Bounded Abstract Interpretation

Maria Christakis[1] and Valentin Wüstholz[2]

[1] Microsoft Research, Redmond, USA
mchri@microsoft.com
[2] The University of Texas at Austin, USA
valentin@cs.utexas.edu

Abstract. In practice, software engineers are only able to spend a limited amount of resources on statically analyzing their code. Such resources may refer to their available time or their tolerance for imprecision, and usually depend on when in their workflow a static analysis is run. To serve these different needs, we propose a technique that enables engineers to interactively bound a static analysis based on the available resources. When all resources are exhausted, our technique soundly records the achieved verification results with a program instrumentation. Consequently, as more resources become available, any static analysis may continue from where the previous analysis left off. Our technique is applicable to any abstract interpreter, and we have implemented it for the .NET static analyzer Clousot. Our experiments show that bounded abstract interpretation can significantly increase the performance of the analysis (by up to 8x) while also increasing the quality of the reported warnings (more definite warnings that detect genuine bugs).

1 Introduction

Software engineers typically have very different resources to devote to checking correctness of their code by running a static program analysis. These resources refer to an engineer's available time, or their tolerance for imprecision and spurious errors. The availability of such resources may depend on several factors, like when in an engineer's workflow static analysis is run (for instance, in the editor, after every build, during code reviewing, or before a release), how critical their code is, or how willing they are to go through all warnings reported by the analysis.

As an example, consider that software engineers might choose to run a static analyzer for a very short amount of time (say, a few minutes) after every build, in which case they expect it to provide immediate feedback even if certain errors might be missed. In contrast, when waiting for code reviewers to sign off, engineers could take advantage of the wait and resume the analysis of their code from where it had previously left off (say, after the last build). However, this time, the analysis can run for a longer period of time since code reviews take at least a few hours to complete [29, 28]. Engineers would now expect to find most errors in their code before making it available to others and are, therefore, more tolerant to spurious errors.

```
1  public void M() {
2    int c = 0;
3    while (*) {
4      if (c < 7) {
5        c++;
6      }
7    }
8    assert c < 585;
9    assert 7 < c;
10 }
```

Label	Interval state for c	
	Intermediate	Final
2	$\bot$	$\bot$
3	$[0, 1]$	$[0, \infty]$
4	$[0, 1]$	$[0, \infty]$
5	$[0, 1]$	$[0, \infty]$
6	$[1, 2]$	$[1, \infty]$
7	$[1, 2]$	$[0, \infty]$
8	$[0, 1]$	$[0, \infty]$
9	$[0, 1]$	$[0, \infty]$
10	$[0, 1]$	$[0, \infty]$

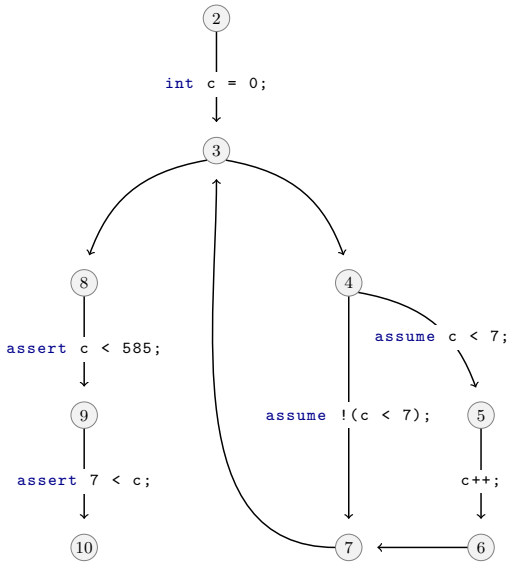

Fig. 1: Example that demonstrates loss of precision in the analysis after performing a widening operation at program point 3 in method `M`. The source code for method `M` is shown on the top left. On the right, we show the control flow graph of the method (program points are depicted as nodes, and edges capture control flow and code). On the bottom left, we show the abstract interval state for variable `c` at an intermediate and the final state.

In this paper, we propose a technique that enables users to *interactively* adapt a static analysis to their available resources. We focus on static analysis in the form of abstract interpretation [11]. When there are no more resources, our technique *soundly* records, in the form of a program instrumentation, any correctness guarantees that have been achieved until that point in the analysis. Consequently, once more resources become available, the static analysis will not start from scratch; it will instead reuse these already collected verification results to continue from where it left off.

As an example, consider method `M` on the top left of Fig. 1. On line 2, we initialize a counter `c`, which is later incremented in the body of a non-deterministic while-loop (lines 3–7). After the loop, there are two assertions, the first of which (line 8) always holds, whereas the second one (line 9) constitutes an error.

When analyzing method `M` with the relatively imprecise interval domain [11], the abstract interpreter generates two warnings, one for each assertion in the code. Our technique allows software engineers to bound the analysis such that it runs faster and only reports the genuine error on line 9. Since the bounded

analysis might miss errors, it is also possible to continue analyzing the code at a later point by reusing the verification results that have been previously collected.

Although bounded abstract interpretation is a general approach for bounding the analysis based on user resources, the possible analysis bounds that we discuss in this paper are time, imprecision, and the number of spurious errors. The bounded abstract interpreter is a verification tool until a bound is reached, at which point it switches to bug-finding while precisely recording its own unsoundness. This technique opens up a new way of building highly tunable abstract interpreters by soundly recording their intermediate verification results.

Our paper makes the following contributions. It presents:

- an algorithm for bounding a static analysis based on abstract interpretation according to the user's available resources,
- verification annotations and an instrumentation for soundly recording any intermediate verification results when a bound of the analysis is reached,
- three application scenarios for interactively tuning an analyzer to reduce the time, imprecision, and number of spurious errors of the analysis, and
- an experimental evaluation that explores the potential of bounded abstract interpretation when applied in a real, industrial analyzer.

Outline. The following section introduces the verification annotations that we use to record any intermediate verification results of the abstract interpreter. In Sects. 3 and 4, we present the algorithm for soundly recording these results and several application scenarios of our approach. In Sect. 5, we discuss our experimental results. We review related work in Sect. 6 and conclude in Sect. 7.

