Dual Analysis for Proving Safety and Finding Bugs

Corneliu Popeea[a], Wei-Ngan Chin[b]

[a] *Max Planck Institute for Software Systems (MPI-SWS)*
[b] *National University of Singapore*

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 *only* on 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.

1. Introduction

Program errors are notoriously difficult to find and eliminate. Traditionally, program testing and model checking [1, 2] 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 uses abstraction on program states and can be used to prove program safety [3, 4]. 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 bugs that do not exist in practice. High incidence of false positives can make static analysis tools impractical to use for finding and eliminating bugs. As reported in the ASTRÉE project [5, 6], manual inspection of alarms (possible bugs) can be a very time-consuming process and may take several days even for simple alarms.

Researchers have proposed to combine different techniques in order to automatically (in)validate alarms raised from static analysis. Software model checkers like Slam [7] or BLAST [8] are based on the counter-example guided abstraction refinement paradigm (CEGAR) [9]. One component of these tools computes state abstraction and is aimed at proving program safety. If a safety proof cannot be found, then a second component analyzes symbolically a counterexample generated by the state abstraction and, if successful, may be able to confirm the presence of a real bug. More recently, there have been

Email addresses: popeea@model.in.tum.de (Corneliu Popeea), chinwn@comp.nus.edu.sg (Wei-Ngan Chin)

some proposals 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) [10, 11]. One main goal of this combination is to search simultaneously for bugs and proofs, and use the information obtained from one search in the other search [10]. 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. It is often useful to explore what can be achieved within a single methodology as an alternative to synergistic combinations of different techniques.

In this paper, we shall propose a dual static analysis that is able to compute safety proofs and also find bugs using a single methodology. Our dual static analysis is different from past approaches as both its components are based on *over-approximation*. Our approach is modular as it computes for each method trigger conditions for each bug expressed symbolically in terms of the method's parameters. Specifically, we compute three conditions for bugs, called *must-bug*, *may-bug* and *never-bug*, respectively. To illustrate the three different kinds of bug conditions, consider the following 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 explicitly using a special `error` construct. We can translate the more conventional (`assert c`) command for bug detection as follows (`if c then () else error`). The method `complexTest(x, y)` denotes a predicate whose outcome cannot be precisely modelled by the static analysis. For example modelling the predicate $x^3 + y^3 \geq 0$ is beyond the capability of linear arithmetic solvers. According to our analysis, the error at location `l1` is a *must-bug* under the condition $(x \leq y \wedge x > 10)$ that *must* lead to the error. In contrast, the error at location `l2` is a *may-bug* as its occurrence is dependent on the second conditional with a statically unknown test. Our analysis determines in this case a trigger condition that *may* lead to this error, namely $(x > y)$. Lastly, the error at location `l3` is a *never-bug* as our analysis determines a trigger condition $(x > y \wedge x < y)$ that can never happen, namely *false*. In general, our classification of bugs is dependent on the precision of the abstract domain used by our dual static analysis. A more precise abstract domain can classify more of the may-bugs as either must-bugs or never-bugs.

To describe how our approach computes bug conditions, we start from some basic observations on program execution. A machine configuration is represented by a tuple $\langle s, e\rangle$ where s denotes the current stack/heap and e denotes the expression to evaluate. The stack s is a mapping from each variable in $dom(s)$ to its corresponding value. A machine configuration $\langle s, e\rangle$ is said to be *closed* if all free variables in e are present in s, i.e., $closed(\langle s, e\rangle) \triangleq vars(e) \subseteq dom(s)$. This definition assumes the absence of global variables. Each reduction step of a closed configuration is formalised using a small-step relation of the form: $\langle s, e\rangle \hookrightarrow \langle s_1, e_1\rangle$. The transitive reflexive closure of the reduction relation is denoted by $\hookrightarrow^*$. Given a closed configuration $\langle s, e\rangle$, we expect one of three possible outcomes for execution: (i) $\mathtt{ok}(\langle s, e\rangle)$ to denote a successful execution $\langle s, e\rangle \hookrightarrow^* \langle s_1, k\rangle$ that results in a final value k, (ii) $\mathtt{err}(\langle s, e\rangle)$ to denote a failed execution $\langle s, e\rangle \hookrightarrow^* \langle s_1, \bot\rangle$ that

results in an error denoted by $\perp$, and (iii) $\texttt{loop}(\langle s, e\rangle)$ for an execution that does not terminate, denoted by $\langle s, e\rangle \not\rightarrow^*$.

Our strategy is to compute two input conditions $\texttt{OK}_{mn}$ and $\texttt{ERR}_{mn}$ for each method mn, which over-approximate *all* executions that may lead to (i) `ok` outcomes, and (ii) `err` outcomes, respectively. In the definitions of these two input conditions, we shall use a relation between program states and symbolic formulae. The consistency relation $s \models \phi$ holds when $vars(\phi) \subseteq dom(s)$ and the values of the variables from the stack/heap s satisfy the formula ϕ, i.e., substituting each free variable v of ϕ with $s(v)$ gives $s(\phi) \equiv true$.

Definition 1. (All Successful Outcomes) *A condition* $\texttt{OK}_{mn}$ *on the inputs* V *of a method* mn *with body* e *leads to all possible* `ok` *outcomes if* $\forall s.((vars(e) \subseteq V \subseteq dom(s)) \wedge \texttt{ok}(\langle s, e\rangle) \rightarrow (s \models \texttt{OK}_{mn}))$.

In other words, every successful execution for a method mn starts from an initial state that is consistent with the formula $\texttt{OK}_{mn}$. Besides inputs that lead to `ok` outcomes, the condition $\texttt{OK}_{mn}$ may also include some inputs that lead to `err` and `loop` outcomes.

Definition 2. (All Failure Outcomes) *A condition* $\texttt{ERR}_{mn}$ *on the inputs* V *of a method* mn *with body* e *leads to all possible* `err` *outcomes if* $\forall s.((vars(e) \subseteq V \subseteq dom(s)) \wedge \texttt{err}(\langle s, e\rangle) \rightarrow (s \models \texttt{ERR}_{mn}))$.

Every failed execution for a method mn starts from an initial state that is consistent with the formula $\texttt{ERR}_{mn}$. Besides inputs that lead to `err` outcomes, the condition $\texttt{ERR}_{mn}$ may also include some inputs that lead to `ok` and `loop` outcomes.

For example, consider the earlier `foo` example with two input parameters, `x` and `y`. Using our static analysis, we compute two conditions that cover all the `ok` outcomes and all the `err` outcomes as follows.

$$\begin{array}{ll} \texttt{OK}_{\texttt{foo}} & \equiv (x \leq y \wedge x \leq 10) \vee (x > y) \\ \texttt{ERR}_{\texttt{foo}} & \equiv (x \leq y \wedge x > 10) \vee (x > y) \end{array}$$

Based on the two over-approximation results `OK` and `ERR`, we determine the conditions for never-bug, must-bug and may-bug for each given method according to the following definitions.

Definition 3. (Never-Bug Condition) *A condition on the inputs* V *of a method* mn *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, denoted by* $\texttt{NEVER\_BUG}_{mn}$*, can be computed using* $\texttt{OK}_{mn} \wedge \neg\texttt{ERR}_{mn}$ *where* $\neg\texttt{ERR}_{mn}$ *ensures that none of the* `err` *outcomes are possible. Formally:* $\forall s.((vars(e) \subseteq V \subseteq dom(s)) \wedge (s \models \texttt{NEVER\_BUG}_{mn}) \rightarrow (\texttt{ok}(\langle s, e\rangle) \vee \texttt{loop}(\langle s, e\rangle)))$.

Definition 4. (Must-Bug Condition) *A condition on the inputs* V *of a method* mn *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, denoted by* $\texttt{MUST\_BUG}_{mn}$*, can be computed using* $\texttt{ERR}_{mn} \wedge \neg\texttt{OK}_{mn}$ *where* $\neg\texttt{OK}_{mn}$ *ensures that none of the* `ok` *outcomes are possible. Formally:* $\forall s.((vars(e) \subseteq V \subseteq dom(s)) \wedge (s \models \texttt{MUST\_BUG}_{mn}) \rightarrow (\texttt{err}(\langle s, e\rangle) \vee \texttt{loop}(\langle s, e\rangle)))$.

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

A graphical illustration of the input conditions computed by our analysis is shown in Figure 1. The two circles denote program inputs consistent with the conditions `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 one extreme, all possible inputs could be classified under the may-bug category (may-bugs represent false positives in the terminology of static analyzers). Naturally it is preferable to minimise the overlap between the `OK` and `ERR` outcomes by using a more precise static analysis. At the other extreme, the dual analysis is considered precise when $\texttt{OK} \wedge \texttt{ERR} \equiv \mathit{false}$. This scenario is desirable as it provides disjoint conditions for proving safety and finding bugs.

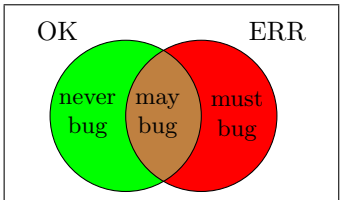

Figure 1: Classifying bugs.

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 show how to identify precisely some non-termination outcomes in Section 4.2. Intuitively, even though `loop` outcomes may appear in each area partitioned by the two circles, 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 never-bug, must-bug and may-bug as follows.

$$
\begin{array}{ll}
\texttt{NEVER\_BUG}_{\texttt{foo}} & \triangleq \texttt{OK}_{\texttt{foo}} \wedge \neg\texttt{ERR}_{\texttt{foo}} \\
& \equiv ((x \leq y \wedge x \leq 10) \vee x > y) \wedge \neg((x \leq y \wedge x > 10) \vee x > y) \\
& \equiv x \leq y \wedge x \leq 10 \\
\texttt{MUST\_BUG}_{\texttt{foo}} & \triangleq \neg\texttt{OK}_{\texttt{foo}} \wedge \texttt{ERR}_{\texttt{foo}} \\
& \equiv \neg((x \leq y \wedge x \leq 10) \vee x > y) \wedge ((x \leq y \wedge x > 10) \vee x > y) \\
& \equiv x \leq y \wedge x > 10 \\
\texttt{MAY\_BUG}_{\texttt{foo}} & \triangleq \texttt{OK}_{\texttt{foo}} \wedge \texttt{ERR}_{\texttt{foo}} \\
& \equiv ((x \leq y \wedge x \leq 10) \vee x > y) \wedge ((x \leq y \wedge x > 10) \vee x > y) \\
& \equiv x > y
\end{array}
$$

Thus, to analyse for must, may and never-bugs, we only need to determine the condition for possible successful executions `OK` and the condition for possible program errors `ERR`.

To summarize, our paper makes the following contributions.

- We propose a new *dual static analysis* to support either bug finding or a proof of the absence of bugs, where possible. Our work captures never, must and may bugs under a single static analysis, rather than using a host of special techniques. We achieve this using only over-approximating analysis by tracking *concurrently* both success and failure outcomes. To the best of our knowledge, this idea has never

been used in mainstream work on static analyses for simultaneously proving safety and finding bugs.

- We propose a technique to classify a sub-class of *definite non-terminations* as bugs. Our technique catches this class of non-termination bugs by explicitly identifying unreachable states during fixed point analysis.
- We prove the correctness of our technique and conduct a set of experiments to validate the feasibility of our proposal.

The current paper is a journal version of our previous work [12]. We have made formal the presentation of the analysis and operators on sets of outcomes (Section 3), and introduced conditions for individual bug classification (Section 4.1). We have also proven the correctness of our approach (Section 5), and provided details of dynamic semantics and proofs (in the Appendix). Lastly, we have provided a more extensive discussion on related work (Section 7).

2. A Core Language with Method Summaries

In this section, we introduce a simple sequential imperative language used to formalize our dual analysis. The syntax of this language is shown in Figure 2. This language retains only few constructs from the better-known C language and its purpose is to make the dual static analysis easier to formulate. A program P written in this language consists of a set of methods, either user-defined or primitive methods. All methods have a return type and a list of parameters. Each parameter has an optional `ref` keyword, a type and a name. The `ref` keyword indicates the passing mechanism of the parameter: if `ref` appears in the parameter declaration, then the parameter is passed-by-reference (any change in its value is visible to the caller); if the `ref` keyword is missing, the parameter is passed-by-value and any change to its value in the callee is not reflected to the caller.

Types represented by t can be either basic types or array type. User-defined method declarations include a method body represented by an expression e. The language is expression-oriented and uses a normalised form for expressions: only variables are allowed as arguments to a method call or a conditional test. The normalized form is obtained with the help of a simple pre-processor and, for brevity, we will use examples in a form that is not always normalized. Expression forms include local variable declaration, assignment, error construct, sequence of expressions, conditional and method call. This core language does not directly support global variables. Global variables can be translated to our core language by making each such global variable appear explicitly as a pass-by-reference parameter. A preprocessor transforms loops into tail-recursive methods and inserts unique labels for each method call and error location.

In our language, both primitive and user-defined method definitions include a method summary Φ. For a user-defined method, the summary is initially not computed and Φ is only a placeholder of the form $\{\texttt{OK} : mn\texttt{OK}(X), \texttt{ERR} : mn\texttt{ERR}(Y)\}$. This placeholder is derived syntactically by a preprocessor, with X and Y as two sets of logical variables: X includes both inputs and outputs of the method, Y includes only inputs of the method. The static analysis shall compute and replace this placeholder with a proper summary.

