Verification Modulo Versions: Towards Usable Verification

Francesco Logozzo,
Shuvendu K. Lahiri, Manuel Fähndrich
Microsoft Research
{logozzo, shuvendu}@microsoft.com,
manuel@fahndrich.com

Sam Blackshear
University of Colorado Boulder
samuel.blackshear@colorado.edu

Abstract

We introduce Verification Modulo Versions (VMV), a new static analysis technique for reducing the number of alarms reported by static verifiers while providing sound semantic guarantees. First, VMV extracts semantic environment conditions from a base program `P`. Environmental conditions can either be sufficient conditions (implying the safety of `P`) or necessary conditions (implied by the safety of `P`). Then, VMV instruments a new version of the program, $\mathtt{P}'$, with the inferred conditions. We prove that we can use (i) sufficient conditions to identify abstract regressions of $\mathtt{P}'$ w.r.t. `P`; and (ii) necessary conditions to prove the relative correctness of $\mathtt{P}'$ w.r.t. `P`. We show that the extraction of environmental conditions can be performed at a hierarchy of abstraction levels (history, state, or call conditions) with each subsequent level requiring a less sophisticated matching of the syntactic changes between $\mathtt{P}'$ and `P`. Call conditions are particularly useful because they only require the syntactic matching of entry points and callee names across program versions. We have implemented VMV in a widely used static analysis and verification tool. We report our experience on two large code bases and demonstrate a substantial reduction in alarms while additionally providing relative correctness guarantees.

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

General Terms Experimentation, Languages, Reliability, Verification.

Keywords Abstract interpretation, Specification inference, Static analysis.

1. Introduction

Program verification is traditionally version *un*-aware: it focuses on the verification of a particular version of a program, independent of past versions of the same code base. Consider the typical verification scenario. The user has a program $\mathtt{P}'$ that she wants to verify. She runs the static verification tool, which produces a long list of alarms. Ideally, she will go over the list, addressing all the issues—fixing these alarms could involve repairing the program or providing additional annotations or contracts. In practice, such a process is too expensive to be realistic—easily requiring several weeks of work on medium-scale projects. Furthermore, she is more interested in fixing the *new* defects, *i.e.*, the ones introduced since the last release `P` of the same project. In fact her confidence in the "old" code is much higher (because of extensive tests, deployments, code reviews and reuse) than her confidence in the new code. Furthermore, even if a bug is found in `P`, it may be the case that she will not choose not to fix it to avoid compatibility issues [4].

Most industrial-strength static analysis tools provide a *syntactic* baselining feature to tackle this issue—*e.g.*, the Polyspace Verifier [31], Coverity SAVE [14], Grammatech CodeSonar [19], and CodeContracts static checker [17]. Syntax-based baselining can be roughly sketched as follows. The tool first creates a database with all unresolved warnings for a base program `P`. In the database, warnings are indexed by source position, type, message, and some other syntactic information. The resulting database is the *analysis baseline*. When the source program `P` is changed to create program $\mathtt{P}'$, each warning generated for $\mathtt{P}'$ is compared against the warnings in the analysis baseline. If the warning is matched, it is automatically hidden; if not, it is reported as a new warning.

By nature, syntactic matching is unreliable and unsound. It can both re-report old warnings and suppress warnings that are genuinely new. For example, let us consider a simple line-based suppression strategy that suppresses all warnings about an assertion failure at line i. Say a warning based on an assertion a_0 at line i was suppressed in the original program. Assume the user inserts a new and potentially failing assertion a_1 at line i in the new program, shifting a_0 back to line $i + 1$. Now, the analyzer will (wrongly) suppress the warning about the failure of a_1, and (wrongly) re-report the warning about the failure of a_0. This is exactly the opposite of the desired behavior! Though this example represents the worst case behavior of an overly simplistic syntactic baselining scheme, our point is that syntactic approaches to baselining will always fall short of correct behavior in some cases because the baseline lacks *semantic* information about why a warning should be suppressed. More importantly, it does not provide any guarantees in cases where it suppresses an "old" alarm or shows a "new" alarm.

Motivated by these deficiencies in verification tools—too many warnings for version-unaware verification *vs.* unreliable and unsound baseline—we seek to provide a new verification technique which delivers most of the benefits of a baseline while additionally providing semantic correctness guarantees.

Contributions. We introduce *Verification Modulo Versions (VMV)*, a technique for automatically inferring and maintaining *semantic* information across program versions. In VMV, the database corresponding to the analysis baseline is populated with *semantic environmental* conditions $\mathcal{E}$ extracted from the base program `P`. When

PLDI'14, June 9–11, 2014, Edinburgh, UK
Copyright © 2014 ACM [to be supplied]. . . $15.00

```
Sufficient():
y = g()
if *
  assert y > 0
```

(a) An example with no necessary condition other than `true` and a sufficient condition `y > 0`. The expression `*` denotes non-deterministic choice.

```
Sufficient'():
y = g()
  // assume y > 0
  // read from baseline
y = y - 2
if *
  assert y > 0
```

(b) An evolved version of the program from Fig. 1a. Though the assertion is unchanged syntactically, VMV($\mathcal{S}$) will report a regression because the baseline condition `y > 0` is no longer sufficient to ensure the safety of the assertion.

```
Sufficient''():
y = g()
  // assume y > 0
  // read from baseline
z = y - 2
if *
  assert z > -12
```

(c) A different evolved version of the program from Fig. 1a. Though the assertion has changed syntactically and semantically, VMV($\mathcal{S}$) proves it by using the sufficient condition `y > 0` from the baseline.

Figure 1: VMV($\mathcal{S}$): Verification Modulo Versions with sufficient conditions.

the static verifier analyzes $\mathtt{P}'$, it reads the semantic information $\mathcal{E}$ from the baseline, instruments $\mathtt{P}'$ with $\mathcal{E}$, and analyzes the instrumented program $\mathtt{P}'_{\mathcal{E}}$. Since $\mathtt{P}'_{\mathcal{E}}$ makes more assumptions than $\mathtt{P}'$, a monotonic analyzer will report fewer warnings for $\mathtt{P}'_{\mathcal{E}}$ than for $\mathtt{P}'$. In order to make VMV practical, we must address three main questions: (i) What semantic information $\mathcal{E}$ should be extracted from $\mathtt{P}$ and saved in the baseline? (ii) Where do we insert $\mathcal{E}$ in $\mathtt{P}'$, that is, how do we obtain $\mathtt{P}'_{\mathcal{E}}$? (iii) Which semantic guarantees do we have on $\mathtt{P}'_{\mathcal{E}}$? We answer these questions by making the following contributions:

- We formally characterize the optimal extracted environment information $\mathcal{E}$ as an abstraction of the trace semantics of $\mathtt{P}$. We consider abstractions such that the extracted information is (i) *sufficient* for the absence of errors, or (ii) *necessary* for the presence of good runs. We consider observing and injecting the extracted information at different levels of abstraction, namely: (i) the history-level as a sequence of states, (ii) the program point-level as sets of states reaching a program point, or (iii) the call-level, as sets of states after method calls.
- We show that if $\mathcal{E}$ is sufficient to remove the errors in $\mathtt{P}$ and $\mathtt{P}'_{\mathcal{E}}$ has an error, then $\mathtt{P}'$ introduced a *regression* w.r.t. $\mathtt{P}$—*i.e.*, under the same environmental assumptions ensuring $\mathtt{P}$ was correct, $\mathtt{P}'$ leads to an error. Otherwise stated, $\mathtt{P}'$ requires more assumptions from the external environment than $\mathtt{P}$ did.
- We show that if $\mathcal{E}$ is necessary for the presence of good runs in $\mathtt{P}$ and $\mathtt{P}'_{\mathcal{E}}$ has no errors, then $\mathtt{P}'$ is *correct relative* to $\mathtt{P}$—*i.e.*, under the same environmental assumptions holding in all *good* $\mathtt{P}$ runs, $\mathtt{P}'$ is correct.
- We show how the concepts above interplay with the abstractions necessary to make the approach practical: (i) the underlying static analyzer or the program verifier; (ii) the inference of the sufficient/necessary conditions; and (iii) the mapping of environment conditions to new versions.
- We have implemented VMV with necessary conditions and call matching in the `cccheck` tool and validate its effectiveness on a sizable suite of real-world C# projects. For the three largest projects in the open source `ActorFx` code base, VMV reduces the number of alarms to investigate by 72.6% while providing relative correctness guarantees. We only needed to add few contracts to bring the number of false alarms to 0. We applied the technique to a larger production-quality code base and we found similar results.

2. Overview of the Solution

The problem Suppose we have two versions of a program: the original program $\mathtt{P}$ and an updated version $\mathtt{P}'$. Assume also that we have some static analyzer or deductive verifier whose internals are opaque to us. The goal of traditional alarm masking (baselining) is to emit the set of alarms for $\mathtt{P}'$ not in $\mathtt{P}$ using some syntactic context matching to identify alarms both in $\mathtt{P}$ and $\mathtt{P}'$. The goal of VMV is to provide a robust verification technique which exploits semantic information present in a prior version $\mathtt{P}$ to produce a *semantically* sound set of alarms for $\mathtt{P}'$. By robust, we mean that the technique should require minimal syntactic matching among program versions. By semantically sound, we mean that VMV provides actual guarantees on its output.

```
Necessary():
y = g()
if *
  assert y > 0
else
  assert false
```

(a) An example with a necessary condition (`y > 0`), but no sufficient condition other than `false`.

```
Necessary'():
y = g();
 // assume y > 0
 // read from the baseline
if (y % 2 == 1)
  assert y > 0
else
  assert y > 1
```

(b) An evolved version of the program from Fig. 2a where the necessary condition (`y > 0`) from the previous program implies the safety of both assertions in the new program.

Figure 2: VMV($\mathcal{N}$): Verification Modulo Versions with necessary conditions.

