfying assignment of formula φ is a partial assignment σ to a subset of the variables in φ such that:*

- $\sigma(\varphi) \equiv \text{true}$
- $\forall \sigma'$ *such that* $\sigma'(\varphi) \equiv \text{true}$, $Cost(\sigma) \leq Cost(\sigma')$

In this paper, we do not address the problem of computing minimum satisfying assignments for formulas. An algorithm for computing minimum satisfying assignments in theories that admit quantifier elimination is described in [3]. Since the theory of linear arithmetic over integers used in this paper admits quantifier elimination, the algorithm of [3] is directly applicable.

Minimum satisfying assignments are useful because they allow us to determine the minimum set of variables that any proof obligation Γ must contain. However, we are not interested in *any* minimum satisfying assignment to $\mathcal{I} \Rightarrow \phi$ since some minimum satisfying assignments make $\mathcal{I} \Rightarrow \phi$ true by falsifying $\mathcal{I}$. Such assignments are not interesting in this context because they violate known invariants $\mathcal{I}$ about the program. Thus, what we want is a minimum satisfying assignment to $\mathcal{I} \Rightarrow \phi$ that is consistent with $\mathcal{I}$.

DEFINITION 6. **(Consistent Minimum Satisfying Assignment)** *A minimum satisfying assignment σ of φ is consistent with φ' if $\sigma(\varphi')$ is satisfiable.*

Suppose σ is a minimum satisfying assignment of $\mathcal{I} \Rightarrow \phi$ consistent with $\mathcal{I}$. Now, if we interpret σ as a logical formula F_σ (i.e., a conjunction of equalities between variables and constants), F_σ is in fact a proof obligation, since, by definition:

$$F_\sigma \models \mathcal{I} \Rightarrow \phi \text{ and } \text{SAT}(F_\sigma \wedge \mathcal{I})$$

Furthermore, F_σ also has minimum cost, but, it is not the weakest proof obligation. More specifically, since F_σ assigns each variable to a concrete value, it is in fact a *strongest* proof obligation with minimum cost and is not very likely to be a valid program invariant.

What we are really after, then, is a formula Γ containing only the variables in σ but that is also the weakest formula containing these variables that still entails $\mathcal{I} \Rightarrow \phi$. In other words, what we want is the *weakest sufficient condition* of $\mathcal{I} \Rightarrow \phi$ containing only the variables in σ.

LEMMA 3. *Let V be the set of variables in a minimum satisfying assignment of $\mathcal{I} \Rightarrow \phi$ consistent with $\mathcal{I}$, and let $\overline{V}$ be the set of variables in $\mathcal{I} \Rightarrow \phi$ but not in V. We can obtain a weakest minimum proof obligation by eliminating the quantifiers from the formula:*

$$\forall \overline{V}.\ (\mathcal{I} \Rightarrow \phi)$$

PROOF 3. *The fact that this formula is a minimum proof obligation follows from Definition 6. The fact that this formula is the weakest such one follows from the well-known result that the weakest sufficient condition of a formula ϕ containing only variables $\overline{V}$ is given by $\forall \overline{V}.\phi$ and from the fact that linear integer arithmetic admits quantifier elimination.*

Remark: While the formula computed as described in Lemma 3 yields a weakest minimum proof obligation, this formula may be redundant. In other words, parts of this formula may already be implied by the known invariants $\mathcal{I}$. Thus, to avoid unnecessary queries, we simplify the formula computed as in Lemma 3 with respect to $\mathcal{I}$. This can be achieved, for instance, by using the simplification algorithm of [4], with $\mathcal{I}$ as the initial critical constraint.

EXAMPLE 2. *We now illustrate the computation of the proof obligation Γ on Example 1, where we already computed $\mathcal{I}$ and ϕ. Here, a minimum satisfying assignment to $\mathcal{I} \Rightarrow \phi$ that is consistent with $\mathcal{I}$ is $\alpha_j^1 = 0$. Now, we eliminate the quantifiers from the formula:*

$$\forall \nu_1, \nu_2, \alpha_i^1.\ (\mathcal{I} \Rightarrow \phi)$$

which, after simplification, yields $\alpha_j^1 \geq 0$. Thus, the assertion in Example 1 can be proven if the user shows that `j` ≥ 0 *after the loop, which is indeed the case. Observe that the query we pose to the user is much simpler compared to ϕ.*

4.1.2 Deciding Proof Obligation Queries

As discussed earlier, a proof obligation Γ is a query presented to a user, and, if true, proves the absence of an error in the program. Formally, we define a valid answer to a proof obligation query:

DEFINITION 7. **(Valid Answer to Proof Obligation Query)** *We say that the answer to a proof obligation query Γ is valid iff:*

- *The answer is either yes or no*
- *If the answer if yes, then Γ holds on* all *program executions (i.e., Γ is a program invariant)*
- *If the answer is no, then there is at least one execution in which Γ is violated*

In practice, a user may not always be able to give a definite yes/no answer to a proof obligation query; thus it is reasonable to accept "I don't know" as an answer. We discuss how to extend our technique to handle "I don't know" answers to queries in Section 5.

