Houdini, an Annotation assistant for ESC/Java

Based on the paper “Houdini, an annotation assistant for ESC/Java” by Cormac Flanagan and K. Rustan M. Leino, Compaq Systems Research Center and the User manual for ESC/Java

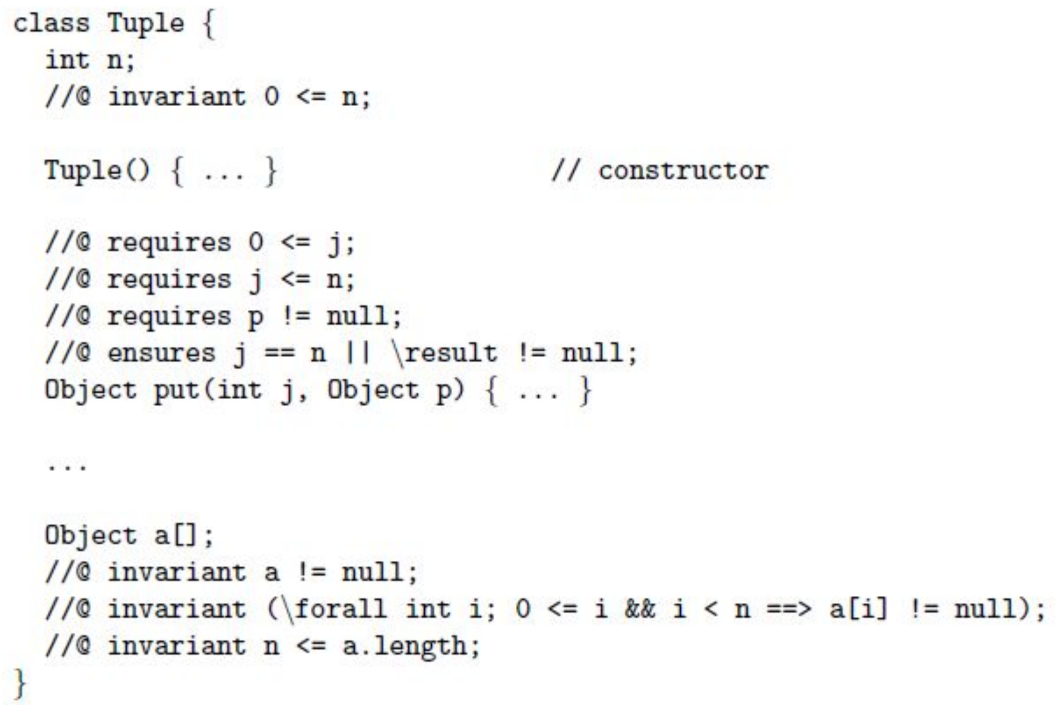
COL731 - Paper presentation

Gokul Sankar (2023CSZ8080)

Introduction to ESC/Java

- Compaq Extended Static Checker for Java (ESC/Java) is a tool for finding defects in Java programs. It takes as input a Java program and produces as output a list of warnings of possible errors in the program
- It catches defects in the software using annotations describing program properties like method preconditions, post-conditions and object invariants
- It uses method local analysis to verify that the annotations are consistent with the program and to verify that each primitive operation will not raise a run-time exception (like dereferencing a null pointer)
- To make ESC/Java simpler, it contains some degree of unsoundness by design, that is it fails to detect genuine errors. Also since the properties that it attempts to check are undecidable, ESC/Java is incomplete and may produce spurious warnings

