DynaMate: Dynamically Inferring Loop Invariants for Automatic Full Functional Verification

Juan P. Galeotti[1], Carlo A. Furia[2], Eva May[1], Gordon Fraser[3], and Andreas Zeller[1]

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

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

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

Abstract. DYNAMATE is a tool that automatically infers loop invariants and uses them to prove Java programs correct with respect to a given JML functional specification. DYNAMATE improves the flexibility of loop invariant inference by integrating static (proving) and dynamic (testing) techniques with the goal of combining their complementary strengths. In an experimental evaluation involving 26 Java methods of `java.util` annotated with JML pre- and postconditions, it automatically discharged over 97% of all proof obligations, resulting in automatic complete correctness proofs of 23 out of the 26 methods.

1 The Challenge of Automating Program Verification

Full automation still eludes generic program verification techniques. The neologism *auto-active*—a portmanteau of *automatic* and *interactive*—has been introduced [11] to characterize some state-of-the-art tools for the formal verification of arbitrary functional properties of code. SMT-based verifiers such as ESC/Java2 [2], Dafny [10], and VCC [3] do not depend on a step-by-step interaction with the user, and hence are not purely interactive tools; but they still require substantially more input than just a program and its functional specification (typically given in the form of pre- and postcondition). For programs with loops, *loop invariants* are a crucial ingredient of any formal correctness proof; but the support to automatically infer loop invariants is generally limited and rarely available as part of the same tools used to perform verification. The general expectation is that users will provide detailed additional annotations (including loop invariants) whenever the tool needs them. DYNAMATE aims at providing more automation in these situations.

How DYNAMATE works. The DYNAMATE tool presented in this paper combines different techniques with the overall goal of providing *fully automatic* verification of programs with loops. The only required input to DYNAMATE is a Java program (method) annotated with a JML functional specification (pre- and postcondition). DYNAMATE will try to construct a correctness proof of the program with respect to the specification; to this end it will infer necessary loop invariants. Even in the cases where it fails to find all required loop invariants, DYNAMATE still may find *some* useful invariants and use them to discard some proof obligations, thus providing *partial* verification.

We presented the details of how DYNAMATE works in a companion paper [8]. Figure 1 highlights its components and their high-level interactions: the program and its JML specification (center) are first fed into a test generator (left, EVOSUITE [6] in the current implementation), which generates executions covering possible behavior. Two dynamic invariant detector techniques (top) suggest possible loop invariants, based both on fixed patterns (DAIKON [5]) and on postconditions (GIN-DYN, a component designed as part of DYNAMATE). The candidates not invalidated by the generated runs are then fed into a static program verifier (right, ESC/Java2 [2] invoked with the `-loopsafe` option for sound verification of unbounded loops). When the verifier cannot produce a program proof (bottom), the test generator initiates another iteration where it tries to produce new tests that falsify candidates unproven as of yet.

While any test case generator could work with DYNAMATE, our prototype integrates the search-based test generator EVOSUITE. 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 candidates that are in fact loop invariants, and hence won't be falsified) do not ultimately limit search effectiveness. A directed search is also useful to guide the successive iterations searching for new tests that specifically try to exercise unproven candidates under new conditions.

Advantages of DYNAMATE. The integration of techniques and tools in DYNAMATE compensates for individual shortcomings and achieves a greater whole in terms of flexibility and degree of automation. Dynamic techniques are capable of conclusively invalidating large amounts of loop invariant candidates, thus winnowing a smaller set of *candidate* invariants that hold in all executions, and can test candidates in isolation (dependencies are not an issue). This leaves the static verifier with a more manageable task in terms of number of candidates to check at once. The GIN-DYN component is an original contribution of DYNAMATE. Based on the observation that loop invariants can often be seen as weakened forms postconditions [7], GIN-DYN derives loop invariant candidates by mutating postconditions. This enables inferring loop invariants that are not limited to predefined templates but stem from the annotated Java program

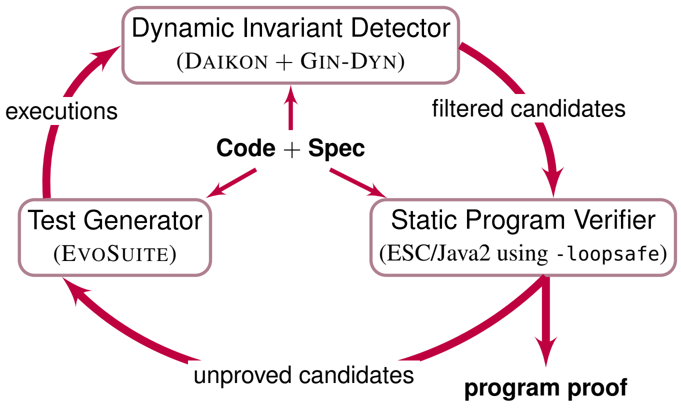


Fig. 1: How DYNAMATE works.

under analysis. DYNAMATE still avails of the advantages of static techniques in terms of soundness: the static verification module scrutinizes the output of the dynamic parts until it can verify it (and uses verified invariants to construct a correctness proof).

