ExplainHoudini: Making Houdini inference transparent

Shuvendu K. Lahiri[1] and Julien Vanegue[2]

[1] Microsoft Research
shuvendu@microsoft.com
[2] Microsoft Corporation
jvanegue@microsoft.com

Abstract. Houdini is a simple yet scalable technique for annotation inference for modular contract checking. The input to Houdini is a set of candidate annotations, and the output is a consistent subset of these candidates. Since this technique is most useful as an annotation assistant for user-guided refinement of annotations, understanding the reason for the removal of annotations is crucial for a user to refine the set of annotations, and classify false errors easily. This is especially true for applying Houdini to large legacy modules with thousands of procedures and deep call chains. In this work we present a method *ExplainHoudini* that explains the reason why a given candidate was removed, purely in terms of the existing candidates. We have implemented this algorithm and provide preliminary experience of applying it on large modules.

1 Introduction

Static analysis aims to check a property on a program with high coverage without executing the code. However, automatic static analysis of most non-trivial properties on real programs is undecidable. This fact either manifests in spurious warnings (false alarms) or non-termination of static analysis tools. For example, tools based on abstract interpretation [5] may generate false alarms when the intermediate invariants cannot be captured in the underlying abstract domains (which are mostly fixed); on the other hand, extensible tools based on predicate abstraction [10] or interpolants [15] may diverge while performing automated refinement of abstractions.

User-specified intermediate contracts or annotations (preconditions, postconditions etc.) are helpful in minimizing the false alarms for static analysis. In recent years, a number of contract-checking tools for real programs have been developed including `ESC/Java` [9] (for Java), `Spec#` [3] (for C#), `HAVOC` [4], and `VCC` [6](for C). These tools can be used to check a range of properties, starting from the absence of runtime defects (such as absence of null dereferences), to simple object type-state properties (such as a lock is acquired before a release) to more complex functional correctness properties. The research problems for these tools vary depending on the type of properties being checked. For example, the main challenges in tools proving functional correctness (mainly in the

hands of verification experts) are in devising efficiently-checkable logics for expressing data invariants, and exploiting data encapsulation for reasoning about composition of multiple modules [3, 6]. On the other hand, the main hurdle for (extended static) checking simple user-defined properties lies in the annotation overhead required for a (non-expert) user to check a property with high coverage and low false alarm [9, 1]. This work concerns the latter.

In the extended static checking [9, 1] scenario for large code bases, the user drives the verification process. The user first adds the property to be checked using procedure contracts and instrumented assertions. The next steps alternate between adding a few core annotations and directing an annotation inference assistant to generate a set of simple intermediate annotations. A desirable property of the inference assistant is to be *configurable* by an average user, be *scalable* to large codebases, and provide *transparency* in providing feedback about the inference.

HOUDINI [8] is a simple yet scalable annotation inference technique that meets most of these requirements. The input to Houdini is a set of candidate annotations (or "guesses"), and the output is a consistent subset of these candidates. It is a greatest fixed point algorithm that removes candidates that cannot be proved by modular checking. It is *configurable* by allowing the user to specify a set of guesses using a set of simple patterns. For example, a user may add a candidate precondition for any procedure that all pointer arguments are non-null [8], or that the lock parameters are released [1], or even guess exceptions to module invariants [13]. The underlying analysis uses a theorem prover and does not require defining new abstract semantics. It is *scalable* as the inference converges linearly in the number of candidates; and often much earlier if only a small number of candidates are removed[3]. It also uses procedure modular checking that does not involve inspecting long interprocedural counterexamples. It is *transparent* to a great extent as the user can inspect the set of annotations that were inferred.

However, when applying HOUDINI on large modules, we have spent most of the effort in understanding the reason a candidate annotation is removed [13, 1]. There are several reasons a candidate could be removed: (i) it does not hold, (ii) the existing annotations are not strong enough, or (iii) prover incompleteness. Understanding the reason is crucial for classifying false alarms from true bugs, and refining the set of annotations. In fact, the authors of the original article of HOUDINI rightly mention [8]:

> *Surprisingly, our experience indicates that presenting the refuted annotations and the causes thereof is the most important aspect of the user interface.*

To address this partially, the tool displayed the call-site where a candidate precondition was removed, and possibly an intraprocedural path that lead to the

[3] In contrast to a least fixed point based forward predicate abstraction approach that can solve the same problem [10].

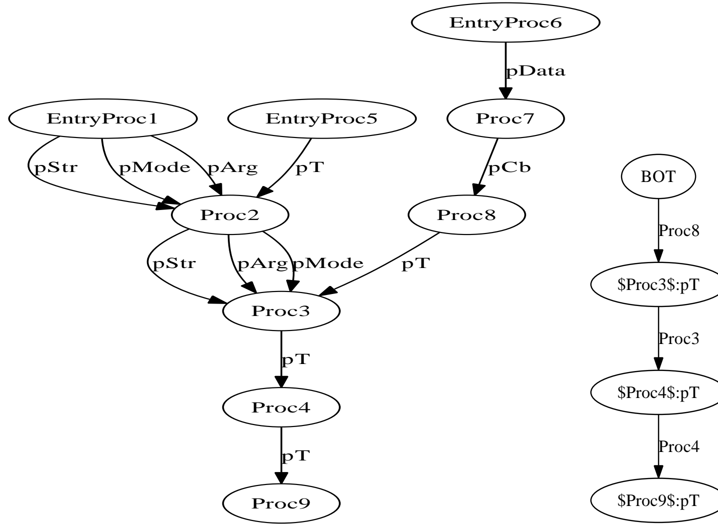

Fig. 1. (a) Dependency of candidates removed for `Proc9`. (b) Output of ExplainHoudini for the candidates removed for `Proc9`.

removal. Although this is useful, it provides very little interprocedural information to understand the cause of the removal.

