

Loop Invariants by Mutation, Dynamic Validation, and Static Checking*

Juan Pablo Galeotti¹, Carlo A. Furia², Eva May¹,
Gordon Fraser³, and Andreas Zeller¹

¹ Software Engineering Chair, Saarland University, Saarbrücken, Germany
lastname@cs.uni-saarland.de

² Chair of Software Engineering, Department of Computer Science, ETH Zurich, Switzerland
caf@inf.ethz.ch

³ Department of Computer Science, University of Sheffield, UK
gordon.fraser@sheffield.ac.uk

Abstract

Some of the intrinsic limitations of the widely used static techniques for loop invariant inference can be offset by combining them with dynamic techniques—based on executing automatically generated tests. We show how useful loop invariant candidates can be generated by systematically mutating postconditions; then, we apply dynamic checking to weed out invalid candidates, and static checking to select provably valid ones. We present a framework that automatically applies these techniques to carry out functional correctness proofs without manually written loop invariants. Applied to 28 methods (including 39 different loops) from various `java.util` classes, our DYNAMATE prototype automatically discharged 97% of all proof obligations, resulting in automatic complete correctness proofs of 25 out of the 28 methods—outperforming several state-of-the-art tools for fully automatic functional verification of programs with loops.

1 Introduction

Loop invariants are a fundamental ingredient to prove properties of programs with loops. Compared to other kinds of annotations, they tend to be difficult to formulate and understand, and even structurally simple loops may require non-trivial invariants to be proved correct. This has motivated a whopping amount of research on inferring loop invariants, and on using inferred invariants to automate proofs of formal correctness.

As we summarize in [Section 7](#), the overwhelming majority of techniques for loop invariant inference are *static*, based on abstract interpretation, constraint-based reasoning, or theorem proving. Static techniques offer clear advantages in terms of *speed* (and hence scalability) and *soundness*. The latter is, in particular, an imperative requirement for loop invariants used for correctness proofs. Nonetheless, static techniques have their own shortcomings, which become particularly critical when one tries to apply them to real-world full-fledged programming language implementations (as opposed to in-the-small algorithmic examples):

- supporting advanced language features (such as calls to external pre-compiled libraries) may reduce *precision* or require manual annotations;
- *robustness* with respect to implementation changes that do not affect functionality (for example, switching from arrays to linked lists) may be difficult to achieve since static techniques typically rely on data structure-specific abstract models;

*An extended version of this summary is currently under consideration for refereed conference publication.

- reliance on hardwired abstract models also typically restricts the applicability of a static inference techniques to predefined templates over well-defined domains (such as scalar inequalities or polynomial constraints), which are often only a subset of what is required to express *functional correctness* properties.

In this paper, we present a novel approach to loop invariant inference that supplements static techniques with *dynamic* techniques—based on executing automatically-generated test cases. Using static techniques ensures that the inference is sound, while dynamic techniques ameliorate some of the aforementioned shortcomings. The key observation is that, given loop invariant *candidates* (assertions that may or may not hold for the loops under analysis), executing tests can efficiently weed out *invalid* candidates, whereas static verification can promptly *validate* candidates—and possibly use them to prove the program correct. Thus, if we can *produce a suitable set of plausible loop invariant candidates*, we have chances to prove the program correct in a fully automatic fashion, without need for manual loop invariant annotations.

A crucial difficulty in producing useful loop invariant candidates is being able to support diverse and expressive functional properties, without being restricted to limited domains defined a priori. To address this issue, in previous work [10, 11] we introduced the idea of *guessing invariant candidates based on mutations of the postcondition*. The rationale for this idea is the observation that a loop drives the program state towards a goal characterized by the postcondition; a loop invariant, which characterizes all program states reached by the loop, can then be seen as a weakened form of the postcondition. Mutating the postcondition is thus a way to obtain possible loop invariant candidates. The fundamental challenge in implementing this idea as a practically applicable technique is dealing with the blow-up in the number of candidates: generation is exponentially expensive and too many useless candidates bog down the program verifier with irrelevant or incorrect information. In this paper, we show how to address this challenge, again by combining static and dynamic techniques. This way, the static program verifier only has to deal with a small, manageable set of likely useful loop invariants, while the invariant generation process remains flexible as it is not restricted to fixed assertion patterns or algorithm implementation details.

We implemented our approach in a tool called DYNAMATE¹, which provide fully automatic verification for Java programs with loops without requiring manually annotated loop invariants. We conducted a case study that applied DYNAMATE to 28 methods from the `java.util` classes in the Java standard library, including the `binarySearch0` method. DYNAMATE discharged 97% of the proof obligations of all the methods, resulting in full correctness proofs for 25 of the 28 methods. As we show in Section 6.2, these results correspond to an over 20% improvement over state-of-the-art tools for automatic verification, such as the CodeContracts static checker `cccheck`, in terms of number of automatically discharged assertions.

2 An Example of Using DYNAMATE

Our tool DYNAMATE inputs Java methods complete with pre- and postcondition specifications written in JML [17]. For example, Figure 1 shows `binarySearch0`, a helper method declared in class `java.util.Arrays` in the standard Java API, with a JML specification expressing that, if the element `key` is found in the input array `a`, its index is returned, otherwise a negative integer is returned. The specification uses model-based *predicates* [22] with descriptive names, such as `¬TArrays.has(a, fromIndex, toIndex, key)` to denote that array `a` has no element `key` over the interval range from `fromIndex` (included) to `toIndex` (excluded).