In this paper, we do not address the problem of verifying that the user's answer to a query is correct. Since our technique only applies in cases where automated verification fails, it is difficult, and sometimes even impossible, to completely eliminate trusted user information. Thus, our goal is to minimize the amount of trusted user information required to verify the program, making the job of the user much easier and improving classification accuracy.

LEMMA 4. *Let Γ be a proof obligation query, and suppose "yes" is a valid answer to this query. Then, the program is error-free.*

PROOF 4. *This follows immediately from the definition of proof obligation and validity of answer to the query.*

4.2 Weakest Minimum Queries to Validate An Error

In this section, we now turn to the complementary problem of validating the *existence* of a real error in the program. Recall from Lemma 2 that the condition

$$\mathcal{I} \models \neg\phi$$

guarantees that the program must contain a real error.

Now, to validate the existence of an error, we need to solve the same kind of abductive inference problem we considered in the previous section. More specifically, we need to find a formula Υ, called a *failure witness*, defined as follows:

DEFINITION 8. **(Failure Witness)** *Given facts $\mathcal{I}$ and success condition ϕ of a program, a* failure witness *is a formula Υ such that:*

$$\mathcal{I} \land \Upsilon \models \neg\phi \quad \textit{and} \quad \textit{SAT}(\Upsilon \land \mathcal{I})$$

As in the case of proof obligations, we are not interested in *any* failure witness, as some witnesses may be more complicated than necessary or overly restrictive. Thus, to be able to ask the user simple and general witness queries, we want to obtain a *weakest minimum failure witness*. For this purpose, we define cost of failure witnesses as follows:

DEFINITION 9. **(Cost of Failure Witness)** *Let Υ be a failure witness for $\mathcal{I}$, ϕ, and let Π_w be a mapping from variables to costs such that $\Pi_w(\nu) = 1$ for input variable ν and $\Pi_w(\alpha) = |\textit{Vars}(\phi) \cup \textit{Vars}(\mathcal{I})|$ for abstraction variable α. Then:*

$$\textit{Cost}(\Upsilon) = \sum_{v \in \textit{Vars}(\Upsilon)} \Pi_w(v)$$

Similar to that for proof obligations, this cost function is biased in favor of queries involving local facts, as it penalizes witness queries involving more abstraction variables. But in contrast to proof obligations, we prefer witness queries involving only inputs to those involving abstraction variables. Since it is in general undesirable to make assumptions about the program's inputs, it is likely that witness queries involving input variables are easy to answer affirmatively, and are thus assigned a lower cost. While this cost definition merely approximates the simplicity and usefulness of a witness query, we found it to work well in practice.

We can now define *weakest minimum failure witness*:

DEFINITION 10. **(Weakest Minimum Failure Witness)** *Given known facts $\mathcal{I}$ about a program P and success condition ϕ, a* weakest minimum failure witness *of P is a formula Υ such that:*

1. *$\Upsilon \land \mathcal{I} \models \neg\phi$ and SAT$(\Upsilon \land \mathcal{I})$*
2. *For any other Υ' that satisfies (1), either Cost$(\Upsilon) <$ Cost(Υ') or Cost$(\Upsilon) =$ Cost$(\Upsilon') \land (\Upsilon \not\Rightarrow \Upsilon' \lor \Upsilon \Leftrightarrow \Upsilon')$ where Cost is defined according to Definition 9.*

The computation of weakest minimum failure witnesses is analogous to the computation of weakest minimum proof obligations:

LEMMA 5. *Let V be the set of variables in a minimum satisfying assignment of $\mathcal{I} \Rightarrow \neg\phi$ consistent with $\mathcal{I}$, and let $\overline{V}$ be the set of variables in $\mathcal{I} \Rightarrow \neg\phi$ but not in V. Then, a weakest minimum failure witness is obtained by eliminating the quantifiers from the formula:*

$$\forall \overline{V}.\ (\mathcal{I} \Rightarrow \neg\phi)$$

4.2.1 Deciding Failure Witness Queries

A failure witness is a query such that if the user answers "yes", then there exists at least one execution in which the program fails. Thus, a *valid answer* to a witness query is defined as follows:

