
Article:
Galeotti, J.P., Furia, C.A., May, E. et al. (2015) Inferring loop invariants by mutation, dynamic analysis, and static checking. IEEE Transactions on Software Engineering, 41 (10). 1019 - 1037. ISSN: 0098-5589

https://doi.org/10.1109/TSE.2015.2431688

Inferring Loop Invariants by Mutation, Dynamic Analysis, and Static Checking

Juan P. Galeotti, Carlo A. Furia, Eva May, Gordon Fraser, and Andreas Zeller

Abstract—Verifiers that can prove programs correct against their full functional specification require, for programs with loops, additional annotations in the form of *loop invariants*—properties that hold for every iteration of a loop. We show that significant loop invariant candidates can be generated by systematically mutating postconditions; then, dynamic checking (based on automatically generated tests) weeds out invalid candidates, and static checking selects provably valid ones. We present a framework that automatically applies these techniques to support a program prover, paving the way for fully automatic verification without manually written loop invariants: Applied to 28 methods (including 39 different loops) from various `java.util` classes (occasionally modified to avoid using Java features not fully supported by the static checker), 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 verification.

Index Terms—Loop invariants, inference, automatic verification, functional properties, dynamic analysis

1 Introduction

Despite significant progress in automating program verification, proving a program correct still requires substantial expert manual effort. For programs with loops, one of the biggest burdens is providing *loop invariants*—properties that hold at the entry of a loop and are preserved by an arbitrary number of loop iterations. Compared to other specification elements such as pre- and postconditions, loop invariants tend to be difficult to understand and to express, and even structurally simple loops may require non-trivial invariants to be proved correct [1].

In this paper, we present and evaluate a novel approach to improve the automation of full program verification through the automatic discovery of suitable loop invariants. The approach is based on a combination of static (program proving) and dynamic (testing) techniques that complement each other's capabilities. The key observation is that, given loop invariant *candidates* (assertions that may or may not hold for the loops under analysis), an automatic testing tool can efficiently weed out *invalid* candidates, whereas a program verifier 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 additional annotations.

- *Juan P. Galeotti and A. Zeller are with the Chair of Software Engineering, Saarland University, Saarbrücken, Germany. E-mail: zeller@acm.org.*
- *Carlo A. Furia is with the Chair of Software Engineering, Department of Computer Science, ETH Zurich, Switzerland.*
- *Eva May is with Google, Inc.*
- *Gordon Fraser is with the University of Sheffield, UK.*


```
private static int binarySearch0(int[] a,
      int fromIndex, int toIndex,
      int key) {
  int low = fromIndex, high = toIndex − 1;
  while (low ≤ high) {
    // midpoint of [low..high]
    int mid = low + ((high − low)/2);
    int midVal = a[mid];
    if (midVal < key) low = mid+1;
     else if (midVal > key) high = mid − 1;
     else return mid; // key found
  }
  return −(low + 1); // key not found
}
```

Fig. 1. Binary search method in `java.util.Arrays`.

```
/*@
  @ requires a ≠ null;
  @ requires TArrays.within(a, fromIndex, toIndex);
  @ requires TArrays.sorted(a, fromIndex, toIndex);
  @
  @ ensures \result ≥ 0 ⟹ a[\result] = key;
  @ ensures \result < 0
  @     ⟹ ¬TArrays.has(a, fromIndex, toIndex, key);
  @*/
```

Fig. 2. Pre- and postcondition of `binarySearch0`.

1.1 Running Example: Binary Search

Figure 1 shows `binarySearch0`, a helper method declared in class `java.util.Arrays` in the standard Java API. Method `binarySearch0` takes a sorted array `a`; if element `key` is found in `a`, it returns its index, otherwise it returns a negative integer. Figure 2 formalizes this behavior as pre- and postcondition written in

JML [2], using model-based *predicates* [3], representing implicit quantified expressions, with descriptive names. For example, the predicate ¬TArrays.has(a, fromIndex,toIndex, key) means that array a has no element key over the interval range from fromIndex (included) to toIndex (excluded). The specification of Figure 2 is what we want to verify binarySearch0 against, and it would be required for any kind of functional validation—be it based on testing or static reasoning.

Fully automatic verifiers such as cccheck [4] (formerly known as Clousot [5]) or BLAST [6] fail to establish the correctness of the annotated program. On the other hand, auto-active verifiers such as ESC/Java2 [7][1] succeed, but only if we provide suitable invariants for the loop in Figure 1, such as those in Figure 3. Compared to pre- and postconditions, it is much more difficult to write loop invariants, since they capture implementation-specific details rather than general input/output behavior. A method implementing a different search algorithm would have the same postcondition as binarySearch0's, but proving it correct would most likely require quite different loop invariants.

```
/*@
  @ loop_invariant fromIndex ≤ low
  @ loop_invariant low ≤ high + 1
  @ loop_invariant high < toIndex
  @ loop_invariant ¬TArrays.has(a,fromIndex,low,key)
  @ loop_invariant ¬TArrays.has(a,high+1,toIndex,key)
  @*/
```

Fig. 3. Loop invariants required for verifying method binarySearch0.

1.2 Summary of the DYNAMATE Approach

In this paper, we present an approach that automates the functional verification of partial correctness of programs with loops by inferring the required loop invariants. The approach combines different techniques as illustrated in Figure 4. Initially, a static verifier runs on the program code and its specification. If the verifier fails to prove the program correct, a round of four steps begins.

Step 1: test cases. To support dynamic invariant detection, a *test case generator* builds 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.)

Step 2: candidate invariants. From the resulting executions, an invariant detector dynamically mines candidates for loop invariants. Our *invariant detector* tools use fixed, generic patterns as well as *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.)

Step 3: invariant verification. The surviving set of loop invariant candidates are fed into a static *program verifier*. The verifier may confirm that some candidates are valid loop invariants.

(The program verifier confirms that high < toIndex is a valid loop invariant, but rejects the other candidate.)

Step 4: program verification and refinement. Using the verified invariants, the static verifier may also be able to produce a proof that the program is correct with respect to its specification. If the proof does not succeed using the loop invariants inferred so far, *another round* of generating, mining, and verifying starts. If the verifier has left loop invariants unproved, 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.)

We implemented this approach in a tool called DYNAMATE[2], a fully automatic verifier for Java programs with loops.

1.3 Loop Invariants from Postconditions

A key functionality in DYNAMATE is GIN-DYN, a flexible mechanism for generating loop invariants. While a vast amount of research on automatically finding loop invariants has been carried out over the years (Section 6), most approaches target restricted classes of loop invariants (such as linear inequalities over scalar numeric variables), which limits wide applicability in practice. In fact, while ESC/Java2 can handle full-fledged JML-annotated Java programs, it provides no support for loop invariant discovery.

To overcome this limitation, in previous work [8], [1] we introduced the idea of *guessing invariant candi-*

1. Like most static verifiers working on real programming languages, ESC/Java2 does not fully support a number of Java and JML language features, which makes it an unsound tool in general. In this work, we only deal with a subset of Java and JML that excludes ESC/Java2's unsupported features, and we always use ESC/Java2 with the -loopSafe option for sound verification of unbounded loops. Therefore, the rest of the paper refers to ESC/Java2 as a *sound but incomplete tool*, with the implicit proviso that we avoid exercising its known sources of unsoundness. See Section 5.1 for more details about this issue with reference to the case study.

2. DYNAMATE = "Dynamic Mining and Testing"

dates 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 relaxation 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 Section 4, we introduce the GIN-DYN technique, which addresses this challenge. GIN-DYN uses dynamic checking to quickly weed out incorrect loop invariant candidates, and static checking to prune formally correct but redundant or irrelevant ones. This way, the 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 simple fixed assertion patterns.

1.4 Experimental Evaluation

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.[3] DYNAMATE automatically discovered all loop invariants in Figure 3 given the code and specification in Figure 1 and Figure 2, resulting in fully automatic verification of the `binarySearch0` method. Across the whole case study, 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 5.3, these results correspond to an over 20% improvement over state-of-the-art tools for automatic verification, such as the CodeContracts static checker, in terms of number of automatically discharged obligations.

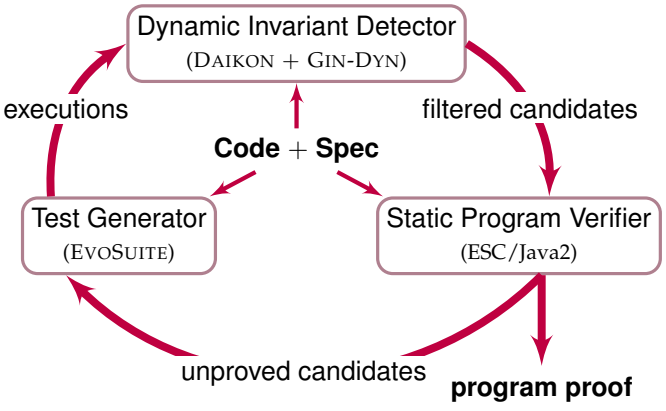


Fig. 4. How DYNAMATE works. The program code (center) is first fed into a test case generator (left), which generates executions covering legal behavior. From these, two dynamic invariant detector tools (top) mine possible loop invariants, based both on fixed patterns (DAIKON) as well as postconditions (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.

