Automated Error Diagnosis Using Abductive Inference

Isil Dillig\textsuperscript{1}  Thomas Dillig\textsuperscript{1}  Alex Aiken\textsuperscript{2}

\textsuperscript{1}Department of Computer Science  
College of William & Mary, Virginia, USA

\textsuperscript{2}Department of Computer Science  
Stanford University, CA, USA

PLDI 2012

Severin Heiniger
void foo(int flag, unsigned int n) {
    int k = 0, i = 0, j = 0, z = 0;
    if (flag) k = n;
    else k = 1;

    while (i <= n) {
        i = i + 1;
        j = j + i;
    }

    int z = k + i + j;
    assert(z > 2 * n);
}
An Ordinary Day in a Developer’s Life

```c
void foo(int flag, unsigned int n) {
    int k = 0, i = 0, j = 0, z = 0;
    if (flag) k = n;
    else    k = 1;

    while (i <= n) {
        i = i + 1;
        j = j + i;
    }

    int z = k + i + j;
    assert(z > 2 * n);
}
```

Static analysis tool error report

Assertion \( z > 2 \times n \) may not always hold.
Manual Report Classification

Program

Some Static Analysis → Success

Potential Error Report

User Decides

Genuine Bug    False Alarm
Manual Report Classification

- Time-consuming
- User repeats all successful reasoning by tool
- Error-prone

Effect

Major impediment to adoption of static analysis tools
Semi-Automated Report Classification

Program

Some Static Analysis → Success

Potential Error Report

Inferred Invariants

This paper: Assist User

Genuine Bug

False Alarm

Severin Heiniger
Semi-Automated Report Classification

Program with Inferred Invariants and Potential Error Report

Identify Sources of Incompleteness

Check

Yes or No

If uncertain: Small, relevant query

User

Genuine Bug False Alarm
Queries

- **Proof Obligation Query:** *Is property* \( P \) *an invariant?*
  - If yes, the program is certainly error-free (false alarm)

- **Failure Witness Query:** *Can property* \( P \) *arise in some execution?*
  - If yes, the program is certainly buggy

**Strategy**

Pose queries in order of increasing cost (easiest first) to minimize the amount of trusted information the user must supply.
We are Here

Program with Inferred Invariants and Potential Error Report

Identify Sources of Incompleteness

Check

Yes or No

If uncertain:
Small, relevant query

User

Genuine Bug

False Alarm

Severin Heiniger

Research Topics in Software Engineering

May 13th, 2013
Program with parameters, local variables, conditionals and while loops

Only linear arithmetic, no function calls

While loops annotated with inferred post-condition $p'$:

```c
while (p) { s } [p']
```

Program ends with an `assert (p)`
Symbolically evaluate the program. At each point in the program, environment $S$ maps program variables to symbolic value sets.

$$S(i) = \{\ldots, (\pi, \phi), \ldots\}$$

Under constraint $\phi$, the value of variable $i$ is the symbolic expression $\pi$.

Constraints $\phi$ keep values from different paths separate. $\pi$ can contain

- **Input Variables** $\nu$ For unknown program inputs
- **Abstraction Variables** $\alpha$ For unknown values due to imprecisions, e.g., after loops
Example

```cpp
void foo(int flag, unsigned int n) {
    int k = 0, i = 0, j = 0, z = 0;

    if (flag) k = n;
    else k = 1;

    while (i <= n) {
        i = i + 1;
        j = j + i;
    }
    int z = k + i + j;
    assert(z > 2 * n);
}
```
Propagate inferred invariants as constraints on abstract variables

\[ \mathcal{I} = (\alpha_i \geq 0 \land \alpha_i > \nu_n \land \nu_n \geq 0) \]
Example

```c
void foo(int flag, unsigned int n) {
    int k = 0, i = 0, j = 0, z = 0;
    S(k) = {(0, true)} S(i) = {(0, true)}
    if (flag) k = n;
    else k = 1;
    S(k) = {(1, ¬νflag), (νn, νflag)}
    while (i <= n) {
        i = i + 1;
        j = j + i;
        [i ≥ 0 ∧ i > n]
        S(i) = {(αi, true)} S(j) = {(αj, true)}
        S(z) = {(1 + αi + αj, ¬νflag), (νn + αi + αj, νflag)}
        int z = k + i + j;
        assert(z > 2 * n);
    }
}
```