P	$::= prim^*\ meth^*$	(program)
$prim$	$::= t\ mn\ (([\texttt{ref}]\ t\ v)^*)\ \texttt{where}\ \Phi$	(primitive method)
$meth$	$::= t\ mn\ (([\texttt{ref}]\ t\ v)^*)\ \texttt{where}\ \Phi\ \{\ e\ \}$	(user-defined method)
t	$::= \texttt{boolean} \mid \texttt{int} \mid \texttt{void} \mid t[]$	(type)
e	$::= v \mid k \mid t\ v;\ e \mid v{:=}e \mid \ell : \texttt{error} \mid e_1; e_2$ $\mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \ell : mn(v^*)$	(expression)
k	$::= \texttt{true} \mid \texttt{false} \mid k^{\texttt{int}} \mid ()$	(constant)
Φ	$::= \{\texttt{OK} : \phi_1,\ \texttt{ERR} : \phi_2\}$	(method summary)
ϕ	$::= i \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \exists v.\phi \mid \neg\phi \mid q(v^*)$	(formula)
i	$::= k^{\texttt{int}}v_1 + \cdots + k^{\texttt{int}}v_n \leq k^{\texttt{int}}$	(linear inequality)
Q	$::= \{(q(v^*) \stackrel{\text{rec}}{=} \phi)^*\}$	(set of constraint abstractions)

Figure 2: A core imperative language and a language of constraint abstractions.

In addition to user-defined methods, our language includes primitive method declarations that lack a method body, but are instead given a symbolic description (summary) Φ. Primitive methods can be used to encode various operators for integer values ($\texttt{plus}, \texttt{minus}, \texttt{multiply}, \texttt{divide}$), for boolean values ($\texttt{or}, \texttt{and}, \texttt{not}$) or calls to (external) library methods for which the analysis does not have access to the code that implements the library. Potentially unsafe operators that handle array values are encoded as calls to primitive methods. For example, an array access $\texttt{a}[\texttt{i}]$ is viewed as $\texttt{sub}(\texttt{a}, \texttt{i})$, while an array update $\texttt{a}[\texttt{i}]{:=}\texttt{v}$ is converted to the primitive call $\texttt{update}(\texttt{a}, \texttt{i}, \texttt{v})$. The array update operation succeeds only if the index $\texttt{i}$ is within the bounds of a non-null array. The corresponding primitive method declaration is represented as follows.

$$\texttt{void update(int[] a, int i, int v) where}$$
$$\{\texttt{OK} : a \neq null \wedge 0 \leq i < a.len,\ \texttt{ERR} : a = null \vee i < 0 \vee i \geq a.len\}$$

In this declaration, the ERR outcome captures the null dereferencing, low bound and high bound errors. In general, we expect the outcomes of a primitive to represent non-overlapping conditions that completely characterize the inputs of the primitive.

The syntax of a formula ϕ is based on the first order theory of linear arithmetic with support for recursive constraint abstractions. The choice of the domain is influenced by the numerical properties that we want to capture. Other theories may also be used, provided they are amenable to the fixed-point analysis described in Section 3.2. While not explicit in the syntax of ϕ, an equality constraint ($k_1v_1 + \cdots + k_nv_n = k$) can be represented as a conjunction of two inequality constraints: ($k_1v_1 + \cdots + k_nv_n \leq k \wedge k_1v_1 + \cdots + k_nv_n \geq k$). A strict inequality ($k_1v_1 + \cdots + k_nv_n < k$) can also be represented using an axiom from integer arithmetic: ($k_1v_1 + \cdots + k_nv_n + 1 \leq k$). We use the syntactic shorthands $true \triangleq i \vee \neg i$ and $false \triangleq i \wedge \neg i$ for some predicate i. The existential quantifier $\exists v.\phi$ is used for eliminating intermediate variables.

For simplifying (via $\equiv$ operator) and checking satisfiability/validity of the non-recursive integer constraints, we use a complete decision procedure for the theory of linear arithmetic implemented in the Omega Test [13]. The Omega Test is an extension of the Fourier-Motzkin variable elimination algorithm to integer arithmetic. Despite its doubly-

exponential worst case complexity, the Omega Test has been shown to be efficient in practice [14, 15].

Using an integer domain, *null* may be modelled by the value 0, while nonnull may be modelled by a value $\geq$1. Due to the use of the integer domain to encode array lengths ($a.len > 0$), boolean values ($false \equiv 0, true \equiv 1$) or nullness ($null \equiv 0, nonnull \equiv \geq 1$), we ensure that derived formulae always satisfy a type-invariant. For example, using the type invariant of a boolean variable `b`, we have the following equivalence: $\neg(b{=}0 \vee b{=}1) \equiv false$. We make use of the negation $\neg$ operator that can be provided precisely for Presburger arithmetic. Nevertheless, our analysis can soundly accept an under-approximating negation operator for an abstract domain where a precise negation operator cannot be provided.

To motivate the presence of recursive constraints in the language of formulae ϕ, we give a high-level view of our static analysis. The input program is first analysed to obtain a call graph. Each node from the call graph represents a user-defined method, while each directed edge links a caller method to the corresponding callee method. Consequently, a group of mutually-recursive methods corresponds to a strongly connected component in the graph. Our analysis is modular [16]: it traverses the call graph in reverse topological order (bottom up) and each method (or group of methods) is analysed individually assuming unknown initial values. Specific to our analysis, each method is passed to a forward reasoning process (see Section 3.1) and an intermediate constraint representation is derived in the form of a constraint abstraction [17]. If the method is non-recursive, then the constraint abstraction is also non-recursive and a method summary can be immediately derived. If the method is recursive, then the constraint abstraction is passed on to a fixed-point approximation process parameterized by an abstract domain (see Section 3.2). In the case of mutually-recursive methods, the fixed-point process is done simultaneously for the corresponding constraint abstractions.

In the next section, we formalize the static analysis that computes method summaries based on forward reasoning and fixed-point analysis.

3. Formalization

The aim of our analysis is to compute for each user-defined method an abstraction for all the executions that may lead to success outcomes and a second abstraction for all the executions that may lead to failure outcomes. Similar to primitive methods, we label the former with `OK`, the latter with `ERR` and group them in a method summary of the following form: $\Phi_{mn} = \{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\}$. The successful outcome ϕ_1 includes a postcondition that tracks the relation between inputs and outputs. A special logical variable *res* is used to identify the result of a method (or expression). State updates are modeled in a symbolic manner through transition formulae using two symbolic values per program variable. Given a program variable `v`, the prime notation v' denotes the new value, while v itself denotes the old value of the program variable. Two transition formulae may be composed using a composition operator $\circ_V$ with updating effects on a set of variables $V = \{v_1, \ldots, v_n\}$, as follows: $\phi_1 \circ_V \phi_2 \triangleq \exists\ r_1, \ldots, r_n . \rho_1\, \phi_1 \wedge \rho_2\, \phi_2$ where $r_1, \ldots, r_n$ are fresh variables and $\rho_1 = [v_i' \mapsto r_i]_{i=1}^n$ and $\rho_2 = [v_i \mapsto r_i]_{i=1}^n$. With this notation, a summary of the method `foo` introduced in Section 1 can be represented as

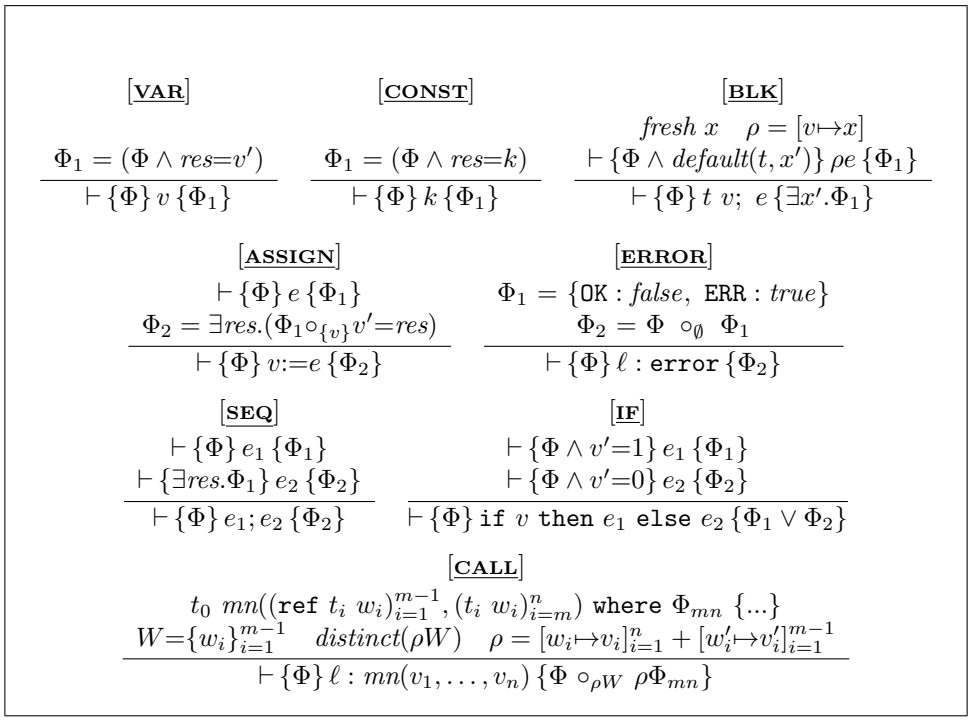

Figure 3: Forward reasoning rules

follows.

$$\Phi_{\texttt{foo}} = \{\texttt{OK} : (x \leq y \wedge x \leq 10 \wedge res = 1) \vee (x > y \wedge res = 2),\\ \texttt{ERR} : (x \leq y \wedge x > 10) \vee (x > y)\}$$

3.1. Forward Reasoning Rules

The first step towards computing method summaries uses forward reasoning to collect from each method body a constraint abstraction. This process is built around a static judgement of form, $\vdash \{\Phi_1\}\, e\, \{\Phi_2\}$, with roots in Hoare logic [18]. 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. The first outcome from Φ_2 is an `OK` transition from the beginning of the current method to the poststate after e's evaluation. The second outcome is an `ERR` condition, in part from errors captured by Φ_1 and also from errors possible during e's evaluation.

Figure 3 shows rules corresponding to each kind of expression from the core language. These rules use logical operators with sets of outcomes as arguments: $\exists V.\Phi$, $\Phi_1 \vee \Phi_1$ and

$\Phi_1 \circ \Phi_2$. These logical operators are distributed to the components of Φ as follows.

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

The rule that involves the composition operator $\circ$ is more complex. The ERR outcome of the result (the condition $\phi_2 \vee (\phi_1 \circ \phi_B)$) indicates either failure from the first argument (the condition ϕ_2), or from the success of the first argument followed by the failure of the second argument (the condition $\phi_1 \circ \phi_B$). The $\circ$ operator is not commutative as there is an implied order in the execution of the two sets of outcomes. For brevity, a singleton set can also be expressed as: $\phi \triangleq \{\texttt{OK}:\phi,\ \texttt{ERR}:\mathit{false}\}$. In that case, the operator $\Phi \wedge \phi$ is a shorthand for: $\Phi \circ_\emptyset \{\texttt{OK}:\phi,\ \texttt{ERR}:\mathit{false}\}$.

For the [**BLK**] rule, constraints provide default values depending on the type of the local variable being defined. According to the language semantics, a possible set of defaults could be given as follows: $\mathit{default}(\texttt{int}, v) \triangleq v{=}0$, $\mathit{default}(\texttt{int}[], v) \triangleq v{=}\mathit{null}$ and $\mathit{default}(\texttt{boolean}, v) \triangleq \mathit{true}$. Note that *true* may be used, if there are no defaults. The [**ERROR**] rule provides Φ_1 which unconditionally leads to a failure outcome. The [**IF**] rule encodes boolean values in the integer domain, as explained earlier, by using 0 for *false*, and 1 for *true*. For the [**CALL**] rule, the substitution ρ maps the formal arguments to the actual arguments from the method call. The premise $\mathit{distinct}(\rho W)$ ensures that the actual arguments passed by reference are distinct and then the resulting outcomes are obtained by composing Φ with the summary of the callee $\rho\Phi_{mn}$.

The last rule in our reasoning process uses an inference judgement $P \Rightarrow P[\mathit{meth}'/\mathit{meth}]$ to process each method declaration *meth* from the program P and updates the program with the inference result meth'. The inference rule is formulated as follows.

$$\frac{\begin{array}{c}
\mathit{meth} = t_0\ mn((\texttt{ref}\ t_i\ v_i)_{i=1}^{m-1}, (t_i\ v_i)_{i=m}^{n})\ \texttt{where}\ \Phi_{mn}\{e\} \\
\Phi_{mn}{=}\{\texttt{OK}{:}\ mn\texttt{OK}(X), \texttt{ERR}{:}\ mn\texttt{ERR}(Y)\} \quad X{=}\{v_1,\ldots,v_n,\mathit{res},v'_1,\ldots,v'_{m-1}\} \quad Y{=}\{v_i\}_{i=1}^{n} \\
\vdash \{\{\texttt{OK}:\mathit{nochange}(Y)\}\}\, e\, \{\{\texttt{OK}:\phi_1, \texttt{ERR}:\phi_2\}\} \\
V{=}\{v'_m,\ldots,v'_n\} \quad R{=}\{\mathit{res},v'_1,\ldots,v'_n\} \quad Q{=}\{mn\texttt{OK}(X) \stackrel{\text{rec}}{=} \exists V.\phi_1,\ mn\texttt{ERR}(Y) \stackrel{\text{rec}}{=} \exists R.\phi_2\} \\
\Phi'_{mn} = \mathit{fix}(Q) \quad \mathit{meth}' = \mathit{meth}[\Phi'_{mn}/\Phi_{mn}]
\end{array}}{P \Rightarrow P[\mathit{meth}'/\mathit{meth}]}$$

The first premises of the inference rule state the expected placeholder $\Phi_{mn} = \{\texttt{OK}:\ mn\texttt{OK}(X), \texttt{ERR}:\ mn\texttt{ERR}(Y)\}$ for the summary not yet computed. X and Y are two sets of logical variables such that X includes both inputs and outputs of the method, while Y includes only inputs of the method. Next, the expression judgement is used to traverse the method body e, using as initial condition $\mathit{nochange}(\{v_1,\ldots,v_n\}) \triangleq \bigwedge_{i=1}^{n} v_i{=}v'_i$. The results of the expression judgement, ϕ_1 and ϕ_2, are used to construct a set of constraint abstractions Q. The analysis invokes an iterative fixed-point analysis $\mathit{fix}(Q)$ to compute closed-form formulas for each of the constraint abstractions. This fixed-point analysis will be described in more detail in Section 3.2. Lastly, the inference rule updates the program P with meth' that includes the newly computed summary.