Let us illustrate the problem with the graph in Figure 1 (a). The graph on the left represents a subset of candidates removed when applying the Houdini implementation in `HAVOC` for annotation inference on a large Windows module measuring more than 250KLOC, containing more than 3000 procedures. The candidates correspond to preconditions that pointer parameters are *trusted*; that is, the pointer was either allocated inside the kernel, or had undergone a sanitization by calling one of the designated procedures [1]. The graph represents a small slice of the overall graph of candidates removed — the slice contains candidates removed from the procedure `Proc9` and its callers.[4] Each node in the graph corresponds to a procedure, and an edge $(\mathtt{p}, c, \mathtt{q})$ represents that the candidate c of procedure `q` was removed while checking the body of `p`. Instead of showing the entire annotation on the edges (which can be verbose), we just show the parameter contained in the candidate.

Although the graph shows the local cause such as the call-site where a precondition was removed, it provides little interprocedural information. For instance, it is not clear if the candidate for `pT` parameter of the method `Proc9` was removed because there is a pointer flowing from one of the roots (say one of the parameters `pMode` of `EntryProc1`) in this graph without sanitization (true bug),

[4] The names of the procedures and the parameters have been obfuscated due to security reasons.

or whether there is a missing precondition or postcondition along the path that would have ensured that the candidate was provable (false alarm). Initial attempt at tracing the call chains from a removed candidate back to the roots of the graph manually was extremely time consuming. Matters get worse when candidate postconditions are involved; one not only has to look at all the transitive callers of a given procedure, but all the callees of these transitive callers as well, resulting in huge graphs. For a module of this size, one can end up spending a few minutes to sometimes several hours trying to understand the root cause of an annotation removal to certify true bugs.

In this work, we provide a step towards making HOUDINI more transparent by providing a more meaningful interprocedural reason of the annotation removal in addition to the above graph. Our goal is much less modest compared to the work on automatic *refinement* (e.g. [11]) to suggest new annotations, or even suggest which procedures to refine. For a run of HOUDINI, we provide a notion of *cause* for the removal of a candidate in terms of another removed candidate whenever possible (or $\perp$ otherwise). These causes can be composed to provide an interprocedural path from $\perp$ to a candidate that was removed; the path captures a sequence of candidate removal ultimately leading to the removal of a candidate. It is an interprocedural explanation of the removal *purely in terms of the candidates removed*. We provide an algorithm EXPLAINHOUDINI, which computes the cause for each removed annotation by adding a simple "replay" mechanism to any HOUDINI implementation. The algorithm is extremely simple (does not require any more functionality from the theorem prover than what HOUDINI requires), modular (does not require analyzing large interprocedural traces that may be infeasible for large modules), and does not require changing the search heuristics in HOUDINI.

Figure 1 (b) displays the output of EXPLAINHOUDINI on the removed candidate for `Proc9`. Here, each node represents a candidate (prefixed with procedure name) and the edge denotes the procedure whose checking removed the annotation. The node `BOT` marks the $\perp$ node. An edge from c' to c denotes that the cause of c is c'. The interprocedural dependency graph suggests that the candidate precondition on the `pT` parameter of `Proc9` cannot be removed until the candidate on `pT` parameter of `Proc3` was removed. However, the removal of this candidate on `pT` parameter from `Proc3` cannot be explained in terms of the existing candidates. The information narrows down the choice of places to inspect to understand the false alarm. The information strongly suggests that the pointer from the root does not flow into `Proc9`, and the false alarm can most likely be found out by looking at the procedures in the path of the explanation. We encourage the reader to revisit the graph after gaining familiarity with the notion of causality captured by these graphs in Section 3.

In the rest of the paper, we provide some background on HOUDINI (Section 2), formalize cause and describe EXPLAINHOUDINI (Section 3), and provide preliminary experience on applying it on a couple of large modules (Section 4).

2 Background

In this section, we formalize the concepts in this paper, and present the HOUDINI algorithm [8].

2.1 Source and assertion languages

We assume that the source language supports standard features of most imperative programming languages such as variable assignment ($\mathtt{x} := e$) statements, conditional statements (if (e) s else t), sequential composition ($s; t$), and procedure calls. Accesses to the heap (dereferencing pointers ($*\mathtt{x}$) or object fields ($\mathtt{x}.f$)) is modeled as reads and writes to global map variables that model the entire heap or individual fields in the heap [9, 4, 3]. To enable a uniform formalism, we assume that loops are modeled as tail-recursive procedures.