2 Verification Annotations

To soundly encode intermediate, partial verification results [30], we use two kinds of annotations. In our previous work, an *explicit assumption* of the form `assumed` P `as` a expresses that an analysis assumed property P to hold at this point in the code without checking it [6, 8, 5, 30]. The unique *assumption identifier* a may be used to refer to this explicit assumption at other program points. In the context of bounded abstract interpretation, an `assumed` statement is used to express that the analyzer unsoundly assumes that a property P holds for all possible program states at a given program point. Actually, P holds only for those states that were explored by the analyzer within its limited resources, which is why the assumption is unsound.

As an example, consider that right before performing the widening operation at program point 3 of method `M` (see Fig. 1), the abstract interpreter runs out of user resources. At this stage of the analysis, we have learned that $\mathtt{c} \in [0, 1]$, as is also shown by the table in Fig. 1. Since all resources have been exhausted, the analyzer (unsoundly) assumes that $\mathtt{c} \in [0, 1]$ at this program point, even though a fixed point has not yet been reached. We capture this unsound assumption by introducing an `assumed` statement at the loop header, as shown on line 4 of Fig. 2.

```
1  public void M() {
2    int c = 0;
3  LH:
4    assumed 0 <= c && c <= 1 as a;            // label 3
5    if (*) {
6      assume 0 <= c && c <= 1 provided a;     // label 4
7      if (c < 7) {
8        assume 0 <= c && c <= 1 provided a;   // label 5
9        c++;
10       assume 1 <= c && c <= 2 provided a;   // label 6
11     }
12     assume 1 <= c && c <= 2 provided a;     // label 7
13     goto LH;
14   }
15   assume 0 <= c && c <= 1 provided a;       // label 8
16   assume c < 585 provided a;
17   assert c < 585;                           // verified
18   assume 0 <= c && c <= 1 provided a;       // label 9
19   assume !(7 < c) provided a;
20   assert 7 < c;                             // falsified
21   assume 0 <= c && c <= 1 provided a;       // label 10
22 }
```

Fig. 2: Verification annotations (highlighted in gray) in method `M` for expressing the intermediate results achieved before performing the widening operation at program point 3 of Fig. 1.

To record intermediate, partial verification results, we introduce a new kind of verification annotations, namely, *partially-justified assumptions* of the form `assume` P `provided` A. These assumptions express that property P has been soundly derived (or inferred) by the analyzer provided that A holds. The condition A is a boolean expression over assumption identifiers, each of which is introduced in an `assumed` statement.

In Fig. 2, we use partially-justified `assume` statements to soundly record the verification results achieved by the analyzer within its bounds. At every program point, we introduce an `assume` statement to document what we have learned about variable `c` so far, also shown in the column for intermediate results of the table of Fig. 1. The properties of the `assume` statements are soundly inferred as long as explicit assumption `a` holds. Note that on line 16 we are also able to explicitly capture the fact that the first assertion in method `M` has been verified provided that `a` holds.

The concrete semantics of these verification annotations is defined in terms of assignments and standard `assume` statements, which are understood by most existing analyzers. For modular analyses, each assumption identifier corresponds to a local boolean variable, which is declared at the beginning of the enclosing method and initialized to true. A statement `assumed` P `as` a is encoded as:

$$a \texttt{ = } a \texttt{ \&\& } P\texttt{;}$$

This means that variable a accumulates the assumed properties for all executions of this statement. Note that this encoding ensures that any unsound assumption is captured at the program point at which it is made by the static analyzer, instead of where it is used.

A statement `assume` P `provided` A is encoded using a standard `assume` statement:

$$\texttt{assume}\ A \Rightarrow P\,;$$

This captures that the derived property P holds provided that the condition A holds, and it precisely reflects that P was soundly learned by the static analyzer under condition A. It also allows us to express that an analyzer found a program point to be unreachable if condition A holds. This can be achieved by instrumenting that program point with the following statement:

$$\texttt{assume false provided}\ A\,;$$

In some cases, an analyzer may even determine that an assertion P never holds at a given program point if condition A holds. We refer to this as a *definite warning*. This can be captured in the following way (line 19 of Fig. 2):

$$\texttt{assume}\ \neg P\ \texttt{provided}\ A\,;$$

3 Sound Intermediate Verification Results

Abstract interpretation uses a fixed-point computation to converge on its verification results. Therefore, verification results only become valid once a fixed point has been reached. As a consequence, abstract interpreters cannot simply be interrupted at any point during the analysis when user-provided resources run out. In this section, we demonstrate a novel technique for soundly sharing verification results at intermediate states during an analysis.

3.1 Analysis

As a first step, we present an algorithm for analyzing a program such that the analysis can be interrupted at any point when a certain bound is reached. We give concrete examples of such bounds in the next section, but for now, the reader may assume that we refer to a user's available time, tolerance for imprecision, or for spurious errors. On a high level, our algorithm, Alg. 1, uses abstract interpretation to analyze a program in the usual way until a bound is reached for a given program point. Now, the abstract state is *not* updated at this program point, and consequently, *no* change is propagated to subsequent program points, contrary to what is typically done. Instead, we only record that a bound is reached at the program point and continue the analysis without modifying the abstract state. As expected, this speeds up the fixed-point computation. Note that our algorithm is independent of a particular abstract domain.

As an example, consider an if-statement with a then- and an else-branch. Also, assume that, at a certain point in the analysis, the then-branch has already

been analyzed and the program point right after the if-statement holds the post-state of the then-branch. After analyzing the else-branch of the if-statement, the program point right after the if-statement should be updated by joining the existing state with the state at the end of the else-branch. Now, imagine that a bound is reached at this point after the if-statement. Instead of performing the join and updating the state, we simply record this program point and continue.