Our reasoning rules resemble those from strongest postcondition/weakest precondition calculi [19, 20] with two important distinctions. Firstly, we use a set of outcomes to

compute simultaneously two over-approximations. Secondly, our integrated approach is entirely forward. Discovering both true bugs and proving safety is made possible by this combination.

Example. Let us illustrate the forward reasoning process using a simple example, a method `g` that assigns the value `0` to those elements from the array `a` that are in 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 use the forward rules to derive formulae at the intermediate points from the method `g`. To improve readability, formulae are simplified and we omit tracking the nullness of the array variable `a`. The analysis of the else branch starts with the condition $\Phi_0 = \{\texttt{OK} : i' = i \wedge i' > 0,\ \texttt{ERR} : \mathit{false}\}$.

From the judgement $\vdash \{\Phi_0\}\,\texttt{update}(\texttt{a},\texttt{i},0)\,\{\Phi_1\}$, the rule [**CALL**] derives:
$\Phi_1 \equiv \{\texttt{OK} : i'{=}i \wedge i'{>}0 \wedge i'{<}a.len,\ \texttt{ERR} : i{>}0 \wedge i{\geq}a.len\}$
From the judgement $\vdash \{\Phi_1\}\,\texttt{i:=i}-1\,\{\Phi_2\}$, the rule [**ASSIGN**] derives:
$\Phi_2 \equiv \{\texttt{OK} : i'{=}i{-}1 \wedge i{>}0 \wedge i{<}a.len,\ \texttt{ERR} : i{>}0 \wedge i{\geq}a.len\}$
From the judgement $\vdash \{\Phi_2\}\,\texttt{g}(\texttt{a},\texttt{i})\,\{\Phi_3\}$, the rule [**CALL**] derives:
$\Phi_3 \equiv \Phi_2 \circ_{\{i\}} \{\texttt{OK} : \texttt{gOK}(a,i_1,i'),\ \texttt{ERR} : \texttt{gERR}(a,i_1)\}$
For the conditional expression, the rule [**IF**] derives:
$\{\texttt{OK} : \phi_{\texttt{OK4}},\ \texttt{ERR} : \phi_{\texttt{ERR4}}\} \equiv \{\texttt{OK} : i'{=}i \wedge i'{\leq}0,\ \texttt{ERR} : \mathit{false}\} \vee \Phi_3$
Finally, the following set of constraint abstractions is computed:
$Q = \{\texttt{gOK}(a,i,i') \stackrel{\text{rec}}{=} \phi_{\texttt{OK4}},\ \texttt{gERR}(a,i) \stackrel{\text{rec}}{=} \exists i'.\phi_{\texttt{ERR4}}\}$

After further simplifications, Q reduces to the following constraint abstractions, one for the `OK` outcome, the other for the `ERR` outcome.

$$\begin{array}{lcl}
\texttt{gOK}(a,i,i') & \stackrel{\text{rec}}{=} & (i \leq 0 \wedge i' = i) \vee \\
 & & (i > 0 \wedge 0 \leq i < a.len \wedge \exists i_1.\ i_1 = i{-}1 \wedge \texttt{gOK}(a,i_1,i')) \\
\texttt{gERR}(a,i) & \stackrel{\text{rec}}{=} & (i > 0 \wedge i \geq a.len) \vee \\
 & & (0 < i < a.len \wedge \exists i_1.\ (i_1 = i{-}1 \wedge \texttt{gERR}(a,i_1)))
\end{array}$$

For this example, the two constraint abstractions are independent, since the analyzed method `g` is tail-recursive. For general recursion, the composition operator from the [**CALL**] rule makes the mn`ERR` abstraction depend on the mn`OK` abstraction. Note however that, even for non-recursive methods, the integrated formulation of our dual analysis leads to sharing of the computation results. One crucial advantage of our integrated analysis is that it is more economical when compared to two analyses computing the over-approximations `OK` and `ERR` independently.

3.2. Fixed-Point Analysis

Once constraint abstractions are built for the `OK` and `ERR` outcomes, we apply fixed-point analysis [3] to derive a closed-form formula for each recursive constraint abstraction. The constraint abstractions can be interpreted over various abstract domains. In this presentation, we fix the abstract domain to the disjunctive completion of the polyhedron

abstract domain [4, 21]. This abstract domain is essentially based on the polyhedron abstract domain [3], 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 [21]. 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 (see Figure 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 $f(x) \sqsubseteq x$. We denote by *fix*(f) the post fixed-point computed using the previous sequence.

The join and the widening operators that we use are based on the notion of affinity to characterize how closely related is a pair of disjuncts. Finding related elements in the conjunctive (base) domain improves the precision of hull and widening operators as they have been lifted to the disjunctive (powerset extension of the base) domain. For further details on these abstract operators, we refer the reader to our previous work [21].

We illustrate the fixed-point analysis by applying it to the constraint abstraction $\texttt{gOK}$.

Example. The fixed-point iteration starts with $\texttt{gOK}_0$ initialized to the least element of the abstract domain, the formula *false*. After a number of iterations, we obtain a formula $\texttt{gOK}_4(a, i, i')$ as follows.

$$
\begin{array}{lcl}
\texttt{gOK}_0(a,i,i') & = & \mathit{false} \\
\texttt{gOK}_1(a,i,i') & \equiv & (i{\leq}0 \wedge i'{=}i) \vee (i{>}0 \wedge 0{\leq}i{<}a.len \wedge \exists i_1.\ i_1{=}i{-}1 \wedge \texttt{gOK}_0(a,i_1,i')) \\
 & \equiv & (i{\leq}0 \wedge i'{=}i) \\
\texttt{gOK}_2(a,i,i') & \equiv & (i{\leq}0 \wedge i'{=}i) \vee (i{>}0 \wedge 0{\leq}i{<}a.len \wedge \exists i_1.\ i_1{=}i{-}1 \wedge \texttt{gOK}_1(a,i_1,i')) \\
 & \equiv & (i{\leq}0 \wedge i'{=}i) \vee (i{=}1 \wedge i'{=}0 \wedge 2{\leq}a.len) \\
\texttt{gOK}_3(a,i,i') & \equiv & (i{\leq}0 \wedge i'{=}i) \vee (i{>}0 \wedge 0{\leq}i{<}a.len \wedge \exists i_1.\ i_1{=}i{-}1 \wedge \texttt{gOK}_2(a,i_1,i')) \\
 & \equiv & (i{\leq}0 \wedge i'{=}i) \vee (i{=}1 \wedge i'{=}0 \wedge 2{\leq}a.len) \vee (i{=}2 \wedge i'{=}0 \wedge 3{\leq}a.len) \\
 & \equiv & (i{\leq}0 \wedge i'{=}i) \vee (1{\leq}i{\leq}a.len{-}1 \wedge i{\leq}2 \wedge i'{=}0) \\
\texttt{gOK}_4(a,i,i') & \equiv & (i{\leq}0 \wedge i'{=}i) \vee (i{>}0 \wedge 0{\leq}i{<}a.len \wedge \exists i_1.\ i_1{=}i{-}1 \wedge \texttt{gOK}_3(a,i_1,i')) \\
 & \equiv & (i{\leq}0 \wedge i'{=}i) \vee (1{\leq}i{\leq}a.len{-}1 \wedge i{\leq}3 \wedge i'{=}0) \\
 & \equiv_W & (i{\leq}0 \wedge i'{=}i) \vee (1{\leq}i{\leq}a.len{-}1 \wedge i'{=}0) \\
\texttt{gOK}_5(a,i,i') & \equiv & (i{\leq}0 \wedge i'{=}i) \vee (i{>}0 \wedge 0{\leq}i{<}a.len \wedge \exists i_1.\ i_1{=}i{-}1 \wedge \texttt{gOK}_4(a,i_1,i')) \\
 & \equiv & (i{\leq}0 \wedge i'{=}i) \vee (1{\leq}i{\leq}a.len{-}1 \wedge i'{=}0)
\end{array}
$$

At the last step, the post fixed-point criterion is satisfied, since we obtained $\texttt{gOK}(\texttt{gOK}_4) \sqsubseteq \texttt{gOK}_4$ where $\texttt{gOK}_5 = \texttt{gOK}(\texttt{gOK}_4)$. The operator $\equiv_W$ denotes a widening step [21], where the constraint $i \leq 3$ is dropped to ensure convergence of analysis.

By a similar analysis, we derive a closed-form formula for the ERR outcome.

$$\texttt{gERR}(a, i) \equiv (i > 0 \wedge i \geq a.len)$$

The summary of the method g consists of the two results from fixed-point analysis.

$$\Phi_{\texttt{g}} = \mathit{fix}(\{\texttt{gOK}, \texttt{gERR}\}) = \{\texttt{OK} : (i \leq 0 \wedge i' = i) \vee (1 \leq i \leq a.len{-}1 \wedge i' = 0),\ \texttt{ERR} : (i > 0 \wedge i \geq a.len)\}$$

The input conditions corresponding to the two over-approximations, $\exists i'.\texttt{gOK}(a, i, i')$ and $\texttt{gERR}(a, i)$, do not overlap. Thus, we have a precise analysis result for every method input (either never-bug or must-bug).

$$\begin{array}{ll} \texttt{NEVER\_BUG}_g & \equiv i \leq 0 \vee 1 \leq i \leq a.len - 1 \\ \texttt{MUST\_BUG}_g & \equiv i > 0 \wedge i \geq a.len \\ \texttt{MAY\_BUG}_g & \equiv \mathit{false} \end{array}$$

We use the same fixed-point analysis for imperative loops. It is folklore that loops can be viewed as tail-recursive functions. Pass-by-reference parameters are used to model variables that may be updated across loop iterations. For example, consider the following loop where only the variables r and i are updated.

```
while (i<n) do { r:=r + 2; i:=i + 1 }
```

To model the effect of this loop, our system transforms it automatically to the following tail-recursive method.

```
void mn_tail(ref int r, ref int i, int n) {
  if (i<n) then { r:=r + 2; i:= i + 1; mn_tail(r,i,n) }
  else () }
```

Using the fixed-point analysis described above, the following summary is computed for mn_tail: $\{\texttt{OK} : (i \geq n \wedge r' = r \wedge i' = i) \vee (i < n \wedge i' = n \wedge r' = r + 2(n - i)),\ \texttt{ERR} : \mathit{false}\}$.

We have found the idea of using tail-recursion for loop analysis helpful towards the construction of a smaller core language for our analysis. The same technique has been used in program analysis, e.g., summary-based aliasing analysis [22].

4. Extensions

In this section, we introduce two techniques to further improve our static analysis. First, we show how to identify the origin of each detected bug. Second, our analysis can capture a sub-class of definite non-termination as bugs.

4.1. Individual Bug Classification

Our analysis may pinpoint the precise location of a discovered error by the notation $\{\texttt{ERR}.\ell : \texttt{e}\}$ where ℓ is a sequence of program locations that corresponds to the method call chain leading to the specified error. As an example, consider the following two method definitions.

```
void foo3(int x) {          void foo4(int x, int y) {
  l1 : foo4(x, x + 1);        if x=y then l3 : error
  l2 : foo4(x, 3);   }        else ()  }
```

The error in `foo4` will only be flagged during execution if `x=y`. Our analysis computes the summary of `foo4` as $\{\texttt{OK} : x \neq y, \texttt{ERR.l3} : x = y\}$. This `ERR.l3` error is impossible when invoked from the context `l1:foo4(x, x+1)`, but can occur when it is invoked from `l2:foo4(x, 3)`. The summary for the `foo3` method is therefore $\{\texttt{OK} : x \neq 3, \texttt{ERR.l1.l3} : \mathit{false}, \texttt{ERR.l2.l3} : x = 3\}$. A *false* condition at `ERR.l1.l3` indicates that the bug at `l1` can never occur. We omit this unreachable error outcome from the summary of `foo3`: $\{\texttt{OK} : x \neq 3,\ \texttt{ERR.l2.l3} : x = 3\}$. In contrast, the bug at call `l2` can occur under the input condition $x = 3$. The label `l2.l3` is used to indicate the call chain leading to the bug at final destination `l3`. Each label and trigger condition characterize a must or a may bug that violates safety.

To provide precise reporting of errors, we only need to change two rules from those shown in Figure 3. The modified rules are shown below.

$$\frac{\Phi_1{=}\Phi \circ_\emptyset \{\texttt{OK} : \mathit{false},\ \texttt{ERR}.\ell : \mathit{true}\}}{\vdash \{\Phi\}\, \ell : \texttt{error}\, \{\Phi_1\}} \quad [\textbf{ERROR}]$$

$$\frac{\begin{array}{c} t_0\ mn((\texttt{ref}\ t_i\ w_i)_{i=1}^{m-1}, (t_i\ w_i)_{i=m}^{n})\ \texttt{where}\ \Phi_{mn}\ \{...\} \\ W{=}\{w_i\}_{i=1}^{m-1} \quad \mathit{distinct}(\rho W) \quad \rho = [w_i{\mapsto}v_i]_{i=1}^{n} + [w_i'{\mapsto}v_i']_{i=1}^{m-1} \end{array}}{\vdash \{\Phi\}\, \ell : mn(v_1, \ldots, v_n)\, \{\Phi \circ_{\rho W}\ \mathit{add}(\rho\Phi_{mn}, \ell)\}} \quad [\textbf{CALL}]$$

The label ℓ is used to trace the calling hierarchy of each error. The *add* command prepends the label ℓ to the label sequence from the summary argument.

$$\mathit{add}(\{\texttt{OK} : \phi,\ (\texttt{ERR}.\ell_i : \phi_i)^*\}, \ell) \triangleq \{\texttt{OK} : \phi,\ (\texttt{ERR}.\ell.\ell_i : \phi_i)^*\}$$

In the case of recursive methods, all bugs that originate from the same location in the recursive method are grouped under the same error outcome. This ensures that the label sequences are always finite, and shall be bounded by the static height of the method call hierarchy. Note that all elements in a set of mutual-recursive methods have the same height in the call hierarchy. Using this representation for method summaries, we introduce a definition to classify each individual bug as follows.