DEFINITION 11. **(Valid Answer to Witness Query)** *We say that the answer to a failure witness query Υ is valid iff:*

- *The answer is either yes or no*
- *If the answer is yes, then there exists at least one program execution where Υ holds*
- *If the answer is no, then there is no execution in which Υ holds, i.e. $\neg\Upsilon$ is a program invariant.*

Observe that answering "yes" to a witness query has very different semantics than answering "yes" to a proof obligation query: In the former case, Υ needs to hold in only one execution (existential semantics), whereas in the latter case, Γ must hold in all executions (universal semantics). Furthermore, observe that answering "no" to a witness query Υ is equivalent to answering "yes" to a proof obligation query $\neg\Upsilon$. Similarly, answering "no" to a proof obligation query Γ is equivalent to answering "yes" to a witness query $\neg\Gamma$.

LEMMA 6. *Let Υ be a failure witness query, and suppose "yes" is a valid answer to this query. Then, the program exhibits an error in at least one execution.*

PROOF 5. *This follows from the definition of failure witness and validity of answer to the witness query.*

4.3 The Full Algorithm

We now describe the algorithm, presented in Figure 6, for systematically generating a sequence of queries until the error report is resolved. This algorithm takes as input the known program invariants $\mathcal{I}$ and the success condition ϕ. The set W is a set of witnesses, i.e., conditions known to hold in some execution of the program.

In lines 5-6 of the algorithm, we first compute proof obligation Γ as described in Section 4.1. Here, $\texttt{compute\_msa}(\varphi, S, \Pi_p)$ computes the set of variables in the minimum satisfying assignment of φ consistent with the set of formulas given by S with respect to cost mapping Π_p (recall Definition 2). As discussed in Section 4.1, the minimum satisfying assignment needs to be consistent with $\mathcal{I}$, since we do not want to issue queries about facts that violate known program invariants. Furthermore, we want the minimum satisfying assignment to be consistent with all learned witnesses in W, since we do not want to ask the user queries about facts for whose violation we already have witnesses. In line 4, the notation $\overline{V_1}$ denotes the set of variables that are in $\mathcal{I} \Rightarrow \phi$ but that are not part of the minimum satisfying assignment. As stated by Lemma 3, we can compute the weakest minimum proof obligation Γ by eliminating the universal quantifiers from the formula $\forall \overline{V_1}.\ \mathcal{I} \Rightarrow \phi$.

Dually, lines 7-8 compute a weakest minimum failure witness Υ as described in Section 4.2: We first compute the minimum satisfying assignment of $\mathcal{I} \Rightarrow \neg\phi$ consistent with $\mathcal{I}$ with respect to cost function Π_w (recall Definition 9). Again, it is important that the minimum satisfying assignment is consistent with $\mathcal{I}$, as we do not want to ask queries that we know cannot hold in any execution. However, in this case, it is not necessary that the assignment is consistent with W since the failure witness needs to hold for only

```
function diagnose_error($\mathcal{I}, \phi$) :

1 :  set $W = \emptyset$;

2 :  while(true){

3 :    if(VALID($\mathcal{I} \Rightarrow \phi$)) return ERROR_DISCHARGED
4 :    if($\exists \psi \in W$. UNSAT($\mathcal{I} \wedge \psi \wedge \phi$)) return ERROR_VALIDATED

5 :    $V_1$ = compute_msa($\mathcal{I} \Rightarrow \phi, W \cup \mathcal{I}, \Pi_p$)
6 :    $\Gamma$ = elim_quantifier($\forall \overline{V_1}.(\mathcal{I} \Rightarrow \phi)$)

7 :    $V_2$ = compute_msa($\mathcal{I} \Rightarrow \neg\phi, \mathcal{I}, \Pi_w$)
8 :    $\Upsilon$ = elim_quantifier($\forall \overline{V_2}.(\mathcal{I} \Rightarrow \neg\phi)$)

9 :    if(Cost($\Gamma$) $\leq$ Cost($\Upsilon$)){
10 :     $Q_1$ = form_invariant_query($\Gamma$)
11 :     if(answer to $Q_1$= YES) return ERROR_DISCHARGED
12 :     $W := W \cup (\neg\Gamma)$
13 :   }

14 :   else{
15 :     $Q_2$ = form_witness_query($\Upsilon$)
16 :     if(answer to $Q_2$= YES) return ERROR_VALIDATED
17 :     $\mathcal{I} := \mathcal{I} \wedge (\neg\Upsilon)$
18 :   }
19 : }
```

