Dual Analysis for Proving Safety and Finding Bugs

Corneliu Popeea
Max Planck Institute
for Software Systems (MPI-SWS)
cpopeea@mpi-sws.org

Wei-Ngan Chin
Department of Computer Science
National University of Singapore
chinwn@comp.nus.edu.sg

ABSTRACT

Program bugs remain a major challenge for software developers and various tools have been proposed to help with their localization and elimination. Most present-day tools are based either on over-approximating techniques that can prove safety but may report false positives, or on under-approximating techniques that can find real bugs but with possible false negatives. In this paper, we propose a *dual* static analysis that is based on *only* over-approximation. Its main novelty is to concurrently derive conditions that lead to either success or failure outcomes and thus we provide a comprehensive solution for both proving safety and finding real program bugs. We have proven the soundness of our approach and have implemented a prototype system that is validated by a set of experiments.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*program analysis*

General Terms

Languages, Reliability, Verification

Keywords

Static analysis, Numerical domains

1. INTRODUCTION

Program errors are notoriously difficult to find and eliminate. Traditionally, program testing and model checking [6, 7] have been applied to detect the presence of real bugs. However, one shortcoming of the testing process is that it is unable to prove the absence of bugs, compromising on program safety. In contrast, static analysis which uses abstraction on program states can be used to prove program safety [4, 20]. It achieves this by showing that bad error states are not reachable via an exhaustive interpretation in the abstract domain. Due to approximation, static analysis may report false positives that are possible bugs that do not exist in practice. High incidents of false positives can make static analysis tools impractical to use for finding and eliminating bugs. As reported in the ASTRÉE project [2, 19], manual inspection of alarms (possible bugs) can be a very time-consuming process and may take several days even for simple alarms.

Recently, there have been some proposals [15, 8] that advocate for over-approximation techniques (based on static analysis) to be synergistically combined with under-approximation techniques (based on concrete execution or program testing). One main goal of this combination, as advocated in [8], is to leverage on the strengths of the two techniques so that program bugs or their absence can be discovered more accurately and effectively. While such a proposal can exploit the complementary strengths of its constituent techniques, it is also more complex to construct due to the need to combine *different* techniques and to consider potential interplays between them. Furthermore, it is often useful to explore what can be achieved within a single methodology before considering synergistic combinations of different techniques, to allow the strengths of each technique to be exploited.

In this paper, we shall propose a dual static analysis that is different from past approaches as both its components are based *primarily on over-approximation*. Our approach is also modular and computes (on a per method basis) trigger conditions for each bug expressed symbolically in terms of the method's parameters. Specifically, we support the *concurrent discovery* of three conditions for bugs, called *must-bug*, *may-bug* and *never-bug*, respectively. To illustrate the three different kinds of bugs, consider a simple example :

```
int foo(int x, int y)
{ if (x≤y) then { if (x>10) then ℓ1:error else 1 }
  else { if complexTest(x,y) then ℓ2:error
         else { if x≥y then 2 else ℓ3:error }
       } }
```

The bugs in our programs shall be flagged using a special `error` construct. This approach is simple but general as we can translate the more conventional (`assert c`) command for bug detection, directly to (`if c then skip else error`). The method `complexTest` denotes a predicate whose outcome cannot be precisely modelled by the underlying static analyser. For example modelling the predicate $x^3+y^3\geq 0$ is beyond the capability of linear arithmetic solvers. Accord-

SAC'10 March 22-26, 2010, Sierre, Switzerland.

ing to our analysis, the error at location ℓ_1 is a *must-bug* under the condition $\texttt{x}{\leq}\texttt{y} \wedge \texttt{x}{>}\texttt{10}$ that *must* lead to the error. In contrast, the error at location ℓ_2 is a *may-bug* as our analysis can only determine a trigger condition $\texttt{x}{>}\texttt{y}$ that *may* lead to this error, since its occurrence is still dependent on the second conditional with a statically unknown test. Lastly, the error at location ℓ_3 is a *never-bug* as our analysis can determine a trigger condition $\texttt{x}{>}\texttt{y}{\wedge}\texttt{x}{<}\texttt{y}$ that can never happen, namely `false`.

To describe our approach, we start from some basic observations on program execution. A machine configuration can be represented by $\langle s, e\rangle$ where s denotes the current stack/heap and e denotes the expression to evaluate. This configuration is said to be *closed* if $vars(e) \subseteq dom(s)$, that is all the free variables in e are present in s. Each reduction step of a closed configuration can be formalised using a small-step transition rule of the form: $\langle s, e\rangle{\hookrightarrow}\langle s_1, e_1\rangle$. Transitive application of the reduction steps are denoted by $\hookrightarrow^*$. The reduction rules are standard and are omitted for brevity.

For each given *complete* execution for a closed configuration, we expect one of three possible outcomes : (i) $\texttt{ok}(\langle s, e\rangle)$ to denote a successful execution $\langle s, e\rangle{\hookrightarrow}^*\langle s_1, k\rangle$ that results in a final value k, (ii) $\texttt{err}(\langle s, e\rangle)$ to denote a failed execution $\langle s, e\rangle{\hookrightarrow}^*\langle s_1, \bot\rangle$ that results in an error denoted by $\bot$, and (iii) $\texttt{loop}(\langle s, e\rangle)$ for an execution that does not terminate, denoted by $\langle s, e\rangle{\not\hookrightarrow}^*$. The consistency relation $s \models \phi$ holds when the values of the variables from the stack/heap s agree with the abstract formula ϕ.

Our strategy is to track two input conditions, denoted by `OK` and `ERR`, that over-approximate *all* executions that may lead to (i) `ok` outcomes, and (ii) `err` outcomes, respectively.