As a result, the imprecision that may have been introduced by joining the states of both branches is avoided. In exchange for not losing precision, the analysis becomes unsound. In particular, any code after the if-statement is analyzed as if the else-branch did not exist. This, on one hand, potentially reduces the number of spurious errors of the analysis but, on the other hand, might make the analysis miss errors. The benefit for a user with limited resources is that the analysis becomes more precise within the allocated resources. Moreover, by recording at which program points unsoundness may have been introduced due to a bound, we soundly capture the verification results using the annotations from Sect. 2, as we explain in Sect. 3.2. This instrumentation achieves two additional goals: (1) the bounded abstract interpreter still provides definite correctness guarantees, and (2) the analysis of the program can still continue from where it left off at a later point, when user resources become available again.

Let us now describe our algorithm in more technical detail. In Alg. 1, procedure Analyze takes as arguments the program p, l_{init}, which denotes the label or program point from which the analysis begins, and pre_{init}, which denotes the initial abstract state, that is, the pre-state of l_{init}. For example, if l_{init} is the first program point in a method body, then pre_{init} could refer to the state that expresses the precondition of the method.

On lines 2–5, we perform necessary initializations. Variable q is a work queue containing the labels that remain to be analyzed. Specifically, it contains tuples whose first element is a label and whose second element is a pre-state of that label. Initially, we push to q a tuple with l_{init} and its pre-state pre_{init}. Variable b is a set of labels for which a bound has been reached, and thus, whose state has not been updated as usual. Set b is initially empty. Variables *pre* and *post* denote maps from a label to its computed pre- and post-state, respectively. Initially, these maps are also empty.

The loop on lines 6–20 iterates until q is empty. While q is not empty, we pop a label and its pre-state from the queue (line 7) and check whether a bound has been reached for this label. This is determined by the generic procedure BoundReached (line 8), which we instantiate in Sect. 4 for different application scenarios. If a bound has indeed been reached for a label l, then we add this label to set b (line 9). Otherwise, we determine the pre-state for l (lines 11–16).

Specifically, if the *pre* map already contains a previous pre-state for label l (line 12) and widening is desired (line 13), we perform widening of the previously computed pre-state of l, $pre(l)$, and its current pre-state (line 14). This current pre-state comes from the work queue and has been computed by performing earlier steps of the analysis, as we show next. If widening is not desired, we instead perform a join of the two pre-states of l (line 16).

Alg. 1 Bounded analysis

```
1  procedure Analyze(p, l_init, pre_init)
2    q ← Push(⟨l_init, pre_init⟩, EmptyQueue())
3    b ← EmptySet()
4    pre ← EmptyMap()
5    post ← EmptyMap()
6    while ¬IsEmpty(q) do
7      ⟨l, pre_l⟩, q ← Pop(q)
8      if BoundReached(pre, post, l, pre_l) then
9        b ← Add(b, l)
10     else
11       pre'_l ← pre_l
12       if l ∈ dom(pre) then
13         if WideningDesired(l, pre, pre'_l) then
14           pre'_l ← ∇(pre(l), pre'_l)
15         else
16           pre'_l ← pre(l) ⊔ pre'_l
17       post_l ← Step(p, pre'_l, l)
18       post(l), pre(l) ← post_l, pre'_l
19       foreach s in Successors(p, l)
20         q ← Push(q, ⟨s, post_l⟩)
21   return pre, post, b
```

On line 17, our algorithm performs one step of analysis for label l, that is, we compute the post-state of l by executing an abstract transformer, and on line 18, we update the *pre* and *post* maps with the new pre- and post-states of l. For each successor s of l in program p (line 19), that is, for each program point immediately succeeding l, we push s and its new pre-state (line 20), $post_l$, to the work queue.

When the work queue becomes empty, procedure Analyze returns the information that has been learned during the analysis for each program point (stored in the *pre* and *post* maps). We also return the set b, which is used in Sect. 3.2 for soundly expressing the intermediate verification results of the analysis, that is, the verification results that were achieved within the available resources.

To illustrate, let us explain Alg. 1 on the example of Fig. 1. We assume that procedure BoundReached returns true whenever a widening operation would be used, in other words, the bound prevents any widening. This is the case at label 3 of method `M` (see graph of Fig. 1), right after the intermediate states shown in the table of Fig. 1 have been computed. As a result, label 3 is added to set b, and the interval state for variable `c` remains $[0, 1]$, which has earlier been stored to the *pre* and *post* maps as $pre(3)$ and $post(2)$, respectively.

Similarly to abstract interpretation, many static analysis techniques, such as data flow analysis, or predicate abstraction [21], also perform a fixed-point computation in order to converge on their verification results. Therefore, our algorithm could also be applied to such analysis techniques that may progressively lose precision on examples like that of Fig. 1.

3.2 Instrumentation

Alg. 2 shows how to instrument the program with intermediate verification results. In particular, procedure Instrument is run after Analyze to record, in the form of the verification annotations presented in Sect. 2, the information that has soundly been learned by the analysis at each program point.

On a high level, our algorithm concretizes the abstract states that have been computed for each program point by procedure Analyze and, after each point, inserts either an `assume` statement or an `assumed` statement, if a bound has been reached for the corresponding program point. Recall that an `assumed` statement denotes that at the corresponding program point the abstract interpreter unsoundly assumed that a property P holds for all possible program states, even though P actually holds only for those states that were explored by the analyzer within its limited resources. An `assume` statement expresses properties that have soundly been derived by the analyzer provided that an (unsound) assumption made by the analysis holds.