VMV has three main phases: (i) *extraction* of semantic environmental conditions $\mathcal{E}$ from the base program $\mathtt{P}$; (ii) *installation* of $\mathcal{E}$ in $\mathtt{P}'$ to obtain an instrumented program $\mathtt{P}'_{\mathcal{E}}$; and (iii) verification of $\mathtt{P}'_{\mathcal{E}}$. VMV is only useful if the set of alarms $\mathtt{P}'_{\mathcal{E}}$ is significantly smaller than that of $\mathtt{P}'$. In this paper we systematically study the (orthogonal) choices for extraction and installation of correctness conditions, the semantic guarantees provided in the concrete and in the abstract, and justifying a particular point of the design space in VMV that we have adopted in the tool `cccheck`.

Extraction VMV extracts *semantic conditions* from $\mathtt{P}$, *i.e.*, constraints on the environment to ensure good executions. We must be careful to differentiate two kinds of assumptions, sufficient *vs.* necessary. A sufficient condition guarantees that the program *always* reaches a good state. A necessary condition holds *whenever* the program reaches a good state. Otherwise stated, if a sufficient condition does not hold, the program may or may not reach a bad state. If a necessary condition does not hold, then the program will definitely reach a bad state. We will see that installing sufficient or necessary conditions will provide different semantic guarantees on the instrumented program $\mathtt{P}'_{\mathcal{E}}$.

The difference between sufficient *vs.* necessary conditions is exemplified in Fig. 1a and Fig. 2a. In the first case, the condition `y > 0` (at program point `0`) is sufficient for the correctness of the program: When `y` is positive, the program will always reach a good state no matter which non-deterministic choice is made. However, the condition is not necessary because the program may still reach a good state even if `y` is not positive. In the second case (Fig. 2a), the condition `y > 0` is necessary for the correctness of the program, but not sufficient. If `y` is not positive, then the assertion at line `2` will always fail. Since there is no condition that we can impose on `y` to ensure that the second assertion will not fail, the program has no sufficient correctness condition other than `false`. In general, finding a sufficient condition requires an under-approximation while finding a necessary one requires an over-approximation.

The diagram in Fig. 3 summarizes our framework for extracting correctness conditions from a program `P`. At the lowest level is the trace semantics $\tau_{\mathtt{P}}^{+}$, which capture all finite executions of `P` (Sec. 3). By abstracting away all traces in $\tau_{\mathtt{P}}^{+}$ that lead to an error state via the abstraction function α_G, we obtain the *success semantics* $G[\![\mathtt{P}]\!]$ (Sec. 3). Necessary and sufficient conditions for a program are suitable abstractions of the success semantics. As our diagram illustrates, we can extract correctness conditions at increasing levels of abstraction. One option (Sec. 4) is to extract correctness conditions for sequences of function calls (history conditions $\vec{\mathcal{S}}, \vec{\mathcal{N}}$). More abstract options (Sec. 5) are: (i) to extract correctness conditions for each program point where a function call appears, *i.e.*, different occurrences of the same function invocation are kept separate (state conditions $\mathcal{S}_\Sigma$, $\mathcal{N}_\Sigma$); or, (ii) to merge together calls to the same function (call conditions, $\mathcal{S}, \mathcal{N}$). The intuition behind this hierarchy of conditions is that the more abstract the condition, the easier it is to inject into $\mathtt{P}'$. In the following we will let VMV($\mathcal{S}$) (resp. VMV($\mathcal{N}$)) denote VMV instantiated with sufficient (resp. necessary) conditions.

Insertion In order to build the instrumented program $\mathtt{P}'_{\mathcal{E}}$, we need to inject the conditions $\mathcal{E}$ into $\mathtt{P}'$ (Sec. 6). The insertion step depends on the abstraction level of the extracted conditions. For instance, history conditions require the exact matching of program points, whereas call conditions require only the matching of entry points and callees among program versions. In our theoretical treatment we assume that we are given a function δ mapping each correctness condition from its program point in `P` to the corresponding program point(s) in $\mathtt{P}'$. Our theoretical results are *parametric* w.r.t. δ. In practice, we cannot ask the user to provide such a map, therefore we pick a function δ requiring a minimal syntactic matching that is easy to compute. Our pragmatic choice is to use call conditions ($\mathcal{S}, \mathcal{N}$) where δ only needs to match entry points and callee functions among program versions. Our hypothesis is that such a matching is much more reliable than mapping arbitrary program points.

Concrete semantic guarantees The next question is: what semantic guarantees do we have on $\mathtt{P}'_{\mathcal{E}}$? We will see that if the extracted correctness conditions $\mathcal{E}$ are sufficient and $\mathtt{P}'_{\mathcal{E}}$ has a bad run, then $\mathtt{P}'$ introduced a δ-regression, *i.e.*, $\mathtt{P}'$ is *not* correct when the same hypotheses under which `P` was correct are mapped to $\mathtt{P}'$ via δ (Th. 1). Intuitively, this means that the correctness of $\mathtt{P}'$ relies on stronger assumptions than the correctness of `P`. Conversely, if the extracted correctness conditions are necessary and there are no errors in $\mathtt{P}'_{\mathcal{E}}$, then $\mathtt{P}'$ is δ-correct with respect to the same environment assumptions in `P` (Th. 2). Otherwise stated, sufficient conditions are useful to find new bugs, while necessary conditions are useful to prove relative correctness (the absence of new bugs).

$$\begin{tikzcd}
\delta & \text{Sufficient} & & \text{Necessary} \\
\text{Call conditions} & \mathcal{S} & & \mathcal{N} \\
\text{State conditions} & \mathcal{S}_\Sigma \arrow[u, "\alpha_{\mathcal{S}}"] & & \mathcal{N}_\Sigma \arrow[u, "\alpha_{\mathcal{N}}"'] \\
\text{History conditions} & \vec{\mathcal{S}} \arrow[u, "\alpha_\Sigma"] & & \vec{\mathcal{N}} \arrow[u, "\alpha_\Sigma"'] \\
\text{Success Semantics} & & G[\![\mathtt{P}]\!] \arrow[ul, "\alpha_s"] \arrow[ur, "\alpha_n"'] & \\
\text{Concrete Semantics} & & \tau_{\mathtt{P}}^{+} \arrow[u, "\alpha_G"] & \\
& & {} \arrow[l, "\text{Underapproximate}"'] \arrow[r, "\text{Overapproximate}"] &
\end{tikzcd}$$

Figure 3: The hierarchy of environmental conditions extraction. The success semantics abstracts the concrete semantics. The correctness conditions (sufficient—ensuring the program is always correct; or necessary—required for all the correct executions) abstract the success semantics. Correctness conditions can be observed at increasing levels of abstraction (history, program point, and calls).

As an example of *bug finding*, let us consider the code in Fig. 1b. VMV($\mathcal{S}$) reads the (sufficient) condition `y > 0` from the analysis baseline and installs it as an assumption about the return value of `g()`. It emits a new alarm because the sufficient condition `y > 0` from `Sufficient` is no longer strong enough to prove the correctness of the assertion in the new version of the program: the correctness of `Sufficient'` requires more environmental assumptions than the correctness of `Sufficient`. Note that a syntactic baselining technique matching asserted expressions might wrongly suppress the alarm, since the expression is `y > 0` in both versions of the program. In Fig. 1c, a syntactic baselining technique also would likely fail to match the alarm from `Sufficient` with the alarm in `Sufficient''` because the line numbers have shifted and the expression being asserted has changed. Under the carried-over assumption `y > 0`, the assertion at line `(3)` holds. Note that any *under*-approximation of `y > 0` (*e.g.*, `false`) would allow us to make the same guarantee, whereas an over-approximation would be incorrect. For instance, using `true` would cause VMV($\mathcal{S}$) to (wrongly) report a regression for `Sufficient''`. Finally, note that the condition read from the baseline is stronger than (*i.e.*, is an under-approximation of) the condition we can extract from `Sufficient''`: `ret > -10`.

As an example of *relative verification* consider the evolved program `Necessary'` in Fig. 2b. VMV($\mathcal{N}$) proves that under the necessary correctness assumption `y > 0` inherited from `Necessary`, `Necessary'` is correct. Furthermore, we deduce that `Necessary'` fixes `Necessary` since the condition that was only necessary but not sufficient for the correctness of `Necessary` is now sufficient for the correctness of `Necessary'`.

Abstraction In practice, we need to instantiate our VMV framework with a real static analysis/verification tool and an effective method for semantic condition inference. We require approximation to make program analysis and condition inference tractable (Sec. 7). The *analysis* tool should over-approximate the concrete trace semantics (*i.e.*, $\tau_{\mathtt{P}}^{+}$ in Fig. 3), whilst the *inference* tool should under-approximate the sufficient conditions ($\underline{\mathcal{S}} \leq \mathcal{S}$) and over-approximate the necessary conditions ($\mathcal{N} \leq \overline{\mathcal{N}}$). It is well-known that the abstraction may lose some of the concrete guarantees, so it is natural to ask what happens to VMV in the abstract.

For sufficient conditions (*i.e.*, VMV($\mathcal{S}$)), if the analysis is complete [18], then the alarms for $\mathtt{P}'_{\underline{\mathcal{S}}}$ are still concrete regressions (*e.g.* as in Fig. 1b). However, if the tool is incomplete, *i.e.*, it fails to prove every fact that is true, then the alarms are *abstract* regressions. An abstract regression denotes an environment $s \in \mathcal{S}$ for which the tool can prove P correct, but cannot prove the correctness of P′. Orthogonally if $\underline{\mathcal{S}} < \mathcal{S}$ then $\mathtt{P}'_{\underline{\mathcal{S}}}$ may suppress real alarms, *i.e.*, some regression may be missed—think for instance of a naive extraction algorithm always suggesting `false`. In general, no guarantee can be given on the assertions not shown in the list of raised alarms.

For necessary conditions (*i.e.*, VMV($\mathcal{N}$)), if the analysis is incomplete or $\mathcal{N} < \overline{\mathcal{N}}$, then some of the reported alarms may be false alarms—the analysis failed to prove the relative correctness either because the inferred condition is too weak (at worst `true`), or the tool is too imprecise. Nevertheless, all the assertions not shown as alarms are correct or relatively correct.

Motivated by the observations above, and the fact that all practical verification and analysis tools are incomplete, and that computing "good" sufficient conditions is harder than computing "good" necessary conditions, and that (relative) verification is more interesting than bug finding, we instantiate VMV with state-based necessary precondition inference (Sec. 8).

