Houdini, an Annotation Assistant for ESC/Java

Cormac Flanagan and K. Rustan M. Leino

Compaq Systems Research Center
130 Lytton Ave., Palo Alto, CA 94301, USA
{cormac.flanagan,rustan.leino}@compaq.com

Abstract. A static program checker that performs *modular checking* can check one program module for errors without needing to analyze the entire program. Modular checking requires that each module be accompanied by annotations that specify the module. To help reduce the cost of writing specifications, this paper presents Houdini, an annotation assistant for the modular checker ESC/Java. To infer suitable ESC/Java annotations for a given program, Houdini generates a large number of candidate annotations and uses ESC/Java to verify or refute each of these annotations. The paper describes the design, implementation, and preliminary evaluation of Houdini.

0 Introduction

The Compaq Extended Static Checker for Java (ESC/Java) is a tool for finding defects in Java programs [3, 12, 7, 13]. It relies on the programmer to supply annotations describing program properties such as method preconditions, postconditions, and object invariants. These annotations allow ESC/Java to catch software defects using a method-local analysis. During this analysis, ESC/Java verifies that the annotations are consistent with the program, and it also uses the annotations to verify that each primitive operation (such as a dereference operation) will not raise a run-time exception (as might happen, for example, if a dereferenced pointer is null).

Other static checkers that follow this modular approach include conventional type checkers, which rely on type annotations to guide the type checking process, and `rccjava` [5], a static race condition checker, which relies on annotations describing the locking discipline.

A limitation of the modular checking approach is the burden on the programmer to supply annotations. Although programmers have grown accustomed to writing type annotations, they have been reluctant to provide additional annotations. In our experience, this reluctance has been the major obstacle to the adoption of ESC/Java. This annotation burden appears particularly pronounced when faced with the daunting task of applying ESC/Java to an existing (unannotated) code base.

To make ESC/Java more useful in catching defects in legacy code, we have developed Houdini, an *annotation assistant* that infers suitable ESC/Java annotations for an unannotated program. Houdini reuses ESC/Java as a subroutine when inferring these annotation. Essentially, Houdini conjectures a large number of possible *candidate* annotations, and then uses ESC/Java to verify or refute each of these annotations.

This paper describes the design, implementation, and preliminary evaluation of Houdini. Our experience indicates that this approach is capable of inferring many useful annotations. These annotations significantly reduce the number of false alarms produced by ESC/Java (as compared with checking the original, unannotated program), and we have found that using Houdini reduces the programmer time required to statically catch defects in unannotated programs.

The presentation of our results proceeds as follows. The following section starts by reviewing ESC/Java. Section 2 introduces the basic architecture of Houdini. Section 3 describes the heuristics for generating candidate annotations. Section 4 describes how Houdini handles libraries. Section 5 describes Houdini's user interface. Section 6 describes our experience using Houdini to catch defects in four test programs totaling 50,000 lines of code. Houdini is actually a third-generation annotation assistant; Section 7 outlines some prior approaches we have tried. Section 8 describes related work, and we conclude in Section 9.

1 Review of ESC/Java

ESC/Java is a tool for finding common programming errors in Java programs. It takes as input a Java program, possibly annotated with ESC/Java light-weight specifications, and produces as output a list of warnings of possible errors in the program. Because of its static and automatic nature, its use is reminiscent of that of a type checker. Under the hood, however, ESC/Java is powered by a more precise semantics engine and an automatic theorem prover.

ESC/Java performs modular checking: Every routine (method or constructor) is given a specification. ESC/Java checks that the implementation of each routine meets its specification, assuming that all routines called meet their specifications. The specification comes from user-supplied annotations. Note that ESC/Java does not trace into the code of a callee, even if the callee code is also given to the tool to be checked. By performing modular checking, ESC/Java can be applied to a class, or even a routine, at a time, without needing the entire program.

Figure 0 shows a simple Java class that demonstrates the use of typical ESC/Java annotations. The class implements an `n`-tuple of non-null values (of type `Object`), where `n` can be changed over the lifetime of the object. The constructor creates an empty tuple and the `put` method sets element `j` of the tuple to the given value `p`, extending the tuple size by 1 if `j==n`, and returning the previous value, if any, of this tuple element.

ESC/Java annotations are given as specially formatted Java comments. If the first character within the Java comment is an `@`-sign, ESC/Java parses the comment and expects to find legal annotations. The expressions occurring in ESC/Java annotations are mostly just side-effect free Java expression, but with

```
class Tuple {
  int n;
  //@ invariant 0 <= n;

  Tuple() { ... }                    // constructor

  //@ requires 0 <= j;
  //@ requires j <= n;
  //@ requires p != null;
  //@ ensures j == n || \result != null;
  Object put(int j, Object p) { ... }

  ...

  Object a[];
  //@ invariant a != null;
  //@ invariant (\forall int i; 0 <= i && i < n ==> a[i] != null);
  //@ invariant n <= a.length;
}
```

Fig. 0. Examples of typical ESC/Java annotations

some additions, including quantified expressions and some special keywords and functions.