¹DYNAMATE = “Dynamic Mining and Testing”

```

1    /*@
2    @ requires a ≠ null;
3    @ requires TArrays.within(a, fromIndex, toIndex);
4    @ requires TArrays.sorted(a, fromIndex, toIndex);
5    @
6    @ ensures \result ≥ 0 ⇒ a[\result] = key;
7    @ ensures \result < 0 ⇒ ¬TArrays.has(a, fromIndex, toIndex, key);
8    @*/
9    private static int binarySearch0
10       (int[] a, int fromIndex, int toIndex, int key) {
11       int low = fromIndex, high = toIndex - 1;
12       while (low ≤ high) {
13         // midpoint of [low..high]
14         int mid = (low + high) >>> 1;
15         int midVal = a[mid];
16         if (midVal < key) low = mid + 1;
17         else if (midVal > key) high = mid - 1;
18         else return mid; // key found
19       }
20       return -(low + 1); // key not found
21     }

```

Figure 1: Binary search method in `java.util.Arrays` annotated with pre- and postcondition.

```

22    //@ loop_invariant fromIndex ≤ low
23    //@ loop_invariant low ≤ high + 1
24    //@ loop_invariant high < toIndex
25    //@ loop_invariant ¬TArrays.has(a, fromIndex, low, key)
26    //@ loop_invariant ¬TArrays.has(a, high + 1, toIndex, key)

```

Figure 2: Loop invariants required for verifying method `binarySearch0`.

Fully automatic verifiers such as `cccheck` [5] or `BLAST` [1] cannot prove the correctness of the program in Figure 1 against its specification; nor can auto-active verifiers such as `ESC/Java2` [4]² unless we provide suitable invariants for the loop—for example, those in Figure 2. `DYNAMATE`, instead, can automatically infer the necessary loop invariants and feed them to `ESC/Java2`, so as to verify `binarySearch0` without additional user input besides what shown in Figure 1.

`DYNAMATE` implements an approach that combines different techniques as illustrated in Figure 3. We first run a static verifier on the program code and its specification. If the verifier fails to prove the program correct, we start a round of our approach:

1. To support dynamic invariant detection, we feed the program code and its specification into a *test case generator*, which generates executions of the program that satisfy the given precondition. (In the `binarySearch0` example, a possible test case searches for 5 in the array `[0, 1, 2]` and returns the index -3.)

²In this work, we always use `ESC/Java2` with options for sound verification of unbounded loops (`-loopsafe`).

2. From the resulting executions, we dynamically mine candidates for loop invariants. Our *invariant detector* tools *variable* and *specific patterns* derived from the given postcondition. The patterns systematically determine candidates; only those candidates that hold for all executions are retained and go to the next stage. (This step generates: `high < toIndex` using the generic patterns; and `¬TArrays.has(a, high, toIndex, key)` from the postcondition. Neither is invalidated by the test case with `a = [0, 1, 2]` and `key = 5`, but the latter candidate does not hold in general.)
3. The surviving set of loop invariant candidates is then fed into a static *program verifier*. The verifier may confirm that some candidates are valid loop invariants; and it may eventually produce a proof, which concludes that the program is correct with respect to its specification. (The program verifier confirms that `high < toIndex` is a valid loop invariant, but rejects the other candidate.)
4. If the proof does not succeed after adding the inferred loop invariants, we *start another round* of generating, mining, and verifying. If the verifier has left loop invariants unproven, the test generator searches for executions that falsify them, thus refining the set of candidates for the next proof attempt. (A test case searching for 2 in `[0, 1, 2]` reveals that `¬TArrays.has(a, high, toIndex, key)` is invalid: it does not hold initially when `high = toIndex - 1`.)

Section 3 gives an overview of the algorithms implemented in DYNAMATE; Section 4 describes each component in detail, and in particular the interplay of static and dynamic techniques. A key functionality in DYNAMATE is GIN-DYN, a flexible mechanism for generating loop invariants; GIN-DYN’s algorithms are described in Section 5.

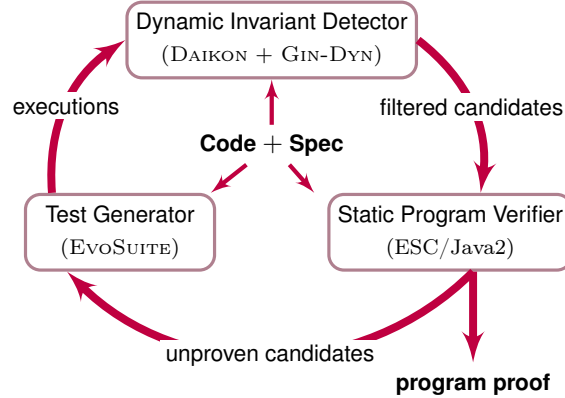


Figure 3: How DYNAMATE works. The program code (center) is first fed into a test case generator (left), which generates executions covering legal behavior. From these, two dynamic invariant detector tools (top) mine possible loop invariants, based both on fixed patterns (DAIKON) as well as postconditions (GIN-DYN). The candidates not invalidated by the generated runs are then fed (together with code and specification) into a symbolic program verifier (right). The verifier then may produce a program proof (bottom right), but may also refute candidates, which initiates another round of executions, and thus refined invariants.

3 Overview of the DYNAMATE Algorithm

DYNAMATE builds on an interplay of three components: a test case generator, a dynamic invariant detector, and a static verifier.