Example of ESC/Java annotated program



```
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

- The example 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 postcondition uses a special keyword `\result` to refer to the value returned by the method.
- The example also contains object invariants. ESC/Java checks that these are established by the constructor and maintained by the routines

Issues with using ESC/Java

- Although ESC/Java works well on annotated programs, catching defects in legacy unannotated programs is an arduous process.
- It is possible to run ESC/Java on an unannotated program but it produces an excessive number of false alarms
- Alternatively, a programmer can manually insert appropriate annotations into the program, but this is very time-consuming especially if the programmer is unfamiliar with the code.

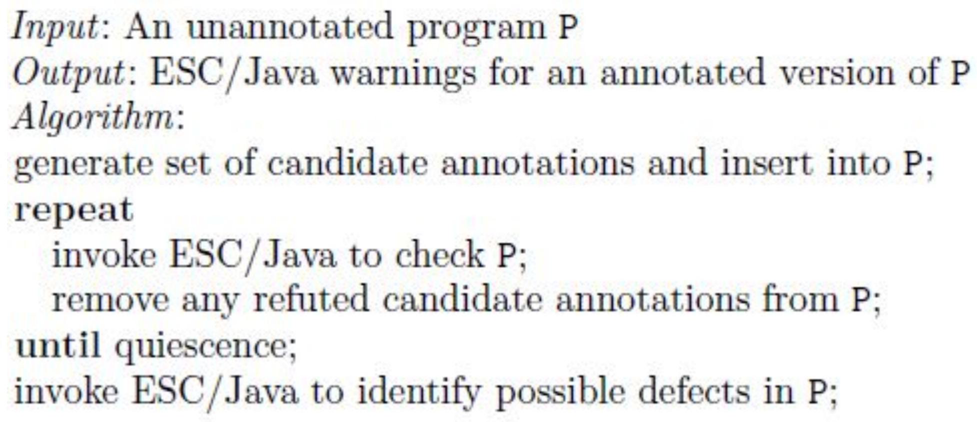
Introduction to Houdini

- It is an annotation assistant for ESC/Java that infers suitable annotations for an unannotated program

- Houdini conjectures a large number of possible candidate annotations, and then uses ESC/Java to verify and refute each of the annotations.

- These annotations significantly reduce the number of false alarms that are produced by ESC/Java as compared with checking the original, unannotated program

Houdini algorithm