The example in Figure 0 shows the `put` method to have some pre- and post-conditions. The `requires` keyword declares a precondition and the `ensures` keyword declares a postcondition. The occurrences of `j` and `p` in these annotations refer to the method's parameters and `n` refers to the field declared in the class. The postcondition uses the special keyword `\result` to refer to the value returned by the method. Since this is a light-weight specification, it does not specify all aspects of the method's behavior.

The example also shows several declarations of object invariants. ESC/Java checks that these are established by the constructor and maintained by other routines. (The details are described in the ESC/Java user's manual [12].) One of the invariant declarations uses a universal quantification and the ESC/Java implication operator `==>`.

To make ESC/Java simpler, it contains some degree of unsoundness by design. That is, it sometimes fails to detect genuine errors. In practice, this limitation does not negatively affect the usefulness of ESC/Java. Also, since the properties that it attempts to check are undecidable in the worst case, ESC/Java is also incomplete and may produce spurious warnings.

2 Houdini architecture

Although ESC/Java works well on annotated programs, catching defects in legacy, unannotated programs using ESC/Java is an arduous process. It is possible to run ESC/Java on an unannotated program, but this produces an excessively large number of false alarms. Alternatively, one can manually insert appropriate annotations into the program, but this is a very time-consuming task for large programs. Preliminary experience with ESC/Java indicates that a programmer can annotate an existing, unannotated program at the rate of a few hundred lines per hour, or perhaps at a lower rate if the programmer is unfamiliar with the code.

Therefore, we would like to automate much of the annotation process by developing an *annotation assistant* that infers suitable ESC/Java annotations for a legacy, unannotated program. The following *Houdini* algorithm implements an annotation assistant. This algorithm leverages off ESC/Java's ability to perform precise method-local analysis.

Input: An unannotated program `P`
Output: ESC/Java warnings for an annotated version of `P`
Algorithm:
generate set of candidate annotations and insert into `P`;
repeat
  invoke ESC/Java to check `P`;
  remove any refuted candidate annotations from `P`;
until quiescence;
invoke ESC/Java to identify possible defects in `P`;

The first step in the algorithm is to generate a finite set of *candidate annotations*. This set is generated from the program text based on heuristics about what annotations might be useful in reasoning about the program's behavior. For example, since a common precondition in manually-annotated programs is that an argument of reference type is non-null, the candidate annotation set includes all preconditions of this form. Other useful heuristics for guessing candidate annotations are described in Section 3.

Many of the candidate annotations will of course be incorrect. To identify these incorrect annotations, the Houdini algorithm invokes ESC/Java on the annotated program. Like any invocation of ESC/Java, this invocation may produce two kinds of warnings. The first kind concerns potential run-time errors, such as dereferencing the null pointer. These warnings are ignored by the Houdini algorithm.

The second kind of warning concerns invalid annotations. During the checking process, ESC/Java may discover that the property expressed by an annotation may not hold at a particular program point (for example, a method precondition may not hold at a call site of the method). The annotation assistant interprets such warnings as refuting incorrect guesses in the candidate annotation set, and removes these refuted annotations from the program text.

Since removing one annotation may cause subsequent annotations to become invalid, this check-and-refute cycle iterates until a fixpoint is reached. This process terminates, because until a fixpoint is reached, the number of remaining candidate annotations is strictly decreased with each iteration. The resulting annotation set is clearly a subset of the candidate set, and is *valid* with respect to ESC/Java, that is, ESC/Java does not refute any of its annotations. The inferred annotation set is in fact a maximal valid subset of the candidate set. Furthermore, this maximal subset is unique. For a proof of these properties, and also a more efficient version of the basic algorithm presented here, we refer the interested reader to our companion paper [6].

Note that the Houdini algorithm works also for recursive methods. The candidate preconditions of a recursive method will be refined (by removing refuted preconditions) until the resulting set of preconditions holds at all call sites of the method, both recursive and non-recursive call sites.

After the check-and-refute loop terminates, the final step in the Houdini algorithm is to run ESC/Java one more time to identify potential run-time errors in the (now annotated) program. These warnings are then presented to the user, and are used as a starting point in identifying defects in the program.

3 Generating the candidate annotation set

The usefulness of the inferred annotations depends crucially on the initial candidate annotation set. Ideally, the candidate set should include all annotations that are likely to be useful in reasoning about the program's behavior. However, the candidate set should not be too large, because this would increase the running time of the algorithm. Based on an inspection of a variety of hand-annotated programs and on our experience with ESC/Java and Houdini, we have developed the following heuristics for generating candidate annotations.