Definition 1. **(Entire Success Outcomes)**
Given a method with body e, we can capture a condition `OK` *on the method's inputs $\vec{v}$ that leads to* all *possible* `ok` *outcomes. This condition is an over-approximation that may include* some `err` *and* `loop` *outcomes. Formally:*
$\forall s \cdot dom(s){=}v \wedge closed(\langle s, e\rangle) \wedge \texttt{ok}(\langle s, e\rangle) \implies (s \models \texttt{OK})$

Definition 2. **(Entire Failure Outcomes)**
Given a method with body e, we can capture a condition `ERR` *on the method's inputs $\vec{v}$ that leads to* all *possible* `err` *outcomes. This condition is an over-approximation that may include* some `ok` *and* `loop` *outcomes. Formally:*
$\forall s \cdot dom(s){=}v \wedge closed(\langle s, e\rangle) \wedge \texttt{err}(\langle s, e\rangle) \implies (s \models \texttt{ERR})$

For example, consider the earlier `foo` example with two input parameters, `x` and `y`. Using our static analysis, we may compute two conditions that cover all the `ok` outcomes and all the `err` outcomes, respectively:
$\texttt{OK} = (\texttt{x}{\leq}\texttt{y} \wedge \texttt{x}{\leq}\texttt{10}) \vee \texttt{x}{>}\texttt{y}$.
$\texttt{ERR} = (\texttt{x}{\leq}\texttt{y} \wedge \texttt{x}{>}\texttt{10}) \vee \texttt{x}{>}\texttt{y}$.
Based on the two over-approximation results `OK` and `ERR`, we can determine the conditions for must-bug, may-bug and never-bug for each given method:

Definition 3. **(Never-Bug Condition)**
A condition `c` *on the inputs $\vec{v}$ of a method with body e is a never-bug condition if each of its inputs leads to either the* `ok` *or* `loop` *outcomes, but never the* `err` *outcome. This condition* `c` *can be computed using* $\texttt{OK} \wedge \neg\texttt{ERR}$ *where* $\neg$`ERR` *ensures that none of the* `err` *outcomes are possible. Formally:*
$\forall s{\cdot}dom(s){=}v{\wedge}closed(\langle s, e\rangle){\wedge}(s \models \texttt{OK}{\wedge}\neg\texttt{ERR}) \implies \texttt{ok}(\langle s, e\rangle){\vee}\texttt{loop}(\langle s, e\rangle)$

Definition 4. **(Must-Bug Condition)**
A condition `c` *on the inputs $\vec{v}$ of a method with body e is a must-bug condition if each of its inputs leads to either the* `err` *or* `loop` *outcomes, but never the* `ok` *outcome. This condition* `c` *can be computed using* `ERR` $\wedge \neg$`OK` *where* $\neg$`OK` *ensures that none of the* `ok` *outcomes are possible. Formally:*
$\forall s{\cdot}dom(s){=}v{\wedge}closed(\langle s, e\rangle){\wedge}(s \models \texttt{ERR}{\wedge}\neg\texttt{OK}) \implies \texttt{err}(\langle s, e\rangle){\vee}\texttt{loop}(\langle s, e\rangle)$

Definition 5. **(May-Bug Condition)**
A condition `c` *(on the inputs) of a method is a may-bug condition if each of its inputs leads to either* `ok`, `err` *or* `loop` *outcomes. This condition* `c` *arises from imprecision of the analysis and can be computed using* `OK` $\wedge$ `ERR` *which covers an overlap where all three outcomes are possible.*

A graphical illustration of these three categories of bug conditions is shown in Fig 1. The two circles denote the conditions for `OK` and `ERR`, while the three areas being partitioned by the two circles are the conditions for *never-bug*, *may-bug* and *must-bug*. At the extreme, all possible inputs could be classified under the may-bug category (may-bugs are sometimes denoted as false positives in the terminology of static analyzers). Our remedy is to minimise the overlap between the `OK` and `ERR` outcomes by using a more precise analysis based on a disjunctive numerical domain [20, 16], where needed. The dual analysis is considered precise when $\texttt{OK}{\wedge}\texttt{ERR} = \texttt{false}$. This scenario is desirable as it provides disjoint conditions for proving safety and finding bugs.

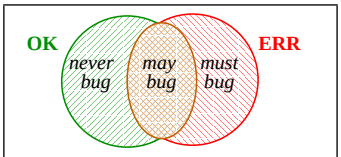

Figure 1: Classifying bugs

Due to our use of over-approximation, our analysis can only guarantee that a bug will occur, assuming the absence of non-termination outcome. This situation is related to partial correctness proofs, since we may sometimes report a must-bug when the outcome is actually a `loop`. Nevertheless, we will discuss how to discover some non-termination outcomes in Sec 3. Intuitively, even though `loop` outcomes may appear everywhere, the condition $\neg(\texttt{OK} \vee \texttt{ERR})$ is expected to capture exclusively `loop` outcomes.

Going back to the `foo` example, we may now compute this method's conditions for must-bug, may-bug and never-bug :

$$\begin{array}{ll}
\texttt{MUST_BUG} & = \neg(\texttt{OK}) \wedge \texttt{ERR} \\
& = \neg(\texttt{x}{\leq}\texttt{y}{\wedge}\texttt{x}{\leq}\texttt{10} \vee \texttt{x}{>}\texttt{y}) \wedge (\texttt{x}{\leq}\texttt{y}{\wedge}\texttt{x}{>}\texttt{10} \vee \texttt{x}{>}\texttt{y}) \\
& = \texttt{x}{\leq}\texttt{y} \wedge \texttt{x}{>}\texttt{10} \\
\texttt{MAY_BUG} & = \texttt{OK} \wedge \texttt{ERR} \\
& = (\texttt{x}{\leq}\texttt{y}{\wedge}\texttt{x}{\leq}\texttt{10} \vee \texttt{x}{>}\texttt{y}) \wedge (\texttt{x}{\leq}\texttt{y}{\wedge}\texttt{x}{>}\texttt{10} \vee \texttt{x}{>}\texttt{y}) \\
& = \texttt{x}{>}\texttt{y} \\
\texttt{NEVER_BUG} & = \texttt{OK} \wedge \neg(\texttt{ERR}) \\
& = (\texttt{x}{\leq}\texttt{y}{\wedge}\texttt{x}{\leq}\texttt{10} \vee \texttt{x}{>}\texttt{y}) \wedge \neg(\texttt{x}{\leq}\texttt{y}{\wedge}\texttt{x}{>}\texttt{10} \vee \texttt{x}{>}\texttt{y}) \\
& = \texttt{x}{\leq}\texttt{y} \wedge \texttt{x}{\leq}\texttt{10}
\end{array}$$