Procedure Instrument takes as arguments the program p, the map *post* from a label to its computed post-state, and the set b of labels for which a bound has been reached. Note that *post* and b are computed and returned by Analyze. For each label l in program p (line 2), we first concretize the bottom state and assign it to $post_l$ (line 3), which denotes the concrete post-state of l. If the *post* map contains an entry for l (line 4), then we update $post_l$ with that entry (line 5). Finally, we check if a bound was reached at label l (line 6). In this case, we instrument the program to add an `assumed` statement for property $post_l$ right after program point l (line 7). We use label l as the unique assumption identifier in the `assumed` statement. In particular, the instrumentation added after program point l looks as follows:

`assumed` $post_l$ `as` l;

If label l is not in set b (line 8), we add an `assume` statement for property $post_l$, whose `provided` clause is a conjunction of all assumption identifiers (or labels) in set b (line 9). Assuming that set b contains n labels, the instrumentation in this case looks as follows:

`assume` $post_l$ `provided` $l_0 \wedge l_1 \wedge ... \wedge l_{n-1}$;

Alg. 2 Instrumentation of sound intermediate verification results

```
1 procedure Instrument(p, post, b)
2   foreach l in Labels(p)
3     post_l ← Concretize(⊥)
4     if l ∈ dom(post) then
5       post_l ← Concretize(post(l))
6     if l ∈ b then
7       InsertAssumed(p, l, post_l)
8     else
9       InsertAssume(p, l, post_l, b)
```

This statement means that $post_l$ has been soundly learned by the abstract interpreter provided that the unsound assumptions made by the analysis at program points $l_0, ..., l_{n-1}$ hold. With this instrumentation, a partially-justified assumption might depend on identifiers of explicit assumptions made later in the code. However, note that the boolean variables corresponding to these identifiers are trivially true at the point of the partially-justified assumption due to their initialization (see Sect. 2).

Recall that for label 3 of method `M`, in Fig. 1, a bound was reached for the widening operation. Therefore, according to Alg. 2, we introduce an `assumed` statement at that program point, as shown in Fig. 2. At all other program points, we insert an `assume` statement, as shown in the same figure. Note that the properties of these statements are the concretized, intermediate abstract states for variable `c`, which are shown in the table of Fig. 1. Also note that the `provided` clauses of the `assume` statements correspond to the unsound assumption made at program point 3. We added line 16 of Fig. 2 to show that the subsequent assertion is verified under assumption `a`. Similarly, we added line 20 of Fig. 2 to show that the subsequent assertion is found to never hold.

For simplicity, we record intermediate verification results for each program point in the code. However, an optimization could remove any `assume` statements that only contribute information that can easily be derived from the remaining ones.

Soundness argument. In standard abstract interpretation, the inferred properties for labels that are not in the work queue are sound, given that the inferred properties for labels that are still in the queue hold. In bounded abstract interpretation, the inferred properties for labels that are not in the work queue are sound, given that the inferred properties for labels that are still in the queue *or in the set of labels for which a bound has been reached* hold. Our verification annotations precisely express these soundness guarantees since the work queue eventually ends up empty.

4 Applications

In this section, we discuss examples of user resources and the corresponding bounds that may be imposed during the analysis. In particular, we describe possible instantiations of procedure BoundReached from Alg. 1 and an application scenario for each such instantiation.

4.1 Bounding time

As a first example, consider running the computationally expensive polyhedra abstract domain [12]. In comparison to other simpler domains, it typically takes longer for the verification results of this domain to reach a fixed point. Simply imposing a timeout on such expensive analyses does not solve the problem for engineers who are often not willing to wait for long-running analyses to terminate. In case a timeout is hit before a fixed point is reached, all intermediate verification results are lost.

The implementation of procedure BoundReached for this scenario is very simple. For BoundReached to be deterministic, there needs to be a symbolic notion of time, instead of actual time that varies from machine to machine. Each symbolic time tick could, for example, refer to a step of the analysis in Alg. 1 (line 17). For instance, this can be implemented using a shared counter of symbolic ticks, which is updated in procedure Step, and a shared bound on the total number of ticks allowed to happen in the analysis, in other words, a symbolic timeout selected by the user. Note that, when reaching this bound, the number of explicit assumptions in the instrumentation is usually small as it is bounded by the size of queue q in Alg. 1 at the point of the timeout.

4.2 Bounding imprecision

We now consider as our limited resource a user's tolerance for imprecision. Specifically, three common sources of imprecision in abstract interpretation are joins, widenings, and calls. The motivation behind having a threshold on the amount of imprecision is the following: the analysis keeps accumulating imprecision while it runs, and after a certain point, many reported warnings are spurious and caused solely by the accumulated imprecision. Note that, for simplicity, we do not consider all possible sources of imprecision since accurately identifying them is an orthogonal issue; imprecision of the analysis generally depends on both the abstract domain and the analyzed program, and existing work [18, 17] could be used to identify possible imprecision sources with more accuracy.

A join operation approximates the union of two sets of concrete program states in the abstract domain. In particular, it is often not possible to precisely express the union in a given abstract domain, thus, resulting in over-approximation. By bounding the number of joins, we are able to control the amount of imprecision that is caused by an abstract domain's inability to accurately express union (or disjunction) of states even though the join operation itself is known to be complete [18, 17].

Alg. 3 shows an implementation of BoundReached that counts the number of imprecise joins. On line 2, we check whether label l is in the domain of the pre map (storing the current pre-state for each label) and whether a join is desired. If this is the case, we compute the join of the current pre-state $pre(l)$ with the pre-state pre_l from the work queue on line 3. If its result pre'_l, which may become the current pre-state for label l, is less than (not possible due to property of join) or equal to $pre(l)$, the join does not lead to additional imprecision and we do not increase the global counter $joins$ for imprecise joins. Otherwise, we do (line 5). Note that, in this algorithm, we conservatively consider any join that generates a different result than $pre(l)$ as imprecise. This may include joins that are actually precise, but efficiently checking for imprecise joins is usually not possible within an abstract domain and the overhead would defeat the purpose. On line 6, we return whether the number of imprecise joins has exceeded the bound.

Similarly to joins, a widening operation is a source of imprecision as its result purposefully over-approximates its arguments in order to reach a fixed point more efficiently. The instantiation of procedure BoundReached in this

Alg. 3 Bounded imprecision due to joins

```
1 procedure BoundReached(pre, post, l, pre_l)
2   if l ∈ dom(pre) ∧ ¬WideningDesired(l, pre, pre_l) then
3     pre'_l ← pre(l) ⊔ pre_l
4     if ¬(pre'_l ⊑ pre(l)) then
5       joins ← joins + 1
6       return maxJoins < joins
7   return false
```

case is similar to the one for joins in Alg. 3. If widening is desired, we increase the global counter for imprecise widenings only if the result of performing the widening is different from the current pre-state in the map pre.