3. Concrete Semantics and Background

We formalize VMV using Abstract Interpretation [8]. Abstract Interpretation is a theory of semantic approximations which is very well suited to describing (under/over) approximations. In formalizing VMV, we will need to consider four orthogonal axes of approximation: (i) the extracted semantic conditions (history, state, or call); (ii) whether they are sufficient or necessary; (iii) the approximation induced by the inference tool (under-approximation of sufficient, over-approximation of necessary); and (iv) the over-approximation induced by the static analysis/verification tool.

Syntax We consider a simple imperative block language with "holes." Blocks are sequences of basic statements. Basic statements are assignments, assertions, assumptions, and function calls. For our theoretical treatment: (i) holes are input values and calls to *unknown* external functions; (ii) non-deterministic choices (`*`) can only appear in assignments (*i.e.*, `assert *` and `assume *` are disallowed); (iii) all contracts (if any) for function calls are explicitly expanded. A program is a directed graph whose nodes are blocks. We assume that a program has a single exit block. The special program points `entry` and `exit` denote function entry and exit points. We use the `BeforeCall` (resp. `AfterCall`) set to denote the set of program points before (resp. after) some method call.

Concrete states A program state maps variables $\mathtt{x}, \mathtt{y} \cdots \in \mathtt{V}$ to values in $\mathcal{V}$. We assume some reserved variable names: $\mathtt{arg}_0, \mathtt{arg}_1, \ldots$ are the actual arguments for function calls and `ret` is the value returned by a function call. Program states $s, s_0 \ldots$ belong to a given set of states Σ. The function $\pi \in \Sigma \to \mathtt{PC}$ maps a state to the corresponding program point, and the function $\pi_s \in \Sigma \to \mathtt{Stm}$ maps states to the corresponding statements. The function $\models \in (\Sigma \times \mathtt{Exp}) \to \{\mathtt{true,\ false}\}$ returns the Boolean value of an expression in a given state.

Small-step operational semantics The *non-deterministic* transition relation $\tau_{\mathtt{P}} \in \wp(\Sigma \times \Sigma)$ associates a state with its possible successors in the program P. When the program P is clear from the context, we write τ instead of $\tau_{\mathtt{P}}$. We often write $\tau(s, s')$ instead of $\langle s, s' \rangle \in \tau$. The definition of τ for our block language is mostly straightforward, hence we omit it here. Function calls are the only interesting case for our exposition. We use function calls to model unknowns from the environment and imprecision in the static analysis. To simplify the theoretical development, we assume that function calls are (i) black boxes – we cannot inspect their bodies, so they can return any value, and (ii) side effect-free, otherwise do as in [13]. If $s \in \Sigma$ is the state corresponding to a function call (that is, $\pi_s(s) = \mathtt{ret} = \mathtt{f}(\mathtt{arg}_0 \ldots \mathtt{arg}_{n-1})$), then the transition relation is such that $\forall s' \in \Sigma.\ \tau(s, s') \Leftrightarrow \exists v \in \mathcal{V}.\ s' = s[\mathtt{ret} \mapsto v]$. In general, side effect-free functions need not be deterministic. Our model of functions accounts for the possibility that a function can be invoked with the same parameters and return a different value. The set of *initial* states is $\mathcal{I} \subseteq \Sigma$. The *final* (or blocking) states have no successors: $\mathcal{F} = \{s \in \Sigma \mid \forall s'.\ \neg\tau(s, s')\}$. The *bad* states $\mathcal{B} \subseteq \mathcal{F}$ are final states corresponding to assertion failures, *i.e.*, $\mathcal{B} = \{s \in \Sigma \mid \pi_s(s) = \mathtt{assert\ e} \wedge s \models \neg\mathtt{e}\}$. The *good* states $\mathcal{G} \subseteq \mathcal{F}$ are non-failing final states: $\mathcal{G} = \mathcal{F} \setminus \mathcal{B}$.

Maximal trace semantics The concrete semantics of a program P is a set of execution traces. Various choices are possible for the trace semantics. Here, for simplicity, we will simply consider finite maximal finite traces, *e.g.*, we ignore non-termination. Traces are sequences of states in Σ. The empty trace ϵ has length 0. A trace $\vec{s} = s_0 s_1 \ldots s_{n-1}$ has length n. We will often write s_i to refer to the i-th state of the trace $\vec{s}$. The set of traces of length $n \geq 0$ is Σ^n. The set of *finite* traces is $\Sigma^* = \{\epsilon\} \cup \bigcup_{n>0} \Sigma^n$. The good traces are $\mathcal{G}^* = \{\vec{s} \in \Sigma^* \mid s_{n-1} \in \mathcal{G}\}$. Similarly, the bad traces are $\mathcal{B}^* = \{\vec{s} \in \Sigma^* \mid s_{n-1} \in \mathcal{B}\}$. We define trace *concatenation* as sequence juxtaposition. We can easily extend trace concatenation and composition to sets of traces. The execution prefixes define the *partial execution* trace semantics. The partial runs of P of length n are those partial executions such that $\vec{\tau}_{\mathtt{P}}^{\,n} = \{\vec{s} \in \Sigma^n \mid \forall i \in [0, n-1).\ \tau_{\mathtt{P}}(s_i, s_{i+1})\}$. The complete or *maximal* runs of P are those partial runs ending with a blocking state:

$$\vec{\tau}_{\mathtt{P}}^{\,+} = \bigcup_{n>0} \{\vec{s} \in \vec{\tau}_{\mathtt{P}}^{\,n} \mid s_0 \in \mathcal{I} \wedge s_{n-1} \in \mathcal{F}\}.$$

Please note that while each trace in $\vec{\tau}_{\mathtt{P}}^{\,+}$ is finite, the set $\vec{\tau}_{\mathtt{P}}^{\,+}$ is in general infinite. This means that while we do not capture infinite executions, we do capture unbounded non-determinism from data sources like input variables and values returned from a function. Finally, we let $\vec{\tau}_{\mathtt{P}}^{\,+}(\mathcal{I})$ denote the set maximal partial traces starting with a state in $\mathcal{I}$.

Recall: Abstract Interpretation In this paper, we use Abstract Interpretation to approximate a concrete semantics consisting of the maximal trace semantics defined above over a concrete domain defined by the complete Boolean lattice $\langle \wp(\Sigma^*), \subseteq, \emptyset, \Sigma^*, \cup, \cap, \neg \rangle$. The concrete semantics can be either over-approximated or under-approximated.

Let $\langle C, \leq \rangle$ and $\langle A, \sqsubseteq \rangle$ be two partial orders. If there exists an abstraction function $\alpha \in C \to A$ and a concretization function $\gamma \in A \to C$ such that $\forall c \in C, a \in A.\ \alpha(c) \sqsubseteq a \Leftrightarrow c \leq \gamma(a)$ we say that $\langle \alpha, \gamma \rangle$ is a Galois connection, and we note that by $\langle C, \leq \rangle \underset{\alpha}{\overset{\gamma}{\longleftrightarrow}} \langle A, \sqsubseteq \rangle$. The composition of Galois connections is a Galois connection. The composition enables the step-wise construction of abstract semantics, and it is the theoretical justification to our construction. Sometimes, we may need a relaxed form of the Abstract Interpretation framework in which we only require the abstraction function α function to be monotonic [9].

Success Semantics The *good* execution (or success) semantics of P considers only the traces in the maximal execution traces of P that terminate in a non-error state. To formalize this, we first define a Galois connection $\langle \wp(\Sigma^*), \subseteq \rangle \underset{\alpha_{\mathcal{G}}}{\overset{\gamma_{\mathcal{G}}}{\longleftrightarrow}} \langle \wp(\Sigma^*), \subseteq \rangle$ with the abstraction $\alpha_{\mathcal{G}} = \lambda T.\ T \cap \mathcal{G}^*$ and the concretization

$\gamma_{\mathcal{G}} = \lambda T.\ T \cup \mathcal{B}^*$. Then we define the good execution trace semantics as $G[\![\mathtt{P}]\!] \triangleq \alpha_{\mathcal{G}}(\vec{\tau}_{\mathtt{P}}^{+})$.

4. History Environment Conditions

We want to extract from $G[\![\mathtt{P}]\!]$ the sufficient and necessary conditions for correct executions. Roughly, a sufficient condition ensures that all the executions exhibiting the condition are good executions. A necessary condition holds for *all* good executions. In this section we unify and generalize concepts from [7, 12] to define the *optimal* sufficient and necessary history conditions as suitable abstractions of the success semantics. First, let us define the Galois connection $\langle \wp(\Sigma^*), \subseteq\rangle \xleftrightarrow[\alpha_n]{\gamma_n} \langle \wp(\Sigma^*), \subseteq\rangle$ where the abstraction

$$\alpha_n = \lambda T.\ \{s_0\vec{e} \mid \vec{s} \in T \wedge \vec{e} = \alpha'(\vec{s})\}$$

uses the commodity function α' defined as

$$\alpha'(\vec{s}) = \begin{cases} \epsilon & \text{if } \vec{s} = \epsilon \\ s_1\alpha'(s_2 \dots s_{n-1}) & \text{if } \pi_s(s_0) = \mathtt{ret\ =\ f(...)} \\ \alpha'(s_1 \dots s_{n-1}) & \text{otherwise.} \end{cases}$$

Intuitively, α_n captures sequences of environment choices from a set of traces T. The abstraction function records the entry state for the function and the state after each method call.