```
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;
```

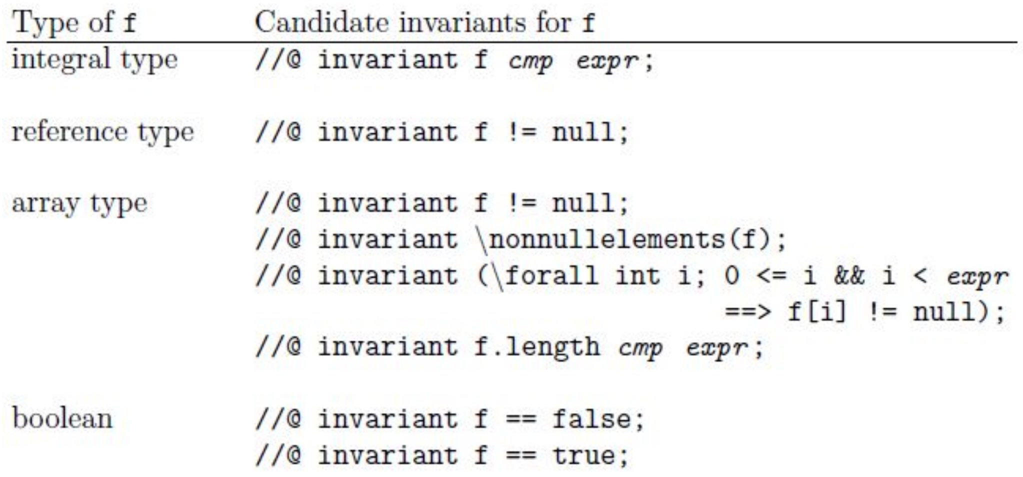
- First, the algorithm generates a finite set of candidate annotations. This set is generated from the program text based on heuristics about what annotations might be useful in reasoning about the program's behaviour

- For example, a common precondition in manually annotated programs is that an argument of reference type is non-null so Houdini includes all annotations of this form
- To identify incorrect annotations, the Houdini algorithm invokes ESC/Java on the annotated program.
- This invocation may produce two kinds of warnings. The first kind is about potential run-time errors. These warnings are ignored by the Houdini algorithm.

- The second kind of warnings are regarding invalid annotations. During the checking process ESC/Java may discover that the property expressed by an annotation may not hold a particular program point. For example, a method precondition may not hold at a call site.
- Houdini interprets these warnings as refuting incorrect guesses in the candidate annotation set, and removes these refuted annotations
- Removing one annotation may cause subsequent annotations to become invalid, so the check and refute cycle iterates until a fixpoint is reached. This process terminates, because the number of candidate annotations is strictly decreased with each iteration.
- The inferred annotation set is 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 a maximal valid subset of the candidate set and is unique.
- The Houdini algorithm also works for recursive methods. The candidate preconditions of a recursive method will be refined until the resulting set of preconditions holds at all call sites of the method.
- After, the check and refute loop, terminates , ESC/Java is run one more time to identify potential run-time errors.

The candidate annotation set

- The usefulness of the inferred annotations depends 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 behaviour without being too large as this would increase the running time of the algorithm.



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;`

- For each integral field `f` the algorithm guesses several inequalities relating `f` to other integral fields and constants. The comparison operator `cmp` ranges over <, <=, ==, !=, >= and > and `expr` is either an integral field declared earlier in the same class or an interesting constant such as the numbers -1, 0, 1 and constant dimensions used in array allocation expressions.
- For each field of array type inequalities regarding `f.length` are guessed.

The candidate annotation set

- Some of the guessed invariants are mutually inconsistent (for eg. f >= 0 and f < 0). Such guesses do not cause a problem as ESC/Java will refute at least one of these invariants.

- The candidate preconditions and postconditions are generated 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 to either argument variables or fields.

- In addition, the candidate postcondition `//@ ensures \fresh(\result);` is generated which states that the result of a method is a newly-allocated object. Also the precondition `//@ requires false;` is generated to help identify dead code. If the annotation is unrefuted the corresponding routine is never called.

- For correctness reasons, it is required that all applicable candidate annotations hold at the program's initial state, so for the program's entry point only `//@ requires \nonnullelements(args)` is generated which is ensured by the Java runtime system

Dealing with Libraries

- The program may be linked with one or more libraries that cannot be analyzed because the source code for the library is not available or such analysis is impractical due to the size of the library.

- If the library in question already includes ESC/Java annotations then it is straightforward to adapt the Houdini algorithm to analyze and annotate the remainder of the program.

- If the library is unannotated, the specifications for the library are guessed to analyze the program.

Dealing with Libraries

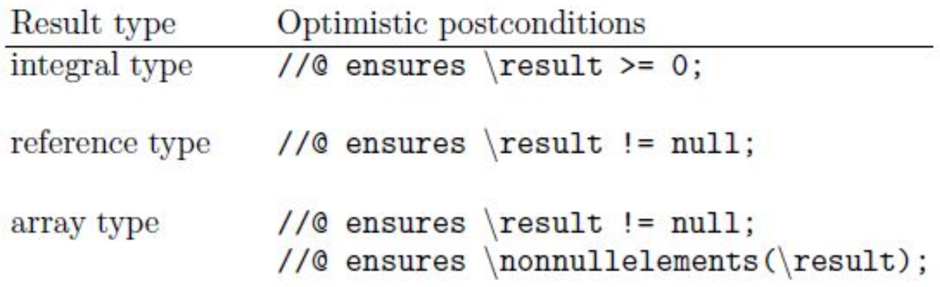
- There are two main strategies for guessing library specifications.
- The first is to make pessimistic assumptions (for eg. all pointers returned by library methods may be null). This can cause a large number of false alarms.
- The second is to make optimistic assumptions about the behaviour of libraries (for eg. all pointers returned by library methods are not null). These assumptions may be unsound and may cause Houdini to miss some run-time errors.
- As the optimistic strategy leads to fewer false alarms, Houdini uses this strategy.

Dealing with Libraries

- For libraries, guessing contradictory annotations leads to a problem. As the implementation of the library method is not checked, none of the contradictory annotations get refuted.

- For example, the annotations for the postcondition of a library method `//@ensures \result < 0;//ensures \result >= 0;` are contradictory but neither get refuted as the implementation of the library method doesn't get checked.

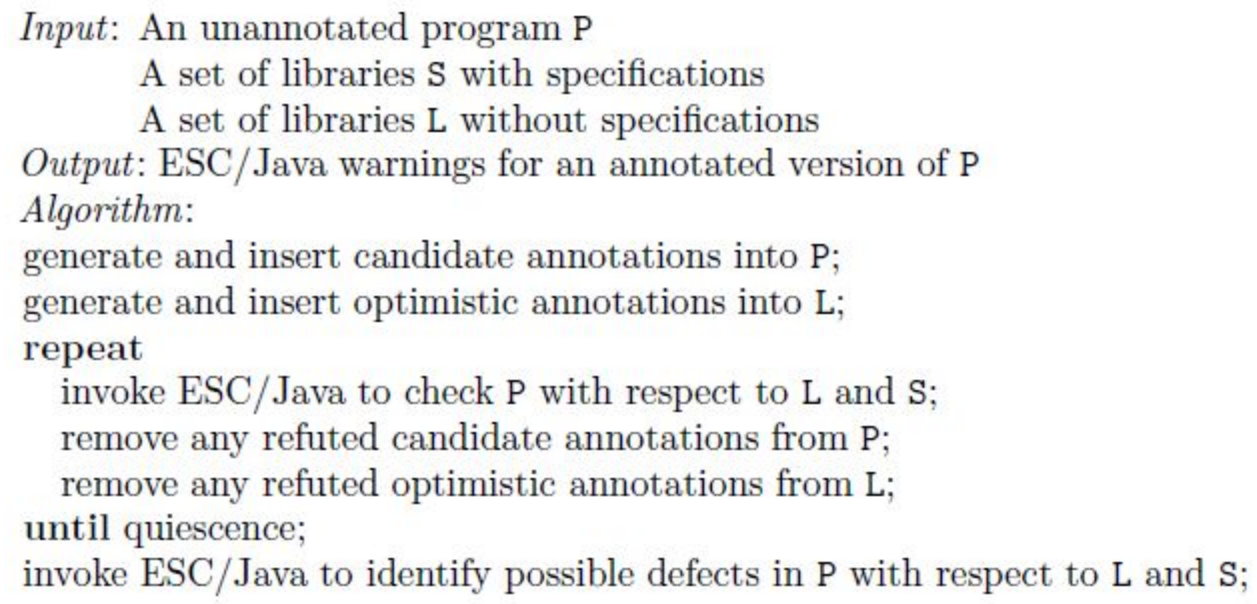
- The postconditions for the library methods are:



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

array type	`//@ ensures \result != null;`
	 `//@ ensures \nonnullelements(\result);`

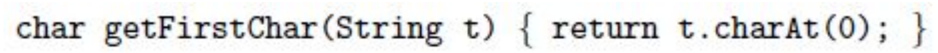
The modified Houdini algorithm