Figure 6. Algorithm for Diagnosing Error Reports

one, rather than for all, executions. Finally, as shown by Lemma 5, we compute the weakest minimum failure witness by eliminating quantifiers from the formula $\forall\overline{V_2}.\ \mathcal{I} \Rightarrow \neg\phi$, where $\overline{V_2}$ is the set of variables in $\mathcal{I} \Rightarrow \neg\phi$, but not in V_2.

At line 9, we compare the costs of the proof obligation Γ and the failure witness Υ to decide whether it is more promising to try to discharge the error or to validate it. If the cost of Γ is cheaper, then at line 10, we formulate the proof obligation Γ as an invariant query (discussed further in Section 4.4). If the user answers the invariant query positively, we have discharged the error. On the other hand, if the user decides that Γ is not an invariant, then we have learned a new witness: We now know there exists at least one execution where Γ is violated; thus we add $\neg\Gamma$ to the set of witness formulas.

On the other hand, if the error validation strategy has lower cost than the error discharge strategy (lines 14-18), we formulate Υ as a witness query (again, see Section 4.4). If the answer to this query is "yes", we have validated the error. On the other hand, if the answer to the witness query is "no", this means there is no execution in which Υ holds. In this case, we have learned that $\neg\Upsilon$ is an invariant, and we conjoin it with the existing invariants $\mathcal{I}$.

Observe that at lines 3-4, we check whether the error can be discharged or validated with the current invariants $\mathcal{I}$ and witnesses W. This check is necessary because even when the user answers "no" to a query, we still learn additional facts as a result.

Also, observe that the queries computed by our algorithm may increasingly become more difficult over time. As the analysis progresses, $\mathcal{I}$ becomes stronger and more witnesses are acquired; thus, the minimum satisfying assignment needs to be consistent with more facts and may therefore contain more variables. In fact, in the unlikely case where the user answers "no" to all queries, the final query may degenerate into the success condition itself.

4.4 From Formulas to Queries

So far, we focused on computing small, relevant formulas that are useful for classifying error reports. We now discuss how to present these formulas in a way that can be easily understood by humans.

The first challenge is to translate analysis variables into program expressions that can be understood without knowledge of internal analysis details. This problem is easily solved in our representation since there is a straightforward mapping between analysis variables and program expressions: Input variables ν_i used in the analysis represent values of program inputs a_i, and abstraction variables α_i^ρ represent values of program variables v_i at program point ρ.

The second challenge is that query formulas presented to the user may be arbitrary boolean combinations of program expressions. Since it is unrealistic to expect programmers not well-versed in logic to easily reason about complicated boolean expressions, we need to decompose queries with complex boolean structure to a series of simpler queries. To achieve this goal, we observe that invariant queries distribute over conjuncts, while witness queries distribute over disjuncts. More specifically, if $\phi_1 \wedge \phi_2$ is an invariant, it is necessary that ϕ_1 and ϕ_2 are independently also invariants. Similarly, if $\phi_1 \vee \phi_2$ is a witness, it is necessary that ϕ_1 is possible in some execution or ϕ_2 is possible in some execution. Thus, we first convert invariant queries to conjunctive normal form (CNF) and treat each clause as a separate, independent query. Dually, we convert witness queries to disjunctive normal form (DNF) and reason about each clause independently.

We also further decompose clauses into simpler queries. For witness queries, observe that if ϕ_1 and ϕ_2 are individually witnesses, this does not mean $\phi_1 \wedge \phi_2$ is also a witness, since ϕ_1 and ϕ_2 may be possible in distinct executions, but never possible in the same execution. In this case, we decompose a conjunct $\phi_1 \wedge \phi_2$ by first querying whether ϕ_1 is possible in some execution and then ask whether ϕ_2 is also possible in an execution where ϕ_1 holds.

For invariant queries, clauses contain disjunctions, which humans typically find difficult to reason about. Thus, for an invariant query $\phi_1 \vee \phi_2$, we first ask whether either ϕ_1 or ϕ_2 are independently invariants, which is often the case. In cases where a truly disjunctive invariant is required, observe that for $\phi_1 \vee \phi_2$ to be an invariant, $\neg\phi_1 \wedge \neg\phi_2$ should *not* be a witness. Thus, to avoid disjunctions, we convert the invariant query $\phi_1 \vee \phi_2$ into the witness query $\neg\phi_1 \wedge \neg\phi_2$ and use the same technique described in the previous paragraph to decompose conjunctive witness queries.