2 Using DYNAMATE

We briefly present DYNAMATE in action on the implementation of binary search available in class `java.util.Arrays` from Java's JDK. [4]

```
1 /*@ requires a ≠ null
2   @ requires TArrays.within(a, fromIndex, toIndex)
3   @ requires TArrays.sorted(a, fromIndex, toIndex);
4   @ ensures \result ≥ 0 ⟹ a[\result] = key;
5   @ ensures \result < 0 ⟹ ¬TArrays.has(a, fromIndex, toIndex, key);    @*/
6   private static int binarySearch0(int[] a, int fromIndex, int toIndex, int key)
```

Fig. 2: JML specification of the binary search method from `java.util`. The specification includes a precondition (**`requires`**) and two postconditions (**`ensures`**)

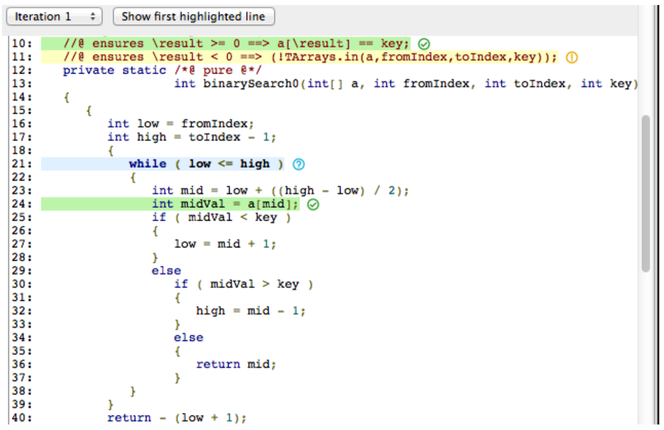
The input to DYNAMATE consists of the method `binarySearch0` annotated with the JML specification of Figure 2. Note that predicate `has` is a shorthand for a quantification over `[fromIndex..toIndex)`; dealing with quantified invariants is a challenges for fully automatic verification. When it starts, DYNAMATE opens an HTML report, which shows the program and specification with all elements (statements or annotations) that trigger an ESC/Java2 warning highlighted in yellow. Clicking on a highlighted element displays its current status, including ESC/Java2's warning messages.

After each iteration of its main loop (Figure 1), DYNAMATE updates the report: elements for which all associated ESC/Java2 warnings have been discharged are highlighted in green. In addition, users can inspect the generated loop invariants by clicking on any loop header. By default only verified loop invariants are shown (in green). Candidate invariants can be viewed (in yellow) by de-selecting a check-box. These candidates have not been falsified by a test, nor have they been verified by ESC/Java2.

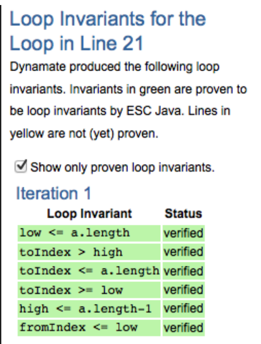
Figure 3 shows a report after the first iteration on `binarySearch0`: DYNAMATE has proven several simple scalar invariants for the selected loop. These simple invariants come from predefined templates; they are sufficient to prove the first postcondition (line 4 in Figure 2) and to show that array accesses are within bounds.

As DYNAMATE continues its search, it uses the postconditions as a basis for more complex invariants. In the example, the postcondition on line 5 in Figure 2 (corresponding to when the search returns unsuccessfully) mentions predicate `¬has(a,fromIndex, toIndex, key)`; DYNAMATE mutates its arguments and checks if any of the mutations are loop invariants. Among many mutations, `¬has(a, fromIndex, low, key)` and `¬has(a, high + 1, toIndex, key)` are valid loop invariants, essential to establishing the postcondition. DYNAMATE finds them during iteration # 9, validates them, and uses them to prove the second postcondition. This concludes DYNAMATE's run, which terminates successfully having achieved full verification. Upon terminating, the