For any field `f` declared in the program, we guess the following candidate invariants for `f`:

Type of `f`	Candidate invariants for `f`
integral type	`//@ invariant f` *cmp expr* `;`
reference type	`//@ invariant f != null;`

array type	`//@ invariant f != null;`
	`//@ invariant \nonnullelements(f);`
	`//@ invariant (\forall int i; 0 <= i && i <` *expr*
	`==> f[i] != null);`
	`//@ invariant f.length` *cmp expr* `;`

boolean	`//@ invariant f == false;` `//@ invariant f == true;`

Many of these candidate invariants are intended to help verify the absence of index-out-of-bounds errors. For each integral field `f` we guess several inequalities relating `f` to other integral fields and constants. The comparison operator *cmp* ranges over the six operators `<`, `<=`, `==`, `!=`, `>=`, and `>`, and *expr* is either an integral field declared earlier in the same class or an *interesting constant*. Interesting constants include the numbers -1, 0, 1, and also constant dimensions used in array allocation expressions (*e.g.*, `new int[4]`). For each field `f` of an array type, we also guess a number of inequalities regarding `f.length`. Although some of these inequalities are more useful than others, we include all of them for completeness.

Some of these guessed invariants are mutually inconsistent. For example, if a class declares an integral field `f` we will guess several invariants, including:

```
//@ invariant f < 0;
//@ invariant f >= 0;
```

Such inconsistent guesses do not cause a problem. When checking a constructor for the class, ESC/Java will refute at least one of these invariants, since the constructed instance cannot simultaneously satisfy both invariants.

We also guess candidates invariants that help verify the absence of null dereference errors. For each field `f` of a reference type, we guess `f != null`. For each field `f` of an array type, in addition to guessing the invariant `f != null`, we also guess the invariant `\nonnullelements(f)`, which states that each entry in the array is not null, and we guess an invariant that all entries in `f` up to ***`expr`*** (a field or an interesting constant) are not null. We have found this last property to be useful in reasoning about the behavior of stack-like data structures implemented using arrays.

We generate candidate preconditions and postconditions in a similar manner for each routine declared in the program. Candidate preconditions may include inequalities relating two argument variables, or relating an argument variable to a field declared in the same class. Candidate postconditions may relate the result variable `\result` to either argument variables or fields. In addition, we generate the candidate postcondition

```
//@ ensures \fresh(\result);
```

which states that the result of a method is a newly-allocated object, and hence not an alias of any previously existing object.

As an aid in identifying dead code, we generate the candidate annotation

```
//@ requires false;
```

for every routine in the program. An unrefuted `requires false` annotation indicates that the corresponding routine is never called.

For correctness reasons, we require that all applicable candidate annotations hold in the program's initial state. Hence, for the program entry point

```
public static void main(String args[]) { ... }
```

we only generate the following precondition, which is ensured by the Java runtime system:

```
//@ requires \nonnullelements(args);
```

4 Dealing with libraries

So far, we have described Houdini as a system that infers annotations based on an analysis of the entire program. However, the program may be linked with a library (or with several libraries) that we cannot analyze, either because we do not have source code for the library or because the size of the library makes the analysis impractical.

If the library in question already includes ESC/Java annotations that specify its interface, or if we are willing to write such an interface specification, then it is straightforward to adapt the Houdini algorithm to analyze and annotate the remainder of the program with respect to this specification. In many cases, however, the size and complexity of the library makes writing an interface specification quite tedious. Hence, we would like to be able to infer annotations for a program even in the absence of ESC/Java specifications for all of the libraries used by the program.

Therefore, we extend Houdini so that it can analyze a program with respect to *guessed* specifications for these libraries. There are two main strategies for guessing library specifications. The first strategy is to make *pessimistic* assumptions, for example, that all pointers returned by library methods may be null. Since many of these pointers will never be null, such pessimistic specifications cause Houdini to produce a large number of false alarms in the rest of the program, and we have not found this approach cost-effective for static debugging. (ESC/Java provides pessimistic assumptions by default in the absence of library annotations.)

An alternative strategy is to make *optimistic* assumptions about the behavior of libraries, for example, that all pointers returned by library methods will be non-null. Since some of these pointers may sometimes be null, this assumption is unsound, and may cause Houdini to miss certain run-time errors. However, in library clients, Houdini will still detect many other run-time errors, and since the optimistic specifications lead to many fewer false alarms, this appears to be a more cost-effective strategy for guessing library specifications.

