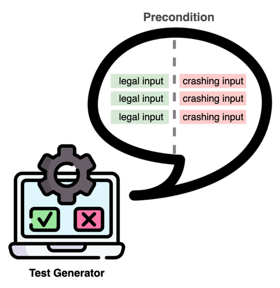
Inferring Natural Preconditions via Program Transformation

Elizabeth Dinella, Shuvendu Lahiri, Mayur Naik

Motivation

Preconditions separate legal inputs from illegal inputs. Existing approaches for generating preconditions often infer predicates which are unnatural and difficult to comprehend. State of the art approaches combine "features" (e.g. `x > 0`, `foo.size() > 0`) from scratch to construct a boolean predicate which separates crashing inputs from non-crashing inputs. The resulting predicate can become unnecessarily complex, difficult to comprehend, and ultimately, unnatural. In contrast, **our approach performs program transformations to the target method to infer natural preconditions as segments of code.**




```
public BigInteger[] DivideAndRemainder(BigInteger val) {
  if (val.m_sign == 0)
    throw new ArithmeticException("Division by zero error");
  BigInteger[] biggies = new BigInteger[2];
  if (m_sign == 0) {
    biggies[0] = Zero;
    biggies[1] = Zero;
  } else if (val.QuickPow2Check()) {
    int e = val.Abs().getBitLength() - 1;
    NetBigInteger quotient = Abs().ShiftRight(e);
    int[] remainder = LastNBits(e);
    biggies[0] =
        val.m_sign == m_sign ? quotient : quotient.Negate();
    biggies[1] =
        new BigInteger(m_sign, remainder, true);
  } else {
    int[] remainder = (int[]) m_magnitude.clone();
    int[] quotient = Divide(remainder, val.m_magnitude);
    biggies[0] =
        new BigInteger(m_sign * val.m_sign, quotient, true);
    biggies[1] =
        new BigInteger(m_sign, remainder, true);
  }
  return biggies;
}
```

```
boolean M_pre(NetBigInteger val) {
  if (val == null)
    return false;
  if (val.m_sign == 0)
    return false;
  return true;
}
```

(a) **Generated by our approach.**

```
(not (val == null))
and (
    (val.getIntValue() == val.m_sign and (
       (val.getIntValue() <= -1) or
       ((not (val.getIntValue() <= -1)) and
           (not (val.getIntValue() <= 0))
       ))
    ) or
    (not (val.getIntValue() == val.m_sign))
)
```

(b) **Generated by Proviso.**

Fig. 1. **A method and its corresponding preconditions.**

Problem Statement

Given an exhaustive tester T which generates a set of input environments E, we aim to infer an M_pre with the following correctness properties for every $e \in E$:

M_pre(e) $\rightarrow$ *True* iff M exits normally on e and
M_pre(e) $\rightarrow$ *False* iff M throws an exception on e.

Solution: Infer a precondition as a boolean returning method seeded by the target method.

Seed generator: Our technique begins by creating a seed through an up-front source transformation on the target method M. We begin with the method body and make transformations to satisfy our problem formulation, requiring M_pre be a non-exceptional, boolean returning function. Furthermore, our seed generation process makes semantics preserving transformations for precise exception check insertion and syntax-guided reduction in later stages. The source transformation is designed such that the seed has the following desirable qualities:

1. M_pre must be boolean returning
2. M_pre must be non-exceptional
3. M_pre localizes crashes

```
BigInteger Sqrt() {
   if (m_sign < 0)
      throw new ArithmeticException();

   return Round(Sqrt(this));
}
```

(a) **Target method M.**

```
boolean Sqrt_pre_init() {
   if (m_sign < 0)
      return false;

   BigInteger var_a = Sqrt(this);
   Round(var_a);
   return true;
}
```

(b) **M after seed transformation.**

```
public boolean Sqrt_pre() {
    if (m_sign < 0)
        return false;
    try {
        Round(Sqrt(this));
    } catch (Exception e) {
        return false;
    }
    return true;
}
```