Thus, to analyse for must, may and never-bugs, we only need to determine the condition for possible successful execution `OK` and the condition for possible program errors `ERR`. Analysing both these outcomes *concurrently* is the main novelty behind our approach for capturing both must and may analyses under a dual static analysis. To the best of our knowledge, this approach has never been used in mainstream work on static analyses for simultaneously proving

$$
\begin{array}{lll}
P & ::= & meth^* \\
meth & ::= & t\ mn\ ((\texttt{[ref]}\ t\ v)^*)\ [\texttt{where}\ \Phi]\ [\{e\}] \\
t & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{int[]} \mid \texttt{void} \\
e & ::= & v \mid k \mid v{:=}e \mid e_1; e_2 \mid \ell : mn(v^*) \mid t\ v;\ e \\
 & & \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \ell : \texttt{error} \\
k & ::= & \texttt{true} \mid \texttt{false} \mid k^{\texttt{int}} \mid () \\
\Phi & ::= & \{\texttt{OK} : \phi_1,\ \texttt{ERR} : \phi_2\} \\
\phi & ::= & a_1{=}a_2 \mid a_1{\leq}a_2 \mid \neg\phi \mid \phi_1{\wedge}\phi_2 \mid \phi_1{\vee}\phi_2 \mid \exists v{\cdot}\phi \mid q(v^*) \\
a & ::= & k^{\texttt{int}} \mid v \mid v' \mid k^{\texttt{int}} * a \mid a_1 + a_2 \mid \cdots
\end{array}
$$

Figure 2: A simple imperative language

safety and finding bugs. In this paper, we formalise the methodology and conduct a set of experiments to validate the approach taken.

2. FORMALIZATION

A simple imperative language used to formalize our dual analysis is shown in Fig 2. We use *meth* for method declaration, t for type, and e for expression. This language is expression-oriented and uses a normalised form for which only variables are allowed as arguments to a method call or a conditional test. A preprocessor can transform loops into tail-recursive methods and handle arbitrarily nested arguments as well as insert unique labels for each method call and error location. Pass-by-reference parameters are marked in each method via the `ref` keyword, while the other parameters from v^* are passed-by-value.

The results of our analysis are encoded as summaries Φ computed for each method body using a modular static analysis approach [3]. Summaries are inferred bottom-up, starting with the methods lowest in the calling hierarchy. A method summary $\Phi = \{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\}$ combines all the traces that may lead to success outcomes under the `OK` label as ϕ_1 and all the traces that may lead to failure outcomes under the `ERR` label as ϕ_2. Our analysis captures each successful `OK` outcome with a postcondition that tracks the relation between inputs and output. The output of a method (or expression) is identified by a special **res** variable.

The syntactic form of ϕ is currently based on the first order theory of linear arithmetic. (Other theories may also be used provided they are amenable to fix-point analysis, as described in Sec 2.2.) State changes are modeled in a symbolic manner through transition formulae using two symbolic values per program variable. Given a program variable `v`, the prime notation $\texttt{v}'$ denotes the new value, while `v` itself denotes the old value of the program variable. Two transition formulae may be composed in a natural way using a composition operator $\circ_V$ with updating effects on a set of variables V, as follows: $\phi_1 \circ_V \phi_2 =_{df} \exists\ r_1..r_n \cdot \rho_1\,\phi_1 \wedge \rho_2\,\phi_2$ where $r_1, \ldots, r_n$ are fresh variables and $\rho_1 = [v_i' {\mapsto} r_i]_{i=1}^n$ and $\rho_2 = [v_i {\mapsto} r_i]_{i=1}^n$.

2.1 Forward Reasoning Rules

To compute method summaries, we shall now propose a set of forward rules shown in Fig 3. These rules resemble those from weakest precondition/strongest postcondition calculi with two important distinctions. Firstly, our integrated approach is entirely forward as it attempts to derive only strongest necessary conditions. Secondly, we use a set of outcomes to compute simultaneously two over-approximations. Discovering both true bugs and proving safety is made possible by this combination.

The rules are written in Hoare-style form using the judgement $\vdash \{\Phi_1\}\, e\, \{\Phi_2\}$. Given the `OK` outcome from Φ_1 (a transition from the beginning of the current method to the prestate before e's evaluation), the judgement derives Φ_2: firstly, a transition from the beginning of the current method to the poststate after e's evaluation; secondly, an `ERR` condition, in part from Φ_1 and also from possible errors happening during e's evaluation. Our rules use logical operators with set of outcomes as arguments: $\exists V{\cdot}\Phi$, $\Phi \vee \Phi$ and $\Phi \circ \Phi$. These logical operators are distributed to the components of Φ as follows:

$$
\begin{array}{l}
\exists V{\cdot}\{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\} \equiv \{\texttt{OK} : \exists V{\cdot}\phi_1, \texttt{ERR} : \exists V{\cdot}\phi_2\} \\
\{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\} \vee \{\texttt{OK} : \phi_A, \texttt{ERR} : \phi_B\} \\
\quad \equiv \{\texttt{OK} : \phi_1{\vee}\phi_A, \texttt{ERR} : \phi_2{\vee}\phi_B\} \\
\{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\} \circ_V \{\texttt{OK} : \phi_A, \texttt{ERR} : \phi_B\} \\
\quad \equiv \{\texttt{OK} : \phi_1{\circ_V}\phi_A, \texttt{ERR} : \phi_2{\vee}(\phi_1{\circ_V}\phi_B)\}
\end{array}
$$

The rule that involves $\circ$ is more complex. The `ERR` outcome of the result (condition: $\phi_2{\vee}(\phi_1{\circ}\phi_B)$) indicates either failure from the first argument (condition: ϕ_2), or from the success of the first argument followed by the failure of the second argument (condition: $\phi_1{\circ}\phi_B$).