In addition to making queries more understandable, decomposing queries into simpler, independent subqueries also has another advantage: We can learn additional facts for every subquery even when the entire query may not be shown. For example, although $\phi_1 \wedge \phi_2$ is not an invariant, ϕ_1 could still be an invariant. It is easy to integrate this additional information learned from subqueries into the algorithm, and our implementation uses this optimization.

5. Implementation

We have implemented the proposed technique on top of the Compass analysis framework for C programs [5–7]. While the simple language from Section 2 contains only integer variables and no function calls, our implementation extends the technique to deal with other features of the C language supported in Compass, such as pointers, arrays, and function calls. In particular, Compass reasons about heap objects and arrays using the techniques described in [5, 6] and uses a summary-based technique for interprocedural analysis [7]. Besides imprecise loop invariants, Compass also has other sources of imprecision, such as non-linear arithmetic or inline assembly, which are also modeled using abstraction variables. In addition, side effects of library functions that we choose not to analyze or whose source code is unavailable are also modeled using abstraction variables.

In the algorithm from Section 4.3, we require users to give only yes/no answers to queries, but, in practice, this requirement is unrealistic, as programmers cannot always answer a query. Thus, we allow users to answer "I don't know" and extend the algorithm

	LOC	Kind	Classification	Manual classification				New technique			
				% correct	% wrong	% ?	Avg. time	% correct	% wrong	% ?	Avg. time
Problem 1	88	synthetic	false alarm	43.5 %	34.8 %	21.7%	297 s	92.3 %	3.9%	3.9 %	57 s
Problem 2	352	real	false alarm	30.8 %	50.0 %	19.2 %	269 s	87.0 %	8.7%	4.4 %	40 s
Problem 3	66	synthetic	false alarm	46.2 %	38.5 %	15.4 %	266 s	79.2 %	20.8%	0.0%	58 s
Problem 4	278	real	real bug	37.5 %	45.8 %	16.7 %	265 s	92.3 %	7.7 %	0.0%	53 s
Problem 5	363	real	false alarm	32.0 %	48.0 %	20.0 %	289 s	100.0 %	0.0 %	0.0 %	46 s
Problem 6	173	real	false alarm	25.0%	54.2 %	20.8%	339 s	92.0 %	8.0 %	0.0 %	54 s
Problem 7	326	real	real bug	40.0 %	56.0 %	4.0%	233 s	79.2 %	8.3 %	12.5 %	55 s
Problem 8	97	synthetic	false alarm	16.7 %	70.8 %	12.5 %	271 s	92.0%	8.0%	0.0%	58 s
Problem 9	116	synthetic	real bug	25.0 %	58.3 %	16.7 %	308 s	92.0 %	4.0%	4.0%	62 s
Problem 10	72	synthetic	real bug	24.0 %	60.0 %	16.0 %	455 s	95.8 %	4.2%	0.0 %	68 s
Problem 11	118	synthetic	real bug	41.7 %	45.8%	12.5%	235 s	84.0 %	16.0%	0.0%	50 s
Average	186	n/a	n/a	32.9 %	51.1 %	16.0 %	293 s	89.6 %	7.3 %	2.3 %	55 s

Figure 7. Results from our user study

from Section 4.3 to deal with such uncertainty. Specifically, our algorithm maintains a set of *potential witnesses* and *potential invariants*. If Γ is an invariant query that the user cannot answer, we add Γ as a potential invariant and $\neg\Gamma$ as a potential witness. Similarly, if Υ is a witness query that the user cannot decide, we add Υ as a potential witness and $\neg\Upsilon$ as a potential invariant. To avoid repeating the same queries, these potential witnesses and invariants are taken into account when computing minimum satisfying assignments: Minimum satisfying assignments in the proof obligation case must be consistent with potential invariants and witnesses, while the minimum satisfying assignments for the witness case must be consistent with potential invariants.

Our technique utilizes minimum satisfying assignments for computing abductions, but no off-the-shelf SMT solver we know of provides this functionality. Thus, we have integrated the required functionality for providing minimum satisfying assignments into our own Mistral SMT solver; the algorithm we implemented for this purpose is described in [3].