The DYNAMATE algorithm, illustrated in Algorithm 1, inputs a program M and its specification—a precondition P and a postcondition Q . Two outcomes of the algorithm are possible: *success* means that DYNAMATE has found a set of valid loop invariants that are sufficient to statically verify M against its specification (P, Q) ; *failure* means that DYNAMATE cannot find new valid loop invariants, and those found are insufficient for static verification. Even in case of failure, the valid loop invariants found by DYNAMATE have a chance of enabling *partial* verification by discharging some proof obligations necessary for a correctness proof.

Algorithm 1 The DYNAMATE algorithm

Require: Program M , precondition P , postcondition Q

```

 $TS \leftarrow \emptyset$            (set of tests)
 $INV \leftarrow \emptyset$    (set of verified loop invariants)
 $C \leftarrow \emptyset$      (set of candidates)
while static verifier cannot prove  $(M, P, Q, INV)$  do
   $T \leftarrow$  execute test case generator on  $(M, P, C)$ 
   $TS \leftarrow TS \cup T$ 
   $I \leftarrow$  execute invariant detector on  $(M, TS)$ 
  if  $I$  has not changed then
    return (“failure”,  $INV$ )
  end if
   $\widehat{M} \leftarrow$  annotate  $M$  with inferred candidate invariants  $I$ 
   $J \leftarrow$  statically check valid invariants of  $(\widehat{M}, P)$ 
   $INV \leftarrow INV \cup J$ 
   $C \leftarrow I \setminus INV$ 
end while
return (“success”,  $INV$ )

```

In case the static verifier fails to prove the program, DYNAMATE’s main loop starts by executing the test case generator, which produces a new set T of test cases that exercise M with inputs satisfying the precondition P . Next, the loop feeds the overall set TS of test cases generated so far to the dynamic invariant detector, which outputs a set of candidate loop invariants I . We call them “candidates” because the invariant detector summarizes a finite set of runs, and hence it may report assertions which are not valid loop invariants.

To find out which candidates are indeed valid, DYNAMATE calls the static verifier on the program annotated with all candidates I ; the verifier returns a set of proven candidates J (a subset of I), which DYNAMATE adds to the set INV of verified loop invariants. Then, using the current INV , it calls the static verifier again, this time trying a full correctness proof of M against (P, Q) . If verification succeeds, DYNAMATE terminates with success. Otherwise, DYNAMATE’s main loop continues as long as the invariant detector is able to find new candidate invariants, to invalidate previous candidates, or both. The loop may also diverge, keeping on finding new candidate invariants until it runs out of resources.

Since we assume a static verifier that is sound but incomplete, unproven candidates in $I \setminus INV$ are not necessarily invalid. Thus, in case of failed proof, DYNAMATE relies on the test case generator to create executions of M that invalidate as many unproven candidates in C as

possible.

4 How DYNAMATE Works

Each run of DYNAMATE takes as input a Java method M with its functional specification consisting of precondition P and postcondition Q . While DYNAMATE can work with JML specifications in any form, we find it effective to follow the principles of the model-based approach to specification (introduced with JML [17] and developed in related work of ours [22, 23]). Specifications refer to underlying mathematical models of the data structures involved; a collection of predicates encapsulate the assertions relevant to the application domain and express them abstractly in terms of the underlying models. The specification of `binarySearch0` in Figure 1 follows this approach, as it uses the predicates `within`, `sorted`, and `has`, which are collected in a class `TArrays` providing notation to capture facts about integer array-like data structures.

4.1 Test Case Generation

While any test case generator could work with DYNAMATE, our prototype integrates EVOSUITE [9], a fully automatic search-based tool using a genetic algorithm.

Within DYNAMATE, EVOSUITE needs to generate test cases that satisfy preconditions written in JML. To this end, we instrument every method with an initial check that its precondition holds: if it does not, the instrumentation code throws an exception that terminates execution and makes the test case invalid. The instrumentation leverages the availability of a runtime semantics for all specification predicates—developed using the model-based style.

4.2 Dynamic Loop Invariant Inference

At the core of the DYNAMATE algorithm lies a component that detects “likely” loop invariants based on the concrete executions provided by the test case generator. The current DYNAMATE implementation relies on two modules with complementary functionalities. Simple “boilerplate” invariants (mostly involving numeric relations between scalar variables or basic array properties) come from DAIKON [6]. On top of it, GIN-DYN produces more complex invariants—involving the same predicates used in the specification—discovered by mutating postcondition clauses.

DAIKON’s and GIN-DYN’s invariants are complementary; for example, neither one suffices for a correctness proof of `binarySearch0`. DAIKON invariants are usually necessary as a basis to establish GIN-DYN invariants (for example, to prove a predicate is well-defined by ensuring absence of out-of-bound errors). Therefore, we run DAIKON first and introduce GIN-DYN invariants as soon as we reach a fixpoint in the overall DYNAMATE algorithm. This scheme is flexible and fosters an effective combination between loop invariant inference techniques with complementary advantages.

When the DYNAMATE algorithm reaches a fixpoint without finding a proof, it is a sign that DAIKON has run out of steam and more complex invariants are required to make progress. For example, `binarySearch0`’s loop invariants on lines 25–26 in Figure 2 are never produced by DAIKON.

To address this fundamental limitation, we provide GIN-DYN: a way to efficiently filter out a large amount of invalid or uninteresting invariant candidates. Section 5 describes in detail how GIN-DYN does the filtering, again based on a combination of dynamic and static techniques. GIN-DYN and DAIKON invariant candidates are used in the very same way within DYNAMATE.

4.3 Static Program Verification

In DYNAMATE, invariant candidates come from dynamic analysis; hence they are only educated guesses based on a finite number of tests. The DYNAMATE algorithm complements dynamic analysis with a *static* program verifier, which serves two purposes: (1) verifying loop invariant candidates, and (2) using verified loop invariants to carry out a conclusive correctness proof.