[4] DYNAMATE's output report for this example is available at `http://goo.gl/7TxE9d`.



```
10:   //@ ensures \result >= 0 ==> a[\result] == key;
11:   //@ ensures \result < 0 ==> (!TArrays.in(a,fromIndex,toIndex,key));
12:   private static /*@ pure @*/
13:                int binarySearch0(int[] a, int fromIndex, int toIndex, int key)
14:   {
15:      {
16:         int low = fromIndex;
17:         int high = toIndex - 1;
18:         {
21:            while ( low <= high )
22:            {
23:               int mid = low + ((high - low) / 2);
24:               int midVal = a[mid];
25:               if ( midVal < key )
26:               {
27:                  low = mid + 1;
28:               }
29:               else
30:                  if ( midVal > key )
31:                  {
32:                     high = mid - 1;
33:                  }
34:                  else
35:                  {
36:                     return mid;
37:                  }
38:            }
39:         }
40:         return - (low + 1);
```


Loop Invariants for the Loop in Line 21

Dynamate produced the following loop invariants. Invariants in green are proven to be loop invariants by ESC Java. Lines in yellow are not (yet) proven.

[✓] Show only proven loop invariants.

Iteration 1

Loop Invariant	Status
`low <= a.length`	verified
`toIndex > high`	verified
`toIndex <= a.length`	verified
`toIndex >= low`	verified
`high <= a.length-1`	verified
`fromIndex <= low`	verified

Fig. 3: DYNAMATE's report after iteration # 1 on `binarySearch0`. Verified statements and annotations (first and last highlighted element) are shown in green, unverified ones in yellow. Loop headers are highlighted in light blue. The right frame shows the proven loop invariants for the selected loop.

tool reports all inferred loop invariants—including those listed in Figure 4—which include both scalar invariants and quantified ones (obtained by mutating postconditions).

In spite of its brevity in lines of code, automatically verifying `binarySearch0` without extra input in the form of loop invariants or other annotations is a task that challenges most fully-automatic verifiers. In fact, we tried to verify the same method against the specification in Figure 2 using the state-of-the-art automatic tools INVGEN [9], BLAST [1], and `cccheck` [4]. None of them could complete a correctness proof of `binarySearch0` against its full functional specification.

```
fromIndex ≤ low ∧ low ≤ high + 1 ∧ high < toIndex
¬TArrays.has(a,fromIndex,low,key)
¬TArrays.has(a,high+1,toIndex,key)
```

Fig. 4: Loop invariants inferred by DYNAMATE.

3 Empirical Evaluation

We evaluated DYNAMATE on 26 methods with loops from the `java.util` standard package of Java, including algorithms operating on data structures such as arrays, lists, deques, and vectors. To account for the randomized algorithm used by EVOSUITE's test-case generator, we ran each example 30 times; column *success rate* in Table 5 reports the percentage of those runs that found a full correctness proof. The other columns report means over the 30 runs: the percentage of proven proof obligations[5]; the number

[5] These include pre- and postcondition checks, class invariant checks, and implicit checks for out-of-bound array accesses and **null** dereferencing.

of iterations of the DYNAMATE algorithm; the number of proven invariants; and the total running time.

DYNAMATE was never successful only with methods `merge0`, `quicksortPartition`, and `sort`, for a total of 4 missing necessary invariants, one for each of `merge0` and `quicksortPartition` and two for `sort`. These invariants have a form that is neither among DAIKON's templates nor among GIN-DYN's mutants. We repeated the experiments by manually adding the four missing invariants; as expected, DYNAMATE successfully verified the methods.

Since mutating postconditions is a heuristic approach, it is bound to fail on *some* examples. However, previous analysis [7] and the results of DYNAMATE's experiments suggest that the heuristics if *often* applicable—and even when it cannot suggest all necessary invariants it often can provide partial, useful instances. In all, we gain in flexibility, but we cannot expect to overcome intrinsic limitations due to dealing with logic fragments including quantifiers that are undecidable in general.