Being able to provide such fully automatic proofs of real Java code is certainly promising. However, we have to adjust our expectations: from a software engineering perspective, the methods DYNAMATE can prove are still very small. Unfortunately, the state of the art is such that proofs of large, complex systems are simply impossible without significant manual effort by highly-trained experts; and the intrinsic complexity of formal verification indicates that they will remain so for the foreseeable future. However, *if* a critical function is similar in complexity to those analyzed in our experiments—that is, mostly array-based algorithm implementations with a handful of loops and conditions—then there are chances that DYNAMATE will be able to prove it. (And, if the proof attempt fails, no human effort is wasted.) From a verification perspective, being able to conduct such proofs automatically on practical functions from production code is an important goal; DYNAMATE neither expects nor requires any loop annotation, user interaction, or expert knowledge.

1.5 Summary of Contributions

The main contributions of this paper are:

1) DYNAMATE: an algorithm to automatically discharge proof obligations for programs with loops, based on a combination of dynamic and static techniques.
2) GIN-DYN: an automatic technique to boost the dynamic detection of loop invariants, based on the idea of syntactically mutating postconditions [8].
3) A prototype implementation of the DYNAMATE algorithm that integrates the EVOSUITE test case generator, the DAIKON dynamic invariant detector, and the ESC/Java2 static verifier, as well as GIN-DYN.
4) An evaluation of our DYNAMATE prototype on a case study involving 28 methods with loops from `java.util` classes.
5) A comparison against state-of-the-art tools for automatic verification based on predicate abstrac-

3. As Section 5.1 explains in detail, we occasionally modified the methods to remove features unsupported by ESC/Java2—without changing the methods' input/output behavior.

tion, abstract interpretation, and constraint-based techniques.

The remainder of this paper is organized as follows. Section 2 overviews the DYNAMATE algorithm in general, followed by a description of the current prototype in Section 3. Section 4 describes in more detail the GIN-DYN technique for loop invariant detection. Section 5 evaluates DYNAMATE on a case study consisting of Java code from selected `java.util` classes. Section 6 discusses related work in loop invariant detection. Section 7 closes with conclusions and future work.

2 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 algorithm is applicable to any language that offers three such components.

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 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*)

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. 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 that are not valid loop invariants in general.

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 proved 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, unproved 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 unproved candidates in C as possible.

3 How DYNAMATE Works

This section details how DYNAMATE works, using `binarySearch0` as running example.

3.1 Input: Programs and Specifications

Each run of DYNAMATE takes as input a Java method M with its functional specification consisting of precondition P and postcondition Q. Pre- and postcondition are written in JML [2]: an extension of the Java syntax for expressions that supports standard logic constructs such as implications and quantifiers. P and Q generally consist of a number of *clauses*, each denoted by the keyword **`requires`** (precondition) and **`ensures`** (postcondition); clauses are implicitly logically conjoined. In Figure 2, for example, the precondition has three clauses and the postcondition has two.

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 [2] and developed in related work of ours [3], [9]). 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 2 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.

Following the model-based specification style entails three main advantages for our work. First, it improves the abstraction and clarity of specifications, and hence it also facilitates reuse with different implementations. For instance, it should be clear that `has(a, fromIndex, toIndex, key)` means that array `a` contains a value `key` within `fromIndex` and `toIndex`. Using predicate `has` also abstracts several details about how arrays are implemented, such as whether they are zero-indexed; in fact, we would write the same specification even if `a` were, say, a dynamic list and `fromIndex` and `toIndex` two pointers to list elements. Predicates are defined only once and then reused for all methods working in a similar domain; in fact, for our experiments (Section 5) we developed a small set of predicates in two classes (`TArrays` and `TLists`), which were sufficient to express the specification and loop invariants of all 28 methods used in the evaluation.

Second, model-based specifications also make it easy to reconcile static and a runtime semantics. When developing predicates in `TArrays` we defined each predicate as a **`static boolean`** method with both a Java implementation and a JML specification. For example, `has`'s precondition declares its domain of definition (in words: `a` is not **`null`** and `[fromIndex..toIndex)` is a valid interval within `a`'s bounds); its postcondition

$$\exists\ \texttt{int i;}\ \texttt{fromIndex} \leq \texttt{i} \wedge \texttt{i} < \texttt{toIndex} \wedge \texttt{key} = \texttt{a[i]}$$

defines the static semantics of the predicate using explicit quantification and primitive operations; its implementation recursively searches through `a` for an element `key` and returns **`true`** iff it finds it within `fromIndex` and `toIndex`. Dynamic tools will then use its implementation to check whether a predicate holds; static tools will instead understand a predicate's semantics in terms of its JML specification.

A third advantage of using model-based specifications is leveraged by the DYNAMATE approach and more precisely by the GIN-DYN invariant detector described in Section 4. Collections of predicates such as `TArrays` provide useful domain knowledge, as they suggest which predicates are likely to be pertinent to the specification of the methods at hand: if a method's postcondition Q includes a predicate p from some class T, it's likely that the loop invariants necessary to prove Q also include p or some other predicates defined in T.

3.2 Test Case Generation

The DYNAMATE algorithm needs concrete executions to dynamically infer loop invariants: DAIKON mines relations that hold in all passing test cases (Section 3.3); and GIN-DYN filters out invalid loop invariant candidates that are falsified by a test case (Section 4). While any test case generator could work with DYNAMATE, our prototype integrates EVOSUITE [10], a fully automatic search-based tool using a genetic algorithm. Besides being a fully automated tool, a specific advantage of EVOSUITE is that its genetic algorithm evolves test suites towards covering all program branches at the same time, and hence infeasible branch conditions (common in the presence of assertions) do not ultimately limit search effectiveness.

```
private static int binarySearch0(int[] a,
       int fromIndex, int toIndex,
       int key) {
  // Precondition clause:
  // TArrays.sorted(a,fromIndex,toIndex)
  if (¬TArrays.sorted(a,fromIndex,toIndex))
    throw new AssertionError("Failed precondition");
  ...
```

Fig. 5. Runtime check for a precondition clause of `binarySearch0`'s.

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 discussed in Section 3.1.

One of `binarySearch0`'s preconditions requires that the input array `a` be sorted within the range from `fromIndex` to `toIndex`; Figure 5 shows the corresponding instrumentation that calls the implementation of the predicate `sorted` and throws an exception if it evaluates to false.

Since EVOSUITE tries to maximize branch coverage, it has a good chance[4] of producing tests that pass all precondition checks and thus represent valid executions according to the specification. For example, EVOSUITE produces the valid test in Figure 6 where the value -1030 is searched for in a trivially sorted array initialized to all zeros.

3.3 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

4. Due to randomization, success may vary between runs.

```
void test() {
 // Element not found
 int[] intArray = new int[8];
 int intV = Arrays.binarySearch0(intArray, 0, 6, −1030);
}
```

Fig. 6. A test that covers the "not found" branch in `binarySearch0`.

TABLE 1
Some loop invariant candidates produced by DAIKON in the first iteration of DYNAMATE. Column VALID reports which candidates are valid.

ID	CANDIDATE	VALID
c_1	`key` $\in \{-1030, 0\}$	NO
c_2	`a` $\neq$ **`null`**	YES
c_3	`a[]`'s elements one of $\{-915, 0\}$	NO
c_4	`Arrays.INSERTION_THRESHOLD` $\neq$ `toIndex`	NO
c_5	`low` $\geq$ `fromIndex`	YES
c_6	`low` $\geq$ `key`	NO
c_7	`high` $<$ `toIndex`	YES
c_8	`high` $>$ `key`	NO
c_9	`high` $\leq$ `a.length` -1	YES
c_{10}	`toIndex` $>$ `fromIndex`	NO

or basic array properties) come from DAIKON [11]. 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. The rest of this section describes how DYNAMATE uses GIN-DYN and DAIKON.

3.3.1 Dynamic Invariant Detection with DAIKON

DAIKON [11] is a widely used dynamic invariant detector which supports a set of basic invariant templates. Given a test suite and a collection of program locations as input, DAIKON instantiates its templates with program variables, and traces their values at the locations in all executions of the tests. The instantiated templates that hold in every execution are retained as likely invariants at those locations. Likely invariants are still just candidates (they may be invalid) because they are based on a finite number of executions.

Since DYNAMATE needs *loop* invariants, it instructs DAIKON to trace variables at four different locations of each loop: before loop entry, at loop entry, at loop exit, and after loop exit. To this end, we instrument the input programs adding dummy methods invoked at these locations, which gives access to all variables within the loops without requiring modifications to DAIKON.

DYNAMATE executes DAIKON on `binarySearch0` using the initial test suite, which produces 62 loop invariant candidates, most of which are invalid due to the incompleteness of the initial test suite. Table 1 shows 10 of the 62 candidates; among others, c_3 is merely a reflection of the fact that the initial test suite only put the values `0` and $-$`915` in the arrays. Another visible characteristic of these invariants is that they are limited to simple properties, and hence they are insufficient to prove `binarySearch0`'s postcondition. In general, DAIKON templates give invariants that are immediately useful to establish simple properties such as absence of out-of-bound accesses, but are hardly sufficient to prove full functional correctness.