Verification of Loop Invariants. The DYNAMATE prototype relies on ESC/Java2 [4] to check candidates for invariance. Invariant candidates to be validated may have dependencies, that is proving that one candidate L_1 is valid requires to assume another candidate L_2 . To find a maximal set of valid loop invariants, DYNAMATE applies the HOUDINI algorithm [7], which first considers all candidates at once, and then iteratively removes those that can’t be verified until all surviving candidates are valid loop invariants.

Program Proof. At the end of every iteration, DYNAMATE uses the current set of valid loop invariants to attempt a correctness proof of the program against its specification. If ESC/Java2 succeeds, the whole DYNAMATE algorithm stops with success; otherwise, it begins another iteration with the goal of improving its invariant set. The first iteration of DYNAMATE on `binarySearch0` does not find a correctness proof, since DAIKON’s invariants are insufficient to prove the specification in Figure 1.

Unproved loop invariant candidates may be over-specific and hence unsound. In case ESC/Java2 fails to establish program correctness, DYNAMATE adds a conditional branch for every such unproven candidate, which helps direct EVOSUITE’s search towards falsifying the candidate—hence providing more accurate tests.

5 GIN-DYN: Loop Invariants from Postconditions

The postcondition Q of a method M characterizes the goal state of M ’s computation; a loop ℓ in M ’s body contributes to reaching the goal by going through a series of intermediate states, which ℓ ’s loop invariants characterize. Based on these observations [11], we suggested [10] the idea of systematically *mutating* Q to guess candidate invariants of ℓ . Applying these ideas in a fully automatic setting on realistic programs requires to address two fundamental challenges: (a) providing a sweeping widely-applicable collection of mutations, and (b) efficiently discarding a huge number of invalid and uninteresting mutations.

The GIN-DYN technique—presented in this section—addresses these challenges by following a three-step strategy:

1. Generate many *mutants* (i.e., variants) of Q ’s clauses, applying a predefined generic sequences of mutation operators.
2. Discard mutants that cannot be loop invariants of ℓ because at least one test case exercising ℓ violates them.
3. Further prune valid but uninteresting loop invariants by eliminating those mutants that are *tautologies*, that is that hold independent of the specific behavior of ℓ .

The mutants that survive both the validation and the tautology elimination phases are likely to be useful to establish Q by virtue of having been derived from it. The following subsections detail the three steps as they are available in DYNAMATE.

```

Q: \result < 0 ==> ¬TArrays.has(a, fromIndex, toIndex, key);
~> low < 0 ==> ¬TArrays.has(a, fromIndex, toIndex, key);
~> low < 0 ==> ¬TArrays.has(a, fromIndex, mid, key);

```

Figure 4: Mutations producing a valid but trivial loop invariant.

```

Q: \result < 0 ==> ¬TArrays.has(a, fromIndex, toIndex, key);
~> ¬TArrays.has(a, fromIndex, toIndex, key);
~> ¬TArrays.has(a, high, toIndex, key);
~> ¬TArrays.has(a, high + 1, toIndex, key);

```

Figure 5: Mutations producing a valid and useful loop invariant.

5.1 Generation of Mutants

The fundamental idea behind GIN-DYN is to syntactically mutate postcondition clauses by substituting an expression for another one. The mutation algorithm maintains a set $\mathcal{M}$ of *mutants* and, for each type $\mathbf{t}$, a set $\mathcal{E}_{\mathbf{t}}$ of available expressions of type $\mathbf{t}$. $\mathcal{M}$ initially consists of the clauses in postcondition Q , and $\mathcal{E}_{\mathbf{int}}$ and $\mathcal{E}_{\mathbf{int}[]}$ include the integer and integer array variables available in the current loop ℓ . The basic mutation operators (or just “mutations” for short) used by GIN-DYN are the following:

- **Substitution:** given a mutant $m \in \mathcal{M}$ and an expression $e \in \mathcal{E}_{\mathbf{t}}$ of type $\mathbf{t}$, replace by e any subexpression of type $\mathbf{t}$ in m . This generates as many mutants as are subexpressions of type $\mathbf{t}$ in m , which are added to $\mathcal{M}$.
- **Weakening:** given a mutant $m \in \mathcal{M}$ and a Boolean expression $b \in \mathcal{E}_{\mathbf{boolean}}$, add to $\mathcal{M}$ the mutants $b \implies m$ and $\neg b \implies m$.
- **Aging:** given a mutant $m \in \mathcal{M}$, replace by $e - 1$ or $e + 1$ any subexpression e of type $\mathbf{int}$ in m . This generates twice as many mutants as are subexpressions of type $\mathbf{int}$ in m , which are added to $\mathcal{M}$.

In addition to these operators, the mutation algorithm can extend the sets of available expressions, thus enabling a larger number of substitutions; in particular, **predicate extraction** is the process of adding to the set $\mathcal{E}_{\mathbf{boolean}}$ of Boolean expressions all predicates belonging to the same collection (such as `TArrays`) as those in the postcondition (see Section 4).

Figure 4 shows one example of applying two substitutions to `binarySearch0`’s second postcondition clause: `\result` is replaced with `low`; and `toIndex` is replaced with `mid`. The produced mutant is clearly not a useful loop invariant because its antecedent is always false in the loop’s context. In contrast, the mutations applied in Figure 5 produce one of `binarySearch0`’s required loop invariants: predicate extraction adds `¬has(a, fromIndex, toIndex, key)` to the set of available Boolean expressions; a first substitution uses it to replace the whole postcondition; one further `int` substitution of `high` for `fromIndex` and one application of aging (turning `high` into `high + 1`) determine a valid loop invariant which is also useful in the correctness proof.