(a) **M_pre without call normalization.**

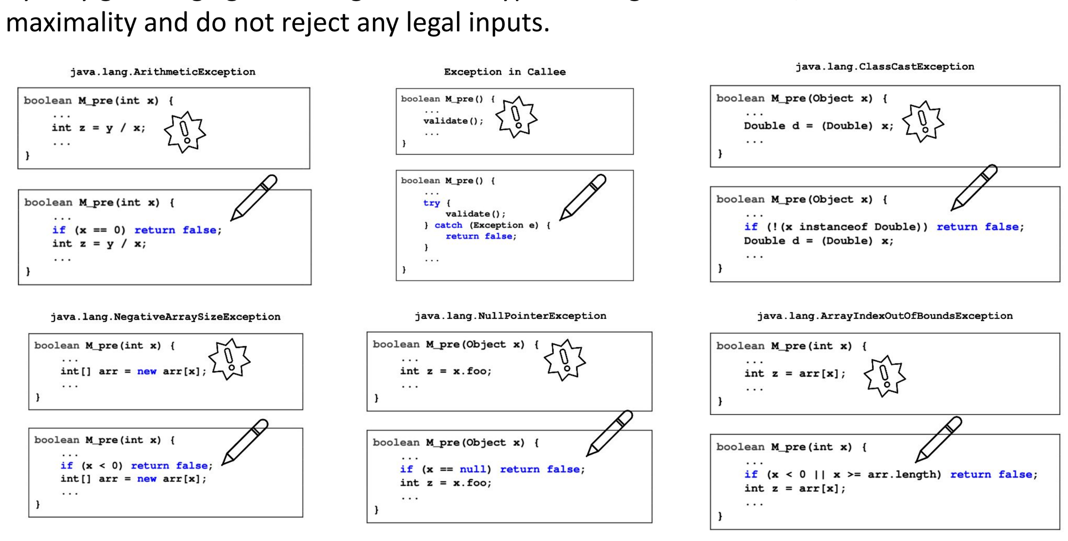
```
public boolean Sqrt_pre() {
    if (m_sign < 0)
        return false;
    try {
        Sqrt(this);
    } catch (Exception e) {
        return false;
    }
    return true;
}
```

(b) **M_pre with call normalization.**

Illustration of call normalization in seed generation.

Call normalization: The process of call normalization is essential for crash localization during the next phase of transformation. The normalization transformation lifts each call to its own source line. Call normalization ultimately results in a more readable precondition. Without call normalization it is not clear whether the exception is occurring in the method `Sqrt` or the method `Round`. On the other hand, performing call normalization ocalizes the exception in `Sqrt`. By localizing the crash we reduce the cognitive load of interprocedural inspection of the exceptional callee.

Check Instrumentor: Here, we describe the process for inserting `false` returning guards prior to crashes found by the test generator. These are inserted such that M_pre will exit normally on an illegal input rather than throwing an exception. The check instrumentor parses a stack trace produced from the execution of the current tests. The stack trace provides a crash type and location, which allows us to make precise AST transformations. By only guarding against the given crash type at the given location, we maintain maximality and do not reject any legal inputs.