Weakest sufficient environment conditions The weakest sufficient environmental conditions capture the largest set of *sequences* of environment choices (initial states and return states) that guarantee that the program execution *always* reaches a good state. We define them via a parameterized Galois connection [7] $\langle \wp(\Sigma^*), \subseteq\rangle \xleftrightarrow[\alpha_s[S]]{\gamma_s[S]} \langle \wp(\Sigma^*), \subseteq\rangle$. Informally, given a base set of traces S, the abstraction $\alpha_s[S]$ first selects the traces in T that terminate in a good state ($\vec{s} \in \alpha_n(\mathcal{G}(T))$) such that their environmental choices ($\alpha_n(\{\vec{s}\})$) are different from the choices of all the traces in S that may lead to a bad state ($\vec{s_{\mathcal{B}}} \in S \cap \mathcal{B}^*$). Then, out of the selected traces, the abstraction retains only the environmental choices. Formally, the parameterized abstraction is

$$\alpha_s[S] = \lambda T.\{\alpha_n(\{\vec{s}\}) \mid \vec{s} \in \alpha_{\mathcal{G}}(T) \wedge \forall \vec{s_{\mathcal{B}}} \in S \cap \mathcal{B}^* \Rightarrow \alpha_n(\{\vec{s}\}) \neq \alpha_n(\{\vec{s_{\mathcal{B}}}\})\}$$

and the (parameterized) concretization is

$$\gamma_s[S] = \lambda T.\{\vec{s} \mid \vec{s} \in \gamma_{\mathcal{G}}(T) \vee \exists \vec{s_{\mathcal{B}}} \in S.\ \alpha_n(\{\vec{s}\}) = \alpha_n(\{\vec{s_{\mathcal{B}}}\})\}.$$

Note that our definition of $\alpha_s[S]$ generalizes the trace-based wlp of [12]. The largest set of environment choices guaranteeing that the program P is always correct is $\vec{\mathcal{S}} \triangleq \alpha_s[\tau_{\mathtt{P}}^{+}](G[\![\mathtt{P}]\!])$.

As an example, let us consider Fig. 1a. The weakest sufficient condition is $\vec{\mathcal{S}} = \{\langle \mathtt{y} \mapsto y_0\rangle\langle \mathtt{y} \mapsto y_1\rangle \mid y_0 \in \mathcal{V}, 0 < y_1\}$. *i.e.*, when g returns a positive number, the program is correct; that is, it will never reach any bad state. The program is correct even if g ensures a stronger property, *e.g.*, the return value is 42 or a positive even number.

Strongest necessary environment conditions We want the set $\vec{\mathcal{N}}$ of sequences of environment choices characterizing good executions. The intuition is that if we have a sequence of environment choices not in $\vec{\mathcal{N}}$, then an execution compatible with those choices will *definitely* lead to a bad state. In particular, we want the *smallest* such set $\vec{\mathcal{N}}$, which we can get by abstracting $G[\![\mathtt{P}]\!]$. Therefore $\vec{\mathcal{N}} \triangleq \alpha_n(G[\![\mathtt{P}]\!])$ is the strongest environment property satisfied by good executions of P. If an environment *does not* make one of the choices admitted by $\vec{\mathcal{N}}$, then the program P will fail for sure. If an environment *does* make a choice allowed by $\vec{\mathcal{N}}$, then we know (by construction) that there is at least one concrete execution of P that terminates in a good state. However, there may also be concrete executions that terminate in a bad state. In the example of Fig. 2a: $\vec{\mathcal{N}} = \{\langle \mathtt{y} \mapsto y_0\rangle\langle \mathtt{y} \mapsto y_1\rangle \mid y_0 \in \mathcal{V}, y_1 > 0\}$. That is, in all the good executions, the value returned by `f` is positive. If `f` is negative then the program will fail for sure. Otherwise, the program may or may not fail, depending on the non-deterministic choice.

```
TwoCallSites(y, z):
0: if *
1:   ret = f(y)
2:   x = ret + 1
3: else
4:   ret = f(z)
5:   x = ret - 1
6: assert x > 100
```

(a) Two distinct invocations of the same function. The sufficient and necessary state conditions coincide, but the call conditions are different.

```
TwoCallSites'(w):
0: ret = f(w)
1: x = ret
2: assert x > 100
```

(b) Evolved version where exact syntactic matching of program points is ambiguous, but function calls matching is well-defined.

Figure 4: An example where the sufficient and necessary call conditions are different (resp. `ret>101` and `ret>99`).

Relation between the two concepts It follows from the definitions above that $\vec{\mathcal{S}} \subseteq \vec{\mathcal{N}}$. In general, it is sound to *under*-approximate $\vec{\mathcal{S}}$, but not to over-approximate it: $\vec{\mathcal{S}}$ characterizes the largest set of concrete executions that always terminate in a good state and any under-approximation of $\vec{\mathcal{S}}$ yields a smaller set of concrete executions that always terminate in a good state. Dually, it is sound to *over*-approximate $\vec{\mathcal{N}}$, but not to under-approximate it. Though sufficient and necessary conditions are not new concepts, we take the novel step of using *both* kinds of conditions to get different guarantees about the warnings reported by our VMV technique—as we will see in Sec. 6.

5. State-Based Environment Conditions

The trace-based environment conditions capture the *sequence* of environment choices that are necessary or sufficient for program correctness. However, in some cases we may be interested in the set of environments that are possible at a given program point regardless of the sequence of environment choices made previously. In such cases, we can abstract the trace-properties of the section above to state properties by collecting all the states observed at each program point and discarding the environment choices made along the way.

State conditions Given a set of traces T, the reachable states abstraction α_Σ collects the set of states that reach each program point. Formally, we have the Galois connection $\langle \wp(\Sigma^*), \subseteq\rangle \xleftrightarrow[\alpha_\Sigma]{\gamma_\Sigma} \langle \mathtt{PC} \to \wp(\Sigma), \dot\subseteq\rangle$, where the abstraction function α_Σ is $\alpha_\Sigma = \lambda T.\ \lambda\mathtt{pc}.\ \{s_i \mid \exists \vec{s} \in T. \vec{s} = s_0 \dots s_i \dots s_{n-1} \wedge \pi(s_i) = \mathtt{pc}\}$ and the concretization γ_Σ is the expected one.

The weakest *state-based* sufficient conditions on the environment are the set of states $\mathcal{S}_\Sigma \triangleq \alpha_\Sigma(\vec{\mathcal{S}})$. The strongest *state-based* necessary conditions on the environment are the set of states $\mathcal{N}_\Sigma \triangleq \alpha_\Sigma(\vec{\mathcal{N}})$. If `entry` denotes the entry point of P, then (i) $\mathcal{S}_\Sigma(\mathtt{entry})$ is the weakest sufficient precondition of [15] and (ii) $\mathcal{N}_\Sigma(\mathtt{entry})$ is the strongest necessary precondition of [10]. It follows from Sec. 4 that $\mathcal{S}_\Sigma \dot\subseteq \mathcal{N}_\Sigma$, where $\dot\subseteq$ is the pointwise functional extension of $\subseteq$. In the example of Fig. 4a, the sufficient and necessary *state-based* conditions coincide: $\mathcal{S}_\Sigma(\mathtt{1}) = \mathcal{N}_\Sigma(\mathtt{1}) = \{s \mid s(\mathtt{ret}) > 99\}$ and $\mathcal{S}_\Sigma(\mathtt{4}) = \mathcal{N}_\Sigma(\mathtt{4}) = \{s \mid s(\mathtt{ret}) > 101\}$.

Method call conditions The state-based environment conditions collect the possible environments at each program point. In some cases, we are only interested in the environment conditions at certain program points. For example, we may only be interested in the environment conditions following a method call. We call this abstraction the *method call conditions*. Let `Callees` be the set of invoked functions. Then the weakest sufficient condition on function calls (*i.e.*, the largest set of states that guarantees the program always terminates in a good state) is the intersection of all the sufficient states at return points. Formally, we need the monotonic abstraction function:

$$\alpha_{\mathcal{S}} = \lambda r.\ \lambda \mathtt{f}.\ \bigcap_{\mathtt{pc} \in \mathtt{Callees(f)}} r(\mathtt{pc}).$$

Therefore, $\mathcal{S} \triangleq \alpha_{\mathcal{S}}(\mathcal{S}_\Sigma)$ are the *weakest* sufficient conditions on callees. They are an under-approximation of $\vec{\mathcal{S}}$. In the example of Fig. 4a, the weakest sufficient condition on `f` is $\mathcal{S}(\mathtt{f}) = \{s \mid s(\mathtt{ret}) > 101\}$. No matter which branch is taken, if `f` returns a value larger than `101`, the assertion holds.

The *necessary* call conditions are given by the Galois connection $\langle \mathtt{PC} \rightarrow \wp(\Sigma), \subseteq \rangle \underset{\alpha_\Sigma}{\overset{\gamma_\Sigma}{\leftrightarrows}} \langle \mathtt{Callees} \rightarrow \wp(\Sigma), \subseteq \rangle$. The abstraction function α_Σ merges the states after all invocations of a given callee `f`. Formally, the abstraction function is

$$\alpha_{\mathcal{N}} = \lambda r.\ \lambda \mathtt{f}.\ \bigcup_{\mathtt{pc} \in \mathtt{Callees(f)}} r(\mathtt{pc})$$

and the concretization function is

$$\gamma_{\mathcal{N}} = \lambda n.\ \lambda \mathtt{pc}. \begin{cases} n(\mathtt{f}) & \mathtt{pc} \in \mathtt{AfterCall} \land \mathtt{pc} \in \mathtt{Callees(f)} \\ \Sigma & \text{otherwise} \end{cases}$$

Therefore, $\mathcal{N} \triangleq \alpha_{\mathcal{N}}(\mathcal{N}_\Sigma)$ are the *strongest* conditions on the callees which are *necessary* for the program to be correct. They are an over-approximation of $\vec{\mathcal{N}}$. In the example of Fig. 4a, the strongest necessary condition on `f` is $\mathcal{N}(\mathtt{f}) = \{s \mid s(\mathtt{ret}) > 99\}$, *i.e.*, if the assertion holds then `f` returns a value larger than 99.

6. Semantic Conditions Injection

Our goal now is to apply the extracted environmental conditions to a new version of the program P′. The idea is that the previous conditions provide a semantic baseline that allow us to report only errors which are *new* in the modified program P′, or to prove its *correctness* under the same environmental assumptions made in the good runs of P.