The number of combined multiple substitutions dominates the combinatorial complexity of generating mutants: if we start with q postcondition clauses in $\mathcal{M}$, each having up to s subexpressions of some type $\mathbf{t}$, and we have e expressions of type $\mathbf{t}$ available, applying all possible n consecutive $\mathbf{t}$ -type substitutions builds a number of mutants in the order of $(qse)^n$. Therefore, for all but the most trivial cases it is unfeasible to exhaustively apply more than few multiple substitutions.

To mitigate this problem, the mutation algorithm combines multiple applications of mutation operators (possibly in combination with predicate extraction) to determine complex mutants that may significantly differ from the initial postcondition. GIN-DYN defines several mutation sequences, which we call *waves*; each wave ω combines a variable number of mutations, which are applied exhaustively starting with the postconditions and finally producing a set $\mathcal{M}_\omega$ of mutants. The current implementation defines 16 waves that cover the most useful patterns found in algorithms [11]; GIN-DYN executes one or more waves in each iteration of DYNAMATE’s main loop (Figure 3). For example, a wave producing the mutant in Figure 5 is the following: (a) *predicate extraction* based on the postconditions; (b) one *substitution* for every Boolean subexpression available; (c) one integer *substitution* for every integer expression available; and (d) one application of *aging*.

5.2 Validation of Mutations

Each wave ω produces a set $\mathcal{M}_\omega$ of mutants. Mutants are educated guesses; the large majority of them are not loop invariants of ℓ and must be discarded. To determine whether a mutant $\mu \in \mathcal{M}_\omega$ is a loop invariant, GIN-DYN injects a check of μ at the entry and exit of ℓ ’s body, runs the test cases available for ℓ (generated as in Section 4.1), and retains in $\mathcal{M}_\omega$ only the mutants that pass all tests.

Being based on dynamic techniques, the validation of mutations performed within GIN-DYN is provisional, in that validated mutants still have to pass muster with the static program prover before we can conclude they are valid loop invariants. One advantage of testing is that we can check mutants for invariance *independently* of one another. In contrast, static verifiers such as ESC/Java2 work modularly: they reason about the program state at the beginning of every loop iteration entirely in terms of the available loop invariants, and hence they may fail to confirm that a given assertion L is indeed a loop invariant if other invariants, necessary to prove L , are not provided. The dependency problem is particularly severe in the presence of nested loops, where there can be circular dependencies.

5.3 Tautology Elimination

Several of the mutants $\mathcal{M}_\omega$ that pass validation are trivial, in that they hold only by virtue of their logic structure—that is, they are tautologies, useless to prove the postcondition Q . For example, many mutants are implications with an identically false antecedent (such as the one in Figure 4). To identify and discard mutants that are tautologies, GIN-DYN uses ESC/Java2 as follows. For each mutant $\mu \in \mathcal{M}_\omega$, it builds a dummy method m_μ with the following structure:

```
public void mμ(t1 v1, t2 v2, ...) {
    //@ assume verified loop invariants
    //@ assert μ
}
```

The arguments of m_μ include all variables occurring in μ ; since ESC/Java2 reasons modularly, it makes no assumption about their values, which is tantamount to having them nondeterministically set. Then, an **assume** lists all loop invariants verified across all DYNAMATE iterations by ESC/Java2 (Section 4.3). Any formula that is a consequence of these already known invariants is redundant and should not be retained. Finally, an **assert** asks ESC/Java2 to establish the mutant μ in the given context; GIN-DYN retains μ only if ESC/Java2 cannot prove the **assert**, and hence μ is not logically derivable from the verified loop invariants. This way, the tautology elimination is sound and complete relative to the static verifier’s capabilities.

6 Case Study

We evaluated the DYNAMATE approach by running our prototype on 28 methods from the `java.util` package of OpenJDK Java 1.6. We initially considered all methods with loops in `java.util`; then, we conducted a preliminary assessment where we spent about three person-weeks trying to write specifications, as complete as possible. We dropped all methods for which we could not get ESC/Java2 to work with non-trivial specification. When overloaded methods operating on different basic types were available, we included the versions operating on `int` and on `Object`; the latter typically require more complex annotations since they have to deal with `null` values. Note that, although the data structure implementations of most methods in our case study use arrays, the model-based style is largely oblivious of this detail, and its abstraction ensures that the general DYNAMATE approach remains applicable in principle to other types of data structures.

All experiments were performed on a Ubuntu GNU/Linux system installed on a 2.0 GHz Intel Xeon processors and 2 GB of RAM. We set a timeout of 120 seconds to every invocation of EVOSUITE and of 180 seconds to every invocation of ESC/Java2. As EVOSUITE uses randomized algorithms, its results are not deterministic; to control for this, we repeated every complete experiment 30 times and report the mean of every measure.

6.1 Experimental Results

Table 1 summarizes the experimental results, where DYNAMATE automatically verified 25 of the 28 methods in at least one of the 30 repeated runs (indicated by a success rate above 0%). A success rate below 100% indicates methods for which EVOSUITE performed below par, and thus DYNAMATE reached a fix point in invariant discovery in some of the 30 repeated runs. Nonetheless, experiments with DYNAMATE normally have high repeatability: 21 of the 25 verified subjects have a success rate above 50%.