For libraries, we need to be careful not to guess contradictory annotations. To illustrate this idea, suppose that we generated the two contradictory postconditions

```
//@ ensures \result < 0;
//@ ensures \result >= 0;
```

for a library method that the program does not override. Since the implementation of the library method is not checked, neither of these guessed annotations will be refuted. ESC/Java would then infer that the method never returns and would not check code following a call to this method. Therefore, we only guess the following consistent postconditions for each library method:

Result type	Optimistic postconditions
integral type	`//@ ensures \result >= 0;`
reference type	`//@ ensures \result != null;`
array type	`//@ ensures \result != null;` `//@ ensures \nonnullelements(\result);`

We guess optimistic preconditions and invariants in a similar manner.

The modified Houdini algorithm for dealing with libraries is as follows:

Input: An unannotated program `P`
A set of libraries `S` with specifications
A set of libraries `L` without specifications
Output: ESC/Java warnings for an annotated version of `P`
Algorithm:
generate and insert candidate annotations into `P`;
generate and insert optimistic annotations into `L`;
repeat
invoke ESC/Java to check `P` with respect to `L` and `S`;
remove any refuted candidate annotations from `P`;
remove any refuted optimistic annotations from `L`;
until quiescence;
invoke ESC/Java to identify possible defects in `P` with respect to `L` and `S`;

5 User interface

To catch defects using Houdini, a user starts by inspecting Houdini's output, which includes the set of warnings that ESC/Java produces for the annotated version of the program. Unlike using ESC/Java, where a warning in one routine often points to a problem with that routine's implementation or specification, the warnings produced by Houdini are more often caused by some other part of the program. For example, suppose one of the warnings points out a possible null dereference of `t` in the following method:

```
char getFirstChar(String t) { return t.charAt(0); }
```

A user's first reaction might be: But I only intend `getFirstChar` to be called with non-null arguments! The ESC/Java user would then add the precondition `t != null`, which suppresses the spurious warning. (This precondition is then checked at call sites.) But the Houdini user will instead ask: Why didn't Houdini infer this precondition?

In our experience with looking at Houdini output, we constantly asked questions such as these. We developed a simple user interface to help answer these questions. The user interface generates a collection of HTML pages. The root page of this collection presents a summary of the kinds of warnings that the final call to ESC/Java produces, followed by the actual list of warning messages. Each warning message contains a hyperlink to the source view of the code at the location of the offending program line.

In the source code view (shown in Figure 1), the user interface displays all of the candidate annotations guessed by Houdini. A refuted annotation is grayed out and hyperlinks to the source line where ESC/Java issued the warning that refuted the annotation. We also insert the warning messages into the code in the source view.

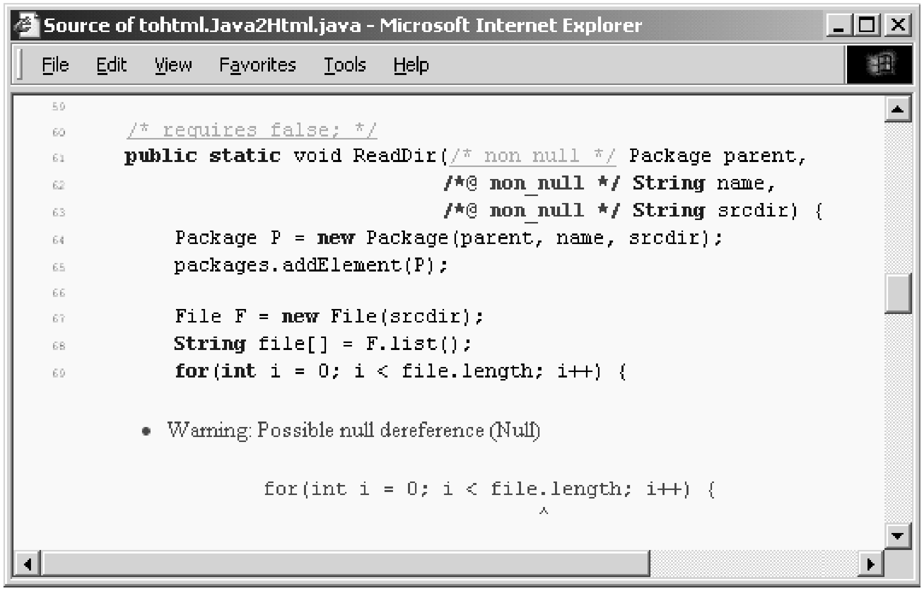

Fig. 1. Screen shot showing the source view of Houdini's user interface. (Declaring a parameter, like `parent`, with the ESC/Java modifier `non_null` serves as an alternate way of writing the precondition `requires parent != null;`.)