Definition 6. (Individual Bug Classification) *Consider a method with the summary:* $\{\texttt{OK} : \phi_0,\ \texttt{ERR}.\ell_1 : \phi_1, \ldots,\ \texttt{ERR}.\ell_n : \phi_n\}$.

- *A bug* $\texttt{ERR}.\ell_i$ *is said to be a never-bug if* $\phi_i \equiv \mathit{false}$.

- *A bug* $\texttt{ERR}.\ell_i$ *is said to be a must-bug if* $(\phi_i \wedge \phi_0 \equiv \mathit{false})$ *and* $\phi_i \wedge (\bigvee_{j \in \{1..n\}-\{i\}} \phi_j)) \equiv \mathit{false}$ *(and* $\phi_i \not\equiv \mathit{false}$*).*
- *A bug* $\texttt{ERR}.\ell_i$ *is said to be a may-bug otherwise.*

Two bugs $\texttt{ERR}.\ell_a$ and $\texttt{ERR}.\ell_b$ are said to be *closely-related* if either $\phi_a \rightarrow \phi_b$ or $\phi_b \rightarrow \phi_a$. As closely-related bugs may be indistinguishable from each other, we shall group them together in the must-bug category, if the condition $(\phi_a \vee \phi_b) \wedge \phi_0$ is unsatisfiable. This amalgamation of closely-related must-bugs allows us to report when a bug from the amalgamated set will definitely be triggered as a must-bug, even if we are unable to pinpoint the exact bug from this set.

4.2. Non-Termination as Bugs

Non-terminating program executions represent another source of bugs that can be detected statically by specialized analyses [23]. 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 conservatively over-approximated, any input state left unreachable after analysis corresponds to some non-terminating execution.

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 $\neg(\exists R.\phi_1 \vee \phi_2)$ can only correspond to executions leading to non-terminating `loop`, where R denotes the set of output variables (including *res*) from ϕ_1. This characterization is done on the assumption that all errors have been modelled and captured under the `ERR` outcome. It leads to the following total summary: $\{\texttt{OK} : \phi_1,\ \texttt{ERR} : \phi_2, \texttt{ERR.fn.LOOP} : \neg(\exists R.\phi_1 \vee \phi_2)\}$. In this case, `fn` denotes the name of the recursive method that is causing the non-termination bug.

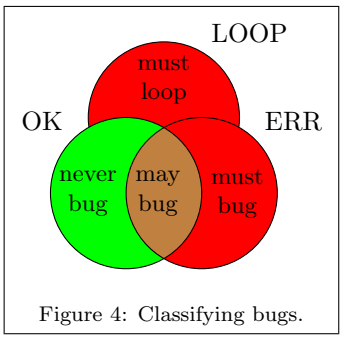

Figure 4: Classifying bugs.

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 obtain the following summary: $\{\texttt{OK} : i \leq 10 \wedge \mathit{res} = 2(10-i)+1\}$. Since the `ERR` condition is *false*, we derive $\neg(\exists \mathit{res}.i \leq 10 \wedge \mathit{res} = 2(10-i)+1) \equiv (i > 10)$, the condition corresponding to a non-termination must-bug. The resulting total summary is the following :

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

Once a non-termination bug has been detected for a given recursive method, it is 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. Note that we can only catch a subset of the non-termination bugs and cannot guarantee that all non-termination bugs are captured.

5. Correctness

In this section, we shall outline the proofs that our analysis is sound in proving safety and finding bugs. While the concrete state is captured by the stack $s \in S$, the abstract state that we infer is captured by a transition formula $\phi \in D$ between the unprimed and primed variables. We shall use the following operators: an abstraction operator $\alpha : S \rightarrow D$, a prestate operator $PreSt(\phi)$ to capture the relation between unprimed variables of ϕ and a poststate operator $PostSt(\phi)$ to capture the relation between primed variables of ϕ. Consistency relations between the concrete and abstract domains are defined to agree either with a state, $s \models \phi$, or with a pair of both pre- and post-state, $(s_1, s_2) \models \phi$. Using the previous notations, Theorem 1 states that the summary outcomes that are inferred are a conservative approximation of the program state expected from the evaluation of the program.

Theorem 1 (Soundness of Summary Outcomes). *For every e_1, s_1 and P_1 such that $(s_1, s_1) \models P_1$ and $\vdash \{P_1\}\, e_1\, \{\{\texttt{OK} : O_1, \texttt{ERR} : E_1\}\}$,*

1. *if $\langle s_1, e_1\rangle \hookrightarrow^* \langle s_n, k\rangle$ then $(s_1, [res \mapsto k] + s_n) \models O_1$ (the success outcome O_1 is sound).*
2. *if $\langle s_1, e_1\rangle \hookrightarrow^* \langle s_n, \bot\rangle$ then $s_1 \models E_1$ (the error outcome E_1 is sound).*

Proof sketch: This result can be shown using induction on the length of the reduction sequence. The main part of the proof is based on a subject reduction lemma that proves the induction step corresponding to an arbitrary reduction step: $\langle s_i, e_i\rangle \hookrightarrow \langle s_{i+1}, e_{i+1}\rangle$. In particular, the lemma proves that the success outcome for e_{i+1} is stronger than the one inferred from e_i: $O_{i+1} \rightarrow O_i$ (similarly $E_{i+1} \rightarrow E_i$). By repeated applications, we can conclude that the success and the error outcomes obtained from the inference of the original expression e_1 are sound. □

The full proof of Theorem 1 is included in Appendix B and is formulated in the style of correctness proofs from the program analysis textbook authored by Nielson, Nielson and Hankin [24]. The proof relies on the definition of the dynamic semantics of the core language given in Appendix A.

We shall now state four results that are corollaries of the main theorem. The first corollary confirms that we have a true error from the must-bug condition; secondly, we can guarantee a safe execution from the never-bug condition. Thirdly, we have a diverging execution from the loop condition. The fourth corollary applies when neither of the previous three cases holds: for inputs that satisfy the may-bug condition, it is possible to have either a safe execution, a true error or a diverging execution.

Corollary 2 (Definite Error from Must-Bug). *For every e_1, s_1 and P_1 such that $(s_1, s_1) \models P_1$, $\vdash \{P_1\}\, e_1\, \{\{\texttt{OK} : O_1, \texttt{ERR} : E_1\}\}$ and $s_1 \not\models PreSt(O_1)$, then either the evaluation is failed $\langle s_1, e_1\rangle \hookrightarrow^* \langle s_n, \bot\rangle$ or $\langle s_1, e_1\rangle \not\hookrightarrow^*$ does not terminate.*

Proof: Theorem 1 confirms that if the execution is successful then the consistency relation $s_1 \models PreSt(O_1)$ holds. Consequently, if the consistency relation does not hold, then the execution cannot be successful; it either fails or diverges. □

Corollary 3 (Definite Safety from Never-Bug). *For every e_1, s_1 and P_1 such that $(s_1, s_1) \models P_1$, $\vdash \{P_1\}\, e_1\, \{\{\texttt{OK} : O_1, \texttt{ERR} : E_1\}\}$ and $s_1 \not\models PreSt(E_1)$, then either the evaluation is successful $\langle s_1, e_1\rangle \hookrightarrow^* \langle s_n, k\rangle$ or $\langle s_1, e_1\rangle \not\hookrightarrow^*$ does not terminate.*

Proof: Theorem 1 confirms that if the execution fails then the consistency relation $s_1 \models PreSt(E_1)$ holds. Consequently, if the consistency relation does not hold, then the execution cannot fail; it is either successful or it diverges. □

Corollary 4 (Definite Non-termination from Loop Condition). *For every e_1, s_1 and P_1 such that $(s_1, s_1) \models P_1$, $\vdash \{P_1\}\, e_1\, \{\{\texttt{OK} : O_1, \texttt{ERR} : E_1\}\}$, $s_1 \not\models PreSt(O_1)$ and $s_1 \not\models PreSt(E_1)$, then $\langle s_1, e_1\rangle \not\hookrightarrow^*$ does not terminate.*

Proof: Using the previous two corollaries, we denote by s_1 a state that does not satisfy neither of the following consistency relations $s_1 \not\models PreSt(O_1)$ and $s_1 \not\models PreSt(E_1)$. Then the execution is neither successful nor failed. The only possible alternative is that $\langle s_1, e_1\rangle \not\hookrightarrow^*$ does not terminate. □

Corollary 5 (Indefinite Execution Outcome from May-Bug). *For every e_1 , s_1 and P_1 such that $(s_1, s_1) \models P_1$, $\vdash \{P_1\}\, e_1\, \{\{\texttt{OK} : O_1, \texttt{ERR} : E_1\}\}$ and $s_1 \models PreSt(O_1) \wedge PreSt(E_1)$, then all the three evaluation alternatives are possible: $\langle s_1, e_1\rangle \hookrightarrow^* \langle s_n, k\rangle$, $\langle s_1, e_1\rangle \hookrightarrow^* \langle s_n, \bot\rangle$ or $\langle s_1, e_1\rangle \not\hookrightarrow^*$ does not terminate.*

Proof: This corollary is vacuously true, since the three alternatives are the only possible outcomes for program execution. □

6. Experimental Results

We have implemented the proposed inference mechanisms in a tool named DUALYZER (from DUAL analYZER). The implementation uses the CIL infrastructure [25] 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 [26]. We used the Omega library [14] 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 [27]. This set of programs includes Fast Fourier Transform, LU decomposition, Successive Over-Relaxation (from SciMark suite [28]) and routines from the Linpack library [29]. Figure 5 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). Our system presently does not track the actual contents of the array, since this may require pointer analysis that is not supported by our system. We have thus only modelled indexes for arrays as integer values that must be bounded by the lower and upper bounds of each array that is being accessed. 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 Figure 5 to make a comparison with the BLAST software verification system [8, 30]. [1] With a similar goal to DUALYZER, the

[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.

Benchmark Programs	Source (lines)	Rec. constr.	Static checks	BLAST		DUALYZER	
				Result	(secs)	Result	(secs)
`binary_search`	31	1	2	×	0.06	√	3.16
`bubble_sort`	39	2	12	×	0.10	√	0.82
`init_array`	11	1	2	√	1.15	√	0.26
`merge_sort`	58	3	24	×	0.12	√	4.63
`queens`	39	2	8	√	3.45	√	1.47
`quick_sort`	43	2	20	√	28.82	√	1.50
`sentinel`	17	1	4	×	1.31	×	0.12
`FFT`	336	9	62	×	0.57	√	13.50
`LU`	191	10	82	√	7.26	√	14.34
`SOR`	84	5	32	√	2.14	√	3.50
`Linpack`	903	25	166	×	408.1	√	38.91

Figure 5: Analysis of programs without bugs. A tick √ indicates that the tool verifies the safety of the checks from a program, while × is used when the tool reports a false bug.

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`, `quick_sort`, `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 constraints ultimately leads to some approximation (e.g., via uninterpreted functions) that could lead unwittingly to false alarms. This scenario does not occur for DUALYZER since the dual analysis helps to 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 example, `mid` = `a/2` is modelled by $2{*}mid \leq a \leq 2{*}mid{+}1$. The `sentinel` example illustrates a pattern that cannot be verified by our tool, as it makes use of a sentinel/guard value against reading past the end of the array. This example is thus reported as one that has a may bug.

In our previous work [27], 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.

6.1. Examples from the SYNERGY *Paper*

The SYNERGY system [10] complements the capabilities of predicate abstraction refinement (as in BLAST) with DART-style testing [2] to prove safety and also find true bugs. To test DUALYZER's capabilities, we used the illustrative programs that were highlighted as figures in the SYNERGY paper [10]. See Figure 6 for results. 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`.

Benchmark Programs	SYNERGY	DUALYZER
ex_fig1	BUG	BUG
ex_fig3	SAFE	SAFE
ex_fig4	BUG	BUG
ex_fig6	SAFE	SAFE
ex_fig7	BUG	BUG
ex_fig8	SAFE	SAFE
ex_fig9	ABORT	BUG-`LOOP`

Figure 6: Examples from the SYNERGY paper [10]

While able to prove safety and also find bugs, the SYNERGY system may fail to terminate due to abstraction refinement. The last example, `ex_fig9`, illustrates a case for which 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; }
```

If restricted to fixed-point analysis with one disjunct, our system is only able to report a must-bug at `l6`. We denote results from such conjunctive fixed-point analysis by subscript (`m=1`), where `m` indicates the number of disjuncts. The loop summary is computed as follows.

$$\texttt{loop}_{(\texttt{m=1})} \equiv \{\texttt{OK} : (x'{=}x \wedge y'{\leq}y \wedge y'{<}0)\}$$

This conjunctive formula is unable to capture the non-termination property of the loop. Combined with the information prior to the loop ($x{=}0 \wedge y{=}0$), the summary of the method can only confirm the presence of a must-bug at `l6` (if the program terminates).

$$\texttt{ex\_fig9}_{(\texttt{m=1})} \equiv \{\texttt{OK} : \mathit{false},\ \texttt{ERR.l6} : \mathit{true}\}$$

Using disjunctive fixed-point analysis (with `m=2` number of allowable disjuncts per formula), we can capture non-termination in the summary of the `while` loop and prove that the error at `l6` is unreachable.

$$\begin{aligned}\texttt{loop}_{(\texttt{m=2})} \quad &\equiv \{\texttt{OK} : (x'{=}x \wedge y'{=}y \wedge y'{<}0) \vee (x'{=}x \wedge x{\leq}y'{\leq}x{+}y \wedge y'{<}0),\\ &\quad\ \texttt{ERR.LOOP} : (x{\geq}0 \wedge y{\geq}0)\}\\ \texttt{ex\_fig9}_{(\texttt{m=2})} &\equiv \{\texttt{OK} : \mathit{false}, \texttt{ERR.l5.LOOP} : \mathit{true}, \texttt{ERR.l6} : \mathit{false}\}\end{aligned}$$

Thus, with increased precision, our analysis is able to re-classify a must-bug more accurately and indicate a source of non-terminating executions.

6.2. 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 [31]. [2] 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 constants.h
#define BUFSZ  BASE_SZ+2
                          // from loop_ok.c file
void  message_write (char * msg,  int len) {
  char buffer[BUFSZ];
  int limit = BUFSZ - 4;
  for (int i=0; i<len;  ){
    for (int j=0; i<len && j<limit;  ){
      ...
      buffer[j] = msg[i];
      j++;
      i++;
      ...
}}}
```

This example contains a nested loop structure, where the terminating condition of the inner loop depends on the values of the variables `j`, `i`, `len` and `limit`. Both `j` and `i` are incremented on every loop iteration, while `len` and `limit` are kept unchanged. However, the variable `limit` is (unwittingly) initialized by the programmer to `0`. Since both the variables `limit` and `j` are initially `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.