For the [BLK] rule, constraints provide default values depending on the respective types. According to the language semantics, a possible set of defaults could be given as follows: *default*(`int`, `v`)≡`v`=`0`, *default*(`int[]`, `v`)≡`v`=`null` and *default*(`void`, `v`)≡`true`. Note that `true` may be used if there are no defaults. For the [CALL] rule, Φ is composed with the summary of the callee Φ_{mn}. In the case of boolean values, we encode them in the integer domain by using `0` for `false`, and `1` for `true`, as can be seen in the [IF] rule.

The most important rule in our reasoning process, [METH] handles method declarations. This rule uses as initial prestate $nochange(\{v_1..v_n\}) \equiv \bigwedge_{i=1}^n v_i{=}v_i'$. The first line of the rule uses the expression judgement to traverse the method body e, using the set of logical variables W to represent the inputs of the current method. For a recursive method mn, the rule [CALL] uses the following placeholder for its summary to be computed: $\Phi_{mn}{=}\{\texttt{OK} : mn\texttt{OK}(X), \texttt{ERR} : mn\texttt{ERR}(W)\}$. For this recursive case, the rules will derive a set of constraint abstractions, one for each outcome of the method. The third line of the rule [METH] collects in Q the constraint abstractions and then invokes an iterative fixed point analysis to compute the summary $\Phi'_{mn} = fix(Q)$. This fixed point analysis will be described in more detail in Sec 2.2.

While the summary of each user-defined method can be inferred, some methods are primitives in that they lack a method body and are provided instead with a summary formula. As an example, consider the following two primitives that may incur divide-by-zero and some array-related errors, respectively.

```
int div(int x, int y) where{OK:y≠0∧res=x/y, ERR:y=0}
void update(int[] a, int i, int v) where
  {OK:a≠null∧0≤i<a.len, ERR:a=null∨i<0∨i≥a.len}
```

Note that `null` may be modelled by the value `0`, while `nonnull` may be modelled by a value $\geq$`1`. In the implementation, we rely exclusively on an arithmetic constraint form as our abstraction domain and solver. Due to the use of the integer domain to encode array lengths (`a.len` > `0`), boolean values (`false` ≡ `0`, `true` ≡ `1`) or nullness (`null` ≡ `0`, `nonnull` ≡ $\geq$`1`), we ensure that derived formulae always

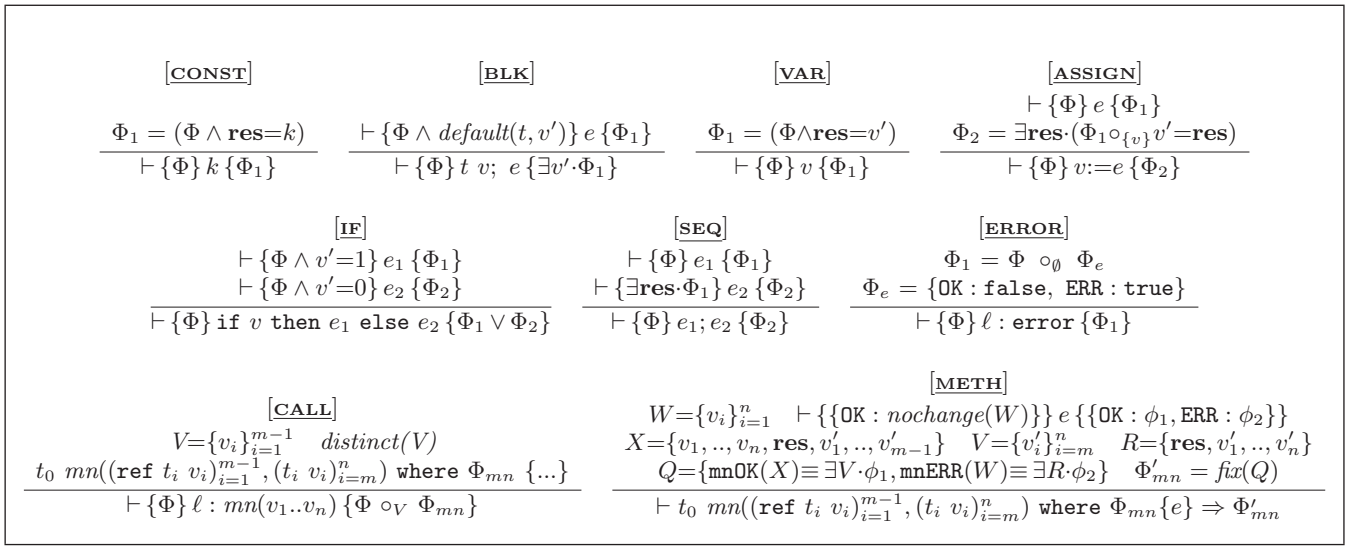

Figure 3: Forward reasoning rules

satisfy a type-invariant. This improves completeness by strengthening the underlying formulae. For example, given a variable of boolean sort `b`, we can add its type invariant to obtain: $\neg(\texttt{b}{=}0 \vee \texttt{b}{=}1) \equiv \texttt{false}$.