Syntactic differences We call P the *base* program and P′ the *new* program. We do not impose any constraint on how much P and P′ control flow graphs differ. For the moment, and for the theoretical treatment, we assume we are given a function $\delta \in \mathtt{PC(P)} \rightarrow \mathtt{PC(P')} \cup \{\bot\}$ that captures the syntactic changes between P and P′. The δ function maps a program point from the base program to its corresponding point in the new version or to $\bot$ if the program point has been removed. For simplicity, we assume: (i) P and P′ have the same variables, otherwise a further projection function mapping variables of P into P′ should be introduced; and (ii) each program point of P corresponds at most to one program point in P′, otherwise we need to lift the co-domain of δ to sets of program points, and add a pair-wise disjunction hypothesis on its image.

We define the image of a trace w.r.t. δ:

$$\alpha_\delta = \lambda \vec{s}. \begin{cases} \epsilon & \text{if } \vec{s} = \epsilon \\ s\alpha(\vec{s'}) & \text{if } \vec{s} = s\vec{s'} \land \delta(\pi(s)) \neq \bot \\ \alpha(\vec{s'}) & \text{if } \vec{s} = s\vec{s'} \land \delta(\pi(s)) = \bot \end{cases}$$

that is, $\alpha_\delta(\vec{s})$ abstracts away all the states in $\vec{s}$ which do not refer to a program point in P′.

Semantic filtering Intuitively, applying the trace environmental conditions $\vec{\mathcal{E}}$ to P′ means restricting the traces in the concrete semantics of P′ to those which are *compatible* with $\vec{\mathcal{E}}$. Formally, the *δ-filtered* maximal trace semantics of P′ is

$$\vec{\tau}^{+}_{\mathtt{P'}} \lfloor_\delta \vec{\mathcal{E}} = \{\vec{s} \in \vec{\tau}^{+}_{\mathtt{P'}} \mid \exists \vec{t} \in \vec{\mathcal{E}}.\ \alpha_\delta(\vec{t}) = \alpha'(\vec{s})\},$$

i.e., we keep only those traces in $\vec{\tau}^{+}_{\mathtt{P'}}$ such that the sequence of choices made by the environment is compatible with the trace condition $\vec{\mathcal{E}}$. It follows trivially from the definition that this filtering yields a subset of the concrete traces, *i.e.*, $\vec{\tau}^{+}_{\mathtt{P'}} \lfloor_\delta \vec{\mathcal{E}} \subseteq \vec{\tau}^{+}_{\mathtt{P'}}$.

VMV($\mathcal{S}$): VMV with sufficient conditions Say that we choose to inject sufficient conditions $\vec{\mathcal{S}}$ from P into P′. What can we deduce on $\mathtt{P'}_{\vec{\mathcal{S}}}$? Intuitively, if $\mathtt{P'}_{\vec{\mathcal{S}}}$ has a bad trace, then a new error has been introduced. We know by construction that, if `id` denotes the identity function, $(\vec{\tau}^{+}_{\mathtt{P}} \lfloor_{\mathtt{id}} \vec{\mathcal{S}}) \cap \mathcal{B}^* = \emptyset$—the history conditions $\vec{\mathcal{S}}$ were sufficient to eliminate all the bad runs for P.

We say that a program P′ has a δ-regression w.r.t. a mapping function δ if there exists a sufficient condition s for P which is no more sufficient to eliminate all the bad runs. More formally, there exists a trace $s \in \vec{\mathcal{S}}$ such that $\vec{\tau}^{+}_{\mathtt{P'}} \lfloor_\delta \{s\} \cap \mathcal{B}^* \neq \emptyset$. From the previous definitions and basic set theory it immediately follows:

THEOREM 1. *If $\underline{\vec{\mathcal{S}}} \subseteq \vec{\mathcal{S}}$ and $(\vec{\tau}^{+}_{\mathtt{P'}} \lfloor_\delta \underline{\vec{\mathcal{S}}}) \cap \mathcal{B}^* \neq \emptyset$ then P′ has a δ-regression.*

An immediate consequence of the theorem is that we can use, *e.g.*, call-based sufficient conditions $\mathcal{S}$ (which under-approximate $\vec{\mathcal{S}}$) to prove δ-regressions.

VMV($\mathcal{N}$): Semantic baselining with necessary conditions Now, let us inject the extracted necessary conditions $\vec{\mathcal{N}}$ into P′, and let us suppose that we find no bad execution—formally, $(\vec{\tau}^{+}_{\mathtt{P'}} \lfloor_\delta \vec{\mathcal{N}}) \cap \mathcal{B}^* = \emptyset$. Intuitively, we can conclude that under the same environment conditions that hold for all the good runs (and maybe some of the bad runs) of P opportunely injected via δ, P′ has no bad runs. As a consequence: (i) P′ does not require stronger environmental hypotheses than P to avoid bad runs, and (ii) if $\vec{\mathcal{S}} \subset \vec{\mathcal{N}}$, *i.e.*, $\vec{\mathcal{N}}$ is strictly not sufficient for the absence of bad runs in P, then P′ fixed the errors in P for which $\vec{\mathcal{N}}$ was insufficient. We call the first property relative δ-correctness (or simply relative correctness) and the latter δ-fixing. What said is summarized by:

THEOREM 2. *Let $\vec{\mathcal{N}} \subseteq \overline{\vec{\mathcal{N}}}$ and $(\vec{\tau}^{+}_{\mathtt{P'}} \lfloor_\delta \overline{\vec{\mathcal{N}}}) \cap \mathcal{B}^* = \emptyset$. Then P′ is relatively δ-correct with respect to P. Furthermore, if also $(\vec{\tau}^{+}_{\mathtt{P}} \lfloor_\delta \overline{\vec{\mathcal{N}}}) \cap \mathcal{B}^* \neq \emptyset$ then P′ δ-fixed a bug in P.*

An immediate consequence is that we can use call-based necessary conditions $\mathcal{N}$, which over-approximate $\vec{\mathcal{N}}$, to prove relative correctness. Note that in general the strongest necessary conditions for P may not be necessary conditions for P′ anymore. For instance, if P′ removes an assertion from P, the necessary conditions for P may preclude some good runs in P′.

Combining VMV($\mathcal{S}$) and VMV($\mathcal{N}$) The results above inspire an immediate algorithm combining VMV($\mathcal{S}$) and VMV($\mathcal{N}$) to prioritize the reported alarms. First, extract $\underline{\mathcal{S}}$ and $\overline{\mathcal{N}}$ from P. Since the extraction routines are completely independent, this can be done in parallel. Second, analyze $\mathtt{P'}_{\underline{\mathcal{S}}}$ and $\mathtt{P'}_{\overline{\mathcal{N}}}$ (in parallel) and collect the reported alarms (say respectively $A_{\mathcal{S}}$ and $A_{\mathcal{N}}$). Third, report the alarms $A_{\mathcal{S}}$, as we know they are δ-regressions. Fourth, report the set of alarms $A_{\mathcal{N}} \setminus A_{\mathcal{S}}$—which may include real bugs masked by $\mathcal{S}$. Fixing all the alarms accomplishes full (relative) verification.

7. Abstraction(s)

The semantic filtering and the two theorems above give us the *best* theoretical solution to the problem of cross-version semantic condition injection and, correspondingly, to the problem of verification modulo versions. Unfortunately, such a theoretical, concrete solution is not practical for many reasons. First, the exact computation of the success semantics and history conditions is infeasible—the exact inference of $\vec{\mathcal{S}}$ and $\vec{\mathcal{N}}$ is impossible in general. Second, the theoretical results are parameterized by the cross-versions mapping function δ—we certainly cannot require a user to provide δ. Third, every static analyzer, deductive verifier, model checker, or type system performs some over-approximation of $\vec{\tau}$: do the results above still hold when an abstract semantics is used? In this section we will address these orthogonal topics.

Abstraction of sufficient conditions Our use of the *weakest* sufficient conditions $\vec{\mathcal{S}}$ is sound and complete for detecting δ-regressions: it guarantees that we capture *all* the regressions of $\mathtt{P}'$ w.r.t. $\mathtt{P}$ for the changes encoded by δ. If we consider some under-approximation $\underline{\mathcal{S}} \subset \vec{\mathcal{S}}$, then all the regressions we capture are definite δ-regressions w.r.t. $\mathtt{P}$ but we may miss some. For instance, consider again the example in Fig. 1a, but now suppose that the tool extracted the sufficient condition `ret > 10`, under-approximating `ret > 0`. When such a condition is injected in `Sufficient'`, the analyzer proves the assertion, and no alarm is raised. Therefore, it failed to spot the δ-regression. Overall, when analyzing $\mathtt{P}'_{\underline{\mathcal{S}}}$, we may suppress too many warnings both because of the nature of sufficient conditions and because of approximation. We judge that if the goal is to provide an advanced bug-finding tool then one should use VMV($\mathcal{S}$).

Abstraction of necessary conditions Our use of the *strongest* necessary conditions is sound and complete for proving relative δ-correctness. However, if we consider an over-approximation $\overline{\vec{\mathcal{N}}} \supset \vec{\mathcal{N}}$, we may fail in proving relative completeness. For instance, consider the example in Fig. 2a. Suppose we extract the condition `ret > -10` from the base program. Then we cannot prove that `Necessary'` fixed `Necessary`. As a consequence, when analyzing $\mathtt{P}'_{\overline{\mathcal{N}}}$ we may not suppress all the warnings from $\mathtt{P}$. In general, necessary conditions are more practical as it is easier to craft useful over-approximations than it is to craft useful under-approximations, though our framework can use either approach. Overall, we judge that VMV($\mathcal{N}$) is useful when the goal is verification. The assumptions carried over from the previous version of the program will significantly reduce the number of alarms to consider, while providing (relative) correctness guarantees. We will see in Sec. 8 that, in our setting, for 77% of the alarms the necessary conditions are also sufficient. The programmer can then focus her efforts on fixing or suppressing the remaining alarms until the number is reduced to zero. The resulting warning-free program can be used as the baseline for future versions of the program. Any warnings reported will represent either definite regressions or new warnings.