The success rate was 0% for the methods `Arrays.merge`, `Arrays.quickSortPartition`, and `Collections.sort`. For all of these methods DYNAMATE failed to infer all necessary invariants. The 4 missing invariants have a form that is neither among DAIKON’s templates nor GIN-DYN’s mutants. We repeated the experiments by manually adding the missing invariants; as expected, DYNAMATE successfully verified the methods.

Besides full functional verification, we also measured the percentage of proof obligations DYNAMATE was able to discharge. We counted 367 proof obligations constructed by ESC/Java2 for the annotated Java programs used in our evaluation; these include all pre- and postcondition checks, the class invariant checks, and also implicit checks for out-of-bound array accesses and `null` dereferencing. DYNAMATE managed to find a set of loop invariants such that 97% of the proof obligations were discharged on average. This dramatically decreases the number of loop invariants the user has to write manually.

The columns about GIN-DYN show the crucial role of dynamic analysis and tautology elimination to discard invalid and irrelevant mutants: while postcondition mutation generated nearly 2500 mutants on average (way too many for ESC/Java2 to handle), dynamic analysis and tautology elimination cut this number down to below 300 (i.e., $2410 \times (1 - 0.82) \times (1 - 0.34)$).

On average, 66% of DYNAMATE’s execution time is spent running EVOSUITE, 15% in GIN-DYN, 14% in ESC/Java2 and 6% in DAIKON. DYNAMATE’s average running time per method (45 minutes) is high compared to other dynamic techniques. There are ample margins to optimize the DYNAMATE prototype for better speed; in particular, using third-party components as black boxes is an obvious source of inefficiency.

Table 1: Experimental results for the case study. Column SUCCESS RATE displays the percentage of runs that found a full correctness proof. The numbers in the other columns are means over 30 DYNAMATE runs: the percentage of PROVEN proof obligations; the number of ITERATIONS of DYNAMATE; the number of PROVEN invariants (in parentheses, how many of them come from GIN-DYN) and how many more would be required for a successful proof (MISSING); the number of WAVES of GIN-DYN candidates considered, how many CANDIDATE invariants they generated, and what percentage of the candidates were FALSIFIED by dynamic analysis or removed in the TAUTOLOGY elimination phase; the TIME spent in TOTAL (seconds).

CLASS METHOD	SUCCESS		# # INVARIANTS				GIN-DYN				TIME SECS.
	RATE	PROVEN	ITERAT.	PROVEN	MISS.		WAVES	CAND.	FALS.	TAUT.	
ArrayDeque contains	57 %	97.95 %	7	14 (2)	0		10	438	98 %	39 %	2158 s
ArrayDeque removeFirstOccurrence	53 %	97.96 %	7	14 (2)	0		11	446	98 %	42 %	2180 s
ArrayDeque removeLastOccurrence	87 %	99.52 %	9	43 (11)	0		6	391	93 %	60 %	3281 s
ArrayList clear	70 %	95.22 %	6	9 (2)	0		6	26	79 %	0 %	1524 s
ArrayList indexOf	23 %	90.83 %	7	16 (3)	0		13	40	89 %	11 %	1914 s
ArrayList lastIndexOf	20 %	93.33 %	6	14 (2)	0		13	77	96 %	0 %	1574 s
ArrayList remove	23 %	92.87 %	7	16 (3)	0		13	40	89 %	13 %	2065 s
Arrays binarySearch0	100 %	100 %	11	30 (17)	0		6	1289	90 %	77 %	4200 s
Arrays equals	100 %	100 %	7	7 (1)	0		3	96	80 %	21 %	2240 s
Arrays fill_a	100 %	100 %	6	5 (1)	0		1	6	83 %	0 %	1391 s
Arrays fill_b	100 %	100 %	7	15 (5)	0		1	55	88 %	21 %	1880 s
Arrays fill_c	100 %	100 %	6	7 (3)	0		1	15	37 %	53 %	1375 s
Arrays fill_d	100 %	100 %	7	18 (8)	0		1	55	70 %	39 %	1857 s
Arrays hashCode_a	100 %	100 %	2	4 (0)	0		0	0	—	—	389 s
Arrays hashCode_b	100 %	100 %	2	4 (0)	0		0	0	—	—	343 s
Arrays insertionSort_a	100 %	100 %	10	74 (38)	0		8	1224	89 %	70 %	4029 s
Arrays insertionSort_b	100 %	100 %	11	73 (34)	0		8	5200	99 %	45 %	4512 s
Arrays merge	0 %	90.48 %	11	78 (62)	1		16	7532	97 %	8 %	8034 s
Arrays quicksortPartition	0 %	93.94 %	9	57 (18)	1		16	42714	99 %	60 %	5657 s
Arrays vecswap	100 %	100 %	8	18 (9)	0		5	1983	96 %	22 %	2698 s
Collections replaceAll	77 %	96.71 %	6	16 (4)	0		8	57	77 %	56 %	1801 s
Collections reverse	21 %	79.30 %	9	34 (18)	0		15	1181	63 %	56 %	6949 s
Collections sort	0 %	72.80 %	9	17 (0)	2		16	4343	96 %	86 %	3933 s
Vector indexOf	100 %	100 %	6	24 (4)	0		2	20	70 %	32 %	1698 s
Vector lastIndexOf	90 %	99.23 %	7	19 (2)	0		3	40	88 %	38 %	1859 s
Vector removeAllElements	100 %	100 %	5	12 (5)	0		1	10	50 %	0 %	1218 s
Vector removeRange	63 %	95.80 %	7	17 (5)	0		7	135	65 %	21 %	2574 s
Vector setSize	100 %	100 %	7	31 (20)	0		2	63	58 %	15 %	2003 s
AVERAGE	71 %	97 %	7	25 (10)	0		7	2410	82 %	34 %	2691 s

6.2 Experimental Comparison

In this section, we report on focused experiments involving few cutting-edge tools that can be directly compared to DYNAMATE, to convincingly show how it improves the state of the art and its specific strengths and weaknesses.