Example: Let us illustrate the forward reasoning process using a simple example, a method that assigns 0 to those elements in the array a from the range i to 1.

```
void g(int[] a, ref int i)
{  if (i≤0) then ()
   else { update(a,i,0); i:=i−1; g(a,i) } }
```

We will use the forward rules to derive formulae at intermediate points from the method g. To improve readability, formulae are simplified and we omit the tracking on nullness of array variables:

For $\texttt{update}(\texttt{a},\texttt{i},0)$, [CALL] rule is applied:
$\Phi_1 \equiv \{\texttt{OK} : \texttt{i}'{=}\texttt{i} \wedge \texttt{i}'{>}0 \wedge \texttt{i}'{<}\texttt{a.len}, \texttt{ERR} : \texttt{i}{>}0 \wedge \texttt{i}{\geq}\texttt{a.len}\}$
For $\texttt{i}{:=}\texttt{i}{-}1$, [ASSIGN] rule is applied:
$\Phi_2 \equiv \{\texttt{OK} : \texttt{i}'{=}\texttt{i}{-}1 \wedge \texttt{i}{>}0 \wedge \texttt{i}{<}\texttt{a.len}, \texttt{ERR} : \texttt{i}{>}0 \wedge \texttt{i}{\geq}\texttt{a.len}\}$
For $\texttt{g}(\texttt{a},\texttt{i})$, [CALL] rule is applied:
$\Phi_3 \equiv \Phi_2 \circ_{\{\texttt{i}\}} \{\texttt{OK} : \texttt{gOK}(\texttt{a},\texttt{i}_1,\texttt{i}'), \texttt{ERR} : \texttt{gERR}(\texttt{a},\texttt{i}_1)\}$
For the conditional expression, the [IF] rule is applied:
$\{\texttt{OK} : \phi_{\texttt{OK4}}, \texttt{ERR} : \phi_{\texttt{ERR4}}\} \equiv \{\texttt{OK} : \texttt{i}'{=}\texttt{i} \wedge \texttt{i}'{\leq}0, \texttt{ERR} : \texttt{false}\} \vee \Phi_3$
For the method's body, the [METH] rule is applied:
$\texttt{Q} = \{\texttt{gOK}(\texttt{a},\texttt{i},\texttt{i}') \equiv \phi_{\texttt{OK4}}, \texttt{gERR}(\texttt{a},\texttt{i}) \equiv \exists \texttt{i}' \cdot \phi_{\texttt{ERR4}}\}$

After simplifications, `Q` reduces to two independent constraint abstractions, one for the `OK` outcome, the other for the `ERR` outcome:

$$\texttt{gOK}(\texttt{a},\texttt{i},\texttt{i}') \equiv (\texttt{i}{\leq}0 \wedge \texttt{i}'{=}\texttt{i}) \vee (\texttt{i}{>}0 \wedge 0{\leq}\texttt{i}{<}\texttt{a.len} \wedge \exists \texttt{i}_1 \cdot\ \texttt{i}_1{=}\texttt{i}{-}1 \wedge \texttt{gOK}(\texttt{a},\texttt{i}_1,\texttt{i}'))$$

$$\texttt{gERR}(\texttt{a},\texttt{i}) \equiv ((\texttt{i}{>}0 \wedge \texttt{i}{\geq}\texttt{a.len}) \vee 0{<}\texttt{i}{<}\texttt{a.len} \wedge \exists \texttt{i}_1 \cdot\ (\texttt{i}_1{=}\texttt{i}{-}1 \wedge \texttt{gERR}(\texttt{a},\texttt{i}_1)))$$

2.2 Fixed-Point Analysis

Our approach to analyzing recursive methods is to build two constraint abstractions for `OK` and `ERR` outcomes. Once built, we can apply traditional fixed point analysis [4] to derive a closed-form formula for each recursive constraint abstraction. The constraint abstractions can be interpreted over various abstract domains. We will fix the abstract domain to the disjunctive completion of the polyhedron abstract domain [20, 16]. This abstract domain is essentially based on the polyhedron abstract domain [4], but is more fine-grained by allowing disjunctions of linear inequalities to be captured.

We briefly review the Kleene's fixed point iteration applied to the disjunctive polyhedron abstract domain. This domain is denoted by $(\mathcal{P}, \sqsubseteq)$, where unions of polyhedra are partially ordered by set inclusion. We write $\bot$ for the least element (in $\mathcal{P}$, the empty polyhedron or its representation, the formula false), and $\top$ for the greatest element (in $\mathcal{P}$, the entire n-dimensional space or its representation, the formula true). The join operation in the disjunctive polyhedron domain is the selective polyhedral hull [16]. A function f that is a self-map of a complete lattice is monotone if $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$. In particular, the constraint abstraction functions are monotone self-maps of the disjunctive polyhedron domain. This can be shown trivially as all the operators used to construct the constraint abstractions (in Fig. 3) are monotone.

The least fixed point of a monotone function f can be obtained by computing the ascending chain $f_0 = \bot$, $f_{n+1} = f(f_n)$, with $n{\geq}0$. If the chain becomes stationary, i.e., if $f_m = f_{m+1}$ for some m, then f_m is the least fixed point of f. In the case of a lattice infinite in height (as the lattice of polyhedra), an ascending chain may be infinite, and a widening operator must be used to ensure convergence. The widening ∇ is a binary operator to ensure that the iteration sequence $f_0 = \bot$, $f_{k+1} = f(f_k)$ followed by $f_{n+1} = f_n \nabla f(f_n)$, with $n > k$, converges. In this case, the limit of the sequence is known as a *post fixed point* of f. A post fixed point is a sound approximation of the least fixed point, and the criterion to verify that x is a post fixed point for f is that $f(x) \sqsubseteq x$.

Example: We apply least fixed point analysis to the constraint `gOK` obtained previously. The fixed point iteration starts with the least element of the abstract domain represented by $\texttt{gOK}_0(\texttt{a},\texttt{i},\texttt{i}'){\equiv}\texttt{false}$. After few iterations, we can

obtain a post fixed point $\texttt{gOK}_4(\texttt{a},\texttt{i},\texttt{i}')$ as follows:

$$\begin{array}{ll}
\texttt{gOK}_1(\texttt{a},\texttt{i},\texttt{i}') & \equiv(\texttt{i}\leq 0 \wedge \texttt{i}'=\texttt{i}) \vee (\texttt{i}>0 \wedge 0\leq\texttt{i}<\texttt{a.len} \\
& \wedge\exists i_1 \cdot\ i_1=i-1 \wedge g\texttt{OK}_0(a,i_1,i')) \equiv \texttt{i}\leq 0 \wedge \texttt{i}'=\texttt{i} \\
\texttt{gOK}_2(\texttt{a},\texttt{i},\texttt{i}') & \equiv (\texttt{i}\leq 0 \wedge \texttt{i}'=\texttt{i}) \vee (\texttt{i}=1 \wedge \texttt{i}'=0 \wedge 2\leq\texttt{a.len}) \\
\texttt{gOK}_3(\texttt{a},\texttt{i},\texttt{i}') & \equiv (\texttt{i}\leq 0 \wedge \texttt{i}'=\texttt{i}) \vee (1\leq\texttt{i}\leq\texttt{a.len}-1 \wedge \texttt{i}\leq 2 \wedge \texttt{i}'=0) \\
\texttt{gOK}_4(\texttt{a},\texttt{i},\texttt{i}') & \equiv_{\texttt{w}} (\texttt{i}\leq 0 \wedge \texttt{i}'=\texttt{i}) \vee (1\leq\texttt{i}\leq\texttt{a.len}-1 \wedge \texttt{i}'=0)
\end{array}$$