6. Experiments

To evaluate the proposed technique, we performed a user study in which we asked professional programmers to classify error reports as genuine bugs or false alarms. Our study consists of 11 problems, five of which are code fragments taken directly or slightly modified from real open source C programs (e.g., Unix coreutils, OpenSSH), and six of which are benchmarks from our program analysis test suite. These benchmarks are available from http://www.cs.wm.edu/~tdillig/pldi12-benchmarks.tar.gz. All benchmarks contain between 66-363 lines of code and one assertion. For examples drawn from real applications, we only included those parts of the code that are relevant to deciding the validity of the assertion (i.e., a manual slice). Five of our benchmark problems contain real errors while the remaining six are error-free. The analysis we performed initially reports potential, but not certain, errors on all eleven benchmarks. The causes of these uncertain error reports are diverse, including imprecise loop invariants, missing annotations on library functions, non-linear arithmetic, and missing facts about the program's execution environment (e.g., relationship between `argc` and `argv`).

For a given problem, each participant was randomly assigned to classify an error report either manually or using the proposed technique. Therefore, each participant classified approximately half the benchmarks manually and half of them using the new technique. In the manual classification case, the participants were asked to decide whether the report is a false alarm, a real bug, or whether they cannot decide. When using our new technique, the participants were asked to give a "Yes", "No", or "I don't know" answer to a series of questions generated by our tool, ranging from one to three questions on these benchmarks. The time for query computation is negligible; in all cases, the computation time is below 0.1s.

For our user study, we recruited 56 professional programmers through ODesk and paid them $40 each for participating in the study. Each participant was required to have at least one year of C programming experience and to have ODesk's "systems programming certification". In addition to our eleven benchmark problems, the study also included three simple diagnostic questions that were not identified as such to the participants. We excluded participants who gave the wrong answer to any of the diagnostic questions, resulting in 49 valid participants for our study. Thus, each benchmark problem was classified manually by about 24 people and classified using our technique by approximately the same number of people.

Figure 7 summarizes the results of our user study. The first three columns in the table give details about each benchmark. The column labeled "LOC" gives the lines of code presented to participants; the column "Kind" identifies whether the benchmark is taken from a real application or from our program analysis test suite (i.e., synthetic). Finally, the column "Classification" indicates the correct classification of the report (false alarm or real bug).

The next four columns in the table ("Manual classification") give statistics about accuracy and speed of manual classification. The column "% correct" indicates the percentage of participants who correctly classified the report; the column "% wrong" gives the percentage of participants who chose the wrong classification. The column labeled "% ?" gives the percentage of participants who answered "I don't know". Finally, the column "Avg. time" shows the average time participants took to classify the report.

The next four columns in the table (marked "New Technique") give the same statistics about accuracy of classification and timing using the new technique presented in this paper.

As the results in Figure 7 demonstrate, the average accuracy of manual report classification is surprisingly low: On average, 32.9% of the participants are able to classify an error report correctly, while 51.1% of the participants incorrectly classify a real bug as a false alarm or a false alarm as a real bug. Also, observe that 16% of participants are unable to classify the error report despite spending significant time. (Our instructions specifically mentioned that participants should only choose "I don't know" after spending at least 8 minutes on each benchmark.) Finally, observe that programmers take an average of approximately 5 minutes to manually classify error reports despite the fact that their classification is often wrong.

These results indicate that manual classification is extremely unreliable and time-consuming even on our small benchmark examples. In fact, the group of programmers who participated in our study do not seem to outperform randomly assigning a classification to each report. We believe this is strong evidence that the usefulness of static analysis techniques can be greatly improved by assisting programmers in classifying and understanding error reports.

As the results in Figure 7 also demonstrate, the accuracy and speed of report classification dramatically improve when programmers use our new technique. While only 32.9% of participants correctly classify an error report using manual classification, 89.6% of participants give the correct answer using our technique. Similarly, the percentage of participants who give the wrong answer drops from 51.1% to 7.3%. In addition to dramatically improving classification accuracy, our technique also substantially reduces the time programmers need to classify reports: The average classification time drops from approximately 5 minutes to under 1 minute.

As standard in user studies, we performed a t-test to evaluate the statistical significance of our results. The p-value of a two-tailed t-test (assuming potentially unequal variance) comparing manual classification accuracy vs. our technique is 5×10^{-8}. This means the probability that our technique has no influence on classification accuracy is less than 1 in 10,000,000. Similarly, the p-value for a t-test comparing classification times is 1.2×10^{-28}, indicating our results are statistically significant.