In addition to assertions present in the program (using **assert** ϕ statements), each procedure can be decorated with preconditions and postconditions. A precondition on a procedure `p` is specified as **requires** ϕ to denote that the condition ϕ is true whenever the procedure `p` is invoked. A postcondition on a procedure `p` is specified as **ensures** ϕ to denote that the condition ϕ is true whenever the procedure `p` returns. The language of ϕ constitutes the *assertion language* for the program. These assertions are well-scoped Boolean expressions over program variables (parameters, globals, return variables), including accesses to fields in objects (e.g. **requires** $\mathtt{x}.f < 5$). The latter is desugared as read over the global map modeling the heap. The assertion language also permits the use of ghost variables and fields to write these specifications. The preconditions and postconditions together constitute the *contract* for a procedure[5]. A program $P = \{p_1, \ldots, p_k\}$ constitutes a set of procedures, where some are marked as *entry* procedures (no callers), and some are marked as *external* procedures (without a body).

2.2 Modular contract checking and contract inference

A modular contract checker verifies that each procedure in a program P can verify its contracts with respect to the contracts of its callees. To perform a modular verification of a procedure p, a procedure call $h(e)$ is replaced by the following sequence of statements: (i) assert the preconditions of h, (ii) scramble all the globals (including the heap map) that could be modified by h and (iii) assume the postconditions of h. For the resultant call-free procedure, a logical formula called the *verification condition* (VC) is generated that encodes the correctness of the procedure p with respect to its contracts. If the formula is valid, then p satisfies its contracts. A failure indicates that some postcondition of p or a precondition of one of the callees of p could not be proved.

[5] For simplicity, we have ignored the issue of modifies clauses in the formalism. For the purpose of this paper, they can be regarded as additional postconditions involving the global heap map that express the facts that remain unchanged.

Let $\mathcal{A}$ denote a set of annotations where $\mathcal{A}_p \subseteq \mathcal{A}$ denotes the annotations for a procedure $p \in P$, i.e., $\mathcal{A} = \bigcup_{p \in P} \mathcal{A}_p$. For a procedure $p \in P$ and two sets of annotations $\mathcal{A}$ and $\mathcal{A}'$, we define $\textsc{CheckVC}(p, \mathcal{A}, \mathcal{A}')$ to be a method that checks the procedure p against the contracts in $\mathcal{A}$, *assuming* the contracts in $\mathcal{A}'$. In other words, the contracts in $\mathcal{A}'$ are "free", i.e. they are not checked. For example, a postcondition $a \in \mathcal{A}'$ for a procedure $p \in P$ can be assumed at a call-site of p but is not checked when the body of p is analyzed. The generalization to allow unchecked assumptions will be useful when we describe the HOUDINI algorithm later in this section.

Definition 1 (Contract checking). *For a set of procedures* $P = \{p_1, \ldots, p_k\}$ *and a set of annotations* $\mathcal{A} = \bigcup_{p \in P} \mathcal{A}_p$*, is* $\textsc{CheckVC}(p, \mathcal{A}, \{\}) = \text{VERIFIED}$ *for every* $p \in P$*?*

Definition 2 (Contract inference). *For a set of procedures* $P = \{p_1, \ldots, p_k\}$ *and a set of annotations* $\mathcal{A} = \bigcup_{p \in P} \mathcal{A}_p$*, does there exist (infer) a set of annotations* $\mathcal{A}' = \bigcup_{p \in P} \mathcal{A}'_p$*, such that* $\textsc{CheckVC}(p, \mathcal{A} \cup \mathcal{A}', \{\}) = \text{VERIFIED}$ *for every* $p \in P$*?*

A restricted form of the contract inference is the *monomial predicate abstraction* problem [10]:

Definition 3 (Monomial predicate abstraction). *For a set of procedures* $P = \{p_1, \ldots, p_k\}$*, a set of annotations* $\mathcal{A} = \bigcup_{p \in P} \mathcal{A}_p$*, and a set of candidate annotations* $\mathcal{C} = \bigcup_{p \in P} \mathcal{C}_p$*, does there exist (infer) a set of annotations* $\mathcal{A}' = \bigcup_{p \in P} \mathcal{A}'_p$*, such that (i)* $\mathcal{A}'_p \subseteq \mathcal{C}_p$ *for each* $p \in P$ *and (ii)* $\textsc{CheckVC}(p, \mathcal{A} \cup \mathcal{A}', \{\}) = \text{VERIFIED}$ *for every* $p \in P$*?*

Care has to be taken to not allow candidate preconditions on entry procedures and candidate postconditions on external procedures that do not have a body. One can trivially satisfy the requirement for contract inference with a candidate precondition of **requires** false for each entry procedure of a program. Similarly, a candidate postcondition **ensures** false for a external procedure can lead to unsoundness. In both these cases, these annotations are always going to be assumed and never checked.

