Search-based Inference of Class Invariants: How far can Simulated Annealing take us?

Juan Manuel Copia[1,2], Facundo Molina[1], Alessandra Gorla[1], Nazareno Aguirre[3,4,5], and Pablo Ponzio[3,4]

[1] IMDEA Software Institute, Madrid, Spain
[2] Universidad Politécnica de Madrid, Madrid, Spain
[3] Universidad Nacional de Río Cuarto, Río Cuarto, Argentina
[4] CONICET, Argentina
[5] Guangdong Technion-Israel Institute of Technology, Shantou, China

Abstract. Many formal verification and software testing techniques rely on repOK routines to verify the consistency and validity of software components with complex data representations. For a given class under analysis, a repOK encodes its so-called *class invariant*, i.e., the state properties required on an instance for it to be considered a valid object of the class. While repOK routines can enable early error detection and simplify debugging, writing correct and complete repOKs can be challenging, even for advanced Large Language Models (LLMs). This paper explores the effectiveness of search-based techniques to automatically generate class invariants in the form of imperative repOKs.
We perform an evaluation using Express, a recent simulated annealing-based framework that automatically generates repOKs, on a set of 7 Java classes from widely-used libraries, and show that the framework can generate *correct repOKs for all cases*, ensuring no valid instances are erroneously discarded. Furthermore, Express produces *complete* repOKs in 3 out of 7 cases. Combining Express with an LLM further enhances effectiveness, generating *correct* and *complete* repOK routines for 5 out of 7 subjects. In addition, Express yields more complete invariants (i.e., it generates stronger properties) than the best related approach in 5 out of 7 cases. These results highlight the potential of search-based techniques for automating consistency checks in software systems, bridging gaps in current manual and automated software analyses.

1 Introduction

Software reliability is a core aspect of software quality, and a primary concern in software engineering [19]. Advances in automated program analysis have enabled the efficient creation of extensive program input sets as well as the examination of large-scale program executions [7, 33, 15, 35], facilitating automated reliability analyses. However, determining whether automatically generated 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 [5], 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 [25, 22]. *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 does not contain duplicate elements, or that a queue's maximum capacity must not be exceeded. Class invariants can be captured by so-called *repOK routines*, methods that check whether receiver objects satisfy their corresponding class invariants [22]. These routines can be used to check the internal consistency of objects during testing, debugging and run-time monitoring, and have proved to be very valuable for test generation [11, 7, 3, 23], bug detection [33, 21], and verification [21, 16, 17, 12], among other tasks. Despite their usefulness and effectiveness for automated analysis, formal class invariant specification is complex, and thus class invariants are often only informally specified as ambiguous natural language comments, limiting practical utility for automated analysis.

To address this gap, various techniques have been proposed to automatically infer specifications, including class invariants [14, 6, 28, 38, 30, 40, 4, 27, 18, 13, 26, 24, 9]. Most of these are symbolic approaches, in the sense that they infer the invariants using a pre-defined set of rules and patterns. Daikon [14], the most prominent tool for dynamic invariant detection, leverages program executions to infer invariants from predefined templates. However, Daikon's template-based approach limits expressiveness, prompting the development of more sophisticated tools such as SpecFuzzer [27], Deryaft [24] and Express [9]. While these tools represent significant progress, the inferred invariants often fall short in practical applicability. For instance, Daikon [14] and SpecFuzzer [27] output invariants as assertion expressions, that are either limited in expressiveness or require some effort to make them executable and integrated into the codebase. Deryaft [24] generates executable repOK routines directly, but their invariants are often incomplete, lacking the expressiveness needed for real-world applications. These limitations motivated the development of Express [9], a novel framework that leverages simulated annealing to automatically generate class invariants in the form of imperative repOK routines. Given a target class and its accompanying test suite, the main goal of Express is to infer a repOK that is *correct* with respect to the test suite, meaning it does not discard any valid instance created during test execution, and that is *complete*, discarding as many invalid instances as possible. Despite its promising direction, Express has not been evaluated on real-world subjects, and its performance compared to existing techniques remains unknown.

