Search-based Inference of Class Invariants

Juan Manuel Copia
IMDEA Software Institute
Universidad Politécnica de Madrid
Spain

Facundo Molina
Alessandra Gorla
IMDEA Software Institute
Spain

Nazareno Aguirre
Pablo Ponzio
Universidad Nacional de Rio Cuarto
Argentina

Abstract

Many techniques in formal verification and software testing rely on repOk routines to verify the consistency and validity of software components with complex data representations. A repOk function encodes the state properties necessary for an instance to be a valid object of the class under analysis, enabling early error detection and simplifying debugging. However, writing a correct and complete repOk can be challenging. This paper introduces Express, the first search-based algorithm designed to automatically generate a correct repOk for a given class. Express leverages simulated annealing, using the source code and test suite of the class under analysis to iteratively construct a repOk. We demonstrate how Express works on the LinkedList class of the Java standard library, and show that it produces a correct and complete repOK.

ACM Reference Format:
Juan Manuel Copia, Facundo Molina, Alessandra Gorla, Nazareno Aguirre, and Pablo Ponzio. 2025. Search-based Inference of Class Invariants. In *Genetic and Evolutionary Computation Conference (GECCO '25 Companion), July 14–18, 2025, Malaga, Spain.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3712255.3726698

1 Introduction

Software reliability is a core aspect of software quality, and a primary concern in software engineering [16]. Advances in automated program analysis have enabled the efficient creation of extensive program input sets and the examination of large-scale program executions [5, 12, 26, 28], facilitating automated reliability analyses. However, determining whether such executions exhibit correct behavior or uncover defects requires a formal specification of the program's intended behavior. This challenge, commonly referred to as the oracle problem [3], remains a significant problem in software testing. In object-oriented design, where the software is organized into classes, intended behavior can be specified through constructs such as preconditions, postconditions, and class invariants. Among these, *class invariants* are particularly important because they define constraints on the state of a class's objects that must hold true throughout the object's lifecycle. For instance, a class invariant might specify that a list-based object representation must never contain duplicate elements.

Class invariants can be captured in code by means of *repOk routines*, which explicitly check whether an object satisfies the invariants defined for its class. A repOk routine is an imperative function that encodes the class's invariants and can be used to verify the consistency of objects during testing, debugging, and runtime monitoring. These routines are particularly valuable in enabling automated analyses such as test generation [1, 5, 8, 18], bug detection [17, 26], and verification [9, 13, 14, 17]. Despite their importance, repOk routines are rarely written manually due to their complexity, and class invariants are often described ambiguously in natural language comments, limiting their utility in practice.

To address this gap, many techniques have been proposed for automatically inferring specifications, including class invariants [2, 4, 10, 11, 15, 19, 21–25, 29]. Daikon [11] pioneered dynamic invariant detection based on predefined templates, but its expressiveness is limited. Tools like SpecFuzzer [22] and Deryaft [19] produce more sophisticated results, but their output is either hard to integrate into code or lacks completeness.

To address these limitations, we propose Express, a framework based on simulated annealing that automatically generates class invariants in the form of imperative repOk routines. Express starts from the source of a target class and its accompanying test suite, performing a search to construct a repOk routine. The primary objective is to ensure the repOk is *correct* with respect to the test suite, meaning it does not discard any valid instance created during test execution. The secondary objective is to achieve *completeness*, discarding as many invalid instances as possible. To support the latter, we generate a set of allegedly invalid instances through mutations of valid instances, inspired by prior work [24].

Using Express on the LinkedList class implementation available in the Java standard library we show that we obtain a correct and complete repOk.

2 Express Approach

Express is a novel search-based algorithm designed to automatically infer class invariants by generating executable repOk routines. The approach consists of two main phases: (1) *Object Generation*, where valid and invalid instances of a class are created, and (2) *Search Process*, which uses simulated annealing to refine the repOk method.

In the Object Generation Phase, valid instances are collected by executing test cases, assuming they represent correct states. Invalid instances are then produced by mutating valid ones using two strategies: modifying reference fields to produce structures that violate heap-constraints (e.g. acyclicity), and altering primitive values to yield structures that violate primitive-constraints (e.g. order of nodes in a BST). These invalid instances play a critical role in refining the repOk method, as the goal is to reject (most of) them while still accepting valid instances.