3.3.2 GIN-DYN: Invariants from Postconditions

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 28–29 in Figure 3 are never produced by DAIKON with its standard templates.

Extending DAIKON with new templates is possible, but even leaving implementation issues aside (e.g., DAIKON only includes non-static methods with no arguments as predicates) we cannot anticipate all possible forms a loop invariant may take. Instead, we apply ideas introduced in our previous work [8]: since loop invariants can be seen as relaxations of the postcondition, syntactically mutating a postcondition Q generates variants of Q which we use as suggestions for invariant *candidates*. We turn this intuition into a practically applicable technique by providing GIN-DYN: a way to efficiently generate and filter out a large amount of invalid or uninteresting invariant candidates. Section 4 describes in detail how GIN-DYN does the filtering, again based on a combination of dynamic and static techniques. The rest of the current section briefly discusses how invariant candidates produced by GIN-DYN are used within DYNAMATE.

When it reaches a fixpoint, DYNAMATE asks GIN-DYN for a new *wave* of invariant candidates, adds them to the set of current candidates, and proceeds with static validation, which is possibly followed by other iterations of the main algorithm. Thus, GIN-DYN and DAIKON invariant candidates are used in the very same way; the fact that they nearly always are disjoint sets increases the chance of eventually getting to a complete set of invariants.

In fact, GIN-DYN produces the two fundamental invariants on lines 28–29 in Figure 3 necessary for

a correctness proof of `binarySearch0`. The final set of *verified* loop invariants includes those of Figure 3 with 28 more,[5] consisting of 13 invariants found by DAIKON and 20 invariants found by GIN-DYN.

3.4 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.

3.4.1 Verification of Loop Invariants

The DYNAMATE prototype relies on the ESC/Java2 [7] static verifier, which works on Java programs and JML annotations.

By default [12], ESC/Java2 handles loops unsoundly by unrolling them a finite number of times; but, by enabling the `-loopSafe` option, it encodes the exact modular semantics of loops based on loop invariants [13, Sec. 5.2.3].[6] Therefore, DYNAMATE always calls ESC/Java2 with the `-loopSafe` option enabled.

To determine whether a candidate L is a valid invariant of some loop ℓ, DYNAMATE annotates ℓ with the assertion **`loop_invariant`** L and runs ESC/Java2 on the annotated program. As part of verifying the input, ESC/Java2 checks whether L is a valid loop invariant, that is it holds initially and is preserved by every iteration. DYNAMATE searches for ESC/Java2 *warnings* that signal whether checking L was successful. Since ESC/Java2 is a sound but incomplete tool,[7] if it produces no warning about L, we conclude that L is a valid loop invariant; if it does produce a warning, it may still be that L is valid but the current information (typically, in the form of other loop invariants) is insufficient to establish it with certainty.

Due to the undecidability of first-order logic, ESC/Java2 may also time out or run out of memory, which also counts as inconclusive output. In all such cases, the DYNAMATE algorithm leverages the static/-dynamic complementarity once again: it initiates another iteration of testing, trying to conclusively falsify the candidates that ESC/Java2 could not validate.

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 [14], which first considers all candidates at once, and then iteratively removes those that cannot be verified until all surviving candidates are valid loop invariants. In the `binarySearch0` example, DAIKON produces 62 loop invariant candidates in the first iteration. Running ESC/Java2 determines that half of them are valid loop invariants, among which c_2, c_5, c_7, and c_9 in Table 1.

3.4.2 Program Proof

At the end of each 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 the invariants in Table 1 are insufficient to prove the specification in Figure 2.

3.5 Refining the Search for Loop Invariants

Unproved loop invariant candidates may be overspecific and hence unsound—such as c_1 in Table 1. Since this may indicate unexplored program behavior, for every such candidate L, DYNAMATE adds the conditional check

```
if (¬L) throw new AssertionError("L failed")
```

at the locations where loop invariants are checked. The check is enclosed by a **`try/catch`** block, so that testing L does not affect testing other candidates. This directs EVOSUITE's search towards trying to explore the new conditional branch, and hence falsifying L. This helps refine the dynamic detection of loop invariants by providing more accurate tests.

4 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 [1], we suggested [8] the idea of systematically *mutating* Q to guess candidate invariants of ℓ.

The throwaway prototype discussed in [8] did not work on a real programming language, and required users to manually select a subset of mutation operators to direct the generation of candidates. In fact, 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.

5. In this work, we do not address minimizing the number of invariants.

6. See also: http://sourceforge.net/p/jmlspecs/mailman/message/31579805/

7. On the Java code we apply it to: see Footnote 1

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 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 in a good position 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.

4.1 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 `t`, a set $\mathcal{E}_{\texttt{t}}$ of available expressions of type `t`. $\mathcal{M}$ initially consists of the clauses in postcondition Q, and $\mathcal{E}_{\texttt{int}}$ and $\mathcal{E}_{\texttt{int[]}}$ 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 $e \in \mathcal{E}_{\texttt{t}}$ of type `t`, replace by e any subexpression of type `t` in m. This generates as many mutants as are subexpressions of type `t` in m, which are added to $\mathcal{M}$.
- **Weakening:** given a mutant $m \in \mathcal{M}$ and a Boolean expression $b \in \mathcal{E}_{\texttt{boolean}}$, add to $\mathcal{M}$ the mutants $b \Longrightarrow m$ and $\neg b \Longrightarrow m$.
- **Aging:** given a mutant $m \in \mathcal{M}$, replace by $e - 1$ or $e + 1$ any subexpression e of type **`int`** in m. This generates twice as many mutants as are subexpressions of type **`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 $\mathcal{E}_{\texttt{boolean}}$ of Boolean expressions all predicates belonging to the same collection (such as `TArrays`) as those in the postcondition (see Section 3.1).

Figure 7 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 8 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.

Q:
`\result` $<$ `0` $\Longrightarrow$ `¬TArrays.has(a,fromIndex,toIndex,key)`
$\rightsquigarrow$ `low` $<$ `0` $\Longrightarrow$ `¬TArrays.has(a,fromIndex,toIndex,key)`
$\rightsquigarrow$ `low` $<$ `0` $\Longrightarrow$ `¬TArrays.has(a,fromIndex,mid,key)`

Fig. 7. Mutations producing a valid but trivial loop invariant.

Q:
`\result` $<$ `0` $\Longrightarrow$ `¬TArrays.has(a,fromIndex,toIndex,key)`
$\rightsquigarrow$ `¬TArrays.has(a,fromIndex,toIndex,key)`
$\rightsquigarrow$ `¬TArrays.has(a,high,toIndex,key)`
$\rightsquigarrow$ `¬TArrays.has(a,high +1,toIndex,key)`

Fig. 8. Mutations producing a valid and useful loop invariant.

4.1.1 Mutation Waves

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 `t`, and we have e expressions of type `t` available, applying all possible n consecutive `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 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; GIN-DYN executes one or more waves in each iteration of DYNAMATE's main loop (Figure 4). The waves are of three kinds: the first kind includes 7 waves that all work by substituting integer expressions in any postcondition *clause*; the second kind includes 4 waves that all work by substituting integer expressions in any *predicate*, in negated or unnegated form, appearing in the *postcondition*; the third kind includes 5 waves that all work by substituting integer expressions in any predicate, in negated or unnegated form,

belonging to the same *collection* (`TArrays` or `TLists`) as those in the postcondition. Waves of the same kind differ in how many integer expressions each generated mutant may contain (from one up to three), in whether only parameterless integer expressions are available for substitution (variables and parameterless method calls), and in whether aging or weakening are applied. Figure 9 describes one wave of each kind: the first and the second wave produce the mutants in Figure 7 and Figure 8.

KIND	WAVE
1st	$\mathcal{M} \leftarrow Q$; **int** sub. ; **int** sub.
2nd	$\mathcal{M} \leftarrow \{p \in Q \cap$ **boolean**$\}$; **int** sub. ; aging
3rd	$\mathcal{M} \leftarrow \{$`T.p` $\in$ **boolean** $\mid \exists q$: `T.`$q \in Q\}$; **int** sub. ; aging

Fig. 9. One example mutation wave of each kind.

The three kinds of waves try to cover the most common patterns found in algorithms [1]: mutations of the postcondition (1st kind), of some predicate in the postcondition (2nd kind), of some predicate of the same family of those in the postcondition (3rd kind).

Then, different waves of the same kind achieve different trade-offs between combinatorial complexity and exhaustiveness of the applied substitutions. If we disregarded generation time completely, we would retain only the most general wave of each kind (for example, including as many three substitutions per mutant as well as aging and weakening) and run it to completion. Unfortunately, this would require an exorbitant amount of time in most cases and, even in the cases where it would not lead to combinatorial explosion, it would generate too many irrelevant mutants. Instead, we introduce waves that are incrementally more complex, so that the more amenable algorithms do not require to run the most complex waves at all. With the current implementation of GIN-DYN, this solution has the drawback of introducing redundancy in the generation of mutants, where some later waves generate mutants that were already generated by previous waves (and hence are immediately discarded). Implementing a mutation generation process that avoids redundancies belongs to future work (although the results in Section 5 indicate that mutation generation is not the bottleneck in DYNAMATE).