2.3 Houdini algorithm

HOUDINI [8] is an algorithm for solving the monomial predicate abstraction problem described earlier. Given a program P annotated with a set of contracts $\mathcal{A}$ and a set of candidate contracts $\mathcal{C}$, the algorithm employs a greatest fixed-point algorithm over the set of candidates: it starts with the entire set of candidates in $\mathcal{C}$ and successively removes a candidate $c \in \mathcal{C}$ if it is unable to prove c using the $\textsc{CheckVC}()$ procedure. It converges when no candidates can be removed. Let $\mathcal{A}' \subseteq \mathcal{C}$ be the set of annotations that were not removed during the fixed-point algorithm — the algorithm guarantees that $\textsc{CheckVC}(p, \mathcal{A}', \mathcal{A}) = \text{VERIFIED}$ for every $p \in P$. Further, if $\textsc{CheckVC}(p, \mathcal{A}' \cup \mathcal{A}, \{\}) = \text{VERIFIED}$ for every

Algorithm 1 HOUDINI

Require: A program $P = \{p_1, p_2, \ldots, p_k\}$
Require: A set of contracts $\mathcal{A} = \bigcup_{p \in P} \mathcal{A}_p$
Require: A set of candidate contracts $\mathcal{C} = \bigcup_{p \in P} \mathcal{C}_p$
Ensure: A set of inferred contracts $\mathcal{A}' = \bigcup_{p \in P} \mathcal{A'}_p$, such that $\mathcal{A'}_p \subseteq \mathcal{C}_p$ for each $p \in P$

```
 1: WL ← P // Add all procedures to the worklist
 2: for p ∈ P do
 3:    R_p ← {}
 4: end for
 5: while ¬IsEmpty(WL) do
 6:    p ← Dequeue(WL)
 7:    if CheckVC(p, C, A) ≠ VERIFIED then
 8:       Let c be the candidate contract that was not proved
 9:       q ← Proc(c)
10:       Cause(c) ← ExplainHoudini(p, c, A, C, R)
11:       C_q ← C_q \ {c}
12:       R_q ← R_q ∪ {c}
13:       if c is a precondition then
14:          WL ← WL ∪ {q}
15:       else
16:          WL ← WL ∪ {q} ∪ Callers(q) //q is the same as p in this case
17:       end if
18:    end if
19: end while
20: return ⋃_{p∈P} C_p
```

$p \in P$, then the algorithm returns "yes" to the monomial predicate abstraction decision problem.

Algorithm 1 describes the algorithm in detail. The highlighted line can be ignored now. The algorithm maintains a worklist WL of procedures that are pending to be analyzed. This is initialized with all the procedures in P (line 5). The variable $\mathcal{R}_p$ tracks the set of candidates removed from $\mathcal{C}_p$ at any instant, and is initialized to $\{\}$ (line 3). At every iteration, the algorithm extracts a procedure p from the worklist (line 6) and checks the procedure p with respect to $\mathcal{A} \cup \mathcal{C}$. Notice that the algorithm uses CHECKVC$(p, \mathcal{C}, \mathcal{A})$ (in line 7) to ensure that none of the assertions in $\mathcal{A}$ are checked (although they may be assumed) during the contract inference stage.

If the check fails for a candidate assertion $c \in \mathcal{C}$, then it extracts the procedure q containing this assertion (using the procedure PROC() in line 9), removes it from $\mathcal{C}_q$ and adds it to $\mathcal{R}_q$. Finally, it updates WL with q and all the callers of q (only when c is a postcondition of q). It returns the set of candidate assertions that have not been removed as the inferred set of annotations. The algorithm also guarantees that the maximum set of candidates are retained irrespective of the removal order [8, 12].

An alternate implementation may use CHECKVC$(p, \mathcal{A} \cup \mathcal{C}, \{\})$ in line 7 and terminate when an annotation $a \in \mathcal{A}$ cannot be proved. The above algorithm

```
typedef struct _B{
  int g;
  int h;
}B, *PB;

PB gb; //global

ensures probed(p)
void Probe(PVOID p);

cand_requires probed(q)
cand_ensures probed(q)
void Inner6(PB q)
{
  assert probed(q);
  q->h = 25;
}

requires probed(gb)
void Inner5()
{
  PB b = gb;
  Inner6(b);
}
```

```
cand_requires probed(q)
cand_ensures probed(q)
void Inner4(PB q)
{
  assert probed(q);
  q->h = 10;
}

cand_requires probed(q)
cand_ensures probed(q)
void Inner3(PB q)
{
  assert probed(q);
  q->g = 5; // TRUE BUG
}

cand_requires probed(q)
cand_ensures probed(q)
ensures q ==> probed(q)
void Inner2(PB q)
{
  if (q)
     Probe(q);
}
```

```
cand_requires probed(q)
cand_ensures probed(q)
void Inner1(PB q)
{
  Inner2(q);
  Inner3(q);
  if (q)
      Inner4(q);
}

cand_ensures probed(p)
void Entry(PB p)
{
  Inner1(p);
  Probe(gb);
  Inner5();
}
```

Fig. 2. Running example. The additional annotations are underlined. The highlighted annotations are the inferred candidates after adding the additional annotations.

tries to find a subset of the candidates in $\mathcal{C}$ that are provable *assuming* the annotations in $\mathcal{A}$. This is often desirable in practice when $\mathcal{A}$ contains multiple annotations; the inferred annotations can serve as additional annotations that can remove some of the false alarms, even though all the annotations in $\mathcal{A}$ may not be provable (and may indeed be bugs).

2.4 Example

Consider the C example in Figure 2 — we will use this example to illustrate concepts in this paper. The example consists of seven procedures with `Entry` as the entry procedure and a set of internal procedures `Innerx`. We check the *probe-before-use* property on this example — every input pointer (pointers reachable from the globals and inputs to the entry procedures) needs to be sanitized with a call to a procedure `Probe` before being used (dereferenced) [1]. This is modeled by a ghost field inside each pointer that tracks whether it has been probed — the formula *probed*(`q`) denotes that the pointer `q` is probed. The procedure `Probe` has a postcondition to ensure that a pointer is probed in the post-state. The property is instrumented by adding an assertion before every dereference of a pointer (in procedures `Inner3`, `Inner4` and `Inner6`). For this example, only the procedure `Inner3` has a violation of the property.