```
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;
```

Usage of Houdini

- To catch defects using Houdini a user starts by inspecting the list of warnings that Houdini produces. For example, if one of the warnings points out a possible null dereference of t in the following method.



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

- Houdini generates a collection of HTML pages where the source code and the annotations are displayed with refuted annotations greyed out. The greyed out annotations hyperlink to the source line where ESC/Java issued the warning that refuted the annotation.

- In the example above, the precondition `t != null` would be greyed out and clicking the annotation would redirect to a call site where as far as ESC/Java could tell the parameter of `t` may be null.

Issues with Houdini

- 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. Particularly, Houdini does not guess disjunctions.

- Another cause of false alarms 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 and the incompleteness of ESC/Java's annotation language.

- The user will have to insert annotations manually or use pragmas to handle the false alarms

Conclusion

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

Extra: Unsoundness in ESC/Java

Taken from Appendix C of ESC/Java user's manual

- As ESC/Java is an extended static checker rather than a program verifier, some degree of unsoundness is incorporated into the checker by design, based on tradeoffs with other properties like false alarms (incompleteness), efficiency etc.

- **Trusting Pragmas**: The `assume`, `axiom` and `nowarn` pragmas allow the user to introduce assumptions in the checking process. If the assumptions are invalid, the checking can miss errors. While Houdini does not use these pragmas in its grammar, running Houdini with these annotations in the source could cause issues, as it is using ESC/Java as a subroutine.

- **Loops:** ESC/Java does not consider all possible execution paths through a loop. It considers only those that execute at most one iteration plus the test for being finished before the second iteration. This avoids the need for loop invariants but is unsound.

Unsoundness in Loops

For example, if the program being checked has the fragment.