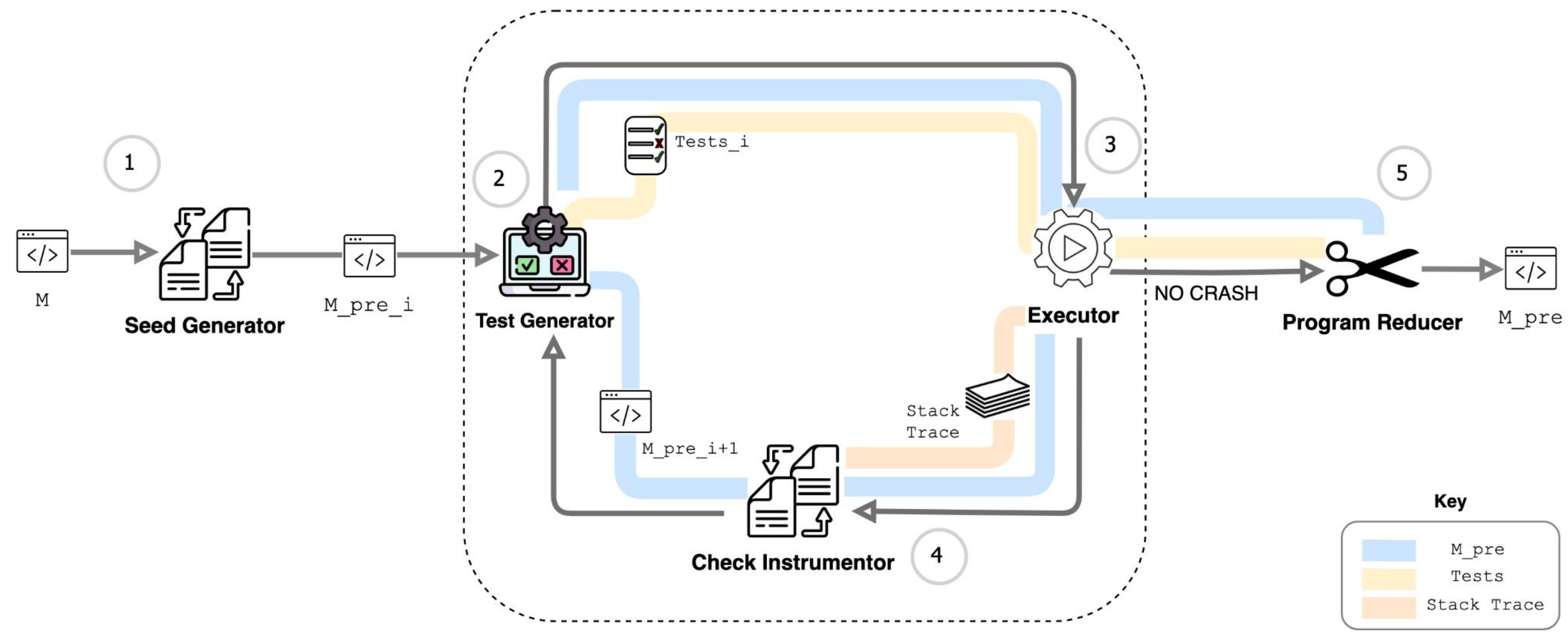


We define six transformations which guard against 99% of the crashes the test generator found on 87 real-world Java projects. Our technique performs AST transformations according to the Algorithm 1. This algorithm works in synergy with our seed generation as it expects localized statements to match the given line number.

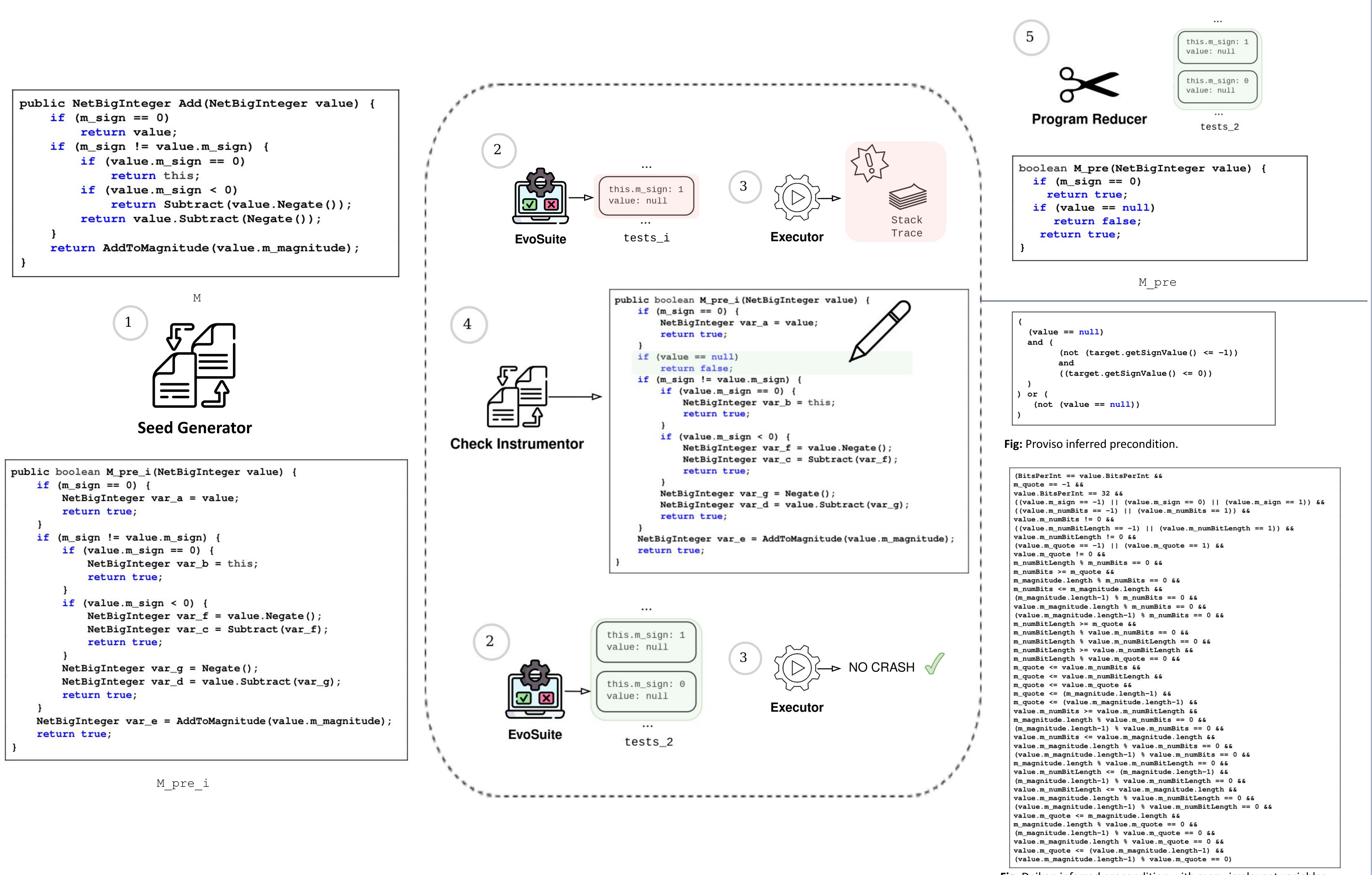
Algorithm 1 Transformation to replace a crashing program fragment on Line L with a new program fragment that guards the crashing exception type.

```
procedure CheckInstrumentorAlgorithm(Exception Type T, Line L, bool isInCallee)
    S ← statement on line L
    P ← S
    if isInCallee then                                        ▷ Interprocedural Exception
        P ← try { S } catch (T exc) { return false; }
        return;
    end if
    Let (expr_1, ..., expr_n) be the list of all expressions in S in the normal order of execution
    for i = 1 : n do
        switch T do
            case java.lang.NullPointerException
                if expr_i is a field access expression of the form object_expr.field then
                    P ← if (object_expr == null) return false;  P
                end if
                if expr_i is an array access expression of the form array_expr[index_expr] then
                    P ← if (array_expr == null) return false;  P
                end if
            case java.lang.ArrayIndexOutOfBoundsException
                if expr_i is an array access expression of the form array_expr[index_expr] then
                    P ← if (index_expr < 0 || index_expr >= array_expr.length) return false;  P
                end if
            case java.lang.ClassCastException
                if expr_i is a class cast expression of the form (cast_type) expr_to_cast then
                    P ← if !(expr_to_cast instanceof cast_type) return false;  P
                end if
            case java.lang.NegativeArraySizeException
                if expr_i is an array creation expression of the form new array_expr[index_expr] then
                    P ← if (index_expr < 0) return false; P
                end if
            case java.lang.ArithmeticException
                if expr_i is a binary operator DIVIDE expression of the form numer/denom then
                    P ← if (denom == 0) return false; P
                end if
    end for
end procedure
```