DYNAMATE's waves target substitutions only[8] of integer (**int** and `Integer`) expressions because they are at the heart of so many algorithms and their functional specifications [1] Applying DYNAMATE to disparate varieties of programs may require introducing mutations involving substitutions of types other than integer.

8. Note that substitutions of Boolean expressions are subsumed by the process of predicate extraction.

4.2 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 3.2), 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. However, 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. Runtime checks do not incur such limitations, and can check mutants for invariance in batches; the batch size is determined by how many assertions we can monitor at once, whereas how we partition the mutants in batches does not affect the correctness of the final result.

4.3 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 7). 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 m_μ(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 setting them nondeterministically. Then, an **assume** lists all loop invariants verified across all DYNAMATE iterations by ESC/Java2 (Section 3.4.1). Any formula that is a consequence of these already known invariants is redundant and should not be retained. The final **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.

5 Case Study

We evaluated the DYNAMATE approach by running our prototype on 28 methods from the `java.util` package of OpenJDK Java 1.6.

5.1 Case Study Selection and Preparation

Table 3 lists the 28 methods used in the case study, which we obtained as follows. We initially considered all methods with loops in `java.util`: there are 421 such methods in 50 classes, for a total of 575 loops.

Since DYNAMATE uses ESC/Java2 as static prover, only methods that ESC/Java2 can verify by *manually* providing suitable loop invariants may be within DYNAMATE's capabilities. To identify them, we conducted a preliminary assessment where we tried to manually specify and annotate with loop invariants, and verify using ESC/Java2, methods with loops of `java.util`. Overall, we spent about four person-weeks in the assessment process. Unsurprisingly, we spent most of the time devising the loop invariants and working around ESC/Java2's limitations—precisely what DYNAMATE can provide automation for—whereas writing pre- and postconditions was generally straightforward and quick. Nonetheless, we had to write pre- and postcondition ourselves rather than reusing Leavens et al.'s [15], which include complex JML features unsupported by ESC/Java2 and in fact have not been used for verification of the method implementations.

During the assessment, we tried to verify every method against functional specifications as complete as possible. In the end, we dropped all methods for which we could not get ESC/Java2 to work with non-trivial functional specifications within the available time budget or without resorting to complex features. Here are some common reasons that lead us to drop methods during the assessment phases:

- Methods relying on language types and operations that are not adequately supported by ESC/Java2's encoding and backend SMT solver: non-linear arithmetic, complex bitwise operations, floating point arithmetic, and strings.
- Methods including calls to native code whose semantics is cumbersome or impossible to specify accurately (although we specified a few basic native methods such as `arraycopy`).
- Methods relying on I/O and other JVM services—most notably, reflection features.
- Methods requiring substantial modifications to their implementation, specification, or both to be verified; in particular, we excluded methods whose verification requires ghost code or complex framing conditions.
- Data-structure classes relying on complex class invariants whose specification is a challenge of its own [16].

When overloaded methods operating on different basic types were available, we included the versions operating on **int** and on `Object` (as placeholder for `Integer` in sorting methods); even if the algorithms are the same, the latter typically require more complex annotations since they have to deal with **null** values. By the same token, we dropped variants of methods operating on different types, such as **short** and **byte**, when they were mere duplicates that carried no new challenges for verification.

Overall, we retained the 26 `java.util` methods listed in Table 2. As described in the last column, we slightly modified a few of the methods to make up for features that needlessly hinder automated reasoning (e.g., external iterators and control-flow breaking instructions) or that ESC/Java2 does not fully support (e.g., bitwise operations and "for each" loops). We also factored out the loops in `sort1` and `mergeSort` into separate methods according to the algorithms they implement. Only three method modifications affected the number of loops. Methods `replaceAll` and `reverse` consist of a conditional at the outermost level, with a loop in each branch that caters to the specific features of the collection such as whether it offers random access or external iterators; in our experiments, we only retained the loops (one per method) in the branch corresponding to `Integer` indexed lists. The single loop in `removeLastOccurrence` visits, in reverse order, the elements of a double-ended queue implemented as a circular array. It turns out that the asymmetry in the index range [`head`..`tail`) of valid elements in the queue (from index `head` *included* to index `tail` *excluded*) makes reasoning more complicated when elements are visited backward (from last to first) than when they are visited forward (from first to last, as in `removeFirstOccurrence`). As a result, we found no simple way to reason about `removeLastOccurrence`'s implementation as is without the help of ghost code; in particular, reversing the approach used with `removeFirstOccurrence` does not seem to work. Instead, we refactored `removeLastOccurrence`'s single loop into three different loops with the same semantics but in a form more amenable to reasoning. None of these modifications altered the input/output semantics of the loops or the essence of the original algorithms, even though it is possible that different modifications would also work.

In all, we obtained the 28 methods listed in Table 3, which we used in our case study. The specifications we wrote also include few boilerplate frame conditions, class invariants, and definitions of exceptional behavior; since these are straightforward and do not impact the core of the correctness proofs, Table 3 does

TABLE 2
Signatures of methods in `java.util` selected as a basis for our case study. We show the number of lines (LOC) of each method's body, the number of loops ($|\ell|$, where $m\{n\}$ denotes m outer loops and n inner loops), and which language features were modified to obtain the corresponding methods of the case study in Table 3.

CLASS	METHOD	LOC	$\lvert\ell\rvert$	MODIFIED
ArrayDeque	contains(Object)	11	1	bitwise mask
ArrayDeque	removeFirstOccurrence(Object)	13	1	bitwise mask
ArrayDeque	removeLastOccurrence(Object)	13	1	bitwise mask, conditional structure
ArrayList	clear()	4	1	
ArrayList	indexOf(Object)	10	2	
ArrayList	lastIndexOf(Object)	10	2	
ArrayList	remove(Object)	14	2	refactored calls to `fastRemove`
Arrays	binarySearch0(**int**[],**int**,**int**,**int**)	13	1	bitwise mask
Arrays	equals(**int**[],**int**[])	11	1	
Arrays	fill(**int**[],**int**)	2	1	
Arrays	fill(**int**[],**int**,**int**,**int**)	3	1	
Arrays	fill(Object[],Object)	2	1	
Arrays	fill(Object[],**int**,**int**,Object)	3	1	
Arrays	hashCode(**int**[])	6	1	"for each" loop expressed as regular **for**
Arrays	hashCode(Object[])	6	1	"for each" loop expressed as regular **for**
Arrays	sort1(**int**[],**int**,**int**)	42	5 {3}	loops factored out into `insertionSort_b` and `quicksortPartition`
Arrays	mergeSort(Object[],Object[],**int**,**int**,**int**)	25	3 {1}	loops factored out into `insertionSort_a` and `merge`, for `Integers`
Arrays	vecswap(**int**[],**int**,**int**,**int**)	2	1	refactored local variables
Collections	replaceAll(List⟨T⟩,T,T)	37	4	retained only one replacement loop that uses no external iterators
Collections	reverse(List⟨T⟩)	13	2	retained only one replacement loop that uses no external iterators
Collections	sort(List⟨T⟩)	7	1	replaced external iterators with direct access
Vector	indexOf(Object,**int**)	10	2	
Vector	lastIndexOf(Object,**int**)	12	2	
Vector	removeAllElements()	4	1	
Vector	removeRange(**int**,**int**)	7	1	
Vector	setSize(**int**)	9	1	
	TOTAL	289	41 {4}	

not list them. We made the resulting fully annotated 28 methods obtained by this process publicly available for repeatability and for related research on verification of real Java code:

https://bitbucket.org/caf/java.util.verified/

ESC/Java2 as a sound verifier. ESC/Java2 does not support a number of Java and JML features [12], [17], such as overflow checks, inherited annotations, string literals, multiple inheritance, and static initializers, whose encoding in unsound with respect to their intended semantics. Unsupported features are compromises that "increase automation, improve performance, and reduce both the number of false positives and the annotation overhead" [18], and as such they are common in the design of static verifiers. To ensure validity of our experiments, we ascertained—by manually inspecting the source code against ESC/Java2's list of documented unsupported features—that the code of our case study uses none of the unsupported Java and JML features on which the verified specifications depend. Regarding loops, our experiments always call ESC/Java2 with the `-loopSafe` option which, as explained in Section 3.4.1, produces a sound semantics of loops thorough invariants.

Positioning. Table 3 gives an overview of the case study subjects, but we should keep in mind that metrics such as lines of code or specification clauses are poor indicators of the complexity of full verification of functional properties. Even the shortest algorithms can be extremely tricky to specify and verify (see [19] for an extreme example), as they may require annotations involving complex disjunctions and quantifiers; the model-based approach helps express complex predicates [1] but it cannot overcome the intrinsic formidable complexity of automated reasoning.

Although the data structure implementations of our examples 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. Besides, even if the full formal verification of complex linked data structures such as hash tables and graphs has made substantial progress in recent years [20], [21], [16], it remains a challenging problem for which full automation is still out of reach. Taking stock, our case study subjects consist of real code that is representative of the state of the art of formal software verification and highlights challenges to providing full automation.