The Search Process Phase consists of a four-stage simulated annealing search that iteratively improves the repOk method. Initially, repOk is a trivial method that always returns true. Through a guided search, Express adds and removes constraints to refine the method: i) Initialization Search: Ensures essential reference constraints (e.g., non-null checks). ii) Traversal Search: Identifies how objects should be traversed, constructing traversal logic based on the class's type graph. iii) Heap Constraints Search: Introduces conditions on reference relationships (e.g., ensuring bidirectional pointers are consistent). iv) Primitive Constraints Search: Incorporates constraints on numerical fields (e.g., ensuring size matches the number of elements). The Objective Function drives the search, prioritizing *correctness* by minimizing false negatives (valid instances wrongly rejected) and false positives (invalid instances wrongly accepted). It also incorporates a secondary penalty to favor concise repOk methods. By leveraging simulated annealing, Express effectively escapes local minima, allowing it to discover correct traversal and structural constraints.

2.1 Object Generation

The first phase of Express involves generating *valid* and *invalid* instances of the target class, which are crucial for guiding the search process. The set of valid instances is derived by executing the provided test cases. Since the test cases interact with the public API of the target class, we assume that the objects generated during their execution represent valid states. Invalid instances, on the other hand, are systematically generated by applying targeted mutations to the valid instances. The goal is to introduce object states that deviate from the intended behavior of the class, thereby challenging the `repOk` method to correctly identify these deviations. Two mutation strategies are employed:

- **Heap-constraint violations:** A reference-typed field of a valid instance is randomly selected, and deliberately modified to simulate an invalid state. Three kind of mutations are performed: assigning `null` to the field, substituting the current value of the field with a newly created object of the same type, make the field reference a different object from the same structure.
- **Primitive-constraint violations:** A primitive-typed field is randomly selected and its value is replaced with a randomly generated value.

It is important to note that not all mutated instances are necessarily invalid. However, this does not hinder the search process as the objective function prioritizes the classification of valid instances, and considers correctly classifying most of the invalid instances as a secondary goal (see Section 2.2). Notice that, the generation of invalid instances through mutation-based strategies is a common practice in related work [24].

2.2 Objective Function

The objective function drives the search process in each stage, balancing correctness and conciseness: Specifically, it evaluates the effectiveness of the current `repOk` method in distinguishing valid instances from invalid ones, and it is defined as follows:

$$\text{Objective Function: } \begin{cases} \text{MAX}, & \text{if } \#\text{FN} > 0 \\ \#\text{FP} + \frac{L}{L_{\max}}, & \text{otherwise} \end{cases}$$

where #FN represents the number of false negatives, i.e., valid instances incorrectly classified as invalid by the current `repOk` method; #FP represents the number of false positives, i.e., invalid instances incorrectly classified as valid; L is the length of the `repOk` method in characters; and $L_{\max}$ is a predefined maximum length.

The function penalizes false negatives heavily to ensure correctness. Thus, the objective function assigns the worst possible value (`MAX`) when #FN > 0.

When the number of false negatives is zero, the goal is to minimize the number of false positives and the length of the `repOk` method. Thus, in this case, the objective function value is given by the total number of false positives plus a linear penalty for longer methods. Concretely, the penalty is introduced by term $\frac{L}{L_{\max}}$, which favors more concise solutions. In our experiments, we set $L_{\max}$ to 5000, ensuring that the penalty for method length remains secondary to the primary objective of minimizing false negatives and false positives. This design balances correctness (avoiding incorrect classifications) with simplicity (favoring shorter solutions).

2.3 The Search Process

Express performs a four-stage simulated annealing search to iteratively refine the `repOk` method. It starts with a trivial method that always returns `true`, then gradually introduces constraints to reject invalid instances while accepting valid ones. The search is guided by an objective function that prioritizes correctness—accepting fewer invalid instances yields a better score. Each stage runs until the temperature reaches zero, controlled by shared hyperparameters: the cooling rate and initial temperature.

Each stage targets a specific type of constraint: *Initialization Search* enforces basic reference field constraints (e.g., nullability); *Traversal Search* builds traversal logic based on the class's type graph; *Heap Constraints Search* adds reference consistency checks (e.g., bidirectional links); and *Primitive Constraints Search* introduces checks over numerical fields (e.g., size).