Technique



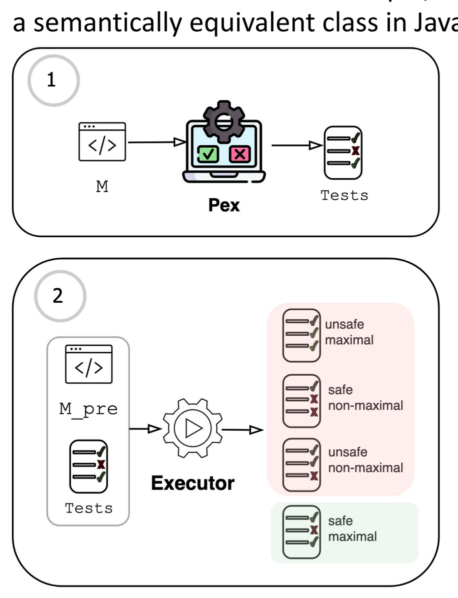
Example



Evaluation

We perform an in-depth comparative evaluation to the state-of-the-art approach on a single real-world project. Here, we evaluate and compare the resulting preconditions inferred by both approaches on two aspects. We aim to answer the following research questions:
RQ1: Correctness: Are the inferred preconditions *safe* and *maximal*?
RQ2: Naturalness: Can humans easily reason over the inferred preconditions?

Experimental Setup:
Baseline. We evaluate in comparison to Proviso and Daikon as they are the state-of-the-art and instantiated in C# and Java which are similar to our target language.
Benchmark. We evaluate on 39 (M, M_pre) pairs from the `NetBigInteger` C# project. In order to evaluate our technique, we manually translate the `NetBigInteger` class to a semantically equivalent class in Java.



RQ1: Correctness. Following the definition of correctness in Proviso, we evaluate the safety and maximally of our inferred preconditions on the benchmark, modulo a test generator. We perform this evaluation for the 39 preconditions inferred by our approach.