7. Related Work

Most program analyses working towards the goal of bug-free programs can be divided broadly depending on their immediate objective: proving safety of programs, finding bugs or approaches trying to prove safety and, at the same time, find bugs where possible. We summarize some representative approaches from these three categories in Figure 7. The second column from the figure lists the direction in which the program is traversed, either forward (FW), backward (BW) or a combination of the two. The presented approaches can also be classified depending on the approximation done: either over-approximating, under-approximating, exact symbolic execution or a combination. Note that the effectiveness

[2] We tested the version 0.1 of the Verisec suite, made available by the authors from `http://se.cs.toronto.edu/index.php/Verisec_Suite`.

Analysis	Direction	Approximation	Modularity	Goal
Suzuki et al.[32]	BW	under	no	safety
ASTRÉE [5]	FW	over	no	safety
VeriSoft [1]	FW	under	no	bugs
DART [2], EXE [36]	FW	under+sym	no	bugs
Saturn [37]	FW	over+under	yes	bugs(1)
SLAM [7], BLAST [8]	FW+BW	over+sym	yes	safety+bugs
Rival[6]	FW+BW	over+under	no	safety+bugs(2)
SMASH [11]	FW+BW	over+under+sym	yes	safety+bugs
DUALYZER (this paper)	FW	over	yes	safety+bugs(2)

Figure 7: Prominent features of various analyses for proving safety and finding bugs

of the symbolic execution is inherently limited by the constraint solver or the theorem prover that is used. Column 4 shows whether the analysis is designed to be modular, where each method is analysed in isolation to derive a summary. The method summary would further be used instead of reanalyzing the method at each of its call sites. Finally, column 5 shows what is the general goal of the analysis: safety, bugs or a combination of the two. The alarms reported by the analysis can be classified as *bugs(1)* (an alarm that is either a true bug or a false positive), *bugs(2)* (an alarm that is a true bug, if the program terminates), or *bugs* (an alarm that is unconditionally a true bug).

Proving safety. The first camp is concerned with proving safety of programs and its proponents are shown in the top lines of the Figure 7. In this case, an analysis 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 [3]. 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. This second approach computes loop invariants using the induction-iteration method pioneered by Suzuki and Ishihata [32]. Various trade-offs between precision of the underlying abstraction and efficiency of the safety analysis have been explored: the interval domain [33], the polyhedron domain [3] and the octagon domain [34] are just a few of the proposed abstractions. In fact, safety analyzers that scale to large critical programs like ASTRÉE [5] or C Global Surveyor [35] use elaborate combinations of abstract domains to achieve maximum efficiency while maintaining an acceptable level of precision. 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 [1]. 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. As an improvement, DART [2] attempts to find more errors in a systematic fashion by keeping a stack of conditional tests encountered during execution. The gathered conditionals are used to generate new test cases that would allow deeper branches to be explored. It also combines concrete with symbolic execution in order to alleviate the limitations of the constraint solver used in symbolic execution. Whenever the constraint solver does not know how to resolve a conditional test, DART simplifies this constraint using the concrete values of the inputs involved in the test. EXE [36] even used symbolic execution to explore conditional branches exhaustively. Symbolic solvers are typically restricted in scope and may be inefficient, but EXE provided some solutions to this difficulty. To reduce the complexity of the automated testing approach, an extension of DART named SMART [38] generates the tests compositionally on a per method basis. Another effort towards making testing more directed is Check 'n' Crash [39] which uses error reports from static checking (possible false positives) to generate test cases. It solves the constraints from an ESC/Java [40] error report to find concrete inputs, which are then confirmed to be true bugs by testing. Because this approach has only an intra-procedural view of a single method, the inputs that crash the analyzed method may not in fact be used by the caller.

Closer in spirit to over-approximating static analyses, bug-finding tools like FindBugs [41] or Saturn [37] take some unsound (under-approximating) decisions in order to minimize the number of false positives and achieve scalability. For example, Saturn only considers some aliasing possibilities between the parameters of a method and only analyzes a bounded number of iterations through a loop. FindBugs uses an approach based on code templates to search for common program bugs in Java programs. While this approach may be heuristic in nature, FindBugs has been shown to be quite useful in practice as it is capable of detecting subtle bugs with less than 50% false positive rate. Due to this combination of heuristic over- and under-approximations, these tools aim to find bugs more precisely and in a scalable fashion, but neither guarantee program safety, nor to report all possible bugs.

Proving safety + Finding bugs. Synergistic approaches for both proving safety and finding bugs usually rely on a combination of over and under approximation. Model checking based on abstraction refinement is often referred as CEGAR (counter-example guided abstraction refinement) and tools like SLAM [7] or BLAST [8] 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 backwards via symbolic reasoning to derive its weakest liberal precondition. If the derived precondition is satisfiable, then the counter-example is deemed to be feasible and a bug is reported. If instead the counter-example is shown to be infeasible, then the abstraction is refined and the search process is iterated.

In another attempt to find real bugs or prove safety where possible, SYNERGY [10] combines predicate abstraction with DART-style program testing. Initially, SYNERGY tries to prove safety by using an over-approximating analysis. If this attempt fails, directed testing is used either to prove the abstract error trace as feasible (true bug) or help

in the abstraction refinement process. The obstacle that SLAM-like tools encounter in refining the abstraction for programs with long deterministic loops is thus overcome using directed testing. SYNERGY uses a fairly complex integration of two distinct techniques for intra-procedural analysis and can only report at most one real bug at a time. Furthermore, it may sometimes fail to find a bug and also fail to prove safety, as DART-style testing remains incomplete. Developed concurrently with our work [42], SMASH [11] generalizes the algorithm proposed in the SYNERGY paper to an inter-procedural analysis based on method summaries. SMASH computes total may-summaries using predicate abstraction and partial must-summaries using directed testing. Impressive experimental results show that the alternating may-must analysis implemented in SMASH is able to scale to a large codebase of Windows 7 device drivers. While DUALYZER is based on a different combination of dual analyses, it also benefits from the compositionality of summary-based analysis.

In order to investigate the origin of the alarms raised by the static analyzer ASTRÉE [5], Rival used iterated forward-backward over-approximating analysis [43] to prove safety of assertions [6, 44]. Despite an 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. In practice, Rival reports that all the considered alarms from the studied set of benchmarks could be classified by these techniques [6]. The classification of alarms is similar to ours in that an alarm indicates either a true bug or a non-terminating program.

Another closely related work by Dillig et al [45] aims for *a sound*, *scalable* and *complete* path-sensitive analysis. The authors developed may/must analyses that are *complete* for programs with finitary states, as they use satisfiability- and validity-preserving formulae transformations. May-analysis captures an over-approximation for *strongest necessary condition*, while must-analysis captures an under-approximation for *weakest sufficient condition*. Comparatively, we have used a dual over-approximating analysis rather than a pair of over- and under-approximating analyses. This gives us a uniform mechanism that has avoided the need for constraint specialization and backward analysis. Secondly, our approach has been designed to identify a sub-class of non-termination bugs, where possible.

As an alternative to fixed-point computation, another approach to proving program safety and finding bugs uses constraint-based invariant generation and exploits recent advances in SAT solving [46]. This work uses an expressive domain containing disjunctions and conjunctions of linear inequalities. Currently their experimental results are limited to small programs, since the constraint system that arises from disjunctive template invariants is quite large. While we do not have access to the binary of the VS3 tool [46], we borrow from their examples the two that took VS3 longest to verify. The first example is based on the McCarthy 91 function instrumented with a safety property (72 seconds of analysis time reported in [46]) and the second example is an array merge procedure (80 seconds of analysis time reported in [46]). While it is unfair to compare running times on different machines, we report that DUALYZER took less than 3 seconds to analyze each of these examples, and obtained a safety proof for the first example and a sufficient safety precondition weaker than the one determined by VS3.

Finally, in a different context, Marriott and Søndergaard develop a dataflow analysis that over-approximates both the success and the failure sets of a logic program [47]. Two interesting applications illustrated in this paper are highlighting errors in a logic program and program specialization. Their formulation uses many-valued logics, which could be a convenient formal framework for our approach too.

8. Conclusion

In this paper, 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 *successful outcomes* and *error outcomes* using only 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 and 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.

We have also conducted some initial experiments on small programs to validate the potential for our proposed system for discovering true bugs and for proving program safety. Future work should consider techniques that would allow our approach to scale to larger programs. We would need to incorporate suitable alias analysis solutions for pointer-based programs, and also engineer suitable mechanisms that would support a good trade-off between precision and scalability through the use of bounded disjunctive formulae. Another avenue for future work is to study the applicability of our dual static analysis to verify numerical properties of domain-specific languages [48, 49] where expressivity is more of a concern than scalability to large codebases.

Acknowledgement. We thank the anonymous reviewers for pointers to closely related work and insightful comments that strengthened our paper.

References

[1] P. Godefroid, Model checking for programming languages using Verisoft, in: Proceedings of the ACM Symposium on Principles of Programming Languages, ACM, 1997, pp. 174–186.

[2] P. Godefroid, N. Klarlund, K. Sen, DART: directed automated random testing, in: V. Sarkar, M. W. Hall (Eds.), Proceedings of the ACM Conference on Programming Language Design and Implementation, ACM, 2005, pp. 213–223.

[3] P. Cousot, N. Halbwachs, Automatic discovery of linear restraints among variables of a program, in: Proceedings of the ACM Symposium on Principles of Programming Languages, ACM, 1978, pp. 84–96.

[4] S. Sankaranarayanan, F. Ivancic, I. Shlyakhter, A. Gupta, Static analysis in disjunctive numerical domains, in: K. Yi (Ed.), SAS, volume 4134 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 3–17.

[5] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival, A static analyzer for large safety-critical software, in: Proceedings of the ACM Conference on Programming Language Design and Implementation, ACM, 2003, pp. 196–207.

[6] X. Rival, Understanding the origin of alarms in Astrée, in: C. Hankin, I. Siveroni (Eds.), SAS, volume 3672 of *Lecture Notes in Computer Science*, Springer, 2005, pp. 303–319.

[7] T. Ball, S. K. Rajamani, Automatically validating temporal safety properties of interfaces, in: M. B. Dwyer (Ed.), SPIN, volume 2057 of *Lecture Notes in Computer Science*, Springer, 2001, pp. 103–122.

[8] T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre, Lazy abstraction, in: Proceedings of the ACM Symposium on Principles of Programming Languages, ACM, 2002, pp. 58–70.
[9] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement, in: CAV, Lecture Notes in Computer Science, Springer, 2000, pp. 154–169.
[10] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, S. K. Rajamani, SYNERGY: a new algorithm for property checking, in: [51], pp. 117–127.
[11] P. Godefroid, A. V. Nori, S. K. Rajamani, S. Tetali, Compositional may-must program analysis: unleashing the power of alternation, in: M. V. Hermenegildo, J. Palsberg (Eds.), Proceedings of the ACM Symposium on Principles of Programming Languages, ACM, 2010, pp. 43–56.
[12] C. Popeea, W.-N. Chin, Dual analysis for proving safety and finding bugs, in: S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal, C.-C. Hung (Eds.), SAC, ACM, 2010, pp. 2137–2143.
[13] W. Kelly, V. Maslov, W. Pugh, et al, The Omega Library Version 1.1.0 Interface Guide, 1996.
[14] W. Pugh, A practical algorithm for exact array dependence analysis, Commun. ACM 35 (1992) 102–114.
[15] W. Pugh, D. Wonnacott, Constraint-based array dependence analysis, ACM Trans. Program. Lang. Syst. 20 (1998) 635–678.
[16] P. Cousot, R. Cousot, Modular static program analysis, in: [52], pp. 159–178.
[17] J. Gustavsson, J. Svenningsson, Constraint abstractions, in: O. Danvy, A. Filinski (Eds.), PADO, volume 2053 of *Lecture Notes in Computer Science*, Springer, 2001, pp. 63–83.
[18] C. A. R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (1969) 576–580.
[19] R. W. Floyd, Assigning meanings to programs, in: Proceedings of the Symposium in Applied Mathematics, American Math. Society, 1967, pp. 19–32.
[20] E. W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Commun. ACM 18 (1975) 453–457.
[21] C. Popeea, W.-N. Chin, Inferring disjunctive postconditions, in: M. Okada, I. Satoh (Eds.), ASIAN, volume 4435 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 331–345.
[22] B. Hackett, A. Aiken, How is aliasing used in systems software?, in: [51], pp. 69–80.
[23] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, R.-G. Xu, Proving non-termination, in: G. C. Necula, P. Wadler (Eds.), Proceedings of the ACM Symposium on Principles of Programming Languages, ACM, 2008, pp. 147–158.
[24] F. Nielson, H. R. Nielson, C. Hankin, Principles of Program Analysis, Berlin: Springer-Verlag, 1999.
[25] G. C. Necula, S. McPeak, S. P. Rahul, W. Weimer, CIL: Intermediate language and tools for analysis and transformation of C programs, in: [52], pp. 213–228.
[26] C. V. Hall, K. Hammond, W. Partain, S. L. Peyton Jones, P. Wadler, The Glasgow Haskell Compiler: A retrospective, in: J. Launchbury, P. M. Sansom (Eds.), Functional Programming, Workshops in Computing, Springer, 1992, pp. 62–71.
[27] C. Popeea, D. N. Xu, W.-N. Chin, A practical and precise inference and specializer for array bound checks elimination, in: R. Glück, O. de Moor (Eds.), PEPM, ACM, 2008, pp. 177–187.
[28] National Institute of Standards and Technology, Java SciMark benchmark for scientific computing, 2004. `http://math.nist.gov/scimark2/`.
[29] J. Dongarra, P. Luszczek, A. Petitet, The LINPACK benchmark: past, present and future, Concurrency and Computation: Practice and Experience 15 (2003) 803–820.
[30] D. Beyer, T. A. Henzinger, R. Jhala, R. Majumdar, The software model checker Blast, STTT 9 (2007) 505–525.
[31] K. Ku, T. E. Hart, M. Chechik, D. Lie, A buffer overflow benchmark for software model checkers, in: R. E. K. Stirewalt, A. Egyed, B. Fischer (Eds.), ASE, ACM, 2007, pp. 389–392.
[32] N. Suzuki, K. Ishihata, Implementation of an array bound checker, in: Proceedings of the ACM Symposium on Principles of Programming Languages, ACM, 1977, pp. 132–143.
[33] P. Cousot, R. Cousot, Static determination of dynamic properties of programs, in: Proceedings of the Second International Symposium on Programming, ACM, 1976, pp. 106–130.
[34] A. Miné, The octagon abstract domain, Higher-Order and Symbolic Computation 19 (2006) 31–100.
[35] A. Venet, G. P. Brat, Precise and efficient static array bound checking for large embedded C programs, in: W. Pugh, C. Chambers (Eds.), Proceedings of the ACM Conference on Programming Language Design and Implementation, ACM, 2004, pp. 231–242.
[36] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, D. R. Engler, EXE: automatically generating inputs of death, in: A. Juels, R. N. Wright, S. D. C. di Vimercati (Eds.), ACM Conference on Computer and Communications Security, ACM, 2006, pp. 322–335.
[37] Y. Xie, A. Aiken, Scalable error detection using boolean satisfiability, in: J. Palsberg, M. Abadi (Eds.), Proceedings of the ACM Symposium on Principles of Programming Languages, ACM, 2005,