Symbolically evaluate the assertion predicate

\[ \phi = (1 + \alpha_i + \alpha_j > 2 \cdot \nu_n \land \neg \nu_{\text{flag}}) \lor (\nu_n + \alpha_i + \alpha_j > 2 \cdot \nu_n \land \nu_{\text{flag}}) \]
The result is a pair of symbolic constraints

\[ I \] All known invariants on abstract variables

\[ \phi \] Condition under which the assertion evaluates to true
The result is a pair of symbolic constraints

\[ \mathcal{I} \quad \text{All known invariants on abstract variables} \]
\[ \phi \quad \text{Condition under which the assertion evaluates to } true \]

Lemma

*If \( \mathcal{I} \models \phi \), then the program is error-free (assertion always succeeds)\n
*If \( \mathcal{I} \models \neg \phi \), then the program must be buggy (assertion always fails)\n
Program with Inferred Invariants and Potential Error Report

Identify Sources of Incompleteness

\[ \mathcal{I}, \phi \]

Yes or No

If uncertain:
Small, relevant query

Genuine Bug
False Alarm

User

Check
Given known facts $\mathcal{I}$ and success condition $\phi$, a \textit{proof obligation} is a formula $\Gamma$ that – together with $\mathcal{I}$ – proves $\phi$:

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

Given known facts $\mathcal{I}$ and success condition $\phi$, a proof obligation is a formula $\Gamma$ that – together with $\mathcal{I}$ – proves $\phi$:

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

**Cost($\Gamma$)**

$$1 \cdot \# \text{abstraction variables } \alpha \in \text{Vars}(\Gamma) + |\text{Vars}(\phi) \cup \text{Vars}(\mathcal{I})| \cdot \# \text{input variables } \nu \in \text{Vars}(\Gamma)$$

- The fewer variables, the better
- No input variables if possible
Given known facts $\mathcal{I}$ and success condition $\phi$, a **failure witness** is a formula $\Upsilon$ that – together with $\mathcal{I}$ – proves $\neg\phi$:

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

**Cost($\Upsilon$)**

$$|\text{Vars}(\phi) \cup \text{Vars}(\mathcal{I})| \cdot \# \text{abstraction variables } \alpha \in \text{Vars}(\Upsilon) + 1 \cdot \# \text{input variables } \nu \in \text{Vars}(\Upsilon)$$

- The fewer variables, the better
- Prefer input variables
Weakest Minimum Queries

Weakest Minimum Proof Obligation $\Gamma$

- costs less than or equal to any other proof obligation, and
- is no stronger than any other proof obligations with same cost

Weakest Minimum Failure Witness $\Upsilon$  Dito
Ask the user the one with lower cost

- **Does \( \Gamma \) hold in all program executions?**
  - **Yes** Program is error-free (because \( \Gamma \land I \models \phi \))
  - **No** Add \( \lnot \Gamma \) to known witnesses and maybe ask another query

- **May \( \Upsilon \) arise in some execution?**
  - **Yes** Program is buggy (because \( \Upsilon \land I \models \lnot \phi \))
  - **No** Add \( \lnot \Upsilon \) to known facts \( I \) and maybe ask another query
Example

```c
void foo(int flag, unsigned int n) {
    int k = 0, i = 0, j = 0, z = 0;
    if (flag) k = n;
    else k = 1;

    while (i <= n) {
        i = i + 1;
        j = j + i;
    }

    int z = k + i + j;
    assert(z > 2 * n);
}
```

Weakest Minimum Proof Obligation $\Gamma = (\alpha_j \geq \nu_n)$

Weakest Minimum Failure Witness $\Upsilon = (\neg \nu_{\text{flag}} \land \alpha_i + \alpha_j < 0)$
Example

```c
void foo(int flag, unsigned int n) {
    int k = 0, i = 0, j = 0, z = 0;
    if (flag) k = n;
    else k = 1;
    while (i <= n) {
        i = i + 1;
        j = j + i;
    }
    int z = k + i + j;
    assert(z > 2 * n);
}
```

\[
\Gamma = (\alpha_j \geq \nu_n) \checkmark \ (\text{false alarm!})
\]

Weakest Minimum Failure Witness \( \Upsilon = (\neg \nu_{flag} \land \alpha_i + \alpha_j < 0) \)
56 professional C programmers
Classify 11 uncertain error reports for real-world code as
  - Genuine bugs (5), or
  - False alarms (6), or
  - *I don’t know*
Randomly assigned to classify manually or using the new technique
User Study: Results

![Bar Chart]

- **Manual Classification**: 5 minutes
- **New Technique**

Percentage Correct:
- Correct
- I Don't Know
- Wrong

Severin Heiniger
Research Topics in Software Engineering
May 13th, 2013
User Study: Results