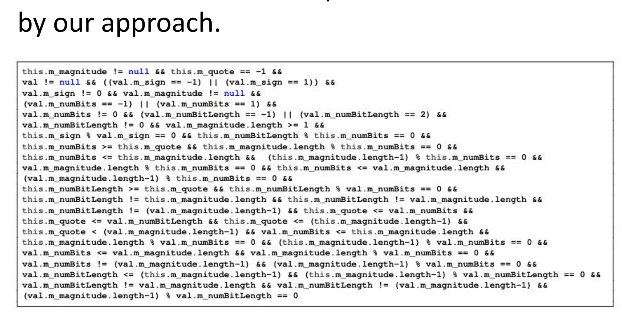


Fig: Daikon inferred precondition with many irrelevant variables.

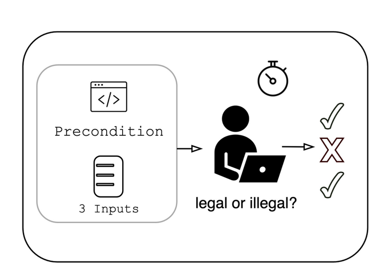
	Safe	Maximal	Correct	Total
Daikon	25	10	6	39
Proviso	37	34	34	39
Ours	33	34	29	39

Table 2: Correctness of Inferred Preconditions.

Through manual inspection, we find that the 10 incorrect preconditions our approach infers are due to EvoSuite incompleteness. The 33 incorrect preconditions Daikon infers are also due to incompleteness as well as including irrelevant variables and relations.

Result 1: Our approach infers correct (safe and maximal) preconditions for **29 of the 39 methods** in the benchmark. The 10 incorrect preconditions were due to EvoSuite incompleteness. Proviso inferred 34 correctly, while Daikon only correctly inferred 6.

RQ2: Naturalness. To better understand if our inferred preconditions are natural, we study a human's ability to reason over its behavior. We compare to Proviso, by conducting a user evaluation of 44 users including computer science PhD students, undergraduates, and industry software engineers split evenly between two groups.



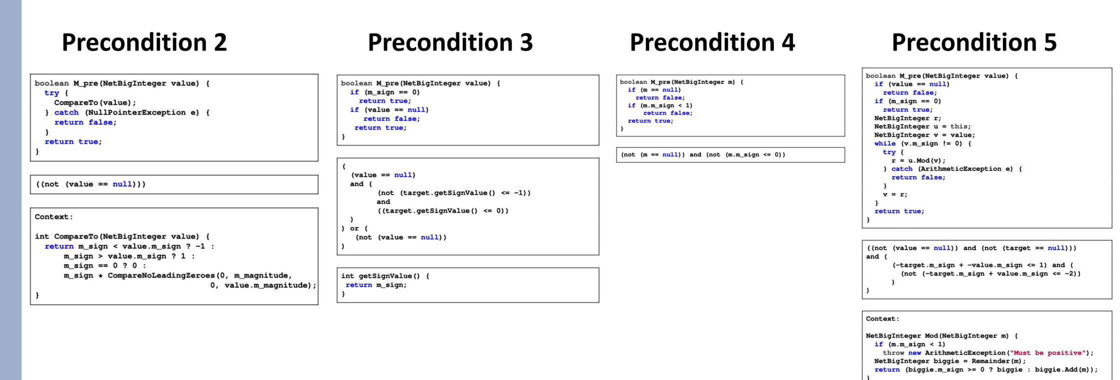
Each user is asked to review a given precondition and three inputs. We ask the user to classify each input as legal or illegal. The accuracy of their answers as well as the time taken to derive the answer are metrics of how natural or easy the precondition is to reason over.

	Accuracy		Time Taken (Sec)	
	Ours	Proviso	Ours	Proviso
Precondition 1	97.78%	77.78%	92.23	309.49
Precondition 2	80.56%	82.05%	250.43	82.19
Precondition 3	100%	73.33%	104.14	216.76
Precondition 4	97.22%	84.85%	61.77	68.38
Precondition 5	68.63%	80.56%	245.88	348.09
Overall	**88.84%**	79.71%	**150.89**	204.98

Table I: **User study results.**

User Study Design
We identify 5 preconditions from our evaluation to use filtered by the following criteria:

1. Both our approach and Proviso infer a correct precondition.
2. The precondition inferred by the tools are syntactically different.
3. The precondition is non-trivial (there exists at least 1 illegal and 1 legal input).



Result 2: On average, users were able to more accurately reason over our preconditions in a shorter time span. Our results were not as strong on preconditions which included interprocedural try-catch blocks.

Contact: edinella@brynmawr.edu