pp. 351–363.
[38] P. Godefroid, Compositional dynamic test generation, in: M. Hofmann, M. Felleisen (Eds.), Proceedings of the ACM Symposium on Principles of Programming Languages, ACM, 2007, pp. 47–54.
[39] C. Csallner, Y. Smaragdakis, Check 'n' crash: combining static checking and testing, in: G.-C. Roman, W. G. Griswold, B. Nuseibeh (Eds.), ICSE, ACM, 2005, pp. 422–431.
[40] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata, Extended static checking for Java, in: Proceedings of the ACM Conference on Programming Language Design and Implementation, ACM, 2002, pp. 234–245.
[41] D. Hovemeyer, W. Pugh, Finding bugs is easy, SIGPLAN Notices 39 (2004) 92–106.
[42] C. Popeea, Disjunctive invariants for modular static analysis, Ph.D. thesis, Dept of Computer Science, National University of Singapore, 2008.
[43] F. Bourdoncle, Abstract debugging of higher-order imperative languages, in: Proceedings of the ACM Conference on Programming Language Design and Implementation, ACM, 1993, pp. 46–55.
[44] X. Rival, Abstract dependences for alarm diagnosis, in: K. Yi (Ed.), APLAS, volume 3780 of *Lecture Notes in Computer Science*, Springer, 2005, pp. 347–363.
[45] I. Dillig, T. Dillig, A. Aiken, Sound, complete and scalable path-sensitive analysis, in: [53], pp. 270–280.
[46] S. Gulwani, S. Srivastava, R. Venkatesan, Program analysis as constraint solving, in: [53], pp. 281–292.
[47] K. Marriott, H. Søndergaard, Bottom-up dataflow analysis of normal logic programs, J. Log. Program. 13 (1992) 181–204.
[48] M. Mernik, J. Heering, A. M. Sloane, When and how to develop domain-specific languages, ACM Comput. Surv. 37 (2005) 316–344.
[49] P. Klint, T. van der Storm, J. J. Vinju, Rascal: A domain specific language for source code analysis and manipulation, in: SCAM, IEEE Computer Society, 2009, pp. 168–177.
[50] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of the ACM Symposium on Principles of Programming Languages, ACM, 1977, pp. 238–252.
[51] M. Young, P. T. Devanbu (Eds.), Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006, ACM, 2006.
[52] R. N. Horspool (Ed.), Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings, volume 2304 of *Lecture Notes in Computer Science*, Springer, 2002.
[53] R. Gupta, S. P. Amarasinghe (Eds.), Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008, ACM, 2008.

Appendix A. Dynamic Semantics

In this section, we shall define a small-step dynamic semantics for the core imperative language from Figure 2. A machine configuration is represented as a tuple $\langle s, e\rangle$ such that s denotes the current stack and e denotes the current program code.

$$\begin{array}{ll} Stack: & s \in Stack = Var \rightarrow_{fin} Value \\ Values: & \delta \in Value = Int \uplus Boolean \uplus Void \end{array}$$

A reduction step is formalised as a small-step transition of the form $\langle s, e\rangle \hookrightarrow \langle s_1, e_1\rangle$. The rules are standard and presented in Figure A.8. As an example, a conditional expression is evaluated depending on the test value. If the value is a boolean constant, then either the rule [**D−IF−1**] (if the constant is *true*) or the rule [**D−IF−2**] apply. If the value is not of boolean type, then in principle the execution would be stuck with a type error. We rely on the fact that the source program is well-typed and such errors cannot

$$\frac{}{\langle s, v\rangle \hookrightarrow \langle s, s(v)\rangle}\ [\textbf{D–VAR}] \qquad \frac{\delta = \mathit{default}(t) \quad \mathit{fresh}\ x \quad \rho = [v \mapsto x]}{\langle s, t\ v; e\rangle \hookrightarrow \langle [x \mapsto \delta] + s, \texttt{ret}(x, \rho e)\rangle}\ [\textbf{D–BLK}] \qquad \frac{}{\langle s, l : \texttt{error}\rangle \hookrightarrow \langle s, \bot\rangle}\ [\textbf{D–ERROR}]$$

$$\frac{\begin{array}{c} mn \in \mathit{Primitives} \\ \langle s', \delta\rangle = \mathit{exec}\langle s, l : mn(v_1, ..v_n)\rangle \end{array}}{\langle s, l : mn(v_1, ..v_n)\rangle \hookrightarrow \langle s', \delta\rangle}\ [\textbf{D–PRIM}]$$

$$\frac{\begin{array}{c} t_0\ mn((\texttt{ref}\ t_i\ w_i)_{i=1}^{m-1}, (t_i\ w_i)_{i=m}^{n})\{e\} \\ s' = [w_i \mapsto s(v_i)]_{i=m}^{n} + s \end{array}}{\langle s, l : mn(v_1, ..v_n)\rangle \hookrightarrow \langle s', \texttt{ret}(\{w_i\}_{i=m}^{n}, [v_i/w_i]_{i=1}^{m-1} e)\rangle}\ [\textbf{D–CALL}]$$

$$\frac{s(v) = \mathit{true}}{\langle s, \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\rangle \hookrightarrow \langle s, e_1\rangle}\ [\textbf{D–IF–1}] \qquad \frac{s(v) = \mathit{false}}{\langle s, \texttt{if}\ v\texttt{then}\ e_1\texttt{else}\ e_2\rangle \hookrightarrow \langle s, e_2\rangle}\ [\textbf{D–IF–2}]$$

$$\frac{}{\langle s, v := \delta\rangle \hookrightarrow \langle s[v \mapsto \delta], ()\rangle}\ [\textbf{D–ASSIGN–1}] \qquad \frac{}{\langle s, \delta; e_2\rangle \hookrightarrow \langle s, e_2\rangle}\ [\textbf{D–SEQ–1}] \qquad \frac{}{\langle s, \texttt{ret}(v^*, \delta)\rangle \hookrightarrow \langle s - \{v^*\}, \delta\rangle}\ [\textbf{D–RET–1}]$$

$$\frac{\langle s, e\rangle \hookrightarrow \langle s', e'\rangle}{\langle s, v := e\rangle \hookrightarrow \langle s', v := e'\rangle}\ [\textbf{D–ASSIGN–2}] \qquad \frac{\langle s, e_1\rangle \hookrightarrow \langle s', e_1'\rangle}{\langle s, e_1; e_2\rangle \hookrightarrow \langle s', e_1'; e_2\rangle}\ [\textbf{D–SEQ–2}] \qquad \frac{\langle s, e\rangle \hookrightarrow \langle s', e'\rangle}{\langle s, \texttt{ret}(v^*, e)\rangle \hookrightarrow \langle s', \texttt{ret}(v^*, e')\rangle}\ [\textbf{D–RET–2}]$$

$$\frac{\langle s, e\rangle \hookrightarrow \langle s', \bot\rangle}{\langle s, v := e\rangle \hookrightarrow \langle s', \bot\rangle}\ [\textbf{D–ASSIGN–3}] \qquad \frac{\langle s, e_1\rangle \hookrightarrow \langle s', \bot\rangle}{\langle s, e_1; e_2\rangle \hookrightarrow \langle s', \bot\rangle}\ [\textbf{D–SEQ–3}] \qquad \frac{\langle s, e\rangle \hookrightarrow \langle s', \bot\rangle}{\langle s, \texttt{ret}(v^*, e)\rangle \hookrightarrow \langle s', \bot\rangle}\ [\textbf{D–RET–3}]$$

Figure A.8: Operational Semantics

occur. For any given complete execution, we expect one of three possible outcomes: $\langle s, e\rangle \hookrightarrow^* \langle s_1, \delta\rangle$ for success, $\langle s, e\rangle \hookrightarrow^* \langle s_1, \bot\rangle$ for failure, or $\langle s, e\rangle \not\hookrightarrow^*$ for non-termination.

The source language is extended with a new construct representing the intermediate result of a method call: the evaluation of the expression $\texttt{ret}(v^*, e)$ starts with evaluating the method's body e (rule [D−RET−2]) and, after its reduction to a value, the parameters passed by value v^* are removed from the current stack (rule [D−RET−1]). If the evaluation of the body reaches an error, then the rule [D−RET−3] will return the error back to the caller. The static semantics includes the following rule for the `ret` construct:

$$\frac{[\underline{\textbf{RET}}] \quad \vdash \{\Phi\}\, e\, \{\Phi_1\}}{\vdash \{\Phi\}\, \texttt{ret}(v^*, e)\, \{\exists (v, v')^*.\Phi_1\}}$$

Appendix A.1. Consistency between Concrete and Abstract Elements

While a concrete state is captured by the stack $s \in S$, an abstract state is represented by a relational constraint $\phi \in D$ between the program variables. When used with both unprimed and primed variables, $\phi \in \mathcal{D} \times \mathcal{D}$ denotes a transition between abstract states. We use $\mathcal{V}(\phi)$ to denote the set of unprimed variables from ϕ except *res*, while $\mathcal{V}'(\phi)$ is the set containing *res* and the primed variables from ϕ. The following definition introduces two operators to distinguish the initial and the final abstract states given a transition formula.

Definition 7 (Prestate and Poststate). *Given a transition $\phi \in \mathcal{D}\times\mathcal{D}$, its prestate $PreSt(\phi)$ captures the relation between unprimed variables of ϕ, while its poststate $PostSt(\phi)$ captures the relation between primed variables of ϕ :*

$$PreSt(\phi) = \exists X.\phi, \quad where\ X = \mathcal{V}'(\phi)$$
$$PostSt(\phi) = \rho(\exists X.\phi), \quad where\ X = \mathcal{V}(\phi)\ and\ \rho = [x' \mapsto x \mid x \in \mathcal{V}(\phi)]$$

To formalise the relation between the concrete and the abstract domain, we introduce an abstraction operator: $\alpha(s) = \bigwedge\{v = \delta \mid [v \mapsto \delta] \in s\}$. If the stack contains two or more variables with the same name, only the leftmost variable is considered. For example, $\alpha([x \mapsto 1, y \mapsto 1, x \mapsto 0]) = (x{=}1 \wedge y{=}1)$.

Two consistency relations between the concrete and abstract domains are defined to either agree in the current state, or in a pair formed from a pre-state and a post-state. These consistency relations rely on the logical implication operator:

$$\frac{\alpha(s) \to \phi}{s \models \phi} \qquad \frac{\alpha(s_1) \wedge \rho\alpha(s_2) \to \phi \qquad \rho = [x \mapsto x' \mid x \in \mathcal{V}(\alpha(s_2))]}{(s_1, s_2) \models \phi}$$

In general, $(s_1, s_2) \models \phi$ implies $s_1 \models PreSt(\phi) \wedge s_2 \models PostSt(\phi)$, but the implication does not hold in the other direction (e.g., for constraints relating both primed and unprimed variables).

Appendix B. Proof of the Main Theorem

In this section, we shall prove that the results obtained by the static semantics correctly reflect what happens during execution, as predicted by the dynamic semantics. The

dynamic semantics is formulated in small-step style and the proof proceeds by showing that some property is preserved by each step of (dynamic) evaluation:

$$\begin{array}{ccccccc} \langle s_1, e_1\rangle & \hookrightarrow & \langle s_2, e_2\rangle & \hookrightarrow & \ldots & \hookrightarrow & \langle s_n, e_n\rangle \\ \updownarrow & & \updownarrow & & & & \updownarrow \\ \vdash \{P_1\}\, e_1\, \{O_1, E_1\} & & \vdash \{P_2\}\, e_2\, \{O_2, E_2\} & & & & \vdash \{P_n\}\, e_n\, \{O_n, E_n\} \end{array}$$

Thus, we will proceed by induction on the length of the (dynamic semantics) reduction sequence:

- base case: prove the property for $\langle s_1, e_1\rangle$.
- induction step: assume the property holds for $\langle s_i, e_i\rangle$ and prove it for $\langle s_{i+1}, e_{i+1}\rangle$.