Note that $\equiv_{\texttt{w}}$ denotes a widening step [4], where the constraint $\texttt{i}\leq 2$ is dropped to ensure convergence of analysis. By a similar analysis, we derive the following closed-form formula: $\texttt{gERR}(\texttt{a},\texttt{i}) \equiv \texttt{i}>0 \wedge \texttt{i}\geq\texttt{a.len}$. The conditions corresponding to the two over-approximations $\exists\texttt{i}' \cdot \texttt{gOK}(\texttt{a},\texttt{i},\texttt{i}')$ and $\texttt{gERR}(\texttt{a},\texttt{i})$ do not overlap and thus we have a precise result (either never-bug or must-bug):

$$\begin{array}{ll}
\texttt{NEVER_BUG} & = \texttt{i}\leq 0 \vee 1\leq\texttt{i}\leq\texttt{a.len}-1 \\
\texttt{MUST_BUG} & = \texttt{i}>0 \wedge \texttt{i}\geq\texttt{a.len} \\
\texttt{MAY_BUG} & = \texttt{false}
\end{array}$$

We use the same fixed-point technique to analyze imperative loops. This is achieved by transforming loops to tail-recursive methods that use pass-by-reference parameters for variables that are updated in the loop.

3. NON-TERMINATION AS BUGS

Non-termination can be considered another source of bugs that is difficult to detect, since static analyses are typically formalised for safety property rather than liveness property. Nevertheless, our computation of both OK and ERR outcomes has the side-effect of being able to detect a sub-class of non-termination bugs. These non-termination bugs are due to recursive methods and may be discovered by fixed point analysis: with both OK and ERR outcomes exhaustively covered, any state left unreachable after analysis would have to belong to the non-termination outcome.

For example, consider a recursive method whose summary has been inferred to be $\{\texttt{OK}:\phi_1,\ \texttt{ERR}:\phi_2\}$. As these two outcomes cover all executions that either succeed or fail, whatever is left in the complement $\neg(\exists\texttt{R}\cdot\phi_1 \vee \phi_2)$ can only be executions leading to non-terminating **loop**, where R denotes the set of output variables including **res** from ϕ_1. This is on the assumption that all errors have been modelled and captured under the ERR outcome. For a precise classification of this class of non-termination bugs, we can use $\{\texttt{OK}:\phi_1,\ \texttt{ERR}:\phi_2, \texttt{ERR.fn.LOOP}:\neg(\exists\texttt{R}\cdot\phi_1 \vee \phi_2)\}$. In this case, fn denotes the name of the recursive method that is causing the non-termination bug.

To illustrate how non-termination bugs can be captured, consider the following recursive method:

```
int foo5(int i) { if i=10 then 1 else 2 + foo5(i+1) }
```

From fixed point analysis, we can obtain: $\{\texttt{OK} : \texttt{i}\leq 10 \wedge \textbf{res}=2(10-\texttt{i})+1\}\}$. Since the ERR condition is false, we can determine $\neg(\exists\textbf{res}\cdot\texttt{i}\leq 10 \wedge \textbf{res}=2(10-\texttt{i})+1)$ which simplifies to $(\texttt{i}>10)$ that is clearly a non-termination must-bug. Our summary can now be modified to the following :

$\{\texttt{OK}:\texttt{i}\leq 10 \wedge \textbf{res}=2(10-\texttt{i})+1,\ \texttt{ERR.foo5.LOOP}:\texttt{i}>10\}\}$

Once a non-termination bug has been detected for a given recursive method, it can be treated like any other bug where it could be propagated, downgraded to a may bug or proven safe, depending on the context of its callers. Lastly, we caution that we can only catch a subset of the non-termination bugs and *cannot* guarantee that all non-termination bugs are captured.

Benchmark Programs	Static checks	BLAST		DUALYZER	
		Result	(secs)	Result	(secs)
binary search	2	*	0.06	√	3.16
bubble sort	12	*	0.10	√	0.82
init array	2	√	1.15	√	0.26
merge sort	24	*	0.12	√	4.63
queens	8	√	3.45	√	1.47
quick sort	20	√	28.82	√	1.50
sentinel	4	*	1.31	*	0.12
FFT	62	*	0.57	√	13.50
LU	82	√	7.26	√	14.34
SOR	32	√	2.14	√	3.50
Linpack	166	*(exc)	408.1	√	38.91

Figure 4: Analysis of array-based programs without bugs. A tick √ indicates the tool verifies a program, while * is used when the tool reports a false bug.

4. EXPERIMENTAL RESULTS

We have implemented the proposed inference mechanisms in a tool named DUALYZER (from DUAL analYZER). The implementation uses the CIL infrastructure [13] and performs a further translation to our smaller CoreC language (e.g. loops and other intraprocedural control-flow are translated away). The prototype system was built using the Haskell language and the Glasgow Haskell compiler. We used the Omega library [18] to solve constraints in the Presburger arithmetic domain. Our test platform was a Pentium 3.0 GHz system with 2GBytes main memory, running Fedora 4.