Thus, in the example above, the Houdini user would look at the preconditions that Houdini guessed initially for `getFirstChar`. These would include a grayed-out precondition `t != null`, and clicking on this refuted annotation would bring the user to a call site where, as far as ESC/Java could tell, the actual parameter of `t` may be null. This may lead the user to understand whether the null dereference warning in `getFirstChar` is a real problem with the program or a spurious warning, or this may just be one of a number of similar steps required to get to the source of the problem. Surprisingly, our experience indicates that presenting the *refuted* annotations and the causes thereof is the most important aspect of the user interface.

6 Experience

We have applied Houdini to tens of thousands of lines of unannotated program code. Here we report on four programs that have been subjected to many test runs of various versions of the tool. They are:

– Java2Html [11], a 500-line program that turns Java programs into color-coded HTML pages,
– WebSampler, a 2,000-line program that performs statistical samplings of trace files generated by the web crawler Mercator [10],
– PachyClient, the 11,000-line graphical user interface of the web-based email program Pachyderm [14], and
– "Cobalt", a proprietary 36,000-line program.

These programs had been tested and used, in some cases extensively, before Houdini was applied to them.

Table 0 shows some statistics for the first three programs. For each program, the table shows three columns. The first of these columns indicates the number of checks performed by ESC/Java to verify various correctness properties. These correctness properties include proving that various run-time exceptions will not occur and that libraries are used in a manner consistent with their manually written specifications.

warning or exception type	Java2Html			WebSampler			PachyClient		
	checks	warnings		checks	warnings		checks	warnings	
		before	after		before	after		before	after
NullPointerException	145	41	2	328	87	12	2619	1016	392
IndexOutOfBoundsExn.	8	3	2	228	112	19	294	57	24
ClassCastException	8	7	7	5	4	3	152	117	103
ArithmeticException	0	0	0	23	2	2	25	7	7
NegativeArraySizeExn.	1	0	0	16	2	0	20	4	2
ArrayStoreException	0	0	0	5	2	0	6	0	0
library annotations	22	5	0	147	43	5	376	49	17
all	184	56	11	752	252	41	3492	1250	545

Table 0. A breakdown of the checks performed by ESC/Java and the warnings produced by ESC/Java before and after applying Houdini.

Checking that NullPointerException is not raised is by far the most common check. The row for IndexOutOfBoundsException counts every array dereference as two separate checks, one for the lower bound and one for the upper bound. ClassCastException checks arise from type cast expressions, which are commonly used with container classes. ArithmeticException is raised in the case of integer division by zero. NegativeArraySizeException is raised if an attempt is made to allocate an array with a negative size. The need for ArrayStoreException checks comes from Java's co-variant array subtyping rule. The libraries we used in checking these programs contained some manually inserted lightweight ESC/Java specifications. The library annotations row shows the number of checks in the program arising from these specifications.

The second and third columns for each program in Table 0 show the number of these checks that ESC/Java is not able to statically verify. Each such unverified check corresponds to a warning produced by ESC/Java. The second column is the number of warnings produced for an unannotated program and the third column is the number of warnings produced after running Houdini to infer annotations for the program.

Java2Html was something of a lucky draw: of the 4 non-cast warnings reported, all 4 indicate real errors in the source code. For example, the program will crash if a line in the input exceeds 1024 characters or if the system call `File.list()` returns null (see Figure 1). Houdini currently does not have support for guessing the annotations needed to verify cast checks. For this reason, Houdini actually uses a command-line option to ESC/Java to suppress all cast warnings, since we have found it more cost effective for users to investigate warnings of other kinds.

The warnings produced on WebSampler led us to find 3 real errors. One of these was part of a class that was borrowed from the web crawler Mercator: the method

```
int read(byte[] b);
```

of `java.io.InputStream` is supposed to read characters into the given array `b`, returning -1 if the input stream is at end-of-file or returning the number of characters read otherwise. However, a Mercator subclass of `InputStream` erroneously returned 0 whenever the length of `b` was 0, even if the stream was at end-of-file.

Houdini also issued several warnings pointing out places where WebSampler assumed its input to have a particular format. For example, in some situations, WebSampler assumed the next line of input to be at least 4 characters long and to consist only of characters that can be interpreted as a decimal number. The warnings pointing out these infelicities were considered spurious, since WebSampler is only intended to work on well-formed input.

We have inspected only about a dozen of the PachyClient warnings. Nevertheless, two of these pointed out infelicities that compelled the author of the code to make code changes. Technically, one can argue that these infelicities were not errors, but they did make the code overly brittle. The author changed the code to guard against possible future failures.

Table 1 shows some statistics for the Cobalt program. Rather than using hand-annotated libraries, we analyzed this program using the two strategies of pessimistic and optimistic library assumptions. For each of these two strategies, Table 1 shows the number of checks that ESC/Java performs to verify various correctness properties, and the number of these checks that ESC/Java is not able to statically verify before and after running Houdini.

