DySy: Dynamic Symbolic Execution for Invariant Inference

Christoph Csallner, Nikolai Tillmann, Yannis Smaragdakis

Dynamic Invariant Inference

DySy: an algorithm and tool for invariant inference using symbolic execution

Likely program invariants
= concrete execution of actual test cases
+ simultaneous symbolic execution over tests

Dynamic Invariant Inference

Drastically increases the relevance of inferred invariants or reduces the size of the test suite required for good results

Succintly summarizes both **expected** program inputs and the subset of program behaviours normal under those inputs

Motivation

An invariant inference system observes program properties that hold at pre-selected program points, with the help of a test suite for application functionality.

- method preconditions/postconditions
- object state

Motivation

```
int testme(int x, int y) {
        int prod = x * y;
    if (prod < 0)
        throw new ArgumentException();

    if (x < y) {
        int tmp = x;
        x = y; y = tmp;
    }

    int sqry = y * y;
    return prod * prod - sqry * sqry;
}
```

Motivation

Symbolic execution observes that:

- if `x*y >= 0`, then `prod = x*y`
- the `x < y` condition is also accumulated
- the expression `y*y*x*x - x*x*x*x` is returned

Motivation

Collected over multiple test cases, we get:

- precondition `x * y >= 0`
- a fully collected postcondition

```
\result == ((
  (x < y) -> y*y*x*x - x*x*x*x)
  else -> (x*x*y*y - y*y*y*y))
```

Motivation

An invariant inference system observes program properties that hold at pre-selected program points, with the help of a test suite for application functionality

- method preconditions/postconditions
- object state

These invariants do not reflect only the behaviour of the program, but also the assumptions and expectations of the test suite.

Daikon

(Michael D. Ernst et al, 2007)

Dynamic invariant inference via pre-set collection of invariant templates

Instantiated to produce **candidate** invariants under examination

Expansion of template library is possible, but the instantiation grows combinatorically

Daikon

(Michael D. Ernst et al, 2007)

Instruments the program

Executes it (production use or regular testing)

Analyzes the produced execution traces

At every method entry/exit, instantiates all invariant templates and tries them out

Summarizes behaviour observed in the traces as invariant and generalizes to other cases

Contributions

- DySy, a tool built on top of the Pex Framework for instrumentation and symbolic execution of .NET programs.
- an algorithm for dynamic invariant inference that can infer deep invariants (e.g. method purity)
- evaluation on Daikon's showcases, inferring a subset of the interesting invariants found with Daikon, while eliminating multiple irrelevant or accidental ones

Inference via Path Conditions

Path condition:

- the result of symbolic execution
- collection of branch conditions for the whole program
- always expressed in terms of program inputs

Inference via Path Conditions

Refine symbolic execution by executing tests.

At the end of tests, the overall path condition is the **precondition** of the program under test

Symbolic values of externally observable variables provide **dynamically inferred postconditions**

Symbolic conditions for all methods of a class become the **class state invariants**

This technique is not...

- lifting conditions from program text and postulating them as invariants
  - has conditions which do not exist in the program as the result of abstraction via symbolic execution over tests
- invariant inference *through* static techniques (abstract interpretation, symbolic execution)
  - missing dynamic nature
- concolic execution

DySy Overview

For a method under examination, all class instance variables, the method's parameters, and the method's result are treated as symbolic variables

Path conditions in symbolic execution are determined purely by the paths taken in the concrete execution.

DySy Overview

When executing a single test case:

- the path condition at the end of the symbolic execution is the symbolic condition for the path of the program, and thus is a **precondition** for that execution
- the symbolic values of the method's result and spawned object instance variables are the **postcondition** for that method

The combined conditions over all test cases are simplified through symbolic reasoning.

DySy Overview

Combining preconditions is done by disjunction of the individual test case preconditions.

Combining postconditions is done by conjunction of the individual test case postconditions.

The number of disjuncts is bounded by the number of program paths under examination.

Symbolic reasoning (eg. theorem prover) is needed to simplify terms into tautologies.

DySy Under the Hood

Pex: dynamic analysis and test generation framework for .NET

Monitors execution of program through instrumentation.

Every actual execution gets a dedicated "shadow interpreter", and every actual instruction callbacks to this shadow interpreter to follow symbolically.

DySy Under the Hood

If a method makes no state updates except local variables of new stackframes and instance fields of newly created objects, it is considered **pure**.

All pure methods are abstracted into terms representing the call.

All recursive calls are abstracted away independent of their purity (like a normal form).

Native calls are not monitored.

DySy Under the Hood

Special treatment for explicit loop behaviour:

- loop iteration variables are treated as symbolic
- loop exit condition does not become part of the path condition if the loop body is entered at all
- effectively: a loop becomes an `if` statement with the symbolic conditions in the body of the loop collapsed per program-point

Evaluation

DySy much slower than Daikon (28s vs 9s)

To detect an approximately equal number of ideal invariants (27), Daikon created 138 intermediate invariants

Daikon created three times more unique subexpressions (a relevant metric for measuring the actual throughput in invariants) than DySy for the same number of ideal invariants

Evaluation

Daikon sometimes creates really weird invariants which DySy **doesn't** do.

Relating the type of the array with the types of elements it holds:

```
\old(this.topOfStack) >= 0) ==>
  (this.array.getClass() != \result.getClass())
```

Evaluation

Daikon sometimes creates really weird invariants which DySy **doesn't** do.

Relating the `topOfStack` variable with the stack's default capacity using a bit-shift operation (?!):

```
\old(this.topOfStack) >= 0) ==>
  ((\old(this.topOfStack) >> stack.DEFAULT_CAPACITY == 0))
```

Conclusion

> In this paper we presented an approach that holds promise for the future of dynamic invariant inference:
> using symbolic execution, simultaneously with concrete test execution in order to obtain conditions for invariants.
> We believe that this technique represents the future of dynamic invariant inference.