To give the reader a flavor of how our technique makes report classification much easier, we briefly describe the reasoning required to classify Problem 6, based on the `chroot` utility from the Unix coreutils. In this program, the value of variable `optind` is correlated with four different return values of function `getopt_long`. To prove the assertion in this example, we have to remember all four return values of `getopt_long` along with the intricate path conditions associated with each of these return points and mentally evaluate several conditionals, all of which are relevant to proving the assertion. In fact, an author of this submission spent approximately half an hour to decide that the report is indeed a false alarm. In contrast, if we use the technique proposed in this paper, the user only needs to answer one simple query asking whether the value of `optind` is always greater than zero after a `while` loop. This query is easy to decide by inspecting only three lines of code immediately preceding the program point associated with the query.

7. Related Work

Explaining Error Traces in Model Checking Several papers focus on explaining error traces identified by model checkers [8–13]. Ball et al. describe an algorithm for localizing the likely cause of an error given a trace produced by a model checker [8]. Groce et al. present an error localization technique that computes the minimum distance between an error trace and a successful execution [9]. Ravi and Somenzi present an algorithm for giving succinct explanations for error traces, and like our technique, they employ minimum satisfying assignments to derive these explanations [13]. Jose and Majumdar give an algorithm for fault localization using maximum satisfiability [10]. This technique computes the maximum satisfiable set of clauses in an extended trace formula, and the complement of this set is identified as a potential cause of the error. In this work, we share the goal of making error reports easier to understand for programmers. However, our technique does not require a counterexample produced by model checkers and can be gainfully combined with any static analysis. Furthermore, our technique is useful for classifying and understanding false alarms, which are not addressed by these techniques.

Counterexample-Guided Abstraction Refinement (CEGAR) Our technique is similar to CEGAR in that both approaches infer relevant conditions that are useful for eliminating false alarms [14–16]. The main difference is that CEGAR-based approaches learn new predicates from one concrete counterexample trace. Thus, while the learned predicate may be sufficient to prevent this *particular* counterexample trace, it may not be sufficient to eliminate other spurious traces reaching the same error. In contrast, proof obligations we compute are, if valid, guaranteed to rule out the false alarm entirely. Furthermore, while CEGAR-based approaches have the advantage of being fully automatic and not requiring assistance from the user, they are not guaranteed to terminate.

Techniques for Error Report Understanding Manevich at al. present a post-mortem backwards dataflow analysis for constructing failing execution traces from a failure point [17]. This technique can sometimes show that the report is a false alarm. Le and Soffa describe a fault correlation technique that can be helpful for diagnosing error reports [18]. Their technique groups error reports according to their root cause, which may aid in understanding these reports. The approach of [19] computes a *patch*, which is a modified version of the program not containing the error, and this patch is included in the bug report. While these techniques may sometimes be insightful for understanding a report, they do not allow the classification of an error report as a real bug or false alarm.

Program slicing [20–22] is useful for identifying statements in the source code that may be relevant to the potential failure. However, program slices are typically large and do not take advantage of facts already shown by the analysis. Unlike our technique, a slice also does not provide any semantic clues about whether the report corresponds to a genuine bug or false alarm.

Explaining Type Errors There has been much work on explaining errors that arise in type inference [23–26]. Since errors during polymorphic type checking result from unification failure, the location associated with a report is often not useful for understanding its cause. The techniques of [23–25] augment type inference with information useful for explaining the chain of inferences that led to the error. The algorithm described in [26] does not modify the original compiler, but searches for a program close to the original one that does type check. We share the goal of making static reasoning transparent to users; however, the afore-mentioned approaches specialize in explaining type inference errors while our technique assists users in classifying reports generated by verification tools.

Combining May and Must Information There has been recent work on combining may and must information to prove both the presence and absence of errors. Must information has been harvested both from dynamic analysis [27–29] and computed purely statically [30]. While these approaches are useful for showing the presence of errors, they do not help when errors can neither be discharged nor proven, which is our focus. However, we believe must information obtained from dynamic analysis could be useful for automatically deciding some of our failure witness queries.

8. Conclusion and Future Work

We have presented a new technique for assisting users with classifying error reports generated by static analyses. Our technique can be gainfully combined with any static analysis to make their results more understandable for users. We have evaluated this technique with a user study; our results indicate that the new technique improves both the time and accuracy of report classification.