Tool selection. We selected other tools for automated verification that are in the same “league” as DYNAMATE; namely, they satisfy the following characteristics: *i)* a working implementation is available; *ii)* they work on a real programming language, or at least a significant subset; *iii)* they support numerical and functional properties; *iv)* they are completely automatic (or we clarify what extra input is needed); *v)* they are cutting-edge (i.e., no other similar tool supersedes them). The first two requirements excluded several techniques without serviceable

implementations or that only work on toy examples. All in all, we identified three “champions” with all characteristics: INVGEN [14] (which uses constraint-based techniques and mainly targets linear numeric invariants); BLAST [1] (a software model-checker using CEGAR and predicate abstraction); and cccheck [5] (the CodeContracts static checker formerly known as Clousot, based on abstract interpretation with domains for arrays).

We translated the classes used in DYNAMATE’s evaluation into a form amenable to each tool (in C for INVGEN and BLAST, and in C# for cccheck). We tried to replicate the salient features of Java’s semantics in each language, without introducing unnecessary complications; for example, Java `Object` objects become C `structs` passed to functions as pointer arguments.

We encoded in each tool as many of the ESC/Java2 proof obligations as possible. We used **asserts** to express proof obligations for which no equivalent language feature was available (for example, class invariants in BLAST). When a tool’s specification language did not support quantified expressions (as is the case in C Boolean expressions), we rendered the semantics using operational formulations, like DYNAMATE does for runtime checking (Section 4.1).

For each tool, we measured the percentage of the 367 proof obligations that was expressible and the percentage that was successfully verified. We also include figures about the bare ESC/Java2, which can discharge a good fraction of the proof obligations that do not require reasoning about loops.

TOOL	PROOF OBLIGATIONS		VERIFIED METHODS	TIME
	EXPRESSIBLE	PROVEN		
ESC/Java2	100 %	231 (63 %)	0/28	122 _s
INVGEN	42 %	60 (39 %)	0/28	78 _s
BLAST	100 %	238 (65 %)	3/28	5431 _s
cccheck	100 %	276 (75 %)	3/28	106 _s
DYNAMATE	100 %	354 (97 %)	25/28	75348 _s

Comparison results. ESC/Java2 gives a good baseline of 63% proof obligations automatically discharged. INVGEN is quite limited in what it can express: the subset of C it inputs does not have support for arrays, and hence it is strictly limited to scalar properties; within these limitations, it is often successful using its predefined templates. BLAST supports the full C language, but does not go much beyond the baseline in terms of what it can prove. In fact, “the predicates tracked by BLAST do not contain logical quantifiers. Thus, the language of invariants is weak, and BLAST is not able to reason precisely about programs with arrays or inductive data structures whose correctness involves quantified invariants” [1]. cccheck’s performance is impressive, also given its full support of .NET, but its abstract domains are still insufficient to express complex quantifications over arrays formalizing, for example, sortedness. DYNAMATE achieves a solid 97% of automatically discharged proof obligations, significantly improving over the state of the art: the proof obligations discharged by DYNAMATE are a superset of those checked by other tools. In particular, DYNAMATE achieved full verification of 25 out of 28 methods, while the other tools verified at most 3 methods.

Given that our selection of 28 methods is based on their analyzability with ESC/Java2 (as we describe at the beginning of Section 6), we do not claim that our results generalize to other kinds of programs. They do, however, provide evidence that dynamic techniques can complement static ones in order to automate full verification of “natural” programs that are currently challenging for state-of-the-art tools. (In particular, our DYNAMATE implementation boosts the capabilities of ESC/Java2.)

The main weakness of DYNAMATE derives from its usage of dynamic randomized techniques: DYNAMATE may require repeated runs, and is one to two orders of magnitude slower than the static tools. Future work will target these shortcomings to further promote the significant results obtained by DYNAMATE’s algorithms. Also part of future work is evaluating DYNAMATE on examples originally used to evaluate INVGEN, BLAST, or cccheck. Our comparison is, however, already significant as it is: DYNAMATE’s architecture can easily accommodate other third-party tools tailored to *some kinds* of invariants, while still offering an edge in general-purpose programs.

7 Related Work

Since we cannot exhaustively summarize the huge amount of work on automating program checking indirectly related to DYNAMATE, we focus this section on the problem of inferring loop invariants to automate functional verification, and in particular on static techniques. See the extended version of this paper for more references and details.

Abstract interpretation is a general framework for computing approximations of program semantics; invariant inference is one of its main applications. Each specific abstract interpreter works on one or more *abstract domains*, which characterize the kinds of invariants that can be computed. Much of the work in abstract interpretation has focused on domains for non-functional properties, such as reachability and other heap shape properties, or on simple “global” correctness properties, such as absence of division by zero or null pointer dereference. While scalar numerical properties are fully supported, only recently has restricted array quantification been made available [12, 5].

Predicate abstraction [8] is a technique to build finite-state over-approximations of programs. Its applicability crucially depends on the set of predicates provided as input, which determine precision and complexity. Predicate abstraction is a fundamental component of the CEGAR (Counter-Example Guided Abstraction Refinement) approach to software model-checking, which features in tools such as BLAST [1].