Computing δ in practice Thus far, we have assumed that we are *given* the syntactic change map δ. In general, automatically computing δ in a meaningful way is impossible because several different δ's can characterize the transformation from $\mathtt{P}$ to $\mathtt{P}'$. For example, consider the program change shown in Fig. 4. The base program contains two invocations of `f`: one at program point `1`, the other at program point `4`. The new program contains only one invocation of `f`. Which (if either) of the two calls in the base program corresponds to the call in the new program? In other words, which is the correct δ: $\langle \mathtt{1} \mapsto \mathtt{0}, \mathtt{4} \mapsto \bot\rangle$, $\langle \mathtt{1} \mapsto \bot, \mathtt{4} \mapsto \mathtt{0}\rangle$, or $\langle \mathtt{1} \mapsto \bot, \mathtt{4} \mapsto \bot\rangle$? For this program, it is ambiguous which δ is the correct choice. However, this arbitrary choice will have semantic ramifications because each state corresponding to a call to `f` in the base program has a different correctness condition associated with it. That is, each (equally valid) choice of δ will result in a different correctness condition being ported to $\mathtt{P}'$. This is undesirable.

Fortunately, we found that we can avoid many of the problems associated with computing δ by using the call condition semantics $\mathcal{S}, \mathcal{N}$. Unlike the state condition semantics or any scheme based on syntactically matching assertions, which require computing a δ that matches all program points, using call conditions only requires syntactically matching callee names across program versions. As long as methods are not renamed, this approach is quite reliable. For example, in Fig. 4, the method `f` is called in both versions of the program. Since we merge the call conditions for *all* invocations of a function, changing the number of calls to a particular function across program versions presents no difficulty. Contrast this with the ambiguity introduced by changing the number of calls to `f` during our (previous) attempt to construct a δ to port the state conditions across versions.

Therefore, for each function `f` that is called in both $\mathtt{P}$ and $\mathtt{P}'$, we simply assume the semantic condition after all calls to `f` in $\mathtt{P}'$. Essentially, we treat the correctness conditions as if they were postconditions for `f` that `f`'s body does not need to prove. We should be careful on how to merge semantic conditions from multiple occurrences of `f` in $\mathtt{P}$. The theory developed in the previous sections guides us. In the example of Fig. 4a suppose $\underline{\mathcal{S}}^1, \overline{\mathcal{N}}^1$ and $\underline{\mathcal{S}}^4, \overline{\mathcal{N}}^4$ are the inferred sufficient and necessary conditions at program points `1` and `4`. Then, after line `0` in Fig. 4b we will inject an under-approximation of $\underline{\mathcal{S}}^1 \wedge \underline{\mathcal{S}}^4$ and/or an over-approximation of $\overline{\mathcal{N}}^1 \vee \overline{\mathcal{N}}^4$. The injected program $\mathtt{P}_{\mathcal{E}}$ can be readily analyzed, and the results of this analysis can be reported to the user.

Approximation introduced by the tool All sound automatic verifiers and static analyzers over-approximate the concrete semantics τ^+. When over-approximation adds execution traces that end in an error state, analyzers will report false alarms. Therefore, the analysis is bound to be incomplete—cannot prove every fact that holds. What guarantees does VMV provide when using a tool $\mathcal{T}$ over-approximating τ^+? The answer depends on whether we are considering sufficient or necessary conditions.
Sufficient conditions. Suppose we use an inference tool generating a sufficient condition $\underline{\mathcal{S}}$. In this case, we require that the inference of sufficient conditions generates a condition $\underline{\mathcal{S}}$ that is provable under $\mathcal{T}$. The condition `false` can always be used as a default in the absence of non-trivial conditions. We define an *abstract δ-regression* if $\mathcal{T}$ does not prove $\mathtt{P}'_{\underline{\mathcal{S}}}$. An abstract regression is *spurious* if $\vec{\tau}^{+}_{\mathtt{P}'} \lfloor_{\delta} \underline{\gamma}(\underline{\mathcal{S}}) \cap \mathcal{B}^* = \emptyset$, for an appropriate $\underline{\gamma}$, but $\mathcal{T}$ cannot prove it. For instance, suppose that in Fig. 1 we instantiate $\mathcal{T}$ with the sign abstract domain [8]. In Fig. 1a, the condition $\mathtt{y} = \mathtt{positive}$ is a sufficient one—$\gamma(\mathtt{positive}) = \{n \in \mathbb{N} \mid n > 0\}$. When using this abstraction in the evolved programs of Fig. 1b and Fig. 1c, at line `1`, $\mathtt{positive} - \mathtt{positive} = \top$, therefore $\mathcal{T}$ reports an abstract $\delta-$regression for both cases. For Fig. 1c, however, it is a spurious abstract δ-regression. The spurious abstract regression can be eliminated if: (i) we use the most precise transfer function for signs [18]; or, (ii) we use a more precise numerical domain, *e.g.*, intervals [8]. We believe abstract δ-regressions (even the spurious ones) may be of interest to the user as they occur not only due to the analysis imprecision, but rather due to a change that can make existing analysis more imprecise. We plan to investigate the concept of (spurious) abstract regressions in the future.
Necessary conditions. Now suppose our inference tool generates a necessary condition $\overline{\mathcal{N}}$. In this case, there is no need to use $\mathcal{T}$ to prove that $\overline{\mathcal{N}}$ is necessary for $\mathtt{P}$. When using $\mathcal{T}$ on the instrumented program $\mathtt{P}'_{\overline{\mathcal{N}}}$, all the proven assertions are $\delta-$correct, but because of incompleteness $\mathcal{T}$ may fail to prove some δ-correct assertion. If $\overline{\mathcal{N}}$ is a good approximation and $\mathcal{T}$ is precise enough, then the alarms generated for $\mathtt{P}'_{\overline{\mathcal{N}}}$ should be far fewer than those

generated for $\mathtt{P}'$ alone. Therefore VMV($\mathcal{N}$) is suitable for relative verification—prove assertions of $\mathtt{P}'$ under the same environmental assumptions that hold in the good runs of P.

Non-Monotonicity. Orthogonal to what has been said above, a subtle problem can arise due to the (often forgotten) fact that practical analysis tools are not monotonic. That is, it is possible for an analyzer to compute a less precise result given more precise inputs (*e.g.*, more assumptions). This can happen for many reasons: the use of widening, timeouts, non-monotonic state space finitization, *etc*. In the context of VMV, this can result in an odd case where analyzing the new program $\mathtt{P}'$ yields *fewer* warnings than analyzing the refined program $\mathtt{P}'_{\mathcal{E}}$. An easy way to solve this (very obscure) problem is to consider the intersection of the warnings generated by the analyses of P and $\mathtt{P}_{\mathcal{E}}$. Note that computing this intersection does not pose any problems with respect to syntactic matching because the only syntactic differences between $\mathtt{P}'$ and $\mathtt{P}'_{\mathcal{E}}$ are the addition of our correctness conditions $\mathcal{E}$.

8. Experience

Tool We implemented VMV on top of the industrial-strength static analyzer `cccheck` [17]. `cccheck` is a modular static analyzer and contract verifier with inter-method inference: it builds an approximation of the call-graph, visits it bottom-up, implements assume/guarantee reasoning for function calls, and infers contracts to reduce the annotation burden. Intra-procedurally, `cccheck` uses abstract interpretation [8] to infer invariants at each program point. It includes abstract domains to model the heap, array contents [11], nullness, integer values [26, 29], floating points [30], enums, among others. It leverages the inferred invariants to prove the assertions in the code. Assertions can be either explicit (contracts) or implicit (null-pointer dereferences, buffer overrun, division by zero, etc.). `cccheck` uses the CPPA algorithm from [12] to infer necessary preconditions. The main source of imprecision (and of false warnings) of `cccheck` is the handling of external code: when no contract is available to describe an API behavior, `cccheck` soundly assumes the worst case. As a consequence, it may report many alarms that appear "dumb" to a programmer who has internalized implicit assumptions about API behavior. A starting point for the development of VMV was to automatically identify these assumptions and inject them into future versions of the code base to suppress "dumb" warnings.

We picked one particular point in the design space enabled by VMV (Fig. 3): VMV($\mathcal{N}$) with call conditions insertion for δ and CPPA for (over-)approximation of $\mathcal{N}$. There are several reasons for this choice: (i) we are interested in understanding the capabilities of `cccheck` as a relative verifier (that is, for proving the correctness of $\mathtt{P}'$ w.r.t P); (ii) the CPPA implementation in `cccheck` is very robust, whereas the tool lacks algorithms to infer sufficient conditions; (iii) designing a sufficient condition inference algorithm good enough to be useful in practice is difficult [10].

We have modified `cccheck`'s CPPA implementation to compute necessary conditions on method entry (entry-assumptions) and at called method return points (callee-assumptions) — *i.e.*, for each function call f, we infer a necessary condition $\overline{\mathcal{N}}_{\mathtt{f}}$. We extended the `cccheck` caching mechanism to store the extracted conditions in a SQL database for retrieval on subsequent analysis runs. We build the instrumented program $\mathtt{P}'_{\overline{\mathcal{N}}}$ by reading those conditions from the database and injecting them as opportune `assume` statements in $\mathtt{P}'$. Entry-assumptions are extra-assumptions at the method entry point. Callee-assumptions are extra-assumptions after method calls — *i.e.*, we add an `assume` $\overline{\mathcal{N}}_{\mathtt{f}}$ statement after each call to f.

Code under analysis We consider two code bases: the open-source `ActorFx` framework and the non-public CB framework. We chose `ActorFx` to enable reproducibility of our results and CB to test VMV on complex code bases. `ActorFx` provides a language-independent model of dynamic and distributed objects for Windows Azure. It is available as open-source project at `actorfx.codeplex.com`. The base changeset is 86788 (randomly picked) and the new one is 87786 (the latest available at the time of this writeup, roughly three months later). CB is a large, complex, and high-quality production code base at Microsoft, made up of 57 C# projects. It includes (among other things) a parser, a compiler, a tracer, a distributed file system and a distributed runtime. The code base interacts with the file system, network, underlying operating system, *etc.*, and it heavily uses the .NET serialization/deserialization mechanism. As a consequence, the correctness of many assertions in the code base rely on implicit assumptions about API behavior which are unknown to `cccheck`. The base version is snapshot 12833 of the code (randomly picked) and the new version is snapshot 13311 (three weeks later). Both projects have a few CodeContracts [16] annotations. Contracts range from simple non-null assertions or arithmetic relationships between variables to complex algebraic properties between the entry and exit states of a method.