The annotations in **requires** , **ensures** and **assert** denote the set $\mathcal{A}$. The annotations in **cand_requires** denote the set of candidate annotations $\mathcal{C}$ — let us ignore them initially. Let us also pretend that the underlined annotations are

not present initially. An attempt at modular contract checking of this example produces three counterexamples, one for each of the **assert** statements.

At this point, a user would add the set of guesses using the following instrumentation (say in `HAVOC` [1]) — for any pointer argument `p` of a procedure, add **cand_requires** *probed*(`p`) (except for entry procedures) and **cand_ensures** *probed*(`p`) (except for external procedures with no bodies). This results in the set of candidate annotations present in the program. Running HOUDINI on this example unfortunately removes all the candidates, and infers an empty set of annotations that does not reduce the set of warnings for the example.

Let us assume that the user now adds the annotations that are underlined. Performing modular checking still yields the same three assertion failures. Running HOUDINI(with the candidates) in the presence of the additional annotations in $\mathcal{A}$ yields new inferred annotations (highlighted). With these inferred annotations, a modular checking only produces one assertion failure (in `Inner3`) corresponding to a true bug.

3 ExplainHoudini

Recall that a candidate annotation $c \in \mathcal{C}$ is removed by HOUDINI when it is unable to prove c while checking the annotations of a procedure modularly. There can be several reasons for the removal of a candidate annotation: (1) it does not hold, (2) the set of annotations of the procedure and its callees are not strong enough to prove c, or (3) the theorem prover may be incomplete in proving c. Ideally, the notion of cause should provide feedback for distinguishing between these cases and perhaps suggest additional annotations in the case of (2) to refine the current annotation set. However, our goal is less ambitious — we simply want to show the user the *interprocedural* reason why HOUDINI failed to prove an annotation, purely in terms of the candidate annotations. It is left to the user to use this information to refine the candidate set, identify true errors, or fix prover incompleteness.

In this section, we first define a notion of local *cause* for the removal of a candidate purely in terms of other candidates, and then describe an algorithm EXPLAINHOUDINI that computes the cause for candidates that are removed. Finally, we illustrate the working of this algorithm on the running example.

3.1 Cause

For a run of the HOUDINI algorithm (Algorithm 1), consider a candidate $c \in \mathcal{C}$ that was removed. Let $\mathcal{R}_c \subseteq \mathcal{C}$ contain the set of candidates that were removed prior to the removal of c. Further, let p_c be the procedure whose checking removed c. For such a c, we define a set CAUSESET$(c) \subseteq \mathcal{R}_c$ as follows: a $c' \in$ CAUSESET(c) iff the following conditions hold:

1. $c' \in \mathcal{R}_c$, and
2. there exists a $\mathcal{R}_1 \subseteq \mathcal{R}_c$ such that:

(a) $\text{CheckVC}(p_c, \{c\}, \mathcal{A} \cup (\mathcal{C} \setminus \mathcal{R}_1)) = \text{VERIFIED}$, and
(b) $\text{CheckVC}(p_c, \{c\}, \mathcal{A} \cup ((\mathcal{C} \setminus \mathcal{R}_1)) \setminus \{c'\}) \neq \text{VERIFIED}$.

Intuitively, a c' belongs to $\text{CauseSet}(c)$ if there is a subset $\mathcal{R}_1$ of $\mathcal{R}_c$ such that (a) the removal of $\mathcal{R}_1$ from $\mathcal{C}$ does not affect the provability of c, and (b) the removal of $\mathcal{R}_1 \cup \{c'\}$ makes c unprovable. Note that the two conditions imply that $c' \notin \mathcal{R}_1$. Moreover, since $c' \in \mathcal{R}_c$, c' has to be different from c.

In other words, $\text{CauseSet}(c)$ contains a set of candidates removed before the removal of c, such that there exists an order of removing candidates where c would not be removed unless c' was removed. The intuition is as follows: if the presence of *all* the candidates in $\mathcal{C}$ is not strong enough to prove c, then $\text{CauseSet}(c)$ will be empty; otherwise, the set of candidates in $\mathcal{C}$ is not strong enough to prove some $c' \in \text{CauseSet}(c)$. Since there is a removal order in which c cannot be removed until c' is removed, we label c' as a likely cause for the removal of c.

Although $\text{CauseSet}(c)$ may contain multiple potential causes for each c, considering all possible causes may be overwhelming and may be expensive to compute. In the rest of paper, we will describe a method that concisely describes a *single candidate removal sequence* for each candidate contract c. Let $\mathcal{R}$ be the set of candidates that are removed when the Houdini algorithm terminates. For each $c \in \mathcal{R}$, we define $\text{Cause}(c) \in \mathcal{R} \cup \{\bot\}$, such that:

1. $\text{Cause}(c) \in \text{CauseSet}(c)$, when $\text{CauseSet}(c) \neq \{\}$, and
2. $\text{Cause}(c) = \bot$, otherwise

For a candidate c that was removed, one can also record the intraprocedural trace that removed c as part of $\text{Cause}(c)$ in addition to the candidate c' that forms the cause. These causes can be composed to provide an interprocedural path from $\bot$ to a candidate c that was removed, denoting a candidate removal sequence that terminates in the removal of the particular candidate c.