Manual Classification
∅ 5 mins

New Technique
∅ 1 min

Correct | I Don't Know | Wrong

Severin Heiniger
Research Topics in Software Engineering
May 13th, 2013
Explaining Error Traces in Model Checking
  Requires counter-example, does not address false alarms

Counterexample-Guided Abstraction Refinement (CEGAR)
  Learn new predicates from concrete counter-example trace
  Fully automatic, but not guaranteed to terminate
Conclusion

- Implementation not (yet) publicly available
- Practical technique to help programmers classify error reports
- Tool-agnostic
Questions
### Language

Program $P$  

\[
\lambda \vec{a}. (\text{let } \vec{v} \text{ in } (s; \text{check}(p)))
\]

Statement $s$  

\[
v = e \mid \text{skip} \mid s_1; s_2 \\
| \text{if}(p) \text{ then } s_1 \text{ else } s_2 \\
| \text{while}^\rho(p)\{s\}[@p']?
\]

Expression $e$  

\[
v \mid c \mid c \ast e \mid e_1 \oplus e_2 \ (\oplus \in \{+,-\})
\]

Predicate $p$  

\[
e_1 \ominus e_2 \ (\ominus \in \{<,>,=\}) \\
| p_1 \land p_2 \mid p_1 \lor p_2 \mid \neg p
\]
Operational Semantics of the Language

\[ S \vdash v : S(v) \quad S \vdash c : c \]
\[ \vdash e_1 : c_1 \quad S \vdash e_2 : c_2 \]
\[ \vdash e_1 \oplus e_2 : c_1 \oplus c_2 \]

\[ S \vdash e_1 : c_1 \quad S \vdash e_2 : c_2 \]
\[ b = \begin{cases} 
\text{true} & \text{if } c_1 \odot c_2 \\
\text{false} & \text{otherwise}
\end{cases} \]
\[ \vdash e_1 \odot e_2 : b \]

\[ \vdash p : b \quad S \vdash e : c \]
\[ S \vdash \neg p : \neg b \quad S \vdash v = e : S[c/v] \quad S \vdash \text{skip} : S \]

\[ S \vdash p : \text{true} \quad S \vdash s_1 : S_1 \quad S \vdash s_1 \text{ else } s_2 : S_1 \quad S \vdash \text{if}(p) \text{ then } s_1 \text{ else } s_2 : S_1 \]

\[ S \vdash s_1 : S_1 \quad S_1 \vdash s_2 : S_2 \]
\[ S \vdash s_1; s_2 : S_2 \]

\[ S \vdash \text{loop}^p(p)\{s\} : S' \quad S' \vdash p' : \text{true} \quad S \vdash p : \text{false} \]
\[ S \vdash \text{while}^p(p)\{s\}[@p'] : S' \]
\[ S = [c_1/a_1, \ldots, c_k/a_k][0/v_1, \ldots, 0/v_n] \]
\[ S \vdash s : S' \quad S' \vdash p : b \]
\[ \vdash \lambda \vec{a}. (\text{let } \vec{v} \text{ in } (s; \text{check}(p)))(c_1, \ldots, c_k) : b \]
Operations on Symbolic Value Sets

\[ \theta_1 = \{(\pi_1, \phi_1), \ldots, (\pi_k, \phi_k)\} \]
\[ \theta_2 = \{(\pi'_1, \phi'_1), \ldots, (\pi'_n, \phi'_n)\} \]
\[ \theta = \bigcup_{i,j} ((\pi_i \oplus \pi'_j), (\phi_i \land \phi'_j)) \]
\[ \vdash \theta_1 \oplus \theta_2 : \theta \]

\[ \phi = \bigvee_{i,j} ((\pi_i \ominus \pi'_j) \land \phi_i \land \phi'_j) \]
\[ \vdash \theta_1 \ominus \theta_2 : \phi \]

\[ \theta' = \bigcup_{(\pi_i, \phi_i) \in \theta} (\pi_i, (\phi_i \land \phi)) \]
\[ \vdash \theta \land \phi : \theta' \]
Given known facts $\mathcal{I}$ and success condition $\phi$, a *proof obligation* is a formula $\Gamma$ such that

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

Let $\Gamma$ be a proof obligation query for $\mathcal{I}, \phi$, and let $\Pi_\rho$ be a mapping from variables to costs such that $\Pi_\rho(\alpha) = 1$ for abstraction variable $\alpha$ and $\Pi_\rho(\nu) = |Vars(\phi) \cup Vars(\mathcal{I})|$ for input variable $\nu$. Then,

$$Cost(\Gamma) = \sum_{\nu \in Vars(\Gamma)} \Pi_\rho(\nu)$$
Definitions for Proof Obligations II