Experimental setup. All experiments were performed on a Ubuntu GNU/Linux system installed on a 2.0 GHz Intel Xeon processor 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. DYNAMATE's ran on the 28 methods of Table 3 annotated with pre- and postconditions but without any loop invariants:

TABLE 3
Annotated methods used as case study, obtained from those in Table 2. We show the number of lines (LOC) of each method's body, the number of pre- ($|P|$) and postcondition ($|Q|$) clauses given as specification, and the number of loops ($|\ell|$, where $m\{n\}$ denotes m outer loops and n inner loops).

CLASS METHOD	LOC	$\|P\|$	$\|Q\|$	$\|\ell\|$
ArrayDeque contains	13	0	2	1
ArrayDeque removeFirstOccurrence	15	0	2	1
ArrayDeque removeLastOccurrence	15	0	2	3
ArrayList clear	9	0	3	1
ArrayList indexOf	12	0	3	2
ArrayList lastIndexOf	12	0	3	2
ArrayList remove	14	0	5	2
Arrays binarySearch0	18	2	2	1
Arrays equals	16	0	1	1
Arrays fill_a (**int** array)	4	1	1	1
Arrays fill_b (**int** array range)	5	2	3	1
Arrays fill_c (Object array)	4	1	3	1
Arrays fill_d (Object array range)	5	1	5	1
Arrays hashCode_a (**int** array)	10	0	1	1
Arrays hashCode_b (Object array)	11	0	1	1
Arrays insertionSort_a (mergeSort, Object array)	15	3	1	2 {1}
Arrays insertionSort_b (sort1, **int** array)	8	3	1	2 {1}
Arrays merge (mergeSort, Object array)	12	8	1	1
Arrays quicksortPartition (sort1, **int** array)	22	3	7	3 {2}
Arrays vecswap	4	5	4	1
Collections replaceAll	39	1	2	1
Collections reverse	15	1	1	1
Collections sort	9	2	3	1
Vector indexOf	12	1	3	2
Vector lastIndexOf	15	1	3	2
Vector removeAllElements	8	0	3	1
Vector removeRange	12	1	3	1
Vector setSize	11	1	3	1
TOTAL	345	37	72	39 {4}

the goal of the experiments was for DYNAMATE to infer loop invariants without additional input.

5.2 Experimental Results

Table 4 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. Specifically, EVOSUITE may fail to find counterexamples to invalid invariants, which thus remain in the set $I \setminus INV$ of candidates neither conclusively falsified nor conclusively proved. This may prevent more general invariants from being inferred, thus jeopardizing the whole verification effort; for example, a surviving invalid invariant $\texttt{i} \in \{0, 1, 2\}$ would shadow a valid invariant $\texttt{i} \geq 0$. Nonetheless, experiments with DYNAMATE normally have high repeatability: 21 of the 25 verified subjects have a success rate above 50%.

DYNAMATE automatically verified 25 of the 28 subjects, with high repeatability.

```
merge:
   destHigh − ic = (mid − p) + (high − q)

quicksortPartition:
   b ≤ c ⟹ x[b] > v

sort:
   ∀ int j: 0 ≤ j ∧ j < a.length ⟹ a[j] = \old(a[j])
   ∀ int j: 0 ≤ j ∧ j < ic ⟹
                  a[j] = l.getElementData()[j]
```

Fig. 10. The four necessary loop invariants that DYNAMATE failed to infer in all runs.

The success rate was 0% for methods `Arrays.merge`, `Arrays.quicksortPartition`, and `Collections.sort`. For these methods, DYNAMATE failed to infer all necessary invariants. The four missing invariants, one for each of `merge` and `quicksortPartition` and two for `sort`, are shown in Figure 10; they have a form that is neither among DAIKON's templates nor GIN-DYN's mutants:

- `merge`'s missing invariant is an equality relation that is not among DAIKON's invariant because it involves six variables but, by default, DAIKON restricts templates to three variables for scalability.
- `quicksortPartition`'s missing invariant is also

TABLE 4

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 PROVED proof obligations; the number of ITERATions of DYNAMATE; the number of PROVED 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 RATE	PROVED	# ITERAT.	# INVARIANTS PROVED	# INVARIANTS MISS.	GIN-DYN WAVES	GIN-DYN CAND.	GIN-DYN FALS.	GIN-DYN TAUT.	TIME SECS.
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

a scalar relation, but DAIKON's templates do not instantiate disjunctive (including implication) templates without additional user input (splitting predicates).

- `sort`'s missing invariants are quantified expressions of the kinds that GIN-DYN supports. However, `sort`'s postcondition has a form that is unrelated to the two invariants, due to `sort`'s peculiar implementation: a call to a native method does the actual sorting of array `a`, and is followed by a loop that copies the result from `a` into `l`. Thus, the loop's invariants are unrelated to sorting, which is just a property carried over by copying.

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 users have to write manually.

On average, DYNAMATE automatically discharged 97% of the proof obligations.

Using exclusively DAIKON's invariants, DYNAMATE could verify only `hashCode_a` and `hashCode_b`; this shows the importance of GIN-DYN to add flexibility to DYNAMATE. The other columns about GIN-DYN in Table 4 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. At the same time, DYNAMATE solves an intrinsically complex task—fully automated correctness proof without loop annotations—compared to the standard goal of dynamic techniques; given the state of the art, we should not have unrealistic expectations regarding its performance.

TOOL	PROOF OBLIGATIONS EXPRESSIBLE	PROOF OBLIGATIONS PROVED	VERIFIED METHODS	TIME
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

TABLE 5
Experimental comparison of DYNAMATE against other tools for automated verification.

5.3 Experimental Comparison

The staggering amount of research on loop invariant generation and automated verification has produced a wide variety of results that are not always directly comparable. 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 or highly specialized examples; we provide more details about some of the most significant exceptions at the end of the section. All in all, we identified three "champions" with these characteristics: INVGEN [22], which uses constraint-based techniques and mainly targets linear numeric invariants; BLAST [6], a software model-checker using CEGAR and predicate abstraction; and `cccheck` [4], the CodeContracts static checker formerly known as Clousot [5], based on abstract interpretation with domains for arrays.

We translated the classes used in DYNAMATE's evaluation (including the specification predicates in `TArrays` and `TLists`) into a form amenable to each tool: to C for INVGEN and BLAST, and to C# for `cccheck`. We tried to replicate the salient features of Java's semantics in each language, without introducing unnecessary complications; for example, Java `Objects` become C **`struct`**s passed to functions as pointer arguments.

We encoded in each tool as many of the ESC/Java2 proof obligations as possible. We used **`assert`**s 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 3.2).

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. Table 5 shows the results.

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" [6]. `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.[9] In particular, DYNAMATE achieved full verification of 25 out of 28 methods, while the other tools verified at most 3 methods.

DYNAMATE automatically verified 28% more proof obligations than state-of-the-art verification tools.

Given that our selection of 28 methods is based on their analyzability with ESC/Java2 (as we describe

9. With the sole exception of a property in `Collections.sort`, which BLAST can check using its context-free option `-cfb`. However, BLAST can also check the negation of the same property, which indicates unsoundness; hence, this exception seems immaterial for the comparison.

at the beginning of Section 5), we do not claim that our results generalize to other kinds of programs or properties such as pointer reachability. 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`; and integrating in DYNAMATE other third-party tools tailored to *some kinds* of invariants.

Other tools. The following table summarizes the crucial features that distinguish DYNAMATE from a few other cutting-edge tools. Only JPF [23] is available (A/U) but is not directly comparable as it is limited to bounded checking (no exhaustive verification). Based on [24], [25], the other techniques also have limitations; see Section 6 for qualitative details.

TOOL	A/U	LIMITATIONS
Java Pathfinder [23]	A	bounded symbolic execution
Vampire [24]	U	linear array access, no nesting
Srivastava et. al [25]	U	requires templates and predicates

5.4 Threats to Validity

External validity concerns the generalization of our results to subjects other than the ones we studied. Regarding specifications, although we followed a particular style (the model-based approach), the style is general enough that it remains applicable to other software [9]. In addition, we wrote specifications as complete as possible with respect to the full functional behavior that can be manually proved with reasonable effort using ESC/Java2; this makes our case study a relevant representative of the state of the art in formal software verification. Targeting DYNAMATE's approach to the verification of properties other than functional, as well as tackling significantly larger programs, belongs to future work.

We took all measures necessary to minimize threats to *internal validity*—which originate in the execution of experiments. To minimize the effect of chance due to the usage of randomized algorithms (EVOSUITE), we repeated each experiment multiple times and considered average behavior; in most cases, chance negatively affected only a small fraction of the runs. We also manually inspected all results (inferred loop invariants), which gives us additional confidence about their internal consistency. DYNAMATE and its modules depend on parameters such as weights, timeouts, and thresholds which might influence experimental results. We ran all subjects using the same parameters, for which we picked default values whenever possible.

Threats to *construct validity* have to do with how appropriate the measures we took are. We measured success in terms of how many proof obligations DYNAMATE discharged automatically. This certainly is a useful metric; assessing other measures such as execution time required per proof obligation belongs to future work.