The table shows that the use of optimistic library annotations can significantly reduce the number of null-dereference and bounds warnings that Houdini produces.

warning or exception type	no lib. anns. checks	warnings before	warnings after	optimistic lib. anns. checks	warnings before	warnings after
NullPointerException	10702	3237	1717	10702	2438	488
IndexOutOfBoundsExn.	982	151	75	982	118	52
ClassCastException	347	278	234	347	271	234
ArithmeticException	17	0	0	17	0	0
NegativeArraySizeExn.	60	13	11	60	1	0
ArrayStoreException	18	0	0	18	0	0
library annotations	0	0	0	9385	787	0
all	12126	3679	2037	21511	3615	774

Table 1. A breakdown of the checks performed by ESC/Java on the Cobalt program, and the warnings produced by ESC/Java before and after applying Houdini, using both pessimistic and optimistic library annotations.

One hundred of the non-cast warnings of Cobalt were inspected and revealed 3 real errors in the code.

For the first three programs, we did not measure the user time required to inspect the warnings, but in the case of Cobalt, this time was measured to be 9 hours to inspect one hundred warnings. Toward the end of this time, the inspection proceeded at a higher pace than in the beginning, partly because of getting more familiar with the tool's output and partly because of repeated spurious warnings. This experience suggests that using Houdini, a user can inspect a program for errors at a rate upwards of 1000 lines per hour.

Despite the precision of ESC/Java, Houdini still produces many false alarms. A major cause of false alarms is that Houdini may fail to guess the right annotations for a given program. In particular, Houdini does not guess disjunctions, such as in the following postcondition of the method `put` from Figure 0:

```
//@ ensures j == n || \result != null
```

Another cause of false alarms in Houdini is the incompleteness of ESC/Java, comprising the incompleteness of the underlying theorem prover, the incompleteness of ESC/Java's axiomatization of Java's operators (for example, the semantics of bitwise-and is not completely axiomatized), and the incompleteness of ESC/Java's light-weight annotation language. An ESC/Java user would know to insert `nowarn`, `assume`, and `axiom` annotations to make up for any such incompleteness (see the ESC/Java user's manual [12]), but Houdini does not infer such annotations.

After inspecting a serious warning, a user would normally fix the error in the program. If instead the user determines the warning to be spurious, a prudent course of action is to convey to Houdini the missing pieces of information. For example, if the cause of a spurious warning is that Houdini didn't guess some annotation, possibly with a disjunction like the one shown above, then the user can manually insert the missing annotation into the program.

Thus, experienced Houdini users are likely to manually insert ESC/Java annotations into the program, just like ESC/Java users would. However, Houdini users insert many fewer annotations. We predict that leaving such a small number of annotations in the code will be acceptable to most programmers.

These manually-inserted annotations give future runs of Houdini more information about the program, which may cause Houdini to generate fewer spurious warnings. Thus, experienced users are likely to rerun Houdini after adding some manual annotations. This iterative process can be quite cost effective, because even small manual interventions can prevent what otherwise might have resulted in a cascade of spurious refutations.

A consequence of this iterative process is that the Houdini running time must not be too long. Since Houdini, like all other static debuggers, competes in practice with software testing, it seems reasonable that a dozen iterations may be done over the course of a couple of weeks. This means that each run of Houdini must be fast enough to complete overnight, say, finishing within 16 hours.

The version of Houdini reported on in this paper does not meet the overnight challenge: the running time on the 36,000-line Cobalt program was 62 hours. We remain optimistic, however, for several reasons. First, some preliminary experiments with algorithmic improvements seem promising. Second, measurements of the work performed during each of Houdini's iterations suggest the operations of Houdini to be parallelizable. And third, we have built a prototype of a dynamic refuter, which creates an instrumented version of the program that, when run, records which candidate annotations are violated (*i.e.*, refuted) during the execution. This significantly reduces the number of candidate annotations that are left to be refuted by ESC/Java.

Since a run of Houdini may take many hours, an important aspect of Houdini's usability is that it is restartable. The system periodically writes a snapshot of its current state to disk, so that if that particular run is abruptly terminated (for example, by a power failure), it can later be restarted at the most recent snapshot.

Finally, we give some measurements that provide a preliminary idea of the effectiveness of the various heuristics described in Section 3 for generating candidate annotations. Table 2 shows the number of candidate annotations generated for the Cobalt program by these heuristics, and the percentage of these annotations that are actually valid. In this table, the count of valid annotations includes many annotations that hold in all Java programs (for example, `a.length >= -1`) and annotations that are subsumed by other annotations (for example, `x != -1` is subsumed by `x > 0`); it remains for future work to avoid such valid but redundant annotations.