The first objective of DUALYZER is to prove the absence of bugs, whenever possible. For this purpose, we tested our system on a set of small programs (up to 1 KLOC) with challenging recursion and intricate numerical computations. This set of programs, taken from [17], includes Fast Fourier Transform, LU decomposition, Successive Over-Relaxation (from SciMark suite [14]) and Linpack [5]. Fig. 4 summarizes the results obtained for each program. To quantify the analysis complexity of the benchmark programs, we counted the total number of static array accesses in the original programs (column 2). An array access operation is captured using a primitive method with an OK outcome (when the index is within the bounds of the array) and an ERR outcome (when the index is out-of-bounds).

We used the set of programs shown in Fig. 4 to make a comparison with the BLAST software verification system [10].[1] With a similar goal to DUALYZER, the BLAST system aims at statically proving safety or finding true bugs otherwise. Compared to our prototype, BLAST performed as well in proving safety for init-array, queens, quicksort, LU and SOR. However, BLAST was not able to prove the safety of binary-search, merge-sort and FFT for which it reported (false) bugs due to division being treated as an uninterpreted function. BLAST also reported a false bug for `bubble sort`; for Linpack the analysis ended prematurely with an exception (raised in the Simplify prover). Though an intended goal of BLAST is to report true bugs where possible, the representation of program states by symbolic con-

[1]We tested the 2.4 version of BLAST, available from http://mtc.epfl.ch/software-tools/blast/. The running times reported for BLAST correspond to several runs of abstraction refinement as we invoked BLAST with the default set of arguments.

straints ultimately leads to some approximation (for e.g. via uninterpreted functions) that could lead unwittingly to false alarms. This scenario does not occur for DUALYZER since the dual analysis helps distinguish program safety from must-bug, but can revert to may-bug reporting whenever there is uncertainty or loss of precision. DUALYZER decides that a program is verified when both must-bug and may-bug conditions of the "main" method are false. The binary-search example can indeed be proven correct using an abstract domain based on linear arithmetic. DUALYZER handles division by a constant, for e.g. `mid = a/2` is modelled by `2*mid<=a<=2*mid+1`. The sentinel example illustrates a pattern that cannot be verified by our tool, as it makes use of a sentinel/guard against reading past the end of the array.

In our previous work [17], we proposed a system for proving program safety based on a combination of forward and backward analysis. Comparatively, DUALYZER was simpler to design and implement (being based only on forward analysis). It is also considerably faster since it avoids the expensive backward analysis. Another difference is the capability of DUALYZER to confirm must-bugs which is illustrated by the next set of experiments.

4.1 Examples from the SYNERGY Paper

The SYNERGY system [8] complements the capabilities of predicate abstraction refinement (as in BLAST) with DART-style testing [7] to prove safety and also find true bugs. To test the Dualyzer capability, we used the set of all illustrative programs that were highlighted as figures in [8]. Our analysis took less than a second on each of these programs. Compared to SYNERGY, we performed equally well in finding real bugs in `ex_fig1`, `ex_fig4`, `ex_fig7`, and also proving safety for `ex_fig3`, `ex_fig6`, `ex_fig8`.

While able to prove safety and also find bugs, the SYNERGY system may fail to terminate due to abstraction refinement. The last example from [8] illustrates a case when SYNERGY fails to terminate as it generates longer and longer test sequences as (`y<0`), (`y+x<0`), (`y+2x<0`), and so on. The code `ex_fig9` is reproduced below:

```
void ex_fig9() { int x, y; x := 0; y := 0;
                 ℓ5 : while (y≥0) { y := y + x; }
                 ℓ6 : error; }
```

Using disjunctive fixed point analysis (with `m = 2` number of allowable disjuncts per formula), we can capture non-termination in the outcome of the `while` loop and prove that the error at ℓ_6 is unreachable:

$$\texttt{loop}_{(\texttt{m=2})} = \{\texttt{OK} : (x'{=}x \wedge y'{=}y \wedge y'{<}0) \vee (x'{=}x \wedge x{\leq}y'{\leq}x{+}y \wedge y'{<}0), \texttt{ERR.LOOP} : (x{\geq}0 \wedge y{\geq}0)\}$$

$$\texttt{ex\_fig9}_{(\texttt{m=2})} = \{\texttt{OK} : \texttt{false}, \texttt{ERR}.\ell_5.\texttt{LOOP} : \texttt{true}, \texttt{ERR}.\ell_6 : \texttt{false}\}$$

4.2 Finding Bugs in the Verisec Benchmark

We have also analyzed several buffer overflow vulnerabilities from the CVE database as grouped in the Verisec benchmark suite [11]. This suite contains testcases with the actual vulnerabilities as well as corrected versions of these testcases. We were surprised that DUALYZER found two must-bugs in the corrected versions of the testcases, bugs that were later confirmed and patched by the authors of the Verisec benchmark. The first must-bug was detected in a testcase extracted from the `Samba` implementation of the SMB networking protocol (CVE-2007-0453). It corresponds to a buffer access with an off-by-one error in the `r_strncpy` function. The second testcase is from the `SpamAssassin` open-source email filter and the must-bug corresponds to a non-termination bug. An excerpt from the corresponding C code is shown below:

```
#define BASE_SZ 2                    // from header file
#define BUFSZ  BASE_SZ+2
void message_write (char * msg, int len) {
  char buffer[BUFSZ];                // from loop_ok.c file
  int limit = BUFSZ − 4;
  for (int i=0; i<len; ){
    for (int j=0; i<len && j<limit; ){
      ...
      buffer[j] = msg[i];
      j++;
      ...
}}}
```

Since both the local variables `limit` and `j` are initialized to `0` and the value of `j` is increased through the inner-loop, the loop condition (`j<limit`) cannot be satisfied causing a non-terminating execution.

5. RELATED WORK

Most program analyses working towards the goal of bug-free programs can be divided broadly depending on their overall goal: proving safety of programs, finding bugs or approaches trying to prove safety and, at the same time, find bugs where possible.