To prove the property for a configuration $\langle s_i, e_i\rangle$, the proof proceeds by induction on the height of the (dynamic semantics) reduction tree:

- base case: prove the property for those reduction rules without premises.
- induction step: assume the property holds for the premises and prove it for the concluding reduction rule.

The proof is completed when all the reduction rules are shown to preserve the required property. More details on the induction principle and proof examples for various program analyses can be found in the "Principles of Program Analysis" book [24, Sec 2.2, Sec 3.2, Sec 4.5.2, Sec 5.2, Appendix B].

We formalize the notion of a sound method summary meaning that the summary is an over-approximation of the outcomes collected from the method's body. A summary is checked to be sound using a static rule [**CHECK–METH**] for a method declaration.

$$\frac{\begin{array}{c}[\textbf{CHECK–METH}] \\ W{=}\{v_i\}_{i=1}^{n} \quad V{=}\{v'_i\}_{i=m}^{n} \quad R{=}\{res, v'_1, .., v'_n\} \\ \vdash \{nochange(W)\}\, e\, \{\{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\}\} \\ \exists V.\phi_1 \rightarrow O_{mn} \qquad \exists R.\phi_2 \rightarrow E_{mn}\end{array}}{\vdash t_0\ mn((\texttt{ref}\ t_i\ v_i)_{i=1}^{m-1}, (t_i\ v_i)_{i=m}^{n})\ \texttt{where}\ \{\texttt{OK} : O_{mn}, \texttt{ERR} : E_{mn}\}\{e\}}$$

Using this soundness notion for method summaries, we can split the main proof in two parts. For the first part, the proof is done assuming a program where all methods are given sound summaries. For the second part, we show that our fixed-point analysis always infers sound method summaries.

Appendix B.1. Soundness of Summary Outcomes

Proof of Theorem 1: The result stated by Theorem 1 can be shown using induction on the length of the reduction sequence. We consider an arbitrary reduction step $\langle s_i, e_i\rangle \hookrightarrow \langle s_{i+1}, e_{i+1}\rangle$ and inference judgements such that the prestate is consistent with dynamic state: $P_i = P_1 \circ \rho\alpha(s_i)$.

Success outcome is sound: $(s_1, [res{\mapsto}\delta]{+}s_n) \models O_1$

The base case proves the property for the last expression in a successful reduction sequence. When e_n is a constant expression δ, we can infer using the rule [**CONST**]: $\vdash \{P_n\}\, \delta\, \{\{\texttt{OK}\colon O_n,\ \texttt{ERR}\colon E_n\}\}$ such that $O_n = P_n \wedge (res{=}\delta) = (P_1 \circ \rho\alpha(s_n)) \wedge (res{=}\delta)$. As a consequence, the following consistency relation holds: $(s_1, [res{\mapsto}\delta]{+}s_n) \models O_n$.

The main part of the proof is based on a subject reduction lemma. This lemma proves the induction step corresponding to an arbitrary reduction step: $\langle s_i, e_i\rangle{\hookrightarrow}\langle s_{i+1}, e_{i+1}\rangle$. In particular, the lemma proves that the success outcome for e_{i+1} is stronger than the one inferred from e_i: $O_{i+1} \rightarrow O_i$. By repeated applications, we can conclude that the success outcome obtained from the inference of the original expression e_1 is sound: $(s_1, [res{\mapsto}\delta]{+}s_n) \models O_1$. □

Failure outcome is sound: $s_1{\models}E_1$

The base case proves the property for the last expression in a failed reduction sequence. When e_n is an `error` expression then from the rule [**ERROR**] we can deduce $\vdash \{\{\texttt{OK} : P_n, \texttt{ERR} : \mathit{false}\}\}\, l : \texttt{error}\, \{\{\texttt{OK} : O_n, \texttt{ERR} : E_n\}\}$ such that $E_n = P_n \wedge \{\texttt{OK} : \mathit{false}, \texttt{ERR} : \mathit{true}\} = P_n$. From the definition of P_n we can conclude that $s_1 \models E_n$.

The subject reduction lemma is used to prove the induction step. As a direct consequence of the lemma, the failure outcome for e_{i+1} is more precise than the one inferred from e_i: $E_{i+1} \rightarrow E_i$. From the base case and the induction step, we can conclude that the failure outcome is sound: $s_1 \models E_1$. □

The proof of Theorem 1 is based on induction over the length of reduction sequence and uses a subject reduction lemma as induction step. This lemma states the properties that are satisfied by an arbitrary reduction step.

Lemma 6 (Subject reduction). *Consider an execution started from the state s_1 and an arbitrary reduction step of this execution: $\langle s_1, e_1\rangle{\hookrightarrow}\ldots{\hookrightarrow}\langle s_i, e_i\rangle{\hookrightarrow}\langle s_{i+1}, e_{i+1}\rangle{\hookrightarrow}\ldots$. Further, consider a transition formula P_i consistent with the execution states: $(s_1, s_i) \models P_i$ and an inference $\vdash \{P_i\}\, e_i\, \{\{\texttt{OK} : O_i, \texttt{ERR} : E_i\}\}$.*

Then there exists P_{i+1} consistent with the execution states $(s_1, s_{i+1}) \models P_{i+1}$ and the results of the inference $\vdash \{P_{i+1}\}\, e_{i+1}\, \{\{\texttt{OK} : O_{i+1}, \texttt{ERR} : E_{i+1}\}\}$ satisfy the following relations:

- *$O_{i+1} \rightarrow O_i$. We also have $PreSt(O_{i+1}){\rightarrow}PreSt(O_i)$ and $PostSt(O_{i+1}){\rightarrow}PostSt(O_i)$.*
- *$E_{i+1} \rightarrow E_i$. We also have $E_i \equiv PreSt(E_i)$.*

Proof: We will prove that there is a relation between the inference result for e_i and the inference result for e_{i+1} by induction on the height of the (dynamic semantics) reduction tree. The induction hypothesis assumes that this relation holds for the reduction steps for the subexpressions of e_i. Various cases are denoted by the name of the evaluation rule that applies in the conclusion.

- Case [**D−VAR**]: With $s_{i+1} = s_i$, $s_i(v) = \delta$, we have the following reduction step:

$$\begin{array}{ccc} \langle s_i, v\rangle & \hookrightarrow & \langle s_{i+1}, s_i(v)\rangle \\ \updownarrow & & \updownarrow \\ \vdash \{P_i\}\, v\, \{P_i \wedge res = v\} & & \vdash \{P_{i+1}\}\, \delta\, \{P_{i+1} \wedge res = \delta\} \end{array}$$

  Let us choose $P_{i+1} = P_i$.

– We can prove P_{i+1} is consistent with the execution states (s_1, s_{i+1}) since P_i is consistent with (s_1, s_i) and $s_{i+1} = s_i$.
– $O_{i+1} \to O_i$ since $O_{i+1} = (P_{i+1} \wedge \mathit{res}{=}\delta)$ and $O_i = (P_i \wedge \mathit{res}{=}v)$.
– $E_{i+1} \to E_i$ since $E_{i+1} = E_i = \mathit{false}$.

- Case [**D−ASSIGN−1**]:

$$\begin{array}{ccc} \langle s_i, v := \delta\rangle & \hookrightarrow & \langle s_i[v \mapsto \delta], ()\rangle \\ \updownarrow & & \updownarrow \\ \vdash \{P_i\}\, v := \delta\, \{P_i \circ_{\{v\}} v'{=}\delta\} & & \vdash \{P_{i+1}\}\, ()\, \{P_{i+1}\} \end{array}$$

Let us choose $P_{i+1} = P_i \circ_{\{v\}} v'{=}\delta$.

– By definition of the consistency relation, $(s_1, s_i[v{\mapsto}\delta]) \models P_{i+1}$ reduces to $\alpha(s_1) \wedge \rho\alpha(s_i[v{\mapsto}\delta]) \to P_i \circ_{\{v\}} v'{=}\delta$. To prove this implication, we rely on the hypothesis that P_i is consistent: $\alpha(s_1) \wedge \rho\alpha(s_i) \to P_i$.
– $O_{i+1} \to O_i$ since $O_{i+1} = O_i$.
– $E_{i+1} \to E_i$ since $E_{i+1} = E_i = \mathit{false}$.

- Case [**D−ASSIGN−2**]:

$$\begin{array}{ccc} \langle s_i, v := e\rangle & \hookrightarrow & \langle s_{i+1}, v := e'\rangle \\ \updownarrow & & \updownarrow \\ \dfrac{\vdash \{P_i\}\, e\, \{\Phi\}}{\vdash \{P_i\}\, v := e\, \{\Phi \circ_{\{v\}} v'{=}\mathit{res}\}} & & \dfrac{\vdash \{P_i'\}\, e'\, \{\Phi'\}}{\vdash \{P_{i+1}\}\, v := e'\, \{\Phi' \circ_{\{v\}} v'{=}\mathit{res}\}} \end{array}$$

We use the induction hypotheses corresponding to the reduction step $\langle s_i, e\rangle{\hookrightarrow}\langle s_{i+1}, e'\rangle$. By these hypotheses, there exists P_i' that satisfies the consistency relation: $(s_1, s_{i+1}) \models P_i'$. Also, the results of the inference judgements $\vdash \{P_i\}\, e\, \{\Phi\}$ and $\vdash \{P_i'\}\, e'\, \{\Phi'\}$ satisfy the relation $\Phi' \to \Phi$.

Let us choose $P_{i+1}{=}P_i'$, where P_i' is the prestate constructed using the induction hypothesis.

– $(s_1, s_{i+1}) \models P_{i+1}$ from the induction hypothesis $(s_1, s_{i+1}) \models P_i'$
– From the induction hypothesis $\Phi' \to \Phi$, we can deduce that $(\Phi' \circ_{\{v\}} v'{=}\mathit{res}) \to (\Phi \circ_{\{v\}} v'{=}\mathit{res})$. This fact implies that $O_{i+1} \to O_i$.
– From the induction hypothesis $\Phi' \to \Phi$, we can deduce that $(\Phi' \circ_{\{v\}} v'{=}\mathit{res}) \to (\Phi \circ_{\{v\}} v'{=}\mathit{res})$. This fact implies that $E_{i+1} \to E_i$.

- Case [**D−SEQ−1**]: We have $s_{i+1} = s_i$:

$$\begin{array}{ccc} \langle s_i, \delta; e_2\rangle & \hookrightarrow & \langle s_{i+1}, e_2\rangle \\ \updownarrow & & \updownarrow \\ \dfrac{\vdash \{\exists \mathit{res}.(P_i \wedge \mathit{res}{=}\delta)\}\, e_2\, \{\Phi\}}{\vdash \{P_i\}\, \delta; e_2\, \{\Phi\}} & & \dfrac{}{\vdash \{P_{i+1}\}\, e_2\, \{\Phi'\}} \end{array}$$

Let us choose $P_{i+1} = P_i$.

- We can prove P_{i+1} is consistent with the execution states (s_1, s_{i+1}) since P_i is consistent with (s_1, s_i) and $s_{i+1} = s_i$.
- From the construction, P_i does not refer to the variable *res*. Consequently, $P_{i+1} = P_i = \exists res.(P_i \wedge res{=}\delta)$. Two inference judgements starting with equivalent prestates will have equivalent summary outcomes: $\Phi' = \Phi$. This implies that $O_{i+1} \rightarrow O_i$.
- $\Phi' = \Phi$ implies that $E_{i+1} \rightarrow E_i$.

- Case [**D–PRIM**]: Since the code for primitive methods is not available for analysis, we assume that the summaries of primitives methods are sound with respect to the operational semantics of the primitive's implementation.

$$\begin{array}{ccc} \langle s_i, mn(v_1,..,v_n)\rangle & \hookrightarrow & \langle s_{i+1}, \delta\rangle \\ \updownarrow & & \updownarrow \\ \vdash \{P_i\}\, mn(v_1,..,v_n)\, \{P_i \circ_{\mathcal{V}(\Phi_{mn})} \Phi_{mn}\} & & \vdash \{P_{i+1}\}\, \delta\, \{P_{i+1} \wedge res{=}\delta\} \end{array}$$

The soundness of the primitive summaries can be formalized with the following condition: $(s_i, [res{\mapsto}\delta]{+}s_{i+1}) \models \Phi_{mn} \wedge nochange(X)$, where $X = \mathcal{V}(s_i) - \mathcal{V}(\Phi_{mn})$.

Let us choose $P_{i+1} = P_i \circ_{\mathcal{V}(\Phi_{mn})} \Phi_{mn}$.

- From $(s_1, s_i) \models P_i$ and $(s_i, s_{i+1}) \models \Phi_{mn} \wedge nochange(X)$ we can prove that $(s_1, s_{i+1}) \models P_i \circ_{\mathcal{V}(\Phi_{mn})} \Phi_{mn}$.
- Using the chosen P_{i+1}, we can derive trivially $O_{i+1} \rightarrow O_i$.
- Using the chosen P_{i+1}, we can derive trivially $E_{i+1} \rightarrow E_i$.

- Case [**D–CALL**]: For this reduction step, we assume that each method is annotated with a sound summary. Using the soundness of the method summary, we can successfully apply the rule [**CHECK–METH**] to the method declaration *mn*:

$$t_0\ mn((\texttt{ref}\ t_i\ w_i)_{i=1}^{m-1}, (t_i\ w_i)_{i=m}^{n})\ \texttt{where}\ \Phi_{mn}\{e\}.$$

Consequently, the result of the judgement $\vdash \{nochange(W)\}\, e\, \{\{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\}\}$ is more precise than Φ_{mn} as follows: $\exists V.\phi_1 \rightarrow O_{mn}$ and $\exists R.\phi_2 \rightarrow E_{mn}$.