7 Other annotation assistants

The design of Houdini was inspired by the experience with two other ESC/Java annotation assistants that we developed. The annotations inferred by Houdini

Type of annotation	Preconditions guessed	Preconditions %valid	Postconditions guessed	Postconditions %valid	Invariants guessed	Invariants %valid	Total guessed	Total %valid
`f ==` *expr*	2130	18	985	18	435	14	3550	17
`f !=` *expr*	2130	35	985	35	435	38	3550	35
`f <` *expr*	2130	26	985	27	435	24	3550	26
`f <=` *expr*	2130	31	985	32	435	36	3550	33
`f >=` *expr*	2130	25	985	21	435	19	3550	32
`f >` *expr*	2130	31	985	36	435	35	3550	23
`f != null`	509	92	229	79	983	72	1721	79
`\nonnullelems(f)`	54	81	21	62	36	64	111	72
`(\forall ...)`	841	27	260	37	125	59	1226	32
`f == false`	47	36	51	25	39	10	137	20
`f == true`	47	28	51	24	39	8	137	25
`\fresh(\result)`	0	0	229	30	0	0	229	30
`false`	780	17	0	0	0	0	780	17
exact type	37	19	11	36	14	57	62	31
Total	15095	30	6762	30	3846	40	25703	31

Table 2. Numbers of candidate annotations generated on the Cobalt program by the heuristics of Section 3, and the percentages of these annotations that are valid.

have been significantly more useful than the annotations inferred by these earlier annotation assistants.

The first annotation assistant starts with an unannotated program and iteratively added annotations. To support this annotation assistant, ESC/Java was modified to output a suggestion with each warning, whenever possible. For example, if a warning points to the dereference of a formal parameter `p` that is not changed by the routine body being checked, the suggestion is to add a precondition `p != null`. Each suggestion has the property that by following the suggestion and rerunning ESC/Java, the warning will be suppressed (but other, new warnings may be generated). The annotation assistant iteratively runs ESC/Java and follows the suggestions until ESC/Java produces no more suggestions.

Although many of the suggestions are good, this annotation assistant has two severe limitations. First, it is hard to produce enough suggestions. Not only do new heuristics become increasingly more complicated, but because of the requirement of only suggesting measures that are sure to suppress the warning, the heuristics for when to make a suggestion frequently have side conditions that are not met (such as "... and the formal parameter is not assigned to by the body"). Second, the suggested annotations are not always correct. The cascading effects of incorrect annotations limit the effectiveness of the annotation assistant.

The second annotation assistant uses a whole-program set-based analysis [4, 9] to identify which variables and fields are never null and inserts corresponding annotations into the program. These annotations are useful in verifying many dereference operations. However, the inferred annotations do not include numeric inequalities (which are necessary for verifying the absence of array bounds errors) and do not include properties such as

```
//@ invariant (\forall int i; 0 <= i && i < expr ==> f[i] != null);
```

(which are necessary for checking stack-like data structures implemented using arrays).

Like the first annotation assistant, Houdini uses ESC/Java as a powerful subroutine. Like the second annotation assistant, Houdini infers only valid annotations. Furthermore, since Houdini's iterative check-and-refute machinery does not depend on the particular annotations contained in the candidate set, Houdini provides a flexible architecture for inferring many kinds of annotations.

Another advantage of the Houdini algorithm is its generality: it is not closely dependent on the underlying checker and can be used to infer annotations for a variety of modular static checkers. We have successfully ported Houdini to a second checker, the race condition checker for Java [5]. This checker extends Java's type system with additional checks that verify the absence of race conditions. The checker relies on additional type annotations to describe aspects of the locking discipline, for example, the protecting lock of each field. Adapting the Houdini algorithm to guess many such type annotations was straightforward, and, as with ESC/Java, we have found the annotations inferred by the system to be useful.

8 Related work

Predicate abstraction is a technique for analyzing an infinite state system, given a set of predicates over the state space of the system [8]. This technique finds a boolean combination of the predicates that holds in all reachable states, and that holds in a minimum of unreachable states.

The Houdini algorithm can be viewed as a variant of predicate abstraction in which each candidate annotation corresponds to a predicate. Interestingly, the Houdini algorithm does not consider arbitrary boolean formulae over these predicates; it only considers conjunctions of predicates. This restriction means that Houdini cannot infer disjunctions or implications of candidate predicates. For example, given the two predicates `j == n` and `\result != null`, Houdini could not infer the property

```
j == n || \result != null
```

This restriction reduces the maximum number of iterations of the algorithm from exponential to linear in the number of predicates; however, it also increases the number of false alarms produced by the system.