In modular abstract interpretation, calls also constitute a source of imprecision as the relation between a method's arguments and its return values is unknown. This relation is, in the best case, described by user-provided specifications, which however might not be precise enough to avoid some over-approximation in the analysis. To bound the amount of imprecision due to calls, BoundReached can be implemented such that the analysis simply terminates after analyzing a limited number of calls.

In addition to joins, widenings, and calls, our technique may be applied to any source of imprecision in a static analysis. For instance, we could bound the imprecision of certain abstract transformers, such as standard `assume` statements. Let us explain this concept through the following concrete example.

Consider an if-statement whose condition checks whether variable x is non-zero. Now, imagine that we are using an interval domain to learn the possible values of x. Right before the if-statement, we have inferred that $x \in [-1, 1]$. In the then-branch, even though we know that x cannot be zero (due to the condition of the if-statement), the analysis still derives that $x \in [-1, 1]$ since there is no way to express that x is either -1 or 1 in the interval domain. Therefore, assuming the condition of the if-statement in the then-branch is imprecise due to lack of expressiveness in this abstract domain. In such cases, our technique can restrict the imprecision that is introduced in the analysis because of such transformers by bounding their number, just like we do for calls.

More generally, one could imagine a slight generalization of Alg. 1 that, in such cases, would execute an under-approximating abstract transformer and add the corresponding label to the set b of labels that have not been fully explored.

Note that, since we are making the analyzer aware of its sources of imprecision, our technique may also be used for ranking any warnings emitted by the analyzer. For example, warnings emitted when introducing fewer sources of imprecision in the analysis could be shown to the user first, before warnings that are emitted when more sources of imprecision may have affected the analysis.

4.3 Bounding spurious errors

The third scenario refers to bounding the number of spurious errors that are reported to the user. This acknowledges the fact that human time, required to

```
1 public void N() {
2   int c = 0;
3   if (*) {
4     while (c < 7) {
5       c++;
6     }
7   }
8   assert c != 5;
9 }
```

Fig. 3: Example for bounding spurious errors.

identify spurious errors, is often more precious than analysis time. On the other hand, it is not always practical to use the most expensive and precise analysis for all code. To strike a good balance, we propose to only run a more precise analysis or one with higher bounds if the less expensive analysis will emit warnings. Before this switch happens, we record the verification results that have been collected until that point by the less expensive analysis so that they are not lost.

Let us consider the example in Fig. 3. When analyzing method `N` with the interval domain, we reach a point right before applying widening where the assertion holds. However, after widening, $\mathtt{c} \in [0, \infty]$ is inferred right before the assertion and an error is reported. To avoid this spurious error, we can interrupt the analysis and record the results before the widening by marking the assertion as verified under an explicit assumption in the loop header, like in Fig. 2.

A second analysis can now take over. It could be the same analysis as before, an analysis with a more precise abstract domain, or one with higher bounds. It could even be dynamic test generation [5, 8]. This second analysis can benefit from the existing verification results since the assertion has already been fully verified for the else-branch (the explicit assumption always holds on that branch). By applying a very inexpensive instrumentation from our previous work [5, 8], we can prune away the fully-verified else-branch. This allows even an analysis of the same precision to verify the resulting instrumented program. In particular, $\mathtt{c} \in [7, \infty]$ is derived right before the assertion, and method `N` is fully verified.

In other words, a subsequent analysis (static or dynamic) can analyze the instrumented program that is produced after running the previous analysis. As demonstrated in this section, the encoded intermediate results make it possible to ignore certain parts of the program state or even entire program paths. Our previous work has specifically shown that this instrumentation is suitable for subsequent analyses, in the context of dynamic test generation [6, 8] and deductive verification [25].

5 Experimental Evaluation

Setup. We have implemented our technique and algorithms in the widely-used, commercial static analyzer Clousot [16], an abstract interpretation tool for .NET and Code Contracts [15].

	Joins	Widenings	Calls
Potential imprecisions	199,354 (55.6%)	3,251 (0.9%)	155,990 (43.5%)

Tab. 1: The number of potential imprecisions in the analysis of 11,990 methods (when no bounds are imposed), categorized as joins, widenings, and calls. Overall, 3,378,692 abstract transformers were executed during the analysis.

For our experiments, we selected a large, popular, open-source C# project on GitHub, namely MSBuild, which is a platform for building applications and can be found at `https://github.com/Microsoft/msbuild`. The MSBuild project consists of five components, MSBuild, Microsoft.Build, Microsoft.Build.Framework, Microsoft.Build.Tasks, and Microsoft.Build.Utilities. We selected a project that has not been annotated with contracts in order not to depend on the quality of user-provided annotations.

We ran our version of Clousot, which we call *ClousotLight*, on one core assembly pertaining to each of MSBuild's components (in the `bb10a5c` changeset), `MSBuild.exe`, `Microsoft.Build.dll`, `Microsoft.Build.Framework.dll`, `Microsoft.Build.Tasks.Core.dll`, `Microsoft.Build.Utilities.Core.dll`. For our evaluation, we used Clousot's default settings, in addition to the configurations of ClousotLight that we describe in the rest of this section. Note that we did not use Clousot's contract inference in order to isolate the effect of bounds within individual methods; in practice however, contract inference can be useful as a complementary approach for reducing imprecision due to calls. We performed all experiments on a 64-bit machine with an Intel Xeon processor at 3.50GHz and 32GB of RAM.

Experiments. We used ClousotLight to analyze 11,990 methods, which are all the methods defined in the five core assemblies of MSBuild.

Tab. 1 summarizes the number of potential imprecisions in the analysis of these methods (when no bounds are imposed), categorized as joins, widenings, and calls. We observed that joins and calls account for the majority (99.1%) of all these possible sources of imprecision. In total, the analysis executed 3,378,692 abstract transformers (that is, calls to procedure Step from Alg. 1). Note that we only counted joins and widenings that result in an update of the program state. For calls, we only counted those that modify any heap location or have a return value. In other words, we did not count joins, widenings, and calls that definitely do not contribute any imprecision to the analysis.