Weakest Minimum Proof Obligation

Given known facts \( \mathcal{I} \) and success condition \( \phi \), a \textit{weakest minimum proof obligation} is a formula \( \Gamma \) such that

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

2. For any other \( \Gamma' \) that satisfies 1, either \( \text{Cost}(\Gamma) < \text{Cost}(\Gamma') \) or \( \text{Cost}(\Gamma) = \text{Cost}(\Gamma') \land (\Gamma \not\Rightarrow \Gamma' \lor \Gamma \Leftrightarrow \Gamma') \)
First, rewrite $\Gamma \wedge \mathcal{I} \models \phi$ as $\Gamma \models \mathcal{I} \Rightarrow \phi$.

**Cost of Partial Assignment**

Let $\sigma$ be a partial assignment for a formula $\phi$ and let $\Pi$ be a mapping from variables in $\phi$ to non-negative integers. The cost of partial assignment $\sigma$ is

$$Cost(\sigma) = \sum_{v \in Vars(\sigma)} \Pi(v)$$

**Minimum Satisfying Assignment**

Given mapping $\Pi$ from variables to costs, a minimum satisfying assignment of formula $\varphi$ is a partial assignment $\sigma$ to a subset of the variables in $\varphi$ such that

- $\sigma(\varphi) \equiv true$
- $\forall \sigma' \text{ such that } \sigma'(\varphi) \equiv true, \ Cost(\sigma) \leq Cost(\sigma')$
Minimum satisfying assignments help determine the minimum set of variables that any proof obligation $\Gamma$ must contain.

**Consistent Minimum Satisfying Assignment**

A minimum satisfying assignment $\sigma$ of $\varphi$ is consistent with $\varphi'$ if $\sigma(\varphi')$ is satisfiable.

Assignments that falsify $\mathcal{I}$ are not interesting. We want a minimum satisfying assignment to $\mathcal{I} \Rightarrow \phi$ that is consistent with $\mathcal{I}$.

Interpret $\sigma$ as a logical formula $F_\sigma$. $F_\sigma$ is a *strongest* proof obligation. It assigns each variable to a concrete value.
We want the *weakest sufficient condition* of $I \Rightarrow \phi$ containing only variables in $\sigma$.

**Lemma**

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

$$\forall V. (I \Rightarrow \phi)$$
Valid Answer to Proof Obligation Query

We say that the answer to a proof obligation query $\Gamma$ is valid iff:

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

Lemma

Let $\Gamma$ be a proof obligation query and suppose yes is a valid answer to this query. Then, the program is error-free.
Translate analysis variables into program expressions (easy)

Decompose complex queries to a series of simpler queries
  - If $\phi_1 \land \phi_2$ is an invariant, so are $\phi_1$ and $\phi_2$
  - If $\phi_1 \lor \phi_2$ is a witness, so are $\phi_1$ and $\phi_2$
  - Convert invariant queries to CNF and witness queries to DNF
  - Treat each clause as separate, independent query

We learn additional facts for every subquery
W := ∅

while (true) {
    if (Valid(\(I \Rightarrow \phi\))) return ERROR_DISCHARGED
    if (\(\exists \psi \in W. UNSAT(I \land \psi \land \phi)\)) return ERROR_VALIDATED
    V_1 = ComputeMSA(\(I \Rightarrow \phi, W \cup I, \Pi_p\))
    \(\Gamma = \text{ElimQuantifier}(\forall V_1. (I \Rightarrow \phi))\)
    V_2 = ComputeMSA(\(I \Rightarrow \neg \phi, W \cup I, \Pi_w\))
    \(\Upsilon = \text{ElimQuantifier}(\forall V_2. (I \Rightarrow \neg \phi))\)

    if (Cost(\(\Gamma\)) < Cost(\(\Upsilon\))) {
        Q_1 = FormInvariantQuery(\(\Gamma\))
        if (answer to Q_1 = YES) return ERROR_DISCHARGED
        W := W \cup \neg \Gamma
    } else {
        Q_2 = FormWitnessQuery(\(\Upsilon\))
        if (answer to Q_2 = YES) return ERROR_VALIDATED
        I := I \land \neg \Upsilon
    }
}
Implementation

- Implemented on top of Compass analysis framework for C programs
- Also reasons about heap objects, arrays and function calls
- Sources of imprecisions are loops, non-linear arithmetic, inline assembly, etc.
- Allow the user to answer *I don’t know*
- Uses own Mistral SMT solver to compute minimum satisfying assignments.
Isill Dillig, Thomas Dillig and Alex Aiken. Automated Error Diagnosis Using Abductive Inference