6 RELATED WORK

Since we cannot exhaustively summarize the huge amount of work on automating program verification indirectly related to DYNAMATE, we focus this section on the problem of inferring loop invariants to automate functional verification; a natural classification is in static and dynamic techniques. Section 5.3 presents a more direct comparison between DYNAMATE and a few selected "champions" of loop invariant inference.

6.1 Static Techniques

Abstract interpretation [26] is a general framework for computing sound 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 pointer reachability [27] and other heap shape properties [28] or on simple "global" correctness properties [29], [30], such as absence of division by zero or null pointer dereference. Domains exist for simple numerical properties such as polynomial equalities and inequalities [31], [32] and interval (bounding) constraints involving scalar variables [33], [34], [35]; these do not include quantified array expressions, which are needed to express the functional correctness of several programs used in DYNAMATE's evaluation, such as searching and sorting algorithms. Only recently have the first abstract domains supporting restricted quantification over arrays been developed [36], [5].

Compared to DYNAMATE, these techniques offer much more scalability and speed, but also incur some limitations in terms of language support and flexibility. In particular, Gulwani et al. [36] require user input in the form of templates that detail the structure of the invariants, and its experiments do not target nested loops (e.g., only inner loops of searching algorithms). The technique behind the CodeContracts static checker [5] also cannot deal with some quantified properties that DYNAMATE can handle (see

Section 5.3 for a direct comparison to cccheck [5]). More generally, abstract interpretation is limited to domains fixed a priori and may lose precision in the presence of unsupported language features, whereas DYNAMATE (largely thanks to its reliance on dynamic analysis) works in principle with any property expressible through JML annotations and arbitrary Java implementations.

Predicate abstraction [37], [38] is a technique to build finite-state over-approximations of programs, which can be seen as a form of abstract interpretation. The applicability of predicate abstraction crucially depends on the set of predicates provided as input, which determine precision and complexity.

Whereas predicates may be collected from the program text following some heuristics [39], [40] or they may be manually specified through annotations [41], [38], DYNAMATE requires no input besides a specified program. However, the two techniques may be usefully combined, with the invariants guessed by DYNAMATE used to build a predicate abstraction.

Predicate abstraction is a fundamental component of the CEGAR (Counter-Example Guided Abstraction Refinement) approach [42] to software model-checking, which features in tools such as SLAM [43] and BLAST [6]. Software model checkers specialize in establishing state reachability properties or other temporal-logic properties, whereas they hardly handle complex invariants involving quantification for functional correctness (such as those central to DYNAMATE's evaluation), as we demonstrate in Section 5.3.

Constraint-based techniques reduce the invariant inference problem to solving constraints over a template that defines the general form of invariants (playing a somewhat similar role to abstract domains in abstract interpretation). The challenge of developing new constraint-based techniques lies in defining expressible yet decidable logic fragments, which characterize extensive template properties.

The state of the art focuses on invariants in the form of Boolean combinations of linear [44], [22], [45] and quadratic [46] inequalities, polynomials [47], [48], [49], restricted properties of arrays [50] and matrices [51], and linear arithmetic with uninterpreted functions [52]. Recent advances include automatically computing least fixed points of recursively defined predicates [53], [54], which are can express the verification conditions of transition systems modeling concurrent behavior.

Since constraint-based techniques rely on decidable logic fragments, they rarely support templates involving quantification. The automata-based approach of [50] 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.

See Section 5.3 for a direct comparison to INVGEN [22].

Using first-order theorem proving. Kovács et al. [55], [24] 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 [56] also uses a modified saturation theorem prover to infer quantified loop invariants describing arrays and linked lists.

Both techniques rely on heuristics that substantially depend on interacting with a custom-modified theorem prover, which limits their practical applicability to programs with "regular" behavior. For example, [55] assumes loops that monotonically increment or decrement a counter variable; more general index arithmetic is not handled. As a result, the example programs demonstrated in [55], [24], [56] are limited to linear access to array elements and no nested loops, and do not include anything as complicated as sorting (also see Section 5.3). While more complex quantified invariants could be generated in principle, doing so normally requires[10] massaging the input programs into a form amenable to the "regularity" assumptions on which the inference technique relies. In contrast, DYNAMATE uses a static prover and other components as black-boxes, and works on real implementation of sorting and searching algorithms.

Totla and Wies [57] apply interpolation techniques to axiomatic theories of arrays and linked lists, which supports invariant inference over expressive domains; however, the examples the technique can handle in practice are still limited to flat loops and straightforward linked list manipulations (the technique for arrays has not been implemented).

Combination of static techniques. HAVOC [58] pioneered using a static verifier to check if candidate assertions are valid: it creates an initial set of candidates (possibly including loop invariants) by applying a fixed set of rules to the available module-level contracts (i.e., module invariants and interface specifications). Like DYNAMATE, HAVOC applies the HOUDINI algorithm to determine which candidates are valid. Using only static techniques, however, may introduce false negatives, that is valid loop invariants being erroneously discarded. In contrast, DYNAMATE discards a large number of invalid candidates by runtime checking, which is immune to false negatives. Static verification is only applied later and, if the program verifier fails, DYNAMATE will enter another iteration of test case generation, trying to bring in additional precision.

JPF (Java Pathfinder) supports heuristics [23] to iteratively strengthen loop invariants from path condi-

10. Laura Kovács, personal communication, 14 March 2014.

tions and intermediate assertions.[11] While JPF mainly targets concurrency errors such as race conditions, these heuristics are also applicable to numeric invariants. However, as the DAIKON experience shows, invariants may not be apparent from the code (e.g., $x < y$ might not appear in the code but be a necessary loop invariant). JPF can also leverage postconditions if they are available, but without mutating them as DYNAMATE does; furthermore, JPF's symbolic-execution approach cannot handle the kind of complex verification conditions that program provers such as ESC/Java2 fully support and, more crucially, is limited to bounded state spaces (such as arrays of bounded size).

HOLA [59] implements an inference technique somewhat similar to JPF, but uses abduction (a mechanism to infer premises from a given conclusion) for strengthening, which gives it more flexibility and generality. However, it is also limited to invariants consisting of Boolean combinations of linear integer constraints (inequalities and modular relations).

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 (also see Section 5.3): 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. In fact, the DYNAMATE approach could also accommodate templates to suggest invariant shapes, thus dramatically narrowing down the search space; conversely, Srivastava and Gulwani's technique could be extended to generate templates from postconditions as DYNAMATE's GIN-DYN does.

Separation logic is an extension of Hoare logic specifically designed to specify and reason about linked structures in the heap [60]. Consequently, the bulk of the work in invariant inference based on separation logic focuses on pointer reachability and other shape properties [61], [62], [63], [64], often by means of abstract interpretation techniques using domains specialized for such properties. Program verifiers based on separation logic also target similar properties, and typically provide a level of automation that is somewhat intermediate between that of "push-button" tools (such as DYNAMATE) and of interactive provers (such as Isabelle and Coq). For example, jStar is a powerful separation-logic verifier that works on real Java code and includes support for loop invariant inference [65]; it focuses on heap reachability properties, requires auxiliary annotations in the form of so-called abstract predicates and custom inference rules, and falls back to user interaction when automated inference is not successful. For these reasons, we did not include separation-logic based tools in our comparison with DYNAMATE. In fact, extending DYNAMATE's techniques to work with separation-logic properties is an interesting direction for future work.

Fully automatic verification requires attacking from multiple angles. In this respect, the DYNAMATE architecture is flexible and can accommodate and benefit from other static approaches.

6.2 Dynamic Techniques

The GUESS-AND-CHECK [66] algorithm infers invariants in the form of algebraic equalities (polynomials up to a given degree). GUESS-AND-CHECK achieves soundness and completeness by targeting a very restricted programming language where all expressions are of Boolean or real type. It starts from concrete program executions, which determine constraints solved using a linear algebra solver; the solution may establish a valid loop invariant or determine a counterexample that can be used to construct new executions. While the overall structure of GUESS-AND-CHECK has some similarities to ours, DYNAMATE targets general-purpose programs, which requires very different techniques.

The work on DAIKON [11] pioneered using dynamic techniques to infer invariants, and has originated a lot of follow-up work. The bulk of it targets, however, assertions such as pre- and postconditions and not loop invariants. Nguyen et al.'s dynamic inference technique [67] is a noticeable exception, which generates loop invariants in the form of polynomial equations over program variables.

A general limitation of dynamic invariant inference is its reliance on predefined templates, which restricts the kinds of invariants that can be inferred. DYNAMATE uses the GIN-DYN approach to work around this limitation: a method's postcondition suggests the possible forms loop invariants may take. The DYNAMATE architecture could also integrate dynamic invariant inference tools with richer or more specialized templates than DAIKON.

6.3 Hybrid Techniques

Recent work in the context of CEGAR techniques has combined static verification and test case generation.

11. The technique of [23] was not fully implemented in JPF at the time of writing (Willem Visser, personal communication, 18 February 2014).

The SYNERGY algorithm [68] avoids unnecessary abstraction refinements guided by concrete inputs generated using directed automatic random testing [69]. The DASH algorithm [70] builds on SYNERGY to handle programs with pointers without whole-program may-alias analysis. Unlike DYNAMATE, these techniques aim at type-state properties (e.g., correct lock usage or absence of resource leaks). Yorsh et al. [71] follow a similar technique of refining abstractions based on the behavior in concrete executions. A model generator determines new concrete states whose execution leads to an unexplored abstract state. The new concrete states could be unreachable, since they are derived based on the abstraction; therefore, they are not as precise as actual tests.