CLASS	METHOD	SUCCESS RATE	OBLIGATIONS PROVEN	DYNAMATE ITERATIONS	INVARIANTS PROVEN	TOTAL TIME
`ArrayDeque`	`contains`	57 %	98 %	7	14	2158 s
`ArrayDeque`	`removeFirstOccurrence`	53 %	98 %	7	14	2180 s
`ArrayDeque`	`removeLastOccurrence`	87 %	99 %	9	43	3281 s
`ArrayList`	`clear`	70 %	95 %	6	9	1524 s
`ArrayList`	`indexOf`	23 %	91 %	7	16	1914 s
`ArrayList`	`lastIndexOf`	20 %	93 %	6	14	1574 s
`ArrayList`	`remove0`	23 %	93 %	7	16	2065 s
`Arrays`	`binarySearch0`	100 %	100 %	11	30	4200 s
`Arrays`	`equals0`	100 %	100 %	7	7	2240 s
`Arrays`	`fill0`	100 %	100 %	6	5	1391 s
`Arrays`	`fill1`	100 %	100 %	7	15	1880 s
`Arrays`	`fillInteger0`	100 %	100 %	6	7	1375 s
`Arrays`	`fillInteger1`	100 %	100 %	7	18	1857 s
`Arrays`	`hashCode0`	100 %	100 %	2	4	389 s
`Arrays`	`hashCodeInteger`	100 %	100 %	2	4	343 s
`Arrays`	`insertionSort1`	100 %	100 %	11	73	4512 s
`Arrays`	`merge0`	0 %	90 %	11	78	8034 s
`Arrays`	`quicksortPartition`	0 %	94 %	9	57	5657 s
`Arrays`	`vecswap`	100 %	100 %	8	18	2698 s
`Collections`	`replaceAll`	77 %	97 %	6	16	1801 s
`Collections`	`sort`	0 %	73 %	9	17	3933 s
`Vector`	`indexOf1`	100 %	100 %	6	24	1698 s
`Vector`	`lastIndexOf1`	90 %	99 %	7	19	1859 s
`Vector`	`removeAllElements`	100 %	100 %	5	12	1218 s
`Vector`	`removeRange0`	63 %	96 %	7	17	2574 s
`Vector`	`setSize`	100 %	100 %	7	31	2003 s
	AVERAGE	72 %	97 %	7	22	2475 s

Table 5: Experimental results for DYNAMATE on methods from `java.util`.

Many more details on the experiments, as well as a detailed comparison of DYNAMATE against state-of-the-art tools are presented in a companion paper [8].

4 Conclusions

DYNAMATE's prototype is currently quite limited in terms of scalability, as it takes a considerable amount of time even on structurally simple methods. However, over 65% of the total time is taken up by testing. Even if dynamic techniques are generally slower than purely static ones, there are significant margins to improve the implementation for speed by customizing the test generator (which is currently used as black box) to cater to DYNAMATE's special requirements.

These details should not, however, distract us from assessing DYNAMATE's specific contributions with the right poise: fully automated program verification features an intrinsic formidable complexity; and even the shortest algorithms (in terms of lines of code) may require complex invariants [12]. DYNAMATE worked successfully on real code annotated with non-trivial (often complete) functional correctness specifications. It automatically built correctness proofs for 23 out of 26 subjects[6]; and discharged over 97% of all proof obligations on average. These results demonstrate the benefits of integrating static and dynamic verification techniques with complementary strengths and shortcomings, and improve over the state of the art in terms of complete automation and flexibility.

Availability: The current prototype of DYNAMATE is available for download at `http://www.st.cs.uni-saarland.de/dynamate-tool/`. The download page includes a virtual-machine image with all dependencies as well as scripts to run the examples mentioned in Section 3.

Acknowledgments: This work was funded by the European Research Council (ERC) Advanced Grant "SPECMATE – Specification Mining and Testing". The second author was partially funded by the Swiss SNF Grant ASII 200021-134976.

References

1. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. STTT 9(5-6), 505–525 (2007)
2. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In: FMCO. pp. 342–363. LNCS, Springer (2006)
3. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: TPHOLs. LNCS, vol. 5674, pp. 23–42. Springer (2009)
4. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: POPL. pp. 105–118. ACM (2011)
5. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE TSE 27(2), 99–123 (2001)
6. Fraser, G., Arcuri, A.: Evolutionary generation of whole test suites. In: QSIC. pp. 31–40. IEEE Computer Society (2011)
7. Furia, C.A., Meyer, B., Velder, S.: Loop invariants: Analysis, classification, and examples. ACM Comp. Sur. 46(3) (2014)
8. Galeotti, J.P., Furia, C.A., May, E., Fraser, G., Zeller, A.: Automating full functional verification of programs with loops. Submitted (July 2014), `http://arxiv.org/abs/1407.5286`
9. Gupta, A., Rybalchenko, A.: InvGen: An efficient invariant generator. In: CAV. LNCS, vol. 5643, pp. 634–640. Springer (2009)
10. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR-16 (Dakar). LNCS, vol. 6355, pp. 348–370. Springer (2010)
11. Leino, K.R.M., Moskal, M.: Usable auto-active verification. In: Usable Verification Workshop. `http://fm.csl.sri.com/UV10/` (2010)
12. Zee, K., Kuncak, V., Rinard, M.C.: Full functional verification of linked data structures. In: PLDI. pp. 349–361. ACM (2008)

[6] The average success rate is below $23/26 = 88\%$ because not all repeated runs succeeded.