At the same time, neural approaches, based on deep learning methods, have been less explored to infer class invariants, with only machine learning techniques being used to train binary classifiers to behave as class invariants [28, 39], but not to produce executable code. Recently, more advanced neural techniques, based on state-of-the-art Large Language Models (LLMs) have been successfully applied to derive other kind of specifications [31, 13, 29, 36]. For instance,

nlp2postcondition [13] generates postcondition assertions by using a zero-shot prompt template to instruct chat-based LLMs (such as GPT-4 [32]). Tratto [31] implements a neuro-symbolic approach to derive axiomatic test oracles, leveraging large code models (such as Code Llama [37]), to produce oracles in a token-by-token fashion. These advances suggest that LLM-based neural techniques, including neuro-symbolic approaches that combine LLMs with symbolic reasoning, can be effective in inferring class invariants as well. To the best of our knowledge, no such techniques have been proposed yet.

In this paper, we aim to study the effectiveness of search-based methods to automatically infer class invariants in the form of imperative repOK routines, and how it compares with other symbolic and neural approaches. Concretely, we evaluate Express, a simulated annealing-based framework that automatically generates repOK methods, on a set of 7 Java classes from widely-used libraries, to understand how correct and complete the generated repOK routines are. More precisely, we compare Express with Daikon, a well-known symbolic invariant inference tool, and with GPT-4, a state-of-the-art neural approach.

Our results show that Express can generate correct repOK routines for all the analyzed subjects, while producing complete repOK routines for 3 subjects, outperforming related techniques. Moreover, we show that invariants produced by LLM-based neural approaches can complement invariants generated by Express, producing correct and complete repOK routines for 5 subjects. This provides initial evidence that hybrid approaches combining search-based symbolic techniques and neural methods can improve class invariant inference.

2 Background

2.1 Class Invariants and repOK Routines

Class invariants are formal conditions that must hold true for an object for it to be considered valid, serving as a foundation for reasoning about the correctness of a class's implementation and playing a critical role in automated analyses such as testing, debugging, and verification.

As an example, consider the `LinkedList` class in Figure 1, a simplified version of the implementation in the `java.util` package. This class represents sequences of elements using a doubly-linked list, with two primary fields: `header`, pointing to the first element, and `size`, storing the number of elements. Each list element is represented by an `Entry` object, containing the fields `element`, `next`, and `previous`. From the fields alone, the developer's intent regarding the data structure's constraints may not be immediately clear. However, such intent can often be inferred from the behavior of the class's methods. If the programmer intended the list to be cyclic, the class invariant must ensure that the list forms a closed loop (the `next` and `previous` pointers are consistent). Figure 1 demonstrates how such a circularity check can be implemented as a `repOK`.

In this paper, we study the challenge of automatically generating class invariants for Java classes, expressed as imperative routines. We study the generation

```
public class Entry {
    private Entry next, previous;
    private Object element;
    ...
}

public class LinkedList {
    private Entry header;
    private int size;

    public boolean repOK() { return isCircular() && isSizeOk(); }

    public boolean isCircular() {
        Set<Entry> visited = new HashSet<Entry>();
        if (header == null) return false;
        visited.add(header);
        Entry current = header;
        while (true) {
            Entry next = current.next;
            if (next == null) return false;
            if (next.previous != current) return false;
            current = next;
            if (!visited.add(next)) break;
        }
        if (current != header) return false;
        return true;
    }
    ...
```

Fig. 1. LinkedList class declaration and its class invariant.

of *correct* and *complete*, executable invariants, which can enhance automated analyses, and enable robust and scalable reliability assessments.

2.2 Simulated Annealing

EXPRESS, the search-based tool we consider in this paper, is based on simulated annealing [20]. Simulated annealing is a probabilistic optimization algorithm that explores a search space by iteratively mutating candidate solutions and evaluating them based on an objective function. The algorithm begins with a random feasible solution and generates new candidates by applying small modifications. A candidate solution is accepted based on the following criteria:

- **Improvement in the objective function:** If the new solution improves the objective function value, it is always accepted.
- **Acceptance of worse solutions:** To escape local minima, worse solutions are also accepted, with the acceptance probability given by:

$$P(\text{accept}) = \exp\left(-\Delta f / T\right),$$

  where $\Delta f = f_{\text{candidate}} - f_{\text{current}}$ represents the change in the objective function value, and T is the *temperature*, a control parameter that determines the likelihood of accepting worse solutions.

The temperature follows a predefined cooling schedule (e.g., linear decay), gradually reducing T over time. As T decreases, the algorithm shifts from exploration to refinement, focusing on optimizing the best solutions found so far.