3.2 Example revisited

Figure 3 shows a graph (for the running example in Figure 2) where a node represents a candidate annotation $c \in \mathcal{R}$ that was removed, and there is an edge (c', p, c) to represent $\text{Cause}(c) = c'$, where $c' \in \{\bot\} \cup \mathcal{R}$ and c was removed during $\text{CheckVC}(p, \ldots)$. Each annotation node has a prefix `$p$k:` to indicate the procedure `p` that the annotation belongs to and the position of the annotation `k` in the list of annotations for `p` (to account for duplicate candidates).

It is useful to understand the candidate removal for the procedures that have false alarms (`Inner4` and `Inner6`). Observe that the edge ($\bot$, `Inner5`, `$Inner6$0`) indicates that the reason for the removal of this precondition cannot be explained in terms of the existing candidates. This is indeed true, as we need the additional annotation **requires** *probed*(`gb`) for `Inner5` (that is not captured by the set of candidates) to retain this candidate precondition. On the other hand, the graph indicates that Houdini cannot remove the precondition of `Inner4` until the postcondition of `Inner3` is present. Similarly, the

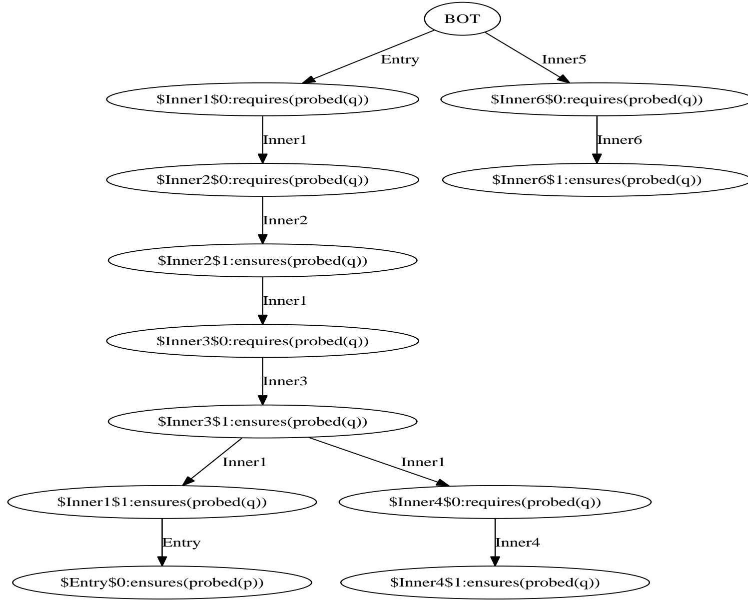

Fig. 3. The output of ExplainHoudini on the running example (in the absence of any of the additional annotations).

postcondition of `Inner3` can be removed only after the precondition of `Inner3` has been removed, which in turn requires the postcondition of `Inner2` to be removed. This process continues all the way where the precondition of `Inner1` is removed inside `Entry`. Although the graph shows the interprocedural causality why HOUDINI removed the precondition of `Inner4`, it is not clever enough to identify the procedure that has to be refined (`Inner2`) and the additional annotation (**ensures** `q` $\implies$ *probed*(`q`)) that can remove the spurious warning. This is consistent with the vision of a transparent inference that performs a scalable analysis to perform simple tasks and leaves the difficult task to the user. While walking the path of causes backwards, it is easy to spot the place that has to be refined and the candidate for a user.

3.3 Algorithm

We now describe the EXPLAINHOUDINI algorithm (Algorithm 2) that computes the CAUSE() for each of the removed candidates. This procedure is invoked during the HOUDINI algorithm (Algorithm 1) just before a candidate c is removed from $\mathcal{C}$ and added to $\mathcal{R}$ — this is indicated by the highlighted line in the algorithm. In addition to c, this procedure is invoked with (i) the procedure p whose checking removes c, (ii) the set of annotations $\mathcal{A}$, (iii) the present set of candi-

Algorithm 2 ExplainHoudini(p, c, $\mathcal{A}$, $\mathcal{C}$, $\mathcal{R}$)

Require: A procedure $p \in P$
Require: A candidate contract $c \in \mathcal{C}_q$, where $q \in \{p\} \cup \text{Callees}(p)$
Require: A set of contracts $\mathcal{A} = \bigcup_{r \in P} \mathcal{A}_r$
Require: The current set of candidate contracts $\mathcal{C} = \bigcup_{r \in P} \mathcal{C}_r$
Require: A set of candidate contracts already removed $\mathcal{R} = \bigcup_{r \in P} \mathcal{R}_r$
Ensure: $c' \in \{\bot\} \cup \mathcal{R}_r$, where $r \in \{p\} \cup \text{Callees}(p)$
1: $\mathcal{Q} \leftarrow (\mathcal{R}_p \cap \{c' \mid c' \text{is a candidate precondition}\})$
2: **for all** $q \in \text{Callees}(p)$ **do**
3: $\quad \mathcal{Q} \leftarrow \mathcal{Q} \cup (\mathcal{R}_q \cap \{c' \mid c' \text{is a candidate postcondition}\})$
4: **end for**
5: $c' \leftarrow \bot$
6: **while** $\text{CheckVC}(p, \{c\}, \mathcal{A} \cup \mathcal{C} \cup \mathcal{Q}) = \text{VERIFIED}$ **do**
7: $\quad c' \leftarrow \text{Dequeue}(\mathcal{Q})$
8: **end while**
9: **return** c'

date annotations $\mathcal{C}$, and (iv) the set $\mathcal{R}$ that contain the candidates removed so far. The algorithm returns the cause of the annotation c.