With the overall goal of improving symbolic execution, Godefroid and Luchaup [72] suggest to guess loop invariants from concrete execution traces; the invariants are then used to summarize loop executions when constructing path conditions. The DYSY approach [73] directly mines invariants from the collected path conditions with no required predefined invariant patterns; like DAIKON, it produces a collection of likely invariants, which may include unsound ones; DYSY's constraint-based techniques, however, may provide more flexibility in terms of the invariant forms that can be mined.

Nimmer and Ernst [74], [75] combine dynamic invariant inference à la Daikon with a static program checker. Loop invariants are out of the scope of that work, since they use ESC/Java with unsound loop approximation (i.e., single unrolling); hence, "loop invariants may need to be strengthened to be proved" [74]. Nguyen et al. [76] perform static inductive validation of dynamically inferred program invariants. Their work targets scalar numerical disjunctive invariants (precisely, expressible as inequalities between maxima of scalar terms), which are useful for numerical programs but cannot express some more complex functional properties such as sortedness. DYNAMATE not only provides loop invariants for full program proofs; it also closes the cycle by means of a test input generator, which makes the overall technique completely automatic.

Nori and Sharma [77] 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—an integer expression showing progress. Thus, invariants based on predefined templates are sufficient for termination but are only a small ingredient of DYNAMATE.

7 Conclusions and Future Work

We have presented 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. A novel and important component of our work is a new fully automatic technique for loop invariant detection based on syntactically mutating postconditions. Our DYNAMATE prototype automatically discharged 97% of all proof obligations for 28 methods with loops from `java.util` classes.

Besides general improvements such as scalability and performance, our future work will focus on the following issues:

Better test generators: As any module in DYNAMATE can be replaced by a better implementation of the same functionalities, we are currently investigating dynamic/symbolic approaches to test case generation [78] as well as hybrid techniques integrating search-based and symbolic approaches [79].

More diverse invariant generators: We are exploring *evolutionary* approaches in which invariants are systematically evolved from a grammar over a small set of primitives [80]. We also plan to apply techniques based on symbolic execution (such as the one implemented in DYSY [73]) to provide for more, and more diversified, loop invariant candidates.

Stronger component integration: If a proof fails, a program verifier may be able to produce a *counterexample*, which would make an ideal input to the test case generation module for a new iteration. We are researching how to leverage such additional information, whenever available, while preserving the low coupling of DYNAMATE's architecture.

DYNAMATE can become a platform on which several approaches to test generation, dynamic analysis, and static verification can work in synergy to produce a greater whole. 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/

Acknowledgments

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement nr. [290914] and EU FP7 grant 295261 (MEALS). The second author was partially funded by the Swiss SNF Grant ASII 200021-134976. Klaas Boesche, Alessandra Gorla, Jeremias Rößler, and Christoph Weidenbach provided helpful comments on earlier revisions of this work.

References

[1] C. A. Furia, B. Meyer, and S. Velder, "Loop invariants: Analysis, classification, and examples," *ACM Comp. Sur.*, vol. 46, no. 3, p. Article 34, January 2014.

[2] 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*, vol. 31, no. 3, pp. 1–38, 2006.

[3] N. Polikarpova, C. A. Furia, and B. Meyer, "Specifying reusable components," in *VSTTE*, ser. LNCS, vol. 6217. Springer, 2010, pp. 127–141.

[4] M. Fähndrich and F. Logozzo, "Static contract checking with abstract interpretation," in *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, ser. Lecture Notes in Computer Science, B. Beckert and C. Marché, Eds., vol. 6528. Springer, 2010, pp. 10–30. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-18070-5_2

[5] P. Cousot, R. Cousot, and F. Logozzo, "A parametric segmentation functor for fully automatic and scalable array content analysis," in *POPL*. ACM, 2011, pp. 105–118.

[6] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker Blast," *STTT*, vol. 9, no. 5-6, pp. 505–525, 2007.

[7] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll, "Beyond assertions: Advanced specification and verification with JML and ESC/Java2," in *FMCO*, ser. LNCS. Springer, 2006, pp. 342–363.

[8] C. A. Furia and B. Meyer, "Inferring loop invariants using postconditions," in *Fields of Logic and Computation*, ser. LNCS, vol. 6300. Springer, 2010, pp. 277–300.

[9] N. Polikarpova, C. A. Furia, Y. Pei, Y. Wei, and B. Meyer, "What good are strong specifications?" in *ICSE*. ACM, 2013, pp. 257–266.

[10] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *QSIC*. IEEE Computer Society, 2011, pp. 31–40.

[11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE TSE*, vol. 27, no. 2, pp. 99–123, Feb. 2001.

[12] J. R. Kiniry, A. E. Morkan, and B. Denby, "Soundness and completeness warnings in ESC/Java2," in *Proceedings of the 2006 Conference on Specification and Verification of Component-based Systems*, ser. SAVCBS '06. New York, NY, USA: ACM, 2006, pp. 19–24. [Online]. Available: http://doi.acm.org/10.1145/1181195.1181200

[13] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *Int. J. Softw. Tools Technol. Transf.*, vol. 7, no. 3, pp. 212–232, Jun. 2005. [Online]. Available: http://dx.doi.org/10.1007/s10009-004-0167-4

[14] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for ESC/Java," in *FME*, ser. LNCS, vol. 2021. Springer, 2001, pp. 500–517.

[15] G. T. Leavens, B. Shilling, K. Becker, K. Boysen, C. Clifton, C. Ruby, and D. R. Cok, http://www.eecs.ucf.edu/~leavens/JML-release/javadocs/java/util/package-summary.html, 1998–2002.

[16] N. Polikarpova, J. Tschannen, C. A. Furia, and B. Meyer, "Flexible invariants through semantic collaboration," in *Proceedings of the 19th International Symposium on Formal Methods (FM)*, ser. Lecture Notes in Computer Science, vol. 8442. Springer, 2014, pp. 514–530.

[17] D. R. Cok, J. R. Kiniry, and D. Cochran, "ESC/Java2 implementation notes," Kind Software, Tech. Rep., October 2008, http://goo.gl/BFn1zh.

[18] M. Christakis, P. Müller, and V. Wüstholz, "Collaborative verification and testing with explicit assumptions," in *FM*, ser. Lecture Notes in Computer Science, vol. 7436. Springer, 2012, pp. 132–146.

[19] J.-C. Filliâtre, "Verifying two lines of C with Why3: an exercise in program verification," in *VSTTE*, ser. LNCS, vol. 7152. Springer, 2012, pp. 83–97.

[20] K. Zee, V. Kuncak, and M. C. Rinard, "Full functional verification of linked data structures," in *PLDI*. ACM, 2008, pp. 349–361.

[21] H. Mehnert, F. Sieczkowski, L. Birkedal, and P. Sestoft, "Formalized verification of snapshotable trees: Separation and sharing," in *VSTTE*, ser. LNCS, vol. 7152. Springer, 2012.

[22] A. Gupta and A. Rybalchenko, "InvGen: An efficient invariant generator," in *CAV*, ser. LNCS, vol. 5643. Springer, 2009, pp. 634–640.

[23] C. S. Pasareanu and W. Visser, "Verification of Java programs using symbolic execution and invariant generation," in *SPIN*, ser. LNCS, vol. 2989. Springer, 2004, pp. 164–181.

[24] K. Hoder, L. Kovács, and A. Voronkov, "Invariant generation in Vampire," in *TACAS*, ser. LNCS, vol. 6605. Springer, 2011, pp. 60–64.

[25] S. Srivastava and S. Gulwani, "Program verification using templates over predicate abstraction," in *PLDI*. ACM, 2009, pp. 223–234.

[26] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*, 1977, pp. 238–252.

[27] D. Nikolic and F. Spoto, "Reachability analysis of program variables," *ACM Trans. Program. Lang. Syst.*, vol. 35, no. 4, p. 14, 2013.

[28] B.-Y. E. Chang and K. R. M. Leino, "Abstract interpretation with alien expressions and heap structures," in *VMCAI*, ser. LNCS, vol. 3385. Springer, 2005, pp. 147–163.

[29] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A static analyzer for large safety-critical software," in *PLDI*. ACM, 2003, pp. 196–207.

[30] "WALA: Watson libraries for analysis," http://wala.sourceforge.net, 2006.

[31] "PPL: Parma polyhedra library," http://bugseng.com/products/ppl/, 2008.

[32] R. Bagnara, P. M. Hill, and E. Zaffanella, "The Parma Polyhedra Library," *Sci. of Comp. Prog.*, vol. 72, no. 1–2, pp. 3–21, 2008.

[33] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *POPL*, 1978, pp. 84–96.

[34] "Apron numerical abstract domain library," http://apron.cri.ensmp.fr/library/, 2009.

[35] B. Jeannet and A. Miné, "Apron: A library of numerical abstract domains for static analysis," in *CAV*, ser. LNCS, vol. 5643. Springer, 2009, pp. 661–667.