3 Evaluating Simulated Annealing for Class Invariant Inference

To assess search-based inference of class invariants, and compare it with other approaches, we perform an empirical evaluation using Express as a representative search-based technique for class invariant inference. We conduct a series of experiments focused on the following research questions:

RQ1 *How effective is search-based class invariant inference?*
RQ2 *How does search-based class invariant inference compare with related approaches?*

RQ1 analyzes the effectiveness of a search-based technique – Express – to generate class invariants for a set of Java classes, while RQ2 compares search-based class invariant inference with two related approaches: Daikon [14], a well-established and widely used dynamic invariant detection tool, and OpenAI's GPT-4 [32], a state-of-the-art LLM. A replication package for all the experiments in the paper and additional experimental data can be found in [2].

3.1 Experimental Setup

Subjects. We analyze seven Java classes with diverse class invariants for which ground truth invariants are already known [8]. These classes are: `LinkedList`, `HashMap`, and `TreeMap` from the `java.util` package of the Java Standard Library; `NodeCachingLinkedList` (NCLL), a specialized linked list implementation from Apache Commons Collections; and classes `AvlTree`, `BinomialHeap`, and `Schedule`, used in prior research on automated verification [10, 8].

Approaches. We consider three approaches for class invariant inference:
- Express: The simulated annealing based approach, representative of search-based inference. To run Express, we first generate a diverse set of valid instances, using EvoSuite [15], a state-of-the-art test generation tool that produces test suites optimized for code coverage. Since EvoSuite generates minimal test suites, we manually added an average of 12 additional test cases per class to increase instance diversity. Express executes these tests and collects the generated instances as the set of valid states, which are then mutated to produce invalid instances. Express then runs its search process to generate a `repOK` that classifies the instances. The parameter values for Express, including the initial temperature, cooling schedule, and number of mutations for generating invalid instances, were selected by experimenting with different configurations, and selecting the best-performing one: 15.0 as initial temperature, an exponential cooling schedule with a cooling rate of 0.005, and 9 mutations per valid instance to generate invalid ones.
- Daikon: A well-known dynamic invariant detection tool that infers invariants from a set of test cases. To ensure a fair comparison, we execute Daikon from exactly the same test suite used for Express. Since Daikon expresses invariants in a textual format, we manually converted them into imperative repOKs for direct evaluation.

Table 1. Effectiveness of EXPRESS to generate class invariants.

Subject	Input Instances		Effectiveness						
	#Valid	#Invalid	TP	TN	FP	FN	Prec.(%)	Rec.(%)	Acc.(%)
TreeMap	4880	1655572	4880	1651219	4353	0	52.9	100	99.7
Schedule	89958	333585	89958	333249	336	0	99.6	100	99.9
HashMap	22136	177371936	22136	174083876	3288060	0	0.7	100	98.1
LinkedList	9	231	9	231	0	0	100	100	100
AvlTree	2698	616347	2698	613236	3111	0	46.4	100	99.5
BinomialHeap	1749224	920023	1749224	717737	202286	0	89.6	100	92.4
NCLL	660	1309	660	1309	0	0	100	100	100

– GPT-4: A state-of-the-art LLM, used as a representative of neural approaches for class invariant inference. We use GPT-4 through ChatGPT [1], OpenAI's web interface, using a zero-shot prompting approach, and providing the model with the complete implementation of the target class, including comments. We query the model once per subject and use its output without further modifications, as it consistently produces repOK routines. While more sophisticated prompting techniques could be explored, this is beyond the scope of this work. The prompt we use is available on the paper's website [2].

Metrics. To evaluate the approaches, we compare the inferred invariants against ground truth invariants using Korat [7], 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, except for HashMap, where we use a scope of 3 due to the large number of generated instances.

We assess each inferred `repOK` using standard classification metrics, based on the following cases. *True Positive (TP)*: A valid instance correctly classified as valid (`repOK` returns `true`). *False Positive (FP)*: An invalid instance incorrectly classified as valid. *True Negative (TN)*: An invalid instance correctly classified as invalid (`repOK` returns `false`). *False Negative (FN)*: A valid instance incorrectly classified as invalid. From these values, we compute Precision, Recall and Accuracy as follows: Precision, computed as *Precision=TP/(TP+FP)*, measures the proportion of all instances predicted as valid that are actually valid. That is, the higher the precision of the inferred invariants, the more *complete* they are (fewer FPs). Recall, computed as *Recall=TP/(TP+FN)*, measures the proportion of all valid instances that were correctly predicted as valid. That is, the higher the recall of the inferred invariants, the more *correct* they are (less FNs). Finally, accuracy, computed as *Accuracy=(TP+TN)/(TP+TN+FP+FN)*, measures the proportion of all instances that are correctly classified.