Across stages, mutation operators add or remove conditional checks of the form:

```
if (<CONSTRAINTS-TO-CHECK>) return false;
```

This design is inspired by the guidelines in Korat [5]. Stage 2 also includes additional operators for traversal construction, described below. Operators are applied randomly to enable exploration and refinement, with removal operators included to backtrack and escape local minima. We now describe each stage in detail.

2.3.1 *Stage 1: Initialization Search.*

In this stage, Express identifies constraints on reference-typed fields that govern their initialization. These constraints address properties such as nullability, non-nullability, and inter-field relationships. For example, in a linked list implementation with a dummy `header` node, one such constraint might ensure that the `header` field is never null:

```
if (header == null) return false;
```

To discover these constraints, this stage employs three generic operators: (1) single-constraint checks, (2) multiple-constraint checks, and (3) constraint removal. Each operator is instantiated multiple times using randomly generated constraints based on the fields of the target class. This process reflects established practices for implementing `repOk` routines, where verifying proper initialization is a crucial step before checking more complex structural invariants and properties [6, 7].

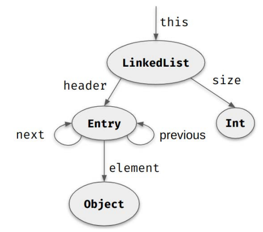

Figure 1: Type graph of the LinkedList class.

```
boolean traverse(<CYCLIC-TYPE> root, Set<CYCLIC-TYPE> visited) {
  /*
    MUTABLE BLOCK OF CODE
  */
  <CYCLIC-TYPE> current = root;
  while (current != null && current.<CYCLIC-FIELD> != root) {
    /*
      MUTABLE BLOCK OF CODE
    */
    if (current.<CYCLIC-FIELD> != null) {
      if (!visited.add(current.<CYCLIC-FIELD>)) {
        return false;
      }
    }
    current = current.<CYCLIC-FIELD>;
  }
  /*
    MUTABLE BLOCK OF CODE
  */
  return true;
}
```

Figure 2: Template for traversing circular references.

2.3.2 Stage 2: Traversal Search. The second stage of Express focuses on identifying suitable traversal strategies for the target class. Traversals are essential in `repOk` methods, as different structures—e.g., circular/non-circular linked lists and trees—require distinct approaches to correctly classify instances.

To guide this, Express constructs a *type graph* representing relationships between the class's fields and their types [21, 24]. For example, in the `LinkedList` type graph (Figure 1), the `next` and `previous` fields are identified as traversable.

This stage uses generic traversal operators that instantiate templates based on the type graph. A key feature is the inclusion of aliasing checks to prevent infinite loops in cyclic structures, ensuring correct handling of both valid and mutated invalid cases.

Figure 2 shows a generic template for traversing circular references. Placeholders for *mutable code blocks* are refined in later stages to tailor the traversal logic. The template captures a general algorithm for navigating cyclic references while addressing aliasing and infinite loops. For instance, substituting `<CYCLIC-TYPE>` and `<CYCLIC-FIELD>` with `Entry` and `next` yields a routine that visits all entries via `next`. Multiple traversal operators can be combined within a `repOk`. Beyond constructing traversals, operators also embed their invocation as conditional checks, e.g.:

```
if (!traverse(...)) return false;
```

This stage utilizes five traversal templates tailored to various scenarios, including arrays, classes implementing the `Iterable` interface, circular references, non-circular references, and multiple references. Furthermore, it incorporates five additional operators to perform operations such as invoking, replacing, removing, and unifying traversal routines. Together, these templates and operators ensure that the traversal strategies are flexible and adaptable to the diverse structural characteristics of the target class.

2.3.3 Stage 3: Heap Constraint Search. Express refines the analysis by focusing on constraints related to heap-structured data. The operators introduced here add conditional checks to validate relationships between reference-typed fields, such as equality, inequality, nullability, and set membership. For instance, the following constraint verifies that the `next` and `previous` fields in a linked structure maintain a correct relationship:

```
if (current.next.previous != current)
  return false;
```

Constraints like this are generated by selecting a random relational operator (e.g., == or !=) and deriving field references from the variable scope where the check will be applied. In the example above, the constraint is derived from field accesses on the local variable `current`.

2.3.4 Stage 4: Primitive Constraints Search. The final stage focuses on constraints involving primitive-typed fields. In this stage, operators introduce comparison checks into the `repOk` routine, utilizing binary operators such as <, >, <=, >=, ==, and !=. These checks are applied to primitive fields and can involve expressions derived from local variables or other field references.