For all experiments that we describe in the rest of this section, we used 15 different configurations of ClousotLight, namely JWC0, J0, J1, J2, J4, J8, W0, W1, W2, C0, C1, C2, C4, C8, and Inf. The configuration names denote the bounds that we imposed on the analysis. For instance, J0 indicates that we allow up to zero joins per method, W1 indicates that we allow up to one widening per method, and C2 that we allow up to two calls per method. JWC0 means that we do not allow any joins, widenings, or calls, and Inf that we do not impose any

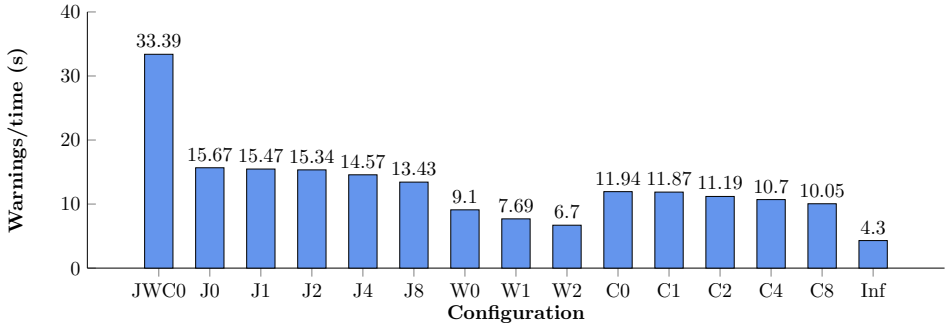

Fig. 4: The average number of warnings in 11,990 methods that were emitted per second of the analysis, for each configuration.

bounds on the analysis (as in Tab. 1). These configurations emerged by choosing different bounds for each separate source of imprecision (e.g., J0, W1, C2) as well as the smallest bound for every source of imprecision (JWC0).

Fig. 4 shows, for each configuration, the average number of warnings that were emitted per second of the analysis. Note that the fastest configuration is JWC0, with approximately 8x more warnings per second over Inf. The slowest configuration is W2, with 1.5x more warnings per second over Inf.

Tab. 2 shows the exact number of warnings (second column) emitted during the duration of the analysis (third column), for each configuration (first

Configuration	Warnings	Time (s)
JWC0	5,304	159
J0	11,777	752
J1	11,913	770
J2	12,008	783
J4	12,137	833
J8	12,240	911
W0	12,466	1,370
W1	12,467	1,621
W2	12,473	1,861
C0	5,214	437
C1	5,214	439
C2	7,195	643
C4	8,403	785
C8	9,683	963
Inf	12,480	2,902

Tab. 2: The number of warnings (second column) emitted during the duration of the analysis in seconds (third column), for each configuration (first column).

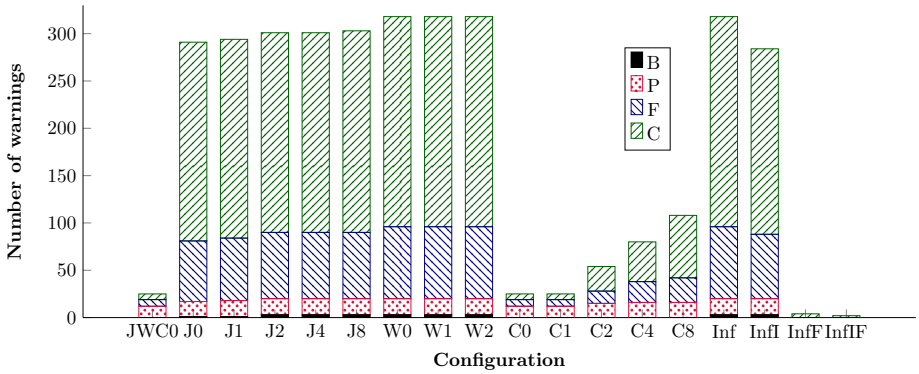

Fig. 5: Manual classification of warnings. We performed this classification for the ten methods that had the largest difference in the number of emitted warnings between configurations JWC0 and Inf. The warnings are classified into four categories: (1) B for genuine, confirmed bugs in the code of the method, (2) P for warnings that depend on the values of the method's parameters, (3) F for warnings that depend on the values of fields in the current object, and (4) C for warnings that depend on what a call modifies, including its return values.

column). As shown in the table, configuration Inf generates 12,480 warnings in approximately 48 minutes. Configuration J0 generates approximately 94% of these warnings in around 13 minutes. Configurations J8 and W0 generate almost all warnings in around 15 and 23 minutes, respectively.

We evaluate the impact of our technique on the number of genuine and spurious errors by manually classifying more than 300 warnings that were emitted by ClousotLight (see Fig. 5). We performed this classification for the ten methods (each consisting of hundreds of lines of code) that had the largest difference in the number of emitted warnings between configurations JWC0 and Inf. Since in a modular setting it is difficult to tell which warnings are spurious, we classify the warnings into four categories: (1) B for genuine, confirmed bugs in the code of the method, (2) P for warnings that depend on the values of the method's parameters and could be fixed by adding a precondition, (3) F for warnings that depend on the values of fields in the current object and could be fixed by adding a precondition or an object invariant, and (4) C for warnings that depend on what a call modifies, including its return values, and could be fixed by adding a postcondition in the callee.

We defined these particular categories as they provide an indication of the severity of a generated warning for the author of a method. For example, the author of a method should definitely fix any bugs in its code (B), and is likely to resolve issues due to the method's parameters (P). We consider B and P high-severity warnings. The author of a method, however, is less inclined to address

warnings caused by the fields of the current object (F), which would affect the entire class, let alone, add postconditions to callees that may have been written by someone else (C). We consider F and C low-severity warnings.