3.2 Effectiveness of Search-based Class Invariant Inference (RQ1)

Table 1 summarizes the results of the effectiveness of the search-based class invariant inference performed by EXPRESS. For each subject, we report the number of valid and invalid instances generated from the ground truth (using Korat). We also report the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN), and the precision, recall, and accuracy

of the inferred invariants. Remarkably, Express produces no false negatives for all subjects, indicating that the inferred class invariants never reject valid instances. Hence, Express achieves perfect recall in all subjects. In other words, all properties included in the repOKs generated by Express are correct. This is an important feature of the approach, since it means that the user does not have to debug and manually fix incorrect properties in the inferred repOKs. Notice that, this can be attributed to the design of Express's simulated annealing algorithm, which aims to prioritize the avoidance of false negatives.

Precision, however, is affected by false positives: invalid instances that are incorrectly accepted by the inferred invariants. Less than 100% precision indicates that some properties of the class's representation invariant remain unlearned. This is expected, as the search space for invariants is vast, and certain properties are difficult to discover without improving Express's search process.

Express achieves over 89% precision in 4 out of 7 classes, with perfect precision in the `LinkedList` and `NCLL` subjects. The low precision observed for `HashMap` is not a reflection of poor performance by Express; instead, it is due to the significant imbalance between valid instances and invalid instances, in the space of instances generated by Korat for `HashMap`. However, as we show in the next section, where we analyze the properties of the ground truth invariants that each tool learns in our case studies, Express learns most of the expected properties, even for the `HashMap` subject.

To conclude, it is important to mention that the average runtime for generating invariants using Express was 13 minutes. In the worst case, Express took 25 minutes. We believe the running times are reasonable for the task of learning high quality (i.e., sound and reasonably complete) representation invariants. The running times for all case studies are reported in the paper's website [2].

3.3 Search-based Class Invariant Inference Compared with Related Techniques (RQ2)

The results of the comparison between Express, Daikon and ChatGPT are summarized in Table 2. Among the three approaches, Daikon produces the most imprecise class invariants. While it achieves perfect recall for four subjects, it has a very low precision, ranging from 0.5% to 39.4% in 6 out of 7 cases. In other words, Daikon is not capable of learning most of the properties of the repOKs in our ground truth, making it the least effective approach. Moreover, there was no class for which Daikon inferred a correct and complete class invariant.

GPT-4 achieves good results, generating sound and complete invariants in 2 subjects, `AvlTree` and `TreeMap`. It produces class invariants with a perfect recall in 5 out of 7 subjects. However, as expected, GPT-4 generates repOK methods with errors, in particular issues that cause the execution of repOK to get stuck in infinite loops while traversing the object state. This happened for two classes, `BinomialHeap` and `NCLL`. In these cases, the invariants are not useful without manual fixes from the developer, or further processing to remove the errors. In summary, Express outperforms GPT-4, as it generates better invariants in 5

Table 2. Comparison of EXPRESS with Daikon and ChatGPT.

Subject	Approach	Performance		
		Precision (%)	Recall (%)	Accuracy (%)
	LinkedList			
LinkedList	Daikon	10.0	100	66.2
	ChatGPT	90.0	100	99.6
	EXPRESS	100	100	100
	EXPRESS+ChatGPT	100	100	100
	NodeCachingLinkedList			
NCLL	Daikon	39.4	100	48.5
	ChatGPT	-	-	-
	EXPRESS	100	100	100
	EXPRESS+ChatGPT	100	100	100
	AvlTree			
AvlTree	Daikon	1.2	100	63.3
	ChatGPT	100	100	100
	EXPRESS	46.4	100	99.5
	EXPRESS+ChatGPT	100	100	100
	Schedule			
Schedule	Daikon	23.4	73.0	43.6
	ChatGPT	23.7	100	31.6
	EXPRESS	99.6	100	99.9
	EXPRESS+ChatGPT	100	100	100
	TreeMap			
TreeMap	Daikon	0.9	100	65.8
	ChatGPT	100	100	100
	EXPRESS	52.9	100	99.7
	EXPRESS+ChatGPT	100	100	100
	HashMap			
HashMap	Daikon	0.5	80.9	98.0
	ChatGPT	0.4	100	96.5
	EXPRESS	0.7	100	98.1
	EXPRESS+ChatGPT	0.7	100	98.1
	BinomialHeap			
BinomialHeap	Daikon	74.9	99.8	77.9
	ChatGPT	-	-	-
	EXPRESS	89.6	100	92.4
	EXPRESS+ChatGPT	89.6	100	92.4