We believe that the approach described in this paper can be further improved by making use of underapproximations. In particular, the analysis described in Section 3 can only prove the program is buggy if $\mathcal{I} \models \neg\phi$. However, since $\mathcal{I}$ is an invariant (i.e., an overapproximation of program behavior), $\mathcal{I} \models \neg\phi$ indicates that every execution of the program must be buggy. By making use of underapproximations obtained through static or dynamic techniques, the analysis can prove that *some* executions of the program must be buggy without requiring the user's help. We believe dynamic analysis could also be very useful for automatically discharging some of the failure witness queries.

References

[1] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: using static analysis to find bugs in the real world. Communications of the ACM **53**(2) (2010) 66–75

[2] Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference (extended version). http://www.cs.wm.edu/~idillig/pldi12-extended.pdf

[3] Dillig, I., Dillig, T., McMillan, K., Aiken, A.: Minimum satisfying assignments for SMT. In: To appear in CAV. (2012)

[4] Dillig, I., Dillig, T., Aiken, A.: Small formulas for large programs: Online constraint simplification in scalable static analysis. Static Analysis Symposium (SAS) (2010) 236–252

[5] Dillig, I., Dillig, T., Aiken, A.: Fluid updates: Beyond strong vs. weak updates. ESOP (2010) 246–266

[6] Dillig, I., Dillig, T., Aiken, A.: Precise reasoning for programs using containers. POPL (2011) 187–200

[7] Dillig, I., Dillig, T., Aiken, A., Sagiv, M.: Precise and compact modular procedure summaries for heap manipulating programs. In: PLDI. (2011) 567–577

[8] Ball, T., Naik, M., Rajamani, S.: From symptom to cause: localizing errors in counterexample traces. POPL **38**(1) (2003) 97–105

[9] Groce, A.: Error explanation with distance metrics. TACAS (2004) 108–122

[10] Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: PLDI, ACM (2011) 437–446

[11] Griesmayer, A., Staber, S., Bloem, R.: Automated fault localization for c programs. Theoretical Computer Science **174**(4) (2007) 95–111

[12] Renieres, M., Reiss, S.: Fault localization with nearest neighbor queries. In: ASE. (2003) 30–39

[13] Ravi, K., Somenzi, F.: Minimal assignments for bounded model checking. TACAS (2004) 31–45

[14] Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Software verification with blast. Model Checking Software (2003) 624–624

[15] Ball, T., Rajamani, S.: The SLAM project: debugging system software via static analysis. In: POPL, NY, USA (2002) 1–3

[16] Henzinger, T., Jhala, R., Majumdar, R., McMillan, K.: Abstractions from proofs. POPL **39**(1) (2004) 232–244

[17] Manevich, R., Sridharan, M., Adams, S., Das, M., Yang, Z.: PSE: explaining program failures via postmortem static analysis. In: FSE. Volume 29., ACM (2004) 63–72

[18] Le, W., Soffa, M.: Path-based fault correlations. In: FSE. (2010) 307–316

[19] Weimer, W.: Patches as better bug reports. In: GPCE. (2006) 181–190

[20] Weiser, M.: Program slicing. In: ICSE, IEEE Press (1981) 439–449

[21] Korel, B., Laski, J.: Dynamic program slicing. Information Processing Letters **29**(3) (1988) 155–163

[22] Sridharan, M., Fink, S., Bodik, R.: Thin slicing. In: PLDI, ACM (2007) 112–122

[23] Beaven, M., Stansifer, R.: Explaining type errors in polymorphic languages. LOPLAS **2**(1-4) (1993) 17–30

[24] Haack, C., Wells, J.: Type error slicing in implicitly typed higher-order languages. Programming Languages and Systems (2003) 284–301

[25] Yang, J.: Explaining type errors by finding the source of a type conflict. Trends in Functional Programming (1999) 49–57

[26] Lerner, B., Flower, M., Grossman, D., Chambers, C.: Searching for type-error messages. In: PLDI, ACM (2007) 425–434

[27] Nori, A., Rajamani, S., Tetali, S., Thakur, A.: The yogi project: Software property checking via static analysis and testing. TACAS (2009) 178–181

[28] Godefroid, P., Nori, A., Rajamani, S., Tetali, S.: Compositional may-must program analysis: unleashing the power of alternation. In: POPL. Volume 45., ACM (2010) 43–56

[29] Yorsh, G., Ball, T., Sagiv, M.: Testing, abstraction, theorem proving: better together! In: ISSTA, ACM (2006) 145–156

[30] Dillig, I., Dillig, T., Aiken, A.: Sound, complete and scalable path-sensitive analysis. In: PLDI, ACM (2008) 270–280