`ActorFx` Results We report in Fig. 5 the results of applying VMV($\mathcal{N}$) to the three largest projects of the `ActorFx` framework. The first column is the name of the project, the second column reports the fraction of the number of external libraries over the total number of libraries referenced by the project. We define a library as external if it is not part of .NET nor of `ActorFx`. External libraries are unknown to `cccheck`, which soundly assumes the worst case for them. The columns $3-6$ report respectively the total number of methods analyzed, proof obligations, raised alarms, and inferred necessary conditions for the base program P. The raised alarms (warnings) are the assertions that the tool could not prove safe. Columns $7-9$ contain the total number of methods analyzed, checks, and warnings emitted by `cccheck` on the new program $\mathtt{P}'$. The next two columns indicate the total number of warnings raised on $\mathtt{P}'_{\overline{\mathcal{N}}}$ and the number of relative verified assertions. Finally, the last three columns report our experience of annotating $\mathtt{P}'_{\overline{\mathcal{N}}}$ to reduce the number of warnings to zero.

The first observation is that VMV($\mathcal{N}$) dramatically reduces the number of alarms to inspect in $\mathtt{P}'_{\overline{\mathcal{N}}}$, while still providing semantic guarantees on the proven assertions. For the `ActorLib'`, `cccheck` proves 2099 assertions absolutely correct and 316 assertions relatively correct when using VMV($\mathcal{N}$), leaving only 137 warns to inspect—roughly reducing the number of warnings by 70%. For `ActorCore'`, 1146 assertions are proven independently of the previous versions and 70 more carrying over hypotheses on the *good* runs of `ActorCore`, so that the user is left a mere 37 warns to look at. The best results are on `System.Cloud`, where VMV($\mathcal{N}$) enables the relative proof of 214 more assertions on top of the 1178 `cccheck` proved on $\mathtt{P}'$ on its own, lowering the number of warnings by more than 80%. Overall, VMV($\mathcal{N}$) proved the relative correctness of 72.6% of the warnings in $\mathtt{P}'$.

Next, we focused on the alarms in $\mathtt{P}'_{\overline{\mathcal{N}}}$, evaluating how much effort is needed to go to zero warnings. It turns out that, as expected, VMV($\mathcal{N}$) eliminated the largest majority of warnings originating from the use of external APIs, allowing the developer to focus only on "interesting" warnings. We added several contracts (mainly object invariants) to the source code, and we found some bugs in the code that we reported to the developers. It is worth observing that: (i) often a contract was enough to prove more than one assertion at the time, *i.e.*, we added fewer annotations than there were warnings; (ii) the total number of checks in the annotated program significantly increased with respect to $\mathtt{P}'$. Overall, the annotation process for the three libraries took us just a couple of hours.

CB Results. In the next experiment, we continued our investigation into VMV($\mathcal{N}$) by quantifying how far from sufficient condi-

	Original code base P					New code base P'			$P'_{\overline{\mathcal{N}}}$		Annotated $P'_{\overline{\mathcal{N}}}$		
Project	Ext. libs	Meth.	Checks	Warns	inf. $\mathcal{N}$	Meth.	Checks	Warns	Warns	Rel. proofs	Checks	Warns	Contracts
`ActorLib`	7 / 15	303	2182	407	324	320	2552	453	137	**316**	2861	0	86
`ActorCore`	4 / 5	149	1284	162	66	159	1253	107	37	**70**	1383	0	32
`System.Cloud`	4 / 12	176	1418	273	226	179	1444	266	52	**214**	1524	0	35

Figure 5: Results of analysis of `ActorFx`, with `cccheck` and VMV($\mathcal{N}$). In bold the number of relatively verified assertions.

tions the inferred necessary conditions are. We also performed a qualitative analysis to find anecdotes of relative verification, and conversely, where relative verification failed. Fig. 6 shows the result of applying VMV($\mathcal{N}$) to `CB`. The first column is the name of the project—we removed projects that did not change at all between the two snapshots. The next three columns report, for `P`, the total number of methods, assertions, and issued warnings.

Necessary conditions, in general, are not sufficient, *i.e.*, they may not prove all assertions in the code. To get an idea of how far from sufficient the inferred conditions are, the 5th column reports the number of warnings issued when applying the inferred necessary conditions $\overline{\mathcal{N}}$ to the base program `P` itself. It is worth noting that in general, the warnings emitted for $P_{\overline{\mathcal{N}}}$ are substantially lower than for the analysis of `P` alone (col. 5 *vs* col. 4). In three projects, the inferred necessary conditions are sufficient to prove all assertions correct, and in six additional projects, there were fewer than 10 warnings remaining. The results were particularly dramatic for project `DF`, where the number of warnings was reduced by 480. Overall, $P_{\overline{\mathcal{N}}}$ has 77.2% fewer warnings than `P`.

The next three columns report the total number of methods, assertions, and warnings issued for P' with no condition injection. The next group of columns list the number of new methods in P', their assertions, and the number of warnings issued for these methods. We classify a method as new if we found no entry for it in the cache when looking up necessary conditions of `P`. This happens both when the method is genuinely new or when the method's name or signature changed from `P` to P'. We separate new methods because VMV does not have any way to add assumptions for them.

The last five columns are the most interesting for our evaluation of VMV. These columns show the number of methods `C'` that we consider changed between `P` and P'. The way we identify if a method has changed is via a semantic hash of the method's instructions, including all pre/post conditions and invariants of the method as well as the methods called from it. If a method's hash is the same as before, then the method did not change between versions, and for those methods VMV($\mathcal{N}$) boils down to just checking that the extracted necessary conditions are also sufficient (*i.e.*, col.5). Therefore, we focus on those methods whose semantic hash changed; these are the cases where the *semantic* assumptions carried over from previous versions should enable the proof of the modified code. The last three columns show the warnings for `C'`, $C'_{\overline{\mathcal{N}}}$, and the consequent number of relatively verified assertions. The results of the analysis of $C'_{\overline{\mathcal{N}}}$ are encouraging. In 10 projects all the code changes have been relatively verified. In 16 additional projects, there are less than 15 warnings issued for the changed methods. The remaining 7 projects have a large number of remaining warnings (21–127). Note the large number of warnings issued on the $P_{\overline{\mathcal{N}}}$ configuration for projects such as `CM`. This indicates that the computed $\overline{\mathcal{N}}$ is sometimes too weak to prove assertions even in the unchanged version of the code base. Since we use (approximations of) necessary conditions, there is always a possibility that the conditions will not be sufficient to prove assertions even in the original version of the program.

Qualitative analysis of remaining warnings. We find that our technique is quite effective at (relatively) proving the correctness of assertions and hence at reducing the number of warnings issued. Overall, VMV($\mathcal{N}$) enabled the relative proof of 568 more assertions, reducing the alarms by more than 50%. However, in order to reduce the number further we must improve the approximations we use to compute necessary conditions. We manually examined the 53 combined warnings of projects `BS`, `MC`, `TTR`, and `U` and determined why each warning was emitted. We found that 13 warnings were emitted properly as they resulted from new assumptions being made in the code, such as additional fields being assumed to be non-null. In one case, a field that was previously initialized in the method was now initialized in the constructor instead, then later used in the same method under the assumption that it was still non-null. In other cases, the warnings originated either from incompleteness in the inference of $\overline{\mathcal{N}}$ or from limitations in using a δ map exclusively based on call-conditions. For instance, a warning occurred inside a for-loop iterating over array elements that are assumed to be non-null. We currently don't infer conditions for loop heads, so there was no program point to attach the necessary condition to. Our measurements indicate that adding loop-heads matching would improve the effectiveness of VMV. Nevertheless, matching loop-heads between programs `P` and P' will introduce some inevitable brittleness induced by the loop-head matching algorithm. Overall, it's worth noting that most of the failures in proving relative correctness we observed during the experiments are related to tool limitations rather than to problems with VMV($\mathcal{N}$) itself. Whenever `cccheck` succeeded in inferring necessary conditions, the conditions were typically installed correctly and used effectively to mask warnings in the later version of the project.

Qualitative analysis of relative verification. We manually investigated the list of assertions in `C'` for which our approach provides relative verification (last column of Fig. 6). We can classify the reasons into one of the following categories: (i) the lines containing the assertions were deleted in the new version; (ii) the assertion appears prior to any changes in the method; (iii) the condition carried over implies the correctness of the new assertion; and (iv) bug fix (warning in $P_{\overline{\mathcal{N}}}$ but not in $P'_{\overline{\mathcal{N}}}$). Except for (i), all other categories demonstrate the use of necessary conditions. One common example of (iii) appears to be when the programmer changed the API, but correctness relied on the same semantic argument. For instance:

```
[--] m_vFileSystem    = new LocalFileSystem(fs)
[++] m_vFileSystemSet = new VirtualFileSystemSet(fs, ..)
```

The correctness condition that ensures that the parameter `fs` is non-null and is valid (preconditions `LocalFileSystem`) of the old version also ensures that the argument to the new API call is non-null (precondition of `VirtualFileSystemSet`). Please note that VMV does not need to match the call to `LocalFileSystem` with that to `VirtualFileSystemSet`: it only uses the semantic information on `fs`.