Since c was removed while checking p, c has to be either (i) a postcondition of p, or (ii) a precondition to one of the callees of p (including p if p calls itself). The set $\mathcal{Q}$ collects the removed candidates that are present as assumptions while checking p — this includes the candidate preconditions of p (line 1) and the candidate postconditions of all the callees of p (line 3).

The loop (line 6) is iterated while $\text{CheckVC}(p, \{c\}, \mathcal{A} \cup \mathcal{C} \cup \mathcal{Q})$ is successful. If this check is not successful, then we return the value in c'. If the body of the loop is never executed, then the returned value is $\bot$. This case indicates that c was not provable with the entire set of candidates in $\mathcal{C}$; therefore $\text{CauseSet}(c) = \{\}$, and consequently $\text{Cause}(c) = \bot$. Otherwise, each loop iteration removes one candidate c' non-deterministically from $\mathcal{Q}$, using the procedure $\text{Dequeue}(\mathcal{Q})$ that extracts a candidate c' from $\mathcal{Q}$. When the returned value in c' is not $\bot$, it is easy to see that $c' \in \text{CauseSet}(c)$. Since $c' \in \mathcal{Q} \subseteq \mathcal{R}$, and c was provable until c' was not removed, c' satisfies the definition of the $\text{CauseSet}(c)$. The only thing to observe is that the while loop (line 6) terminates. If $\mathcal{Q}$ ever becomes empty, we are guaranteed that the check $\text{CheckVC}(p, \{c\}, \mathcal{A} \cup \mathcal{C})$ will fail, because this check is the same as performed in the Houdini algorithm before invoking ExplainHoudini. However, in many cases, the loop will terminate much earlier than $\mathcal{Q}$ becoming empty. Finally, although the soundness of the algorithm does not depend on the choice of implementation of the $\text{Dequeue}(\mathcal{Q})$ procedure, our implementation maintains a priority queue sorted by latest removal time of the candidates in $\mathcal{Q}$.

3.4 Optimizations

The above presentation of the algorithm only requires $\text{CheckVC}(p, \mathcal{C}, \mathcal{A})$ to check a procedure p with respect to a set of annotations A and return either

VERIFIED or an annotation c that was violated. This makes it easy to integrate with any off-the-shelf implementation of HOUDINI. However, the algorithm can benefit from richer functionalities of CHECKVC$(p, \mathcal{C}, \mathcal{A})$. We discuss a couple of such optimizations that are likely to signficantly reduce the number of times the loop in line 6 has to be executed:

- If CHECKVC$(p, \mathcal{C}, \mathcal{A})$ (in HOUDINI algorithm) can return an intraprocedural control-flow path starting from the entry of p to the annotation that was violated, the set of CALLEES(p) in line 2 can be restricted to only the ones along the intraprocedural path. This utility can be easily integrated into the verification condition generation algorithm [14], and is available in several existing tools such as `ESC/Java` and `Boogie` [2].
- Let the CHECKVC$(p, \{c\}, \mathcal{A} \cup \mathcal{C} \cup \mathcal{Q})$ (in EXPLAINHOUDINI) also return a minimal unsatisfiable core of annotations $\mathcal{A}' \subseteq \mathcal{A} \cup \mathcal{C} \cup \mathcal{Q}$ when the check succeeds. Intuitively, $\mathcal{A}'$ represents the annotations that were responsible for proving the candidate c. If such functionality is present, then one can choose c' in line 7 from the annotations in $\mathcal{A}' \cap \mathcal{Q}$. This will have the advantage of choosing those annotations for c' that are more likely to be the cause of c.

4 Evaluation

We have integrated EXPLAINHOUDINI with the implementation of HOUDINI in the `HAVOC` property checker for C. The HOUDINI implementation uses the `Boogie` [2] verification condition generator and the `Z3` [7] SMT solver to implement the CHECKVC() procedure. Our current implementation does not contain the optimizations mentioned in Section 3.4 yet.

We report preliminary experience on using EXPLAINHOUDINI on two large modules in Windows (we have concealed the names of the components for security reasons). The property being checked is the *probe-before-use* described earlier in Section 2.4 and the set of candidates are similar to the ones in the running example. Instead of inspecting the reason for every candidate removed by HOUDINI, a user typically looks at the candidates for the procedures that reported the alarms.