For example, the following constraint verifies that the `size` field is consistent with the actual number of elements in the `visited` set:

```
if (size != visited.size()) return false;
```

This stage employs 12 mutators that integrate these constraints into either the main `repOk` method or the mutable sections of traversal templates created in earlier stages. By refining these sections, Express adapts the `repOk` method to validate primitive constraints, ensuring that both structural and value-based properties of the target class are consistently maintained.

2.3.5 Design of the Algorithm. The design of Express is inspired by our own observations about the common structure of `repOk` routines, when following Korat's guidelines. Each stage of the algorithm builds iteratively upon the results of the previous stage, targeting a specific set of constraints that can be identified independently of others. This staged approach reduces the complexity of the search space, enabling the algorithm to efficiently identify the required constraints for the target class.

Simulated annealing is well-suited to this problem because it can accept worse intermediate solutions. For instance, adding a traversal increases the `repOk` length and worsens the objective function. This flexibility enables exploration of regions that may lead to better solutions, helping the algorithm escape local minima and discover correct traversal strategies or structural constraints.

3 Preliminary Evaluation

We evaluate our approach focusing on the following research question: *How effective is Express in generating class invariants?* We selected `LinkedList` as case study, for which ground truth invariants are already known from our previous work [6, 7]. To run Express, we first generated a diverse set of valid instances using

EvoSuite [12]. We then manually added additional test cases to increase instance diversity. We used such test executions to collect the generated instances as the set of valid instances.

We compared Express inferred invariants against ground truth invariants using Korat [5], a bounded exhaustive input generation tool. Given a class and its ground truth invariant, we collect all valid and invalid instances explored by Korat within a scope of 9.

For the LinkedList class under analysis, Korat generated 9 valid and 231 invalid input instances. The invariant generated by Express correctly infers properties such as the circularity of the list implementation and the consistency between the size of the list and the actual number of elements. Moreover, it accepts all 9 valid instances, and correctly rejects all 231 invalid instances. Thus, for this case study, Express produces invariants that are both *correct* and *complete*.

4 Related Work

Several approaches have been proposed to infer class invariants. Deryaft infers representation invariants as imperative Java methods [19], similar to Express, but relies on fixed templates like Daikon, limiting expressiveness. Machine learning has also been explored. Molina et al. [23] train neural networks to classify valid and invalid instances based on test executions. While sometimes effective, neural networks hinder interpretability and debugging. Express, in contrast, generates transparent, executable invariants. Other work focuses on postconditions rather than class invariants. SpecFuzzer [22] and EvoSpex [24] generate postconditions via grammar-based fuzzing and genetic algorithms, respectively. They use a declarative assertion language [22, 24] with reachability operators. Express uses an operational language (Java), which is more practical for automated tools and familiar to developers. GAssert [29] also employs evolutionary techniques, but supports only simple logical and arithmetic constraints, lacking quantification and reachability. SpecFuzzer, EvoSpex, and GAssert all target postconditions. Precis [27] learns method contracts using a basic assertion language, but cannot express structural properties such as acyclicity, which Express captures. Another direction uses NLP to infer specifications from comments. Jdoctor [4] extracts method specs from Javadoc, and nlp2postcondition [10] applies LLMs. Unlike these, Express learns invariants from class behavior, requiring no comments, which may be absent.

5 Conclusion

We presented Express, a novel approach for automatically inferring class invariants in the form of `repOk` methods. Express takes as input a target class and a test suite, and leverages a four-stage simulated annealing search to iteratively refine and learn structural constraints that characterize valid object states.

Our evaluation demonstrates that Express infers very complete and correct class invariants on the case study we used as example.

Acknowledgments

This work is supported by the Ramón y Cajal fellowship RYC2020-030800-I and by the Spanish Government through grants TED2021-132464B-I00 (PRODIGY) and PID2022-142290OB-I00 (ESPADA).