We found a few examples of (iv), where the change turns a possibly-null value into a non-null value:

```
[--] tArgs = ResolveTarget(rtTask.Arguments, target);
[++] tArgs = ResolveTarget(rtTask.Arguments != null ?
[++]            rtTask.Arguments : string.Empty, target);
```

We also encountered cases where the necessary condition inferred a universally quantified condition on all elements in a collec-

	Original code base P				Updated code base P′										
					Total for P′			New code			Changed code C'				
Project	Meth.	Checks	Warns P	Warns $P_{\overline{\mathcal{N}}}$	Meth.	Checks	Warns	Meth.	Checks	Warns	Meth.	Checks	Warns	Warns $C'_{\overline{\mathcal{N}}}$	Rel. proofs
BS	113	1065	99	16	116	1090	98	3	9	0	30	369	34	3	**31**
C	302	2611	273	36	314	2620	270	21	180	16	10	220	25	0	**25**
CBCBC	25	595	98	13	25	598	99	0	0	0	1	136	32	0	**32**
CBF	323	1923	351	80	323	1922	351	0	0	0	1	5	5	0	**5**
CM	388	4129	641	223	436	4392	682	154	2553	408	58	922	167	127	**40**
DF	855	8074	622	142	868	8132	639	70	216	56	89	1967	177	94	**83**
EMSBD	101	1439	145	18	101	1445	149	3	185	13	4	148	31	21	**10**
EMSBWH	15	321	17	6	17	583	29	3	242	7	5	246	17	12	**5**
ES	53	857	148	36	47	757	123	4	126	8	5	135	44	13	**31**
MAF	3	37	3	0	3	34	3	0	0	0	2	33	3	0	**3**
MC	84	665	62	11	85	667	62	1	2	0	4	62	10	2	**8**
MRM	27	326	39	1	27	334	39	0	0	0	6	159	11	0	**11**
MS	53	575	52	11	53	575	52	0	0	0	1	4	0	0	**0**
MSBE	22	109	20	0	58	1252	109	41	1149	99	10	97	10	9	**1**
MT	16	107	5	0	23	138	5	7	41	0	2	8	0	0	**0**
PMSBG	30	477	92	36	30	482	100	1	21	2	12	388	94	69	**25**
PMSBMI	25	539	57	32	18	325	32	1	62	10	3	110	11	8	**3**
PP	24	507	66	12	24	507	69	0	0	0	6	199	32	13	**19**
PT	49	482	47	11	49	482	47	19	307	44	17	131	1	1	**0**
R	56	653	57	10	52	630	53	37	465	38	5	51	10	0	**10**
SI	172	1208	102	11	188	1453	144	53	561	84	9	120	4	3	**1**
TC	69	711	107	16	76	784	115	8	76	8	5	167	35	9	**26**
TCDPV	84	2298	183	36	84	2271	183	0	0	0	5	99	6	5	**1**
TCDTV	151	3821	394	90	153	3850	397	2	7	2	7	1530	104	49	**55**
TDD	25	506	46	2	25	418	45	1	4	3	3	274	19	1	**18**
TFC	12	162	22	6	12	162	22	1	84	15	0	0	0	0	**0**
TPGF	4	66	3	2	4	66	3	0	0	0	1	63	3	2	**1**
TTH	10	192	22	5	10	192	22	0	0	0	1	32	5	1	**4**
TTPG	52	1289	132	47	52	1304	132	1	23	0	1	230	10	10	**0**
TTR	46	1460	141	40	153	1864	170	108	429	37	21	1028	104	40	**64**
TTTCM	142	961	199	18	189	1572	295	69	855	140	44	414	100	58	**42**
U	381	5164	219	45	401	5266	237	24	85	16	14	305	22	8	**14**
VFS	20	209	45	16	48	458	84	47	457	84	0	0	0	0	**0**

Figure 6: Breakdown of warnings in new code base and effect of necessary baseline. C' denotes the set of changed methods in $P \cap P'$.

tion being non-null in the old version to (relatively) verify elements in the new version.

Semantic versus syntactic baselines. Our work on VMV was inspired by the limitations of the syntactic baseline feature implemented in `cccheck`, which uses warning type-based syntactic matching to suppress warnings. In 5+ years of using only the syntactic baseline feature, we found that that syntactic baselining masks too many alarms, including interesting ones. If our goal is simply to suppress warnings to ease the load on the programmer, syntactic baselining provides an acceptable alternative to VMV. However, if our goal is to review the code for potential new problems, VMV using necessary conditions provides better guarantees that no new code issues have been introduced. We believe that the two techniques have fundamentally different strengths and that both should be provided to users of the tool.

9. Related Work

Baselining Industrial strength-tools like Coverity [14], Klocwork [23], or Polyspace [31] provide a syntactic baseline feature to mask warnings. However, we could not find good documentation detailing the internals of their alarm suppression mechanism. Essentially, they enable suppression by manually providing annotations, creating XML filter files, or adding custom function stubs. FindBugs [21] uses a syntactic baselining mechanism based mostly on warning types and errs on the side of suppressing too much. The matching does not take variable names or line numbers into account and thus will usually suppress all warnings of type t for a given method if the baseline already contains a warning of type t for that method. This strategy seems quite reasonable for a tool like FindBugs that makes no exhaustiveness guarantees, but would not be acceptable for a sound tool.

Differential static analysis Differential static analysis [25] also seeks to utilize multiple versions of a program to improve analysis results. The differential algorithm in [22] filters away inputs causing alarms in a sequential version of a program to isolate the alarms that are unique to the concurrent version of the program. However, their approach is limited to bounded programs without loops and recursion. Recent work on differential assertion checking (DAC) [24] proves relative correctness between assertions across two versions of a program. The approach reduces checking DAC to a single composed program that can be analyzed by existing program verification techniques. Like our work, DAC is useful for identifying when program changes introduce bug fixes or regressions. Relational verification approaches based on *product programs* [3] have also been used to verify equivalence and information-flow safety of a pair of programs.

VMV differs from those works in several ways. Our framework provides the flexibility to extract and utilize either necessary or sufficient conditions from the previous version of the program, whereas other works typically fix the kind of conditions to extract. VMV can easily be added to any static analysis tool, whereas it is not clear how applicable differential techniques are to existing tools. For example, [22] only shows how to prune false positives in concurrent program analysis, [27] only applies when a program has many semantically related alarms, and [3, 25] necessitates the construction of a composed program that require the static analysis to analyze two versions of the program simultaneously. Finally, we showed that VMV can be applied to large code bases, whereas it is unclear how scalable the approaches above are.

Annotation inference Inferring annotations can significantly reduce the number of false alarms a static analyzer reports for a program [28, 33]. The tool `cccheck` already performs contract inference, but we find that on realistic code bases it still raises too many warnings (Sec. 8). A main reason for that is the difficulty of inferring annotations for external APIs. VMV turns out to be very useful in this case: it can infer a (necessary or sufficient) condition on the particular API, and then automatically insert in the next version of the program.

Several authors addressed the problem of under-approximating sufficient conditions—even if they did not explicitly state it in the terms of this paper. For instance, previous work in specification mining [2] and interface inference [1, 20] under-approximates sufficient history conditions $\vec{\mathcal{S}}$: their goal is to describe safe API usage by inferring sequences of API calls that are sufficient to prevent crashes in library code. Similarly, works on the inference of sufficient preconditions [5, 32] under-approximate our method call sufficient conditions $\mathcal{S}$. In the worst case, such approximations result in the sufficient condition `false`. Few authors addressed the inference of necessary condition. For instance, Bouaziz *et al.* [6] used necessary history conditions to infer a sequences of method calls that inevitably lead to a crash, and Cousot *et al.* [10] used them in the context of modular analysis. In the worst case, such approximations yield the necessary condition `true`.

10. Conclusion

In this paper, we addressed the problem of reducing the number of warnings, yet providing soundness guarantees, in real-life static analyzers by carrying over semantic information from previous versions. We showed that sufficient conditions (VMV($\mathcal{S}$)) are useful for bug finding whereas necessary conditions are best for relative verification (VMV($\mathcal{N}$)). We implemented VMV($\mathcal{N}$) in the popular `cccheck` tool, and showed that it drastically reduces the number of alarms while still providing semantic guarantees. In the future, we plan further investigation of VMV($\mathcal{S}$), the concept of abstract regression, and the combination of VMV($\mathcal{S}$) and VMV($\mathcal{N}$).

Acknowledgments. We would like to thank Sophia Drossopoulou for the carefull reading of an earlier version of this paper and the many the constructive comments.

References

[1] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *POPL*, 2005.

[2] G. Ammons, R. Bodík, and J.R. Larus. Mining specifications. In *POPL*, 2002.

[3] G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *FM*, 2011.

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

[5] S. Blackshear and S. K. Lahiri. Almost-correct specifications: a modular semantic framework for assigning confidence to warnings. In *PLDI*, 2013.

[6] M. Bouaziz, F. Logozzo, and M. Fähndrich. Inference of necessary field conditions with abstract interpretation. In *APLAS*, 2012.

[7] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 277(1-2), 2002.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[9] P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[10] P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic inference of necessary preconditions. In *VMCAI*, 2013.

[11] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, 2011.

[12] P. Cousot, R. Cousot, and F. Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *VMCAI*, 2011.

[13] P. Cousot, R. Cousot, F. Logozzo, and M. Barnett. An abstract interpretation framework for refactoring with application to extract methods with contracts. In *OOPSLA*, 2012.

[14] Coverity. Coverity static analysis verification engine. `http://www.coverity.com/products/coverity-save.html`.

[15] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8), 1975.

[16] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *ACM SAC*, 2010.

[17] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, 2010.

[18] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, 2000.

[19] GrammaTech. CodeSonar. `http://www.grammatech.com/codesonar`.

[20] T.A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *ESEC/FSE-13*, 2005.

[21] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12), 2004.

[22] S. Joshi, S.K. Lahiri, and A. Lal. Underspecified harnesses and interleaved bugs. In *POPL*, 2012.

[23] Klocwork. Klocwork inspect. `http://www.klocwork.com/products`.

[24] S.K. Lahiri, K.L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *FSE*, 2013.

[25] S.K. Lahiri, K. Vaswani, and C.A.R. Hoare. Differential static analysis: opportunities, applications, and challenges. In *FoSER*, 2010.

[26] V. Laviron and F. Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *VMCAI*, 2009.

[27] W. Lee, W. Lee, and K. Yi. Sound non-statistical clustering of static analysis alarms. In *VMCAI*, 2012.

[28] F. Logozzo. Automatic inference of class invariants. In *VMCAI*, 2004.

[29] F. Logozzo and M. Fähndrich. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *SAC*, 2008.

[30] F. Logozzo and M. Fähndrich. Checking compatibility of bit sizes in floating point comparison operations. In *3rd workshop on Numerical and Symbolic Abstract Domains*, ENTCS, 2011.

[31] Mathworks. Polyspace verifier. `http://www.mathworks.com/products/polyspace/`.

[32] Y. Moy. Sufficient preconditions for modular assertion checking. In *VMCAI*, 2008.

[33] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *ICSE*, 2011.