Figure 4 presents a summary of the results for two examples named `A` and `B`. For each example, we report the number of candidates and the number of annotations retained. We compare the overhead of EXPLAINHOUDINI by reporting the number of calls to CHECKVC() and the overall runtime with and without using EXPLAINHOUDINI (indicated by "w" and "w/o" respectively). The absolute runtimes are not very representative of an optimized implementation of HOUDINI — the current implementation has huge overhead due to parsing files for input to `Boogie` and can be improved by at least an order of magnitude. In terms of the number of calls to CHECKVC(), the overhead of EXPLAINHOUDINI is around 51% for `A` and 33% for `B`. In terms of the overall runtime, the overhead of EXPLAINHOUDINI is around 36% for `A` and 29% for `B`. Given that it can help reduce manual post-processing effort, this is fairly acceptable. We believe that the optimizations in Section 3.4 will further reduce this overhead.

Ex	LOC	# Pr	# Cand	# Infrd	w/o ExplainHoudini					w ExplainHoudini				
							Explain size					Explain size		
					# Chk	Time	Avg	Min	Max	# Chk	Time	Avg	Min	Max
A	260K	3108	3026	1415	4972	312m	2.46	1	19	7514	426m	1.45	1	7
B	88K	680	550	269	1097	48m	2.27	1	14	1456	62m	1.22	1	3

Fig. 4. Overhead and quality of ExplainHoudini on two modules. "#" is used to denote the number of various entities. "LOC" denotes the lines of code. "Pr" denotes procedures, "Cand" denotes candidates, "Infrd" denotes inferred, "Chk" is a call to CheckVC(). "Time" denotes the total time for inference in minutes. "Explain size" denotes the size (number of edges) of the graph per removed candidate.

More interestingly, we measured the size of the causality graphs ("Explain size") generated with and without ExplainHoudini per removed candidates. The graph without ExplainHoudini contains all the annotations on the (transitive) callers that were removed before a procedures candidate was removed (similar to the graph in Figure 1 (a)). We report the average, maximum and minimum sizes (denoted as "Explain Size") of these graphs for each module. With ExplainHoudini, the average size of the graphs is reduced by more than 40% for both the examples and the maximum size has considerably reduced. The average graph size is small for these benchmarks because more than 50% of the graphs have a size of 1. However, for the example `A`, there are 119 graphs with more than 5 nodes without ExplainHoudini, but only 2 graphs with more than 5 nodes with ExplainHoudini. Similarly, for the example `B`, there are 40 graphs with more than 3 nodes without ExplainHoudini, but 0 graphs with more than 3 nodes with ExplainHoudini. The results suggest that ExplainHoudini can often provide simpler explanations for annotation removal.

To understand the reason for the shallow explanation graphs, we sampled a few of the removed candidates. In most cases, a candidate *probed*(`p`) for a parameter `p` was removed because the caller passed the value of a field `q->f` to `p`, where `q` is a parameter of the caller. Since we only track *probed*(`q`) at the caller, Houdini was unable to prove *probed*(`q->f`) at the call site. In other cases, caller passed a global or the return value of another procedure to `p` for which there are no candidates at present.

5 Conclusion

For large software modules, understanding the cause of (candidate) contract violations is often most the time-consuming step in refining annotations and classifying errors. We have augmented the Houdini algorithm with ExplainHoudini to provide concise explanation for the removal of a candidate contract. We are exploring the optimizations in Section 3.4 to reduce the overhead of ExplainHoudini and providing the user with likely annotations to refine the existing set of annotations.

Acknowledgements. We thank the anonymous reviewers for useful suggestions on improving the paper.

References

1. T. Ball, B. Hackett, S. K. Lahiri, S. Qadeer, and J. Vanegue. Towards scalable modular checking of user-defined properties. In *Verified Software: Theories, Tools, Experiments (VSTTE '10)*, LNCS 6217, pages 1–24, 2010.
2. M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO '05)*, LNCS 4111, pages 364–387, 2005.
3. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS '05)*, LNCS 3362, pages 49–69, 2005.
4. J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *Principles of Programming Languages (POPL '09)*, pages 302–314, 2009.
5. P. Cousot and R. Cousot. Abstract interpretation : A Unified Lattice Model for the Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Principles of Programming Languages (POPL '77)*, pages 238–252, 1977.
6. M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. Vcc: Contract-based modular verification of concurrent c. In *International Conference on Software Engineering, (ICSE '09), Companion Volume*, pages 429–430, 2009.
7. L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, LNCS 4963, pages 337–340, 2008.
8. C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Symposium of Formal Methods in Europe (FME '01)*, pages 500–517, 2001.
9. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI'02)*, pages 234–245, 2002.
10. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification (CAV '97)*, LNCS 1254, pages 72–83, June 1997.
11. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Symposium on Principles of Programming Languages (POPL)*, pages 232–244. ACM, 2004.
12. S. K. Lahiri and S. Qadeer. Complexity and algorithms for monomial and clausal predicate abstraction. In *International Conference on Automated Deduction (CADE '09)*, LNCS 5663, pages 214–229, 2009.
13. S. K. Lahiri, S. Qadeer, J. P. Galeotti, J. W. Voung, and T. Wies. Intra-module inference. In *Computer Aided Verification (CAV '09)*, LNCS 5643, pages 493–508, 2009.
14. K. R. M. Leino, T. Millstein, and J. B. Saxe. Generating error traces from verification-condition counterexamples. *Sci. Comput. Program.*, 55(1-3), 2005.
15. K. L. McMillan. Interpolation and sat-based model checking. In *Computer Aided Verification (CAV '03)*, LNCS 2725, pages 1–13, 2003.