As shown in Fig. 5, we found genuine bugs in MSBuild, three null-dereferences in particular, which we have reported to the project's developers and have now been fixed (issue #452 on GitHub). All three bugs are found by J2, J4, J8, the W configurations, and Inf. Configurations J0 and J1 detect only one of these errors, which happens to share its root cause with the other two. In other words, by addressing this one warning, all three errors could be fixed. Moreover, J0 and J1 report the warning as a definite one, that is, the analyzer proves that there is a null-dereference, which is made explicit by our annotations (see Sect. 2).

In general, the J configurations miss few warnings of high severity (between 15% for J0 and 0% for J8), as shown in the figure. The W configurations find exactly the same warnings as Inf. The C configurations perform very aggressive under-approximation of the state space and, consequently, miss all genuine bugs. This could be remedied by a less coarse under-approximation for calls.

As we observed in Fig. 5, the J and W configurations report a number of warnings very close to that reported by Inf. However, this is achieved with a significant speedup of the analysis. To further reduce the number of reported warnings in one run of a single analysis, a user may turn on orthogonal techniques, such as contract inference or filtering/ranking. For comparison, we include three variants of the Inf configuration with nearly identical running times: InfI (with Clousot's contract inference enabled), InfF (with Clousot's low-noise filtering enabled), and InfIF (with both of these options enabled). We observe that, by enabling Clousot's noise filtering, all high-priority warnings are missed and very few low-priority warnings are reported. We believe that a very fast configuration such as JWC0 strikes a much better balance as it reports 60% of the high-priority warnings after a much shorter running time. We confirm that contract inference does help in reducing low-priority warnings noticeably. However, this complementary technique shows similar benefits when used with configurations that perform bounding and also benefits from their speedup.

To further evaluate whether our technique improves the detection of genuine bugs, we reviewed all definite warnings that were reported for MSBuild by configurations Inf, J0, and JWC0. Recall that the term "definite warnings" refers to messages that warn about an error that will definitely occur according to the analysis, instead of an error that *may* occur. Clousot prioritizes such warnings when reporting them to users. Inf reported 25 definite warnings, none of which corresponded to genuine bugs. JWC0 reported 15 definite warnings and detected one genuine bug, while J0 reported 44 definite warnings and detected two different and independent genuine bugs (including one of the three bugs from the manual classification). We have reported these bugs, which have been confirmed by the developers (issues #569 and #574 on GitHub). One of them has already been fixed, and the other will be addressed by deleting the buggy code.

This experiment suggests that a bounded analysis can significantly increase the number of definite warnings, which should be reported to the user first, as

well as the number of genuine bugs detected by these warnings. For J0, the number of these warnings is increased by almost 2x. The aggressive JWC0 reports fewer definite warnings than Inf, however, a genuine bug is still detected by these warnings, whereas for Inf, it is missed.

Discussion. The experiments we have presented in this section demonstrate two beneficial aspects of bounded abstract interpretation: (1) it increases the performance of the analysis (by up to 8x) by interactively ignoring certain parts of the state space, and (2) when bounding imprecision, it can improve the quality of the reported warnings, either by increasing the number of definite warnings or the number of genuine bugs detected by these warnings, while only missing few high-severity warnings.

We have evaluated these aspects of bounded abstract interpretation in a modular analyzer, which is already highly scalable partly due to its modularity. We expect that the impact of our technique on the scalability of whole-program analysis should be even greater. Moreover, modular analysis is typically more prone to reporting spurious errors, which our technique alleviates with the second aspect that we described above.

For our experiments, we did not evaluate the first and third application scenarios, about bounding time and spurious errors, respectively. Bounding the time of the analysis is easy and very useful in practice, however, the analysis results are too unpredictable for us to draw any meaningful conclusions, for instance about the number of genuine bugs. We did not evaluate how the collaboration of multiple analyses helps reduce the number of spurious errors as we have experimented with several such scenarios in our previous work [6, 8, 5, 30].

6 Related Work

Model checking. Bounded model checking [4, 9] is an established technique for detecting errors in specifications of finite state machines by only considering a bounded number of state transitions. This core idea has been extended in numerous tools, such as CBMC [10], for bug-finding in infinite state programs. In contrast, bounded abstract interpretation is based on a sound verification technique for infinite state programs [11]. It provides a way to terminate the analysis based on a chosen bound (e.g., the maximum number of joins) and to soundly capture the verification results as a program instrumentation, which can be understood by a wide range of other analyses. Furthermore, an advantage of our technique is that it can analyze an infinite number of paths even before a bound is hit.

Conditional model checking (CMC) [2, 3] combines different model checkers to improve overall performance and state-space coverage. In particular, CMC aims at encoding verification results of model checkers right before a spaceout or timeout is hit, which are typical limitations of model checking techniques. A complementary model checker may then be used to continue the analysis from where the previous tool left off. A conditional model checker takes as input a program and its specification as well as a condition that describes the states that

have previously been verified. It produces another such condition to encode the new verification results.

There are the following significant differences between CMC and our technique. CMC focuses on alleviating the effect of exhausting physical bounds, such as time or memory consumption. Our notion of bounds is more flexible and driven primarily by the user's resources. Moreover, the verification annotations that we use for representing partial static analysis results are suitable for modular analyses and relatively compact (there typically is a constant instrumentation overhead in terms of the original program size). Also, our annotations are universally understood by most program analyzers, as opposed to the verification condition of CMC.

Symbolic execution. Our technique features the following significant differences over approaches based on symbolic execution [23]: (1) it produces sound results, (2) reasoning over abstract domains is typically more efficient than reasoning about logical domains (e.g., using SMT solvers), and (3) by relying on abstract interpretation, it inherently avoids path explosion and is more scalable.

Angelic verification. There are two sources of spurious errors in static analysis: those that are generated in the presence of an unconstrained environment, and those that are caused by an imprecision of the analysis itself. Angelic verification [13] aims at constraining a verifier to report warnings only when no acceptable environment specification exists to prove a given property. In other words, angelic verification reduces the number of spurious errors generated due to an unconstrained environment, whereas our technique can address both sources of spurious errors.