References

[1] Pablo Abad, Nazareno Aguirre, Valeria S. Bengolea, Daniel Alfredo Ciolek, Marcelo F. Frias, Juan P. Galeotti, Tom Maibaum, Mariano M. Moscato, Nicolás Rosner, and Ignacio Vissani. 2013. Improving Test Generation under Rich Contracts by Tight Bounds and Incremental SAT Solving. In *ICST*. 21–30.
[2] Angello Astorga, Shambwaditya Saha, Ahmad Dinkins, Felicia Wang, P. Madhusudan, and Tao Xie. 2021. Synthesizing contracts correct modulo a test generator. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27.
[3] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE TSE* 41, 5 (2015), 507–525.
[4] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *ISSTA*, 242–253.
[5] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: automated testing based on Java predicates. In *ISSTA*, 123–133.
[6] Juan Manuel Copia, Facundo Molina, Nazareno Aguirre, Marcelo F. Frias, Alessandra Gorla, and Pablo Ponzio. 2023. Precise Lazy Initialization for Programs with Complex Heap Inputs. In *ISSRE*.752–762.
[7] Juan Manuel Copia, Pablo Ponzio, Nazareno Aguirre, Alessandra Gorla, and Marcelo F. Frias. 2022. LISSA: Lazy Initialization with Specialized Solver Aid. In *ASE*. 67:1–67:12.
[8] Marcelo d'Amorim, Carlos Pacheco, Tao Xie, Darko Marinov, and Michael D. Ernst. 2006. An Empirical Comparison of Automated Generation and Classification Techniques for Object-Oriented Unit Testing. In *ASE*. 59–68.
[9] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. 2006. Modular verification of code with SAT. In *ISSTA*,109–120.
[10] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions? *Proc. ACM Softw. Eng.* 1, FSE (2024), 1889–1912.
[11] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1-3 (2007).
[12] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT FSE*. ACM, 416–419.
[13] Carlo A. Furia, Martin Nordio, Nadia Polikarpova, and Julian Tschannen. 2017. AutoProof: auto-active functional verification of object-oriented programs. *Int. J. Softw. Tools Technol. Transf.* 19, 6 (2017), 697–716.
[14] Juan P. Galeotti, Nicolás Rosner, Carlos López Pombo, and Marcelo F. Frias. 2010. Analysis of invariants for efficient bounded verification. In *ISSTA*,25–36.
[15] Aayush Garg, Renzo Degiovanni, Facundo Molina, Maxime Cordy, Nazareno Aguirre, Mike Papadakis, and Yves Le Traon. 2023. Enabling Efficient Assertion Inference. In *ISSRE*. 623–634.
[16] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. 2002. *Fundamentals of Software Engineering* (2nd ed.).
[17] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. 2005. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.* 55, 1-3 (2005), 185–208.
[18] Lisa (Ling) Liu, Bertrand Meyer, and Bernd Schoeller. 2007. Using Contracts and Boolean Queries to Improve the Quality of Automatic Test Generation. In *TAP*.
[19] Muhammad Zubair Malik, Aman Pervaiz, Engin Uzuncaova, and Sarfraz Khurshid. 2008. Deryaft: a tool for generating representation invariants of structurally complex data. In *(ICSE)*, 859–862.
[20] Bertrand Meyer. 1997. *Object-Oriented Software Construction, 2nd Edition.*
[21] Facundo Molina, César Cornejo, Renzo Degiovanni, Germán Regis, Pablo F. Castro, Nazareno Aguirre, and Marcelo F. Frias. 2019. An evolutionary approach to translating operational specifications into declarative specifications. *Sci. Comput. Program.* 181 (2019), 47–63.
[22] Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. 2022. Fuzzing Class Specifications. In *ICSE*. 1008–1020.
[23] Facundo Molina, Renzo Degiovanni, Pablo Ponzio, Germán Regis, Nazareno Aguirre, and Marcelo F. Frias. 2019. Training binary classifiers as data structure invariants. In *ICSE*, 759–770.
[24] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo F. Frias. 2021. EvoSpex: An Evolutionary Algorithm for Learning Postconditions. In *ICSE*.
[25] Facundo Molina, Alessandra Gorla, and Marcelo d'Amorim. 2025. Test Oracle Automation in the Era of LLMs. In *ACM Trans. Softw. Eng. Methodol.*
[26] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *(ICSE)*. 75–84.
[27] Corina S. Pasareanu. 2020. *Symbolic Execution and Quantitative Reasoning: Applications to Software Safety and Security.* Morgan & Claypool Publishers.
[28] Corina S. Pasareanu, Willem Visser, David H. Bushnell, Jaco Geldenhuys, Peter C. Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Autom. Softw. Eng.* 20, 3 (2013), 391–425.
[29] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. 2020. Evolutionary Improvement of Assertion Oracles. In *ESEC/FSE*.1178–1189.