```
//@ loop_invariant E;
while (B) {
  S
}
```

ESC/Java will consider one execution of

```
//@ assert E;       // but giving a LoopInv warning
if (!(B)) break;
S
```

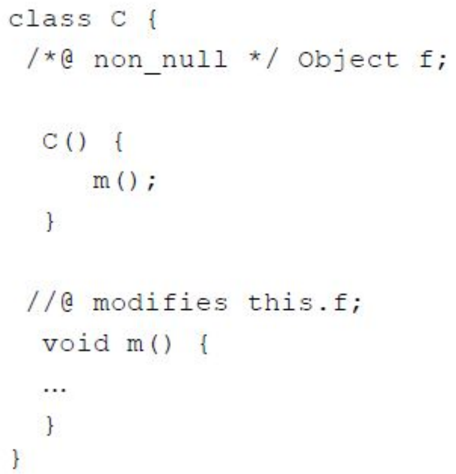
plus an additional execution of

```
assert E;             // but giving a LoopInv warning
if (!(B)) break;
```

This behaviour can be changed, at the cost of efficiency, by adding more loop unrolling using the `-loop` command line option (but only to a finite extent)

Unsoundness in ESC/Java

- **Arithmetic Overflow:** ESC/Java reasons about integer arithmetic as though machines integers are of unlimited magnitude. This is a source of both unsoundness and incompleteness
- **Ignored Exceptional Conditions:** ESC/Java checks for conditions that could give rise to NullPointerException, IndexOutOfBoundsException, ClassCastException, ArrayStoreException, ArithmeticException, or NegativeArraySizeException. It ignores other cases where instances of unchecked exception classes (eg OutOfMemoryError, StackOverflowError, ThreadDeath, SecurityException) might be thrown, except by explicit throw statements.
- **Initialization of fields declared non_null:** Consider the following program



```
class C {
 /*@ non_null */ Object f;

  C() {
     m();
  }

 //@ modifies this.f;
  void m() {
  ...
  }
}
```

When checking the implementation for the constructor for `c`, ESC/Java will assume that the method `m()` returns with `this.f` set to a non null value.
While checking the body of `m`, ESC/Java will check that any assignments to `f` assign non-null values.
However if the body of m can complete normally without assigning to `this.f` then ESC/Java's assumption will be unsound.

Incompleteness in ESC/Java

- The incompleteness of Simplify (Theorem prover for ESC/Java) goes beyond the undecidability of extended static checking.
  - Simplify has no built in semantics for multiplication except by constants
  - Simplify does not support mathematical induction

- Incomplete modeling of Java semantics
  - ESC/Java's built-in semantics for floating point operations are extremely weak – not strong enough to prove `1.0 + 1.0 == 2.0`
  - ESC/Java treats exceptions thrown by the run-time system as errors, even in programs that include code to catch them
  - ESC/Java's built-in semantics for strings are quite weak – strong enough to prove `“Hello World” != null` but not strong enough to prove the assertion `c == ‘l’` after the assignment `c = “Hello World”.charAt(3)`

Thank You