Constraint-based techniques reduce the invariant inference problem to solving constraints over a template that defines the general form of invariants. The state of the art focuses on invariants in the form of Boolean combinations of linear [14, 13] inequalities, polynomials [24], and linear arithmetic with uninterpreted functions [2]. Since constraint-based techniques rely on decidable fragments, they rarely support templates involving quantification. The automata-based approach of [3] is an exception, but its experiments are still limited to flat linear loops with simple control logic, and it is not applicable to linked structures.

Using first-order theorem proving. Kovács et al. [16, 15] target the inference of loop invariants for array-manipulating programs. Their technique encodes the semantics of loops directly as recurrence relations and then uses a properly modified saturation theorem prover to derive logic consequences of the relations that are syntactically loop invariants. This can generate invariants involving alternating quantifiers, which are out of the scope of most other techniques. McMillan [18] also uses a modified saturation theorem prover to infer quantified loop invariants. Both techniques rely on heuristics that depend on interacting with a custom-modified theorem prover, which may constrain the kinds of programs that they can process. For example, [16] assumes loops that monotonically increment or decrement a counter variable; more general index arithmetic is not handled directly.

Combination of techniques. Srivastava and Gulwani [25] combine predicate abstraction and template-based inference to construct quantified invariants that can express complex properties such as sortedness. Their tool takes as input a program and a set of templates and

predicates; for example, inferring invariants specifying sortedness of an array A requires a template $\forall k : \square \implies \square$ and predicates $0 \leq x$, $x < y - 1$ and $A[k] \leq A[k + 1]$. They demonstrate the approach on several sorting algorithms (as well as simple linked list operations). While these results are impressive, DYNAMATE still offers some complementary advantages: it works on real implementations and supports the full Java programming language; by using model-based annotations, it is more flexible with respect to the used data structures (e.g., lists vs. arrays); and does not require extra user input in the form of templates and predicates.

Nimmer and Ernst [19, 20] combine dynamic invariant inference a la Daikon with a static program checker. Loop invariants are out of the scope of that work, since the static checker (ESC/Java, not ESC/Java2) approximates loops unsoundly by unrolling them once; hence, “loop invariants may need to be strengthened to be proved” [19].

Nori and Sharma [21] use a program verifier in combination with automatic test case generation to automatically produce termination proofs. The kinds of loop invariants required for termination proofs are simpler than those discovered by DYNAMATE, since they only have to constrain the values of a ranking function.

8 Conclusions

We present a fully automated approach to discharge proof obligations of programs with loops. The approach combines complementary techniques: test case generation, dynamic invariant detection, and static verification. We are committed to make the DYNAMATE framework publicly available, including all subjects required to replicate the results in this paper. For details, see:

<http://www.st.cs.uni-saarland.de/dynamate/>

References

- [1] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *STTT*, 9(5-6):505–525, 2007.
- [2] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *VMCAI*, volume 4349 of *LNCS*, pages 378–394. Springer, 2007.
- [3] M. Bozga, P. Habermehl, R. Iosif, F. Konečný, and T. Vojnar. Automatic verification of integer array programs. In *CAV*, volume 5643 of *LNCS*, pages 157–172. Springer, 2009.
- [4] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *FMCO*, LNCS, pages 342–363. Springer, 2006.
- [5] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, pages 105–118. ACM, 2011.
- [6] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, Feb. 2001.
- [7] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME*, volume 2021 of *LNCS*, pages 500–517. Springer, 2001.
- [8] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202. ACM, 2002.
- [9] G. Fraser and A. Arcuri. Evolutionary generation of whole test suites. In *QSIC*, pages 31–40. IEEE Computer Society, 2011.
- [10] C. A. Furia and B. Meyer. Inferring loop invariants using postconditions. In *Fields of Logic and Computation*, volume 6300 of *LNCS*, pages 277–300. Springer, 2010.

- [11] C. A. Furia, B. Meyer, and S. Velder. Loop invariants: Analysis, classification, and examples. *ACM Comp. Sur.*, 46(3):Article 34, January 2014.
- [12] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246. ACM, 2008.
- [13] S. Gulwani, S. Srivastava, and R. Venkatesan. Constraint-based invariant inference over predicate abstraction. In *VMCAI*, volume 5403 of *LNCS*, pages 120–135. Springer, 2009.
- [14] A. Gupta and A. Rybalchenko. InvGen: An efficient invariant generator. In *CAV*, volume 5643 of *LNCS*, pages 634–640. Springer, 2009.
- [15] K. Hoder, L. Kovács, and A. Voronkov. Invariant generation in Vampire. In *TACAS*, volume 6605 of *LNCS*, pages 60–64. Springer, 2011.
- [16] L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *FASE*, volume 5503 of *LNCS*, pages 470–485. Springer, 2009.
- [17] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.
- [18] K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *TACAS*, volume 4963 of *LNCS*, pages 413–427. Springer, 2008.
- [19] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *RV*, 2001.
- [20] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *ISSTA*, pages 229–239, 2002.
- [21] A. V. Nori and R. Sharma. Termination proofs from tests. In *ESEC/FSE*, pages 246–256. ACM, 2013.
- [22] N. Polikarpova, C. A. Furia, and B. Meyer. Specifying reusable components. In *VSTTE*, volume 6217 of *LNCS*, pages 127–141. Springer, 2010.
- [23] N. Polikarpova, C. A. Furia, Y. Pei, Y. Wei, and B. Meyer. What good are strong specifications? In *ICSE*, pages 257–266. ACM, 2013.
- [24] S. Sankaranarayanan, H. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In *POPL*, pages 318–329. ACM, 2004.
- [25] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *PLDI*, pages 223–234. ACM, 2009.