Abstract interpretation [1] is a standard framework for developing and describing program analyses. We can view the Houdini algorithm as an abstract interpretation, where the abstract state space is the power set lattice over the candidate annotations and the checker is used to compute the abstract transition relation. As usual, the choice of the abstract state space controls the conservative approximations performed by the analysis. In our approach, it is easy to tune these approximations by choosing the set of candidate annotations appropriately, provided that this set remains finite and that the annotations are understood by the checker.

An interesting aspect of our approach is that the checker can use arbitrary techniques (for example, weakest preconditions in the case of ESC/Java) for performing method-local analysis. If these local analysis techniques allow the checker to reason about sets of intermediate states that cannot be precisely characterized using the abstract state space, then the Houdini algorithm may yield more precise results than a conventional abstract interpretation that exclusively uses abstract states to represent sets of concrete states.

PREfix is a static programming tool that warns about possible errors in C and C++ code [0]. There are no annotations involved in using PREfix, which is mostly an advantage. We find the presence of annotations in Houdini's output, including the refuted annotations, helpful when inspecting the tool's output. Annotations also provide a general and convenient way for users to supply the tool with missing, uninferred facts. The technology underlying PREfix is different from the precise semantics engine and automatic theorem prover that underly ESC/Java, but perhaps the differences are bigger than they need to be.

Daikon is a system that uses an empirical approach to find probable invariants [2]. These invariants are found by creating an instrumented version of the program that records a trace of intermediate program states, running the instrumented program on a test suite, and then analyzing the generated traces off-line to determine properties that hold throughout all runs. Given a suitably complete test suite, the inferred properties are likely to be true program invariants.

9 Conclusions

This paper describes a technique for building an annotation assistant for a modular static checker. The annotation assistant reuses the checker as a subroutine; it works by guessing a large number of candidate annotations and using the checker to verify or refute each candidate annotation.

We have used to this technique to develop an annotation assistant, called Houdini, for the modular program checker ESC/Java. Houdini is capable of inferring a large number of useful annotations, which significantly reduce the number of false alarms produced by ESC/Java (as compared with checking an unannotated program). These inferred annotations also reduce the programmer time required to check an existing, unannotated program for defects.

In our experience, a natural strategy for using Houdini is to maintain a code base with a few manually-inserted annotations that Houdini cannot infer, and to rely on Houdini to infer additional annotations whenever the code needs to be checked. Thus, we expect that Houdini will be considered by users as a static checker in itself, and not just as an annotation assistant for the underlying checker ESC/Java.

A number of issues remain for future work, including refining the heuristics for the generation of the candidate annotations, improving the performance of Houdini (significant progress has already been made in this direction), and enhancing the user interface. However, the system developed to date has already proven capable of catching defects in several real-world programs.

Acknowledgements

We are grateful to our colleagues who contributed in various ways to the Houdini project: Yuan Yu suggested and helped us develop the dynamic refuter. He also helped inspect the Houdini output on WebSampler. Steve Freund ported Houdini to `rccjava`. Raymie Stata helped create the earlier annotation assistant that used set-based analysis. Jim Saxe helped think about how to get the underlying theorem prover to work well with Houdini's demands. Lawrence Markosian and Dr. Maggie Johnson of Reasoning, Inc., Mountain View, CA inspected the Houdini output on Cobalt. Roy Levin suggested "Houdini" as the name of "the great ESC wizard".

References

0. William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice & Experience*, 30:775–802, 2000.
1. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.
2. Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland*, June 2000.
3. Extended Static Checking home page, Compaq Systems Research Center. On the Web at `research.compaq.com/SRC/esc/`.
4. Cormac Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, Houston, Texas, May 1997.
5. Cormac Flanagan and Steven N. Freund. Type-based race detection for Java. In *Proceedings of the 2000 ACM SIGPLAN conference on Programming Design and Implementation (PLDI)*, pages 219–232, 2000.
6. Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. Annotation inference for modular checkers. *Information Processing Letters*, 2001. To appear.
7. Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages*. ACM, January 2001. To appear.
8. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *CAV 97: Computer Aided Verification*, Lecture Notes in Computer Science 1254, pages 72–83. Springer-Verlag, 1997.

9. Nevin Heintze. Set-based analysis of ML programs. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 306–317, 1994.
10. Allan Heydon and Marc A. Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, December 1999.
11. Java2html, Compaq Systems Research Center. On the Web at `research.compaq.com/SRC/software/`.
12. K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Note 2000-002, Compaq Systems Research Center, October 2000.
13. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999. Also available as Technical Note 1999-002, Compaq Systems Research Center, from `research.compaq.com/SRC/publications/`.
14. The Pachyderm email system, Compaq Systems Research Center. On the Web at `research.compaq.com/SRC/pachyderm/`, 1997.