The reduction step corresponding to a method call follows:

$$\begin{array}{ccc} \langle s_i, mn(v_1,..,v_n)\rangle & \hookrightarrow & \langle [w_i{\mapsto}s_i(v_i)]_{i=m}^{n}{+}s_i, \texttt{ret}(\{w_i\}_{i=m}^{n}, e'\rangle \\ \updownarrow & & \updownarrow \\ & & \vdash \{P_{i+1}\}\, e'\, \{\Phi'\} \\ \hline \vdash \{P_i\}\, mn(v_1,..,v_n)\, \{P_i \circ_{\rho W} \rho\Phi_{mn}\} & & \hline \vdash \{P_{i+1}\}\, \texttt{ret}(\{w_i\}_{i=m}^{n}, e')\, \{\exists (w_i, w_i')^*.\Phi'\} \end{array}$$

where $e' = [v_i/w_i]_{i=1}^{m-1} e$, $W = \{v_i\}_{i=1}^{n}$ and $\rho = [v_i/w_i]_{i=1}^{n} + [v_i'/w_i']_{i=1}^{m-1}$.

Let us choose $P_{i+1} = P_i \wedge \bigwedge_{i=m}^{n} (w_i{=}v_i)$.

– From the induction hypothesis $(s_1, s_i) \models P_i$, we can prove that the following holds: $(s_1, [w_i \mapsto s_i(v_i)]_{i=m}^{n} + s_i) \models P_i \wedge \bigwedge_{i=m}^{n}(w_i = v_i)$.

– We can prove a pre-transition lemma that allows us to use a given judgement $\vdash \{\mathit{nochange}(W)\}\, e\, \{\Phi\}$. This in turn let us deduce a related judgement where a transition formula Φ_{pre} is used to translate both the prestate and the poststate: $\vdash \{\Phi_{pre} \circ_W \mathit{nochange}(W)\}\, e\, \{\Phi_{pre} \circ_W \Phi\}$. Note that the formula $(\Phi_{pre} \circ_W \mathit{nochange}(W))$ can be simplified to Φ_{pre}.

Using this lemma with $\Phi_{pre} = \rho^{-1} P_i$, the judgement $\vdash \{\mathit{nochange}(W)\}\, e\, \{\{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\}\}$ can be transformed to the following: $\vdash \{\rho^{-1} P_i\}\, e\, \{\rho^{-1} P_i \circ_W \{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\}\}$. After proper renamings, this last judgement is equivalent with $\vdash \{P_i\}\, \rho e\, \{P_i \circ_{\rho W} \rho\{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\}\}$, which we denote as (JUDG-1).

The judgement from the induction hypothesis $\vdash \{P_{i+1}\}\, e'\, \{\Phi'\}$ can be used together with (JUDG-1) to conclude that $\Phi' \equiv P_i \circ_{\rho W} \rho\{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\}$. Since we know that $\{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\} \rightarrow \Phi_{mn}$, we can finally conclude that $\Phi' \rightarrow P_i \circ_{\rho W} \rho\Phi_{mn}$. From this implication, we can directly derive that $O_{i+1} \rightarrow O_i$.

– From the above proof, we can also conclude that $E_{i+1} \rightarrow E_i$.

- Case [**D–BLK**]:

$$\begin{array}{ccc} \langle s_i,\ t\ v; e\rangle & \hookrightarrow & \langle [x \mapsto \delta] + s, \texttt{ret}(x, \rho e)\rangle \\ \updownarrow & & \updownarrow \\ \dfrac{\vdash \{P_i \wedge \mathit{default}(t, x')\}\, \rho e\, \{\Phi\}}{\vdash \{P_i\}\ \ t\ v; e\, \{\exists x'.\Phi\}} & & \dfrac{\vdash \{P_{i+1}\}\, \rho e\, \{\Phi'\}}{\vdash \{P_{i+1}\}\, \texttt{ret}(x, \rho e)\, \{\exists (x, x').\Phi'\}} \end{array}$$

Let us choose $P_{i+1} = P_i \wedge \mathit{default}(t, x')$.

– From the induction hypothesis $(s_1, s_i) \models P_i$ and $\delta = \mathit{default}(t)$, we can prove that $(s_1, [x \mapsto \delta] + s_1) \models (P_i \wedge \mathit{default}(t, x'))$.

– From the induction hypothesis $\Phi' \rightarrow \Phi$ and the fact that x does not appear in the formulae Φ and Φ', we can conclude that $\exists (x, x').\Phi' \rightarrow \exists x'.\Phi$. Consequently, we have $O_{i+1} \rightarrow O_i$.

– By a similar reasoning as above, we can conclude that $E_{i+1} \rightarrow E_i$.

- Case [**D–RET–1**]:

$$\begin{array}{ccc} \langle s_i, \texttt{ret}(v^*, \delta)\rangle & \hookrightarrow & \langle s_i - \{v^*\}, \delta\rangle \\ \updownarrow & & \updownarrow \\ \dfrac{\vdash \{P_i\}\, \delta\, \{P_i \wedge \mathit{res} = \delta\}}{\vdash \{P_i\}\, \texttt{ret}(v^*, \delta)\, \{\exists (v, v')^*.(P_i \wedge \mathit{res} = \delta)\}} & & \dfrac{}{\vdash \{P_{i+1}\}\, \delta\, \{P_{i+1} \wedge \mathit{res} = \delta\}} \end{array}$$

Let us choose $P_{i+1} = \exists (v, v')^*.P_i$.

- From the induction hypothesis $(s_1, s_i) \models P_i$, we can prove that $(s_1, s_i - \{v^*\}) \models \exists(v, v')^*.P_i$.
- We have $O_i = \exists(v, v')^*.(P_i \wedge res{=}\delta)$ and $O_{i+1} = (\exists(v, v')^*.P_i) \wedge res{=}\delta$, where v^* are either local variables or parameters passed by value. We can conclude that $O_{i+1} \rightarrow O_i$.
- We can conclude that $E_{i+1} \rightarrow E_i$ from $E_{i+1} = E_i = \mathit{false}$.

- Case [**D−ERROR**]: We have $s_{i+1} = s_i$:

$$\begin{array}{ccc} \langle s_i, l : \texttt{error} \rangle & \hookrightarrow & \langle s_i, \bot \rangle \\ \updownarrow & & \updownarrow \\ \vdash \{P_i\}\, l : \texttt{error}\, \{\{\texttt{OK} : \mathit{false}, \texttt{ERR} : \mathit{true}\}\} & & \vdash \{P_i\} \bot \{\{\texttt{OK} : \mathit{false}, \texttt{ERR} : \mathit{true}\}\} \end{array}$$

  Let us choose $P_{i+1} = P_i$.

  - $(s_1, s_{i+1}) \models P_{i+1}$ since $(s_1, s_i) \models P_i$.
  - We have $O_{i+1} = O_i = \mathit{false}$.
  - We have $E_{i+1} = E_i = \mathit{true}$.

- Case [**D−IF−1**]: We have $s_{i+1} = s_i$:

$$\begin{array}{ccc} \langle s_i, \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rangle & \hookrightarrow & \langle s_{i+1}, e_1 \rangle \\ \updownarrow & & \updownarrow \\ \dfrac{\vdash \{P_i \wedge v'{=}1\}\, e_1\, \{\Phi\} \quad \vdash \{\mathit{false}\}\, e_2\, \{\mathit{false}\}}{\vdash \{P_i\}\, \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\, \{\Phi \vee \mathit{false}\}} & & \vdash \{P_{i+1}\}\, e_1\, \{\Phi'\} \end{array}$$

  Let us choose $P_{i+1} = P_i$.

  - We can prove P_{i+1} is consistent with the execution states (s_1, s_{i+1}) since P_i is consistent with (s_1, s_i) and $s_{i+1} = s_i$.
  - The reduction step assumes that $s_i(v) = \mathit{true}$. Since P_i is consistent with the execution state s_i, we can conclude that the prestate $P_i \wedge v'{=}0$ simplifies to the *false* formula. Consequently, two inference judgements for e_2 with equivalent prestates P_i and P_{i+1} will have equivalent poststates: Φ and Φ'. This implies that $O_{i+1} \rightarrow O_i$.
  - $\Phi' = \Phi$ implies that $E_{i+1} \rightarrow E_i$

- Case [**D−IF−2**]: We have $s_{i+1} = s_i$:

$$\begin{array}{ccc} \langle s_i, \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rangle & \hookrightarrow & \langle s_{i+1}, e_2 \rangle \\ \updownarrow & & \updownarrow \\ \dfrac{\vdash \{\mathit{false}\}\, e_1\, \{\mathit{false}\} \quad \vdash \{P_i \wedge v'{=}0\}\, e_2\, \{\Phi\}}{\vdash \{P_i\}\, \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\, \{\Phi \vee \mathit{false}\}} & & \vdash \{P_{i+1}\}\, e_2\, \{\Phi'\} \end{array}$$

  Let us choose $P_{i+1} = P_i$.

– We can prove P_{i+1} is consistent with the execution states (s_1, s_{i+1}) since P_i is consistent with (s_1, s_i) and $s_{i+1} = s_i$.
– The reduction step assumes that $s_i(v) = \mathit{false}$. Since P_i is consistent with the execution state s_i, we can conclude that the prestate $P_i \wedge v'{=}1$ simplifies to the *false* formula. Consequently, two inference judgements for e_1 with equivalent prestates P_i and P_{i+1} will have equivalent poststates: Φ and Φ'. This implies that $O_{i+1} \rightarrow O_i$.
– $\Phi' = \Phi$ implies that $E_{i+1} \rightarrow E_i$

- Case [**D−SEQ−2**]: by induction hypothesis (similar to [**D−ASSIGN−2**]).
- Case [**D−RET−2**]: by induction hypothesis (similar to [**D−ASSIGN−2**]).

Appendix B.2. Soundness of the Fixed-Point Analysis

To complement the proof of Theorem 1, we will show that our fixed-point analysis always infers sound method summaries.

Theorem 7 (Soundness of the Fixed-Point Analysis). *Given a method declaration, we can show that the fixed-point inference applied to the constraint abstraction (obtained via the forward reasoning rules) would result in a sound summary* Φ_{mn}.

Proof: The proof is done using induction on the height of the call graph dominated by the method *mn*. The base case where *mn* does not call methods other than itself, can be proven as a special case of the induction step. For the induction step, the induction hypothesis assumes that the inference of methods called by *mn* has computed sound summaries. Using this hypothesis, we aim to prove that the fixed-point analysis computes a sound summary for the method *mn*.

The first step in the inference process is to derive constraint abstractions from a given method declaration using the forward reasoning rules. These rules are applied recursively on sub-expressions of the method body and, with one exception, derive constraints equivalent to the respective sub-expression. The exception is the rule [**CALL**], where, rather than equivalent, the constraint that is derived is an over-approximation of the method call since the callee has a sound summary (from the induction hypothesis). Consequently, we can show that the constraint abstractions $mnOK$ and $mnERR$ are consistent with the method declaration from which they are derived using the forward reasoning rules. Furthermore the constraint abstractions are monotonic functions defined on the abstract domain (e.g., disjunctive polyhedron domain) with values in the same domain. The domain contains elements that are formulae over a fixed set of variables $\{v_1, .., v_n, v'_1, .., v'_{m-1}\}$, where $v_1, .., v_{m-1}$ are parameters passed by reference and $v_m, .., v_n$ are parameters passed by value.

The fixed-point analysis computes iteratively a sequence starting with the least element of the domain (the formula *false*). Being applied to a monotonic function, this computation will result in an ascending sequence. To ensure convergence of this sequence, a widening operator is used. The result is then guaranteed to be an upper approximation of the least fixed point for the constraint abstractions $mnOK$ and $mnERR$.

Given that the results of the fixed-point analysis O_{mn} and E_{mn} are over-approximations of the least fixed points (lfp) of $mnOK$ and $mnERR$ abstractions, we can apply the

judgement for the method body e and prove that the implications required by the checking rule [**CHECK−METH**] hold as follows:

$$\frac{\begin{array}{c}\vdash \{nochange(W)\}\, e\, \{\{\texttt{OK} : \phi_1, \texttt{ERR} : \phi_2\}\} \\ \exists V.\phi_1 \rightarrow O_{mn} \qquad \exists R.\phi_2 \rightarrow E_{mn}\end{array}}{\vdash t_0\ mn((\texttt{ref}\ t_i\ v_i)_{i=1}^{m-1}, (t_i\ v_i)_{i=m}^{n})\ \texttt{where}\ \{\texttt{OK} : O_{mn}, \texttt{ERR} : E_{mn}\}\{e\}}$$

The formulae ϕ_1 and ϕ_2 correspond to the constraint abstraction functions $mnOK$ and $mnERR$ where the recursive calls are replaced by O_{mn} and E_{mn}. Since O_{mn} and E_{mn} are over approximations of the lfp, they are reductive points of these functions [24, Sec 4.2, Appendix A.4]. As a consequence, the results from the judgement are more precise formulae and the following implications hold: $\exists V.\phi_1 \rightarrow O_{mn}$ and $\exists R.\phi_2 \rightarrow E_{mn}$. Thus the premises of the checking rule are satisfied and Φ_{mn} is shown to be a sound summary for its corresponding method declaration. □

The current proof can immediately be extended to handle mutual recursive functions. In this case, fixed points are computed simultaneously for all the mutually recursive constraint abstraction functions.

We further argue that our fixed-point analysis always terminates. Forward analysis comprises of two main parts (i) to build two constraint abstractions per method, (ii) fixed point analysis for each recursive abstraction. The forward reasoning traverses each program via a well-founded recursion over the expression and is therefore guaranteed to terminate for programs of finite code size. The termination property of fixed point analysis is dependent on the abstraction domain and techniques used for approximation and widening. For linear arithmetic domain, we can use the result of [50] whereby hulling and widening are used to ensure that constraints encountered during conjunctive fixed-point have at most finite variations. This result extends also to k-bounded disjunctive formulae [4].