Proving safety: The first camp is concerned with proving safety of programs. It needs to find a way to abstract all the possible concrete executions into a statically computable form. The abstraction may represent an over-approximation of the state at some program point computed using a forward traversal of the program as in the seminal paper of Cousot and Halbwachs [4]. Alternatively, the statically computed abstraction may represent an under-approximation of the state leading to a program error derived using a backward traversal of the program [21]. Various trade-offs between precision of the underlying abstraction and efficiency of the safety analysis have been explored: the interval domain, the polyhedron domain and the octagon domain are just a few of the proposed abstractions. In fact, safety analyzers that scale to large critical programs like ASTRÉE [2] use elaborate combinations of abstract domains to achieve maximum efficiency. As a summary for all these analyses, when they cannot prove safety, alarms that may include false positives will be signaled. The user of the analyzer is left with the job of manually distinguishing false alarms from real bugs.

Finding bugs: The second camp is primarily concerned with finding bugs in software, so that faulty programs could be quickly remedied. Traditionally, program testing has been used for detecting incorrect programs. More recently, systematic testing or concrete state space exploration has been implemented in model checkers like VeriSoft [6] or DART [7]. It attempts to search through all the feasible paths of the program, uncovering real bugs (with no false positives). Systematic testing cannot achieve full path coverage, so its results represent an under-approximation of all the concrete executions of the program. As search may not terminate in a reasonable amount of time, an upper limit is set in practice on the number of paths that are covered.

Proving safety + Finding bugs: Synergistic approaches for both proving safety and finding bugs usually rely on a combination of over and under approximation. In contrast, our proposal uses only forward over-approximations. Model checking based on abstraction refinement is often referred as CEGAR (counter-example guided automated refinement) and tools like SLAM [1] or BLAST [10] are based on this paradigm. In a first step, SLAM and BLAST perform a forward-directed over-approximating search for possible bugs. If no bugs are found, then the safety of the program has been proven. Otherwise, a counter-example trace is analyzed backward via symbolic reasoning to derive its weakest liberal precondition. If the counter-example is shown to be feasible, then a true bug is reported. If the counter-example is shown to be infeasible, the abstraction is refined and the search process is iterated.

In order to investigate the origin of the alarms raised by the static analyzer ASTRÉE [2], Rival used iterated forward-backward over-approximating analysis to prove safety of assertions [19]. Despite elaborate combination of abstractions, some alarms cannot be resolved by over-approximation alone. Understanding if an alarm is a true bug is facilitated by under-approximating techniques such as input selection or restriction to an execution pattern. The input selection process is not currently automated, but made easier by semantic slicing techniques. The process of restriction to an execution pattern and guiding the analysis towards true bugs is in general incomplete and may not converge. However, [19] reports that in practice all considered alarms from their set of benchmarks could be classified by their techniques. The classification of alarms is similar to ours in that an alarm indicates either a true bug or a non-terminating program. As an alternative to fixed-point computation, another approach to proving program safety and finding bugs uses constraint solving and exploits recent advances in SAT/SMT solving [9]. The novelty of this work is their use of an expressive domain containing disjunctions and conjunctions of linear inequalities. Currently their methodology is limited to small programs, since the constraint system that arises from disjunctive template invariants is quite large.

In a different context, [12] develops a dataflow analysis which over-approximates both the success and the failure sets of a logic program. There is no counterpart to the must results computed by our approach : the must-bug and never-bug conditions.

6. CONCLUSION

We advocate for a dual static analyser that is aimed at proving safety or discovering true bugs. To achieve both goals, a key innovation is the simultaneous capture of *error outcomes* and *successful outcomes*. A novelty of our formulation is the use of dual over-approximating analysis. This approach has allowed us to develop a single machinery for distinguishing must from may bugs, or prove safety in their absence. We have proven the correctness of our approach but for lack of space we omit the proof from the current paper. The main results confirm that a program never fails from an input of never-bug condition, never succeeds from an input of must-bug condition, and diverges from an input of loop condition.

Acknowledgements: We are grateful to Florin Craciun, Cristina David, David Lo and Alex Stefan for many useful comments on a previous version of this paper. This work was supported by A*STAR-funded project R-252-000-233-305.

7. REFERENCES

[1] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop*, pages 103–122, 2001.

[2] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, 2003.

[3] P. Cousot and R. Cousot. Modular static program analysis. In *CC*, 2002.

[4] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM POPL*, pages 84–96, 1978.

[5] J.J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:1–18, 2003.

[6] P. Godefroid. Model checking for programming languages using VeriSoft. In *ACM POPL*, 1997.

[7] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, 2005.

[8] B.S. Gulavani, T.A. Henzinger, Y. Kannan, A.V. Nori, and S.K. Rajamani. SYNERGY: A new algorithm for property checking. In *ACM FSE*, 2006.

[9] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *ACM PLDI*, pages 281–292, 2008.

[10] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *ACM POPL*, 2002.

[11] K. Ku, T. E. Hart, M. Chechik, and D. Lie. A buffer overflow benchmark for software model checkers. In *ASE*, 2007.

[12] K. Marriott and H. Søndergaard. Bottom-up dataflow analysis of normal logic programs. *J. Log. Program.*, 13(2&3), 1992.

[13] G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, 2002.

[14] National Institute of Standards and Technology. Java SciMark benchmark for scientific computing. `http://math.nist.gov/scimark2/`.

[15] C.S. Pasareanu, R. Pelánek, and W. Visser. Concrete model checking with abstract matching and refinement. In *CAV*, pages 52–66, 2005.

[16] C. Popeea and W.N. Chin. Inferring disjunctive postconditions. In *ASIAN CS Conference*, 2006.

[17] C. Popeea, D.N. Xu, and W.N. Chin. A practical and precise inference and specializer for array bound checks elimination. In *ACM SIGPLAN PEPM*, 2008.

[18] W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.

[19] X. Rival. Understanding the origin of alarms in ASTRÉE. In *SAS*, 2005.

[20] S. Sankaranarayanan, F. Ivancic, I. Shlyakhter, and A. Gupta. Static analysis in disjunctive numerical domains. In *SAS*, 2006.

[21] N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *ACM POPL*, pages 132–143, 1977.