[36] S. Gulwani, B. McCloskey, and A. Tiwari, "Lifting abstract interpreters to quantified logical domains," in *POPL*. ACM, 2008, pp. 235–246.

[37] S. Graf and H. Saïdi, "Construction of abstract state graphs with PVS," in *CAV*, ser. LNCS, vol. 1254. Springer, 1997, pp. 72–83.

[38] C. Flanagan and S. Qadeer, "Predicate abstraction for software verification," in *POPL*. ACM, 2002, pp. 191–202.

[39] Y. Jung, S. Kong, B.-Y. Wang, and K. Yi, "Deriving invariants by algorithmic learning, decision procedures, and predicate abstraction," in *VMCAI*, ser. LNCS, vol. 5944. Springer, 2010, pp. 180–196.

[40] P. H. Schmitt and B. Weiß, "Inferring invariants by symbolic execution," in *VERIFY*, vol. 259. CEUR-WS.org, 2007.

[41] B. Weiß, "Predicate abstraction in a program logic calculus," *Sci. Comput. Program.*, vol. 76, no. 10, pp. 861–876, 2011.

[42] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, ser. LNCS, vol. 1855, 2000, pp. 154–169.

[43] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *PLDI*. ACM, 2001, pp. 203–213.

[44] M. Colón, S. Sankaranarayanan, and H. Sipma, "Linear invariant generation using non-linear constraint solving," in *CAV*, ser. LNCS, vol. 2725. Springer, 2003, pp. 420–432.

[45] S. Gulwani, S. Srivastava, and R. Venkatesan, "Constraint-based invariant inference over predicate abstraction," in *VMCAI*, ser. LNCS, vol. 5403. Springer, 2009, pp. 120–135.

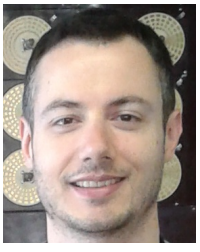
[46] S.Gulwani, S. Srivastava, and R. Venkatesan, "Program analysis as constraint solving," in *PLDI*. ACM, 2008, pp. 281–292.

[47] S. Sankaranarayanan, H. Sipma, and Z. Manna, "Non-linear loop invariant generation using Gröbner bases," in *POPL*. ACM, 2004, pp. 318–329.

[48] D. Kapur, "Automatically generating loop invariants using quantifier elimination," in *Deduction and Applications*, ser. Dagstuhl Seminar Proceedings, vol. 05431, 2006.

[49] E. Rodríguez-Carbonell and D. Kapur, "Generating all polynomial invariants in simple loops," *Journal of Symb. Comp.*, vol. 42, no. 4, pp. 443–476, 2007.
[50] M. Bozga, P. Habermehl, R. Iosif, F. Konečný, and T. Vojnar, "Automatic verification of integer array programs," in *CAV*, ser. LNCS, vol. 5643. Springer, 2009, pp. 157–172.
[51] T. A. Henzinger, T. Hottelier, L. Kovács, and A. Voronkov, "Invariant and type inference for matrices," in *VMCAI*, ser. LNCS. Springer, 2010, pp. 163–179.
[52] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko, "Invariant synthesis for combined theories," in *VMCAI*, ser. LNCS, vol. 4349. Springer, 2007, pp. 378–394.
[53] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *SAT*, ser. LNCS, vol. 7317. Springer, 2012, pp. 157–171.
[54] A. R. Bradley, "SAT-based model checking without unrolling," in *VMCAI*, ser. LNCS, vol. 6538. Springer, 2011, pp. 70–87.
[55] L. Kovács and A. Voronkov, "Finding loop invariants for programs over arrays using a theorem prover," in *FASE*, ser. LNCS, vol. 5503. Springer, 2009, pp. 470–485.
[56] K. L. McMillan, "Quantified invariant generation using an interpolating saturation prover," in *TACAS*, ser. LNCS, vol. 4963. Springer, 2008, pp. 413–427.
[57] N. Totla and T. Wies, "Complete instantiation-based interpolation," in *POPL*. ACM, 2013, pp. 537–548.
[58] S. K. Lahiri, S. Qadeer, J. P. Galeotti, J. W. Voung, and T. Wies, "Intra-module inference," in *CAV*, ser. LNCS, vol. 5643. Springer, 2009, pp. 493–508.
[59] I. Dillig, T. Dillig, B. Li, and K. L. McMillan, "Inductive invariant generation via abductive inference," in *OOPSLA*. ACM, 2013, pp. 443–456.
[60] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 2002, pp. 55–74.
[61] S. Magill, A. Nanevski, E. Clarke, and P. Lee, "Inferring invariants in separation logic for imperative list-processing programs," in *Proceedings of the 3rd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management*, 2006.
[62] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay, "Automatic numeric abstractions for heap-manipulating programs," in *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2010, pp. 211–222.
[63] F. Vogels, B. Jacobs, F. Piessens, and J. Smans, "Annotation inference for separation logic based verifiers," in *Formal Techniques for Distributed Systems (FMOODS/FORTE)*, ser. Lecture Notes in Computer Science, vol. 6722. Springer, 2011, pp. 319–333.
[64] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," *Journal of the ACM*, vol. 58, no. 6, p. 26, 2011.
[65] D. Distefano and M. J. Parkinson, "jStar: towards practical verification for Java," in *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2008, pp. 213–226.
[66] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. Nori, "A data driven approach for algebraic loop invariants," in *ESOP*, ser. LNCS, vol. 7792. Springer, 2013.
[67] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Using dynamic analysis to discover polynomial and array invariants," in *ICSE*. IEEE, 2012, pp. 683–693.
[68] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani, "Synergy: a new algorithm for property checking," in *SIGSOFT FSE*. ACM, 2006, pp. 117–127.
[69] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *PLDI*. ACM, 2005, pp. 213–223.
[70] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons, "Proofs from tests," in *ISSTA*. ACM, 2008, pp. 3–14.
[71] G. Yorsh, T. Ball, and M. Sagiv, "Testing, abstraction, theorem proving: better together!" in *ISSTA*. ACM, 2006, pp. 145–156.
[72] P. Godefroid and D. Luchaup, "Automatic partial loop summarization in dynamic test generation," in *ISSTA*. ACM, 2011, pp. 23–33.
[73] C. Csallner, N. Tillmann, and Y. Smaragdakis, "DySy: dynamic symbolic execution for invariant inference," in *ICSE*. ACM, 2008, pp. 281–290.
[74] J. W. Nimmer and M. D. Ernst, "Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java," in *RV*, 2001.
[75] J. Nimmer and M. Ernst, "Automatic generation of program specifications," in *ISSTA*, 2002, pp. 229–239.
[76] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Using dynamic analysis to generate disjunctive invariants," in *36th International Conference on Software Engineering (ICSE)*, 2014, pp. 608–619.
[77] A. V. Nori and R. Sharma, "Termination proofs from tests," in *ESEC/FSE*. ACM, 2013, pp. 246–256.
[78] K. Jamrozik, G. Fraser, N. Tillmann, and J. de Halleux, "Generating test suites with augmented dynamic symbolic execution," in *TAP*, 2013, pp. 152–167.
[79] J. Malburg and G. Fraser, "Combining search-based and constraint-based testing," in *ASE*. IEEE, 2011, pp. 436–439.
[80] S. Ratcliff, D. R. White, and J. A. Clark, "Searching for invariants using genetic programming and mutation testing," in *GECCO*. ACM, 2011, pp. 1907–1914.

Juan P. Galeotti is a post-doctoral researcher at Saarland University, Saarbrücken, Germany. He received a PhD in computer science from the University of Buenos Aires, Argentina, in 2011. His research interests include software verification, program analysis, automatic test case generation and programming languages design. He has been awarded with the José A. Estenssoro doctoral grant from the YPF Foundation.



Carlo A. Furia is a senior researcher at the Chair of Software Engineering in the Department of Computer Science of ETH Zurich. His main research interests are in formal methods for software engineering. He obtained a PhD in Computer Science from Politecnico di Milano, a Master of Science in Computer Science from the University of Illinois at Chicago, and a Laurea degree in Computer Science Engineering also from Politecnico di Milano.

Eva May was a research assistant at Saarland University, Saarbrücken, Germany until early 2014. Her work focused on program verification, specification mining and automated test case generation.

Andreas Zeller Andreas Zeller is a full professor for Software Engineering at Saarland University in Saarbrücken, Germany. His research concerns the analysis of large software systems and their development process; his students are funded by companies like Google, Microsoft, or SAP. In 2010, Zeller was inducted as Fellow of the ACM for his contributions to automated debugging and mining software archives. In 2011, he received an ERC Advanced Grant, Europe's highest and most prestigious individual research grant, for work on specification mining and test case generation. In 2013, he co-founded Testfabrik AG, a start-up for automatic testing of Web applications.

Gordon Fraser is a lecturer in Computer Science at the University of Sheffield, UK. He received a PhD in computer science from Graz University of Technology, Austria, in 2007. The central theme of his research is improving software quality, and his recent research concerns the prevention, detection, and removal of defects in software. More specifically, he develops techniques to generate test cases automatically, and to guide the tester in validating the output of tests by producing test oracles and specifications.