Unsoundness. Our previous work on documenting any unsoundness in a static analyzer [6–8, 5] focuses on making explicit fixed and deliberate unsound assumptions of the analysis, for example, that arithmetic overflow can never occur. Such *fixed* assumptions are used in most practical static analyzers and are deliberately introduced *by designers* (e.g., to reduce spurious errors). In contrast, this work focuses on providing users with a flexible way to interactively bound the analysis while soundly recording intermediate results such that the analysis can be resumed later on. Unsound assumptions are encountered *dynamically* during the analysis based on the *user-provided* bounds.

Ranking of warnings. Work on ranking analysis warnings, for instance by using statistical measures [24], is orthogonal to ours. Such techniques typically constitute a post-analysis step that is not aware of the internal imprecision of the abstract interpreter, even though there is some work that considers sources of imprecision encountered during the analysis [22]. Instead of performing a post-analysis, we rely on bounding the analysis itself to suppress warnings.

Work on differential static analysis [1] and verification modulo versions [26] suppresses warnings based on another program or program version. Our work, on the other hand, does not require such a reference program or version.

An existing approach [17] makes it possible to prove completeness of an analysis for a given program. This provides a way to rank higher warnings for which the analysis is shown to be complete. Unlike our approach, it requires a

dedicated proof system and input from the designer of the analysis, which makes it more difficult to apply in practice.

In addition, there are orthogonal approaches for controlling imprecision (e.g., due to joins [27], widenings [19, 20], and generally incompleteness of abstract domains [14]) of an existing analysis. In contrast to our work, these approaches focus on refining the analysis to reduce imprecision while still reaching a fixed-point (usually at the cost of performance). The scenario described in Sect. 4.2 is just one possible application of our approach for bounding an analysis based on user resources. Even with approaches that refine the analysis, it can make sense to use our approach since they inherently cannot avoid every imprecision.

7 Conclusion

We have presented a technique for imposing different bounds on a static analysis based on abstract interpretation. Although such bounds make the analysis unsound, we are able to express its verification results in a sound way by instrumenting the program. This opens up a new way for building highly tunable (with respect to precision and performance) analyzers by soundly sharing intermediate analysis results early on. In our experiments, we evaluate several such bounds in an analyzer and show that the analysis time can be significantly reduced while increasing the quality of the reported warnings. The trade-off between analysis time and unsoundness is also beneficial in determining when in the workflow of a software engineer a static analysis should run and with which bounds.

Acknowledgments. We thank Rainer Sigwald for promptly confirming and helping us resolve the MSBuild bugs as well as the anonymous reviewers for their constructive feedback.

References

1. T. Ball, B. Hackett, S. K. Lahiri, S. Qadeer, and J. Vanegue. Towards scalable modular checking of user-defined properties. In *VSTTE*, volume 6217 of *LNCS*, pages 1–24. Springer, 2010.
2. D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. Conditional model checking. *CoRR*, abs/1109.6926, 2011.
3. D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. Conditional model checking: A technique to pass information between verifiers. In *FSE*, pages 57–67. ACM, 2012.
4. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.
5. M. Christakis. *Narrowing the Gap between Verification and Systematic Testing*. PhD thesis, ETH Zurich, 2015.
6. M. Christakis, P. Müller, and V. Wüstholz. Collaborative verification and testing with explicit assumptions. In *FM*, volume 7436 of *LNCS*, pages 132–146. Springer, 2012.
7. M. Christakis, P. Müller, and V. Wüstholz. An experimental evaluation of deliberate unsoundness in a static program analyzer. In *VMCAI*, volume 8931 of *LNCS*, pages 336–354. Springer, 2015.

8. M. Christakis, P. Müller, and V. Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *ICSE*. ACM, 2016. To appear.
9. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *FMSD*, 19:7–34, 2001.
10. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.
11. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.
12. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM, 1978.
13. A. Das, S. K. Lahiri, A. Lal, and Y. Li. Angelic verification: Precise verification modulo unknowns. In *CAV*, volume 9206 of *LNCS*, pages 324–342. Springer, 2015.
14. V. D'Silva, L. Haller, and D. Kroening. Abstract conflict driven learning. In *POPL*, pages 143–154. ACM, 2013.
15. M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *SAC*, pages 2103–2110. ACM, 2010.
16. M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, volume 6528 of *LNCS*, pages 10–30. Springer, 2010.
17. R. Giacobazzi, F. Logozzo, and F. Ranzato. Analyzing program analyses. In *POPL*, pages 261–273. ACM, 2015.
18. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47:361–416, 2000.
19. D. Gopan and T. W. Reps. Lookahead widening. In *CAV*, volume 4144 of *LNCS*, pages 452–466. Springer, 2006.
20. D. Gopan and T. W. Reps. Guided static analysis. In *SAS*, volume 4634 of *LNCS*, pages 349–365. Springer, 2007.
21. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.
22. Y. Jung, J. Kim, J. Shin, and K. Yi. Taming false alarms from a domain-unaware C analyzer by a Bayesian statistical post analysis. In *SAS*, volume 3672 of *LNCS*, pages 203–217. Springer, 2005.
23. J. C. King. Symbolic execution and program testing. *CACM*, 19:385–394, 1976.
24. T. Kremenek and D. R. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *SAS*, volume 2694 of *LNCS*, pages 295–315. Springer, 2003.
25. K. R. M. Leino and V. Wüstholz. Fine-grained caching of verification results. In *CAV*, volume 9206 of *LNCS*, pages 380–397. Springer, 2015.
26. F. Logozzo, S. K. Lahiri, M. Fähndrich, and S. Blackshear. Verification modulo versions: Towards usable verification. In *PLDI*, pages 294–304. ACM, 2014.
27. L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP*, volume 3444 of *LNCS*, pages 5–20. Springer, 2005.
28. L. Nguyen Quang Do, K. Ali, E. Bodden, and B. Livshits. Toward a just-in-time static analysis. Technical Report TUD–CS–2015–1167, Technische Universität Darmstadt, 2015.
29. C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *ICSE*, pages 598–608. IEEE Computer Society, 2015.
30. V. Wüstholz. *Partial Verification Results*. PhD thesis, ETH Zurich, 2015.