out of 7 cases (including two cases where GPT-4 generates completely unsound invariants), whereas GPT-4 generates better invariants in 2 cases.

Express+GPT-4. This technique combines the invariants generated by GPT-4 and EXPRESS. It operates by first invoking the `repOK` of EXPRESS to check if the structure is valid. If EXPRESS identifies a violation of the invariant, `false` is returned. Otherwise, the result of the `repOK` generated by GPT-4 is returned. This approach leads to improvements in three subjects, `AvlTree`, `Schedule`, and `TreeMap`, where the precision of the combined invariants reaches 100%. This suggests that the invariants generated by GPT-4 can in fact complement the ones inferred by EXPRESS, and represents a promising direction for future research, where hybrid approaches can be developed to combine the strengths of search-based techniques and LLMs.

Analysis of the Inferred Invariants. To better assess invariant quality, we manually analyzed the inferred invariants in terms of the properties they express

Table 3. Properties inferred by each approach.

Subject	Expected Property	Daikon	ChatGPT	Express	Express+GPT
Treemap	Is acyclic binary tree	✗	✓	✓	✓
	Parent of root is null	✓	✓	✓	✓
	Parent references match	✗	✓	✓	✓
	Size is consistent	✗	✓	✓	✓
	Balanced tree	✗	✓	✗	✓
	Sorted keys	✗	✓	∼	✓
AvlTree	Is acyclic binary tree	✗	✓	✓	✓
	Size is consistent	✗	✓	✓	✓
	Sorted keys	✗	✓	∼	✓
	Balanced tree	✗	✓	✗	✓
	Height fields are correct	✗	✓	∼	✓
LinkedList	Header $\neq$ null	✓	✓	✓	✓
	Previous and next match	✗	∼	✓	✓
	List is circular DLL.	✗	✓	✓	✓
	Size is consistent	✗	✓	✓	✓
NCLL	Header $\neq$ null	✓	✓	✓	✓
	Header is circular DLL.	✗	✗	✓	✓
	Size is consistent	✗	✓	✓	✓
	Cache is acyclic	✗	✗	✓	✓
	Size of cache is consistent	✗	✓	✓	✓
	No node sharing	✗	✗	✓	✓
HashMap	Buckets are acyclic	✗	✓	✓	✓
	Buckets don't share nodes	✗	∼	✓	✓
	Size is consistent	✗	✓	✓	✓
	Size $\leq$ threshold	✗	✓	✓	✓
	Keys and hashes match	∼	✓	✓	✓
	Keys are in correct bucket	✗	✗	✗	✗
	No repeated keys	✗	✓	✓	✓
Schedule	P0 = null	✓	✗	✓	✓
	P1, p2, p3, blockqueue $\neq$ null	✗	✗	✓	✓
	Queues do not share jobs	✗	✓	✓	✓
	Queues have no aliasing	✗	✗	✓	✓
	Queues are acyclic DLL	✗	✓	✓	✓
	Job.priority matches queue	✗	✓	✗	✓
	Job.val are unique	✗	✗	✓	✓
	Job.val $\geq$ 0	✗	✗	✓	✓
	CurProc is in queues	✗	✓	✓	✓
	Job.priority $\geq$ 0	✗	∼	✓	✓
	Job.priority $\leq$ MAXPRIO	✗	∼	✓	✓
	NumProc is consistent	✗	✗	✓	✓
	List.mem_count is consistent	✗	✓	✓	✓
	NumProc $\leq$ allocProcNum	✗	✓	✓	✓
BinomialHeap	Is acyclic binary tree	✗	∼	✓	✓
	Parent of root points to null	✓	✗	✓	✓
	Parent references match	✗	✓	✓	✓
	Size is consistent	✗	✓	✓	✓
	Sorted keys	✗	✗	✓	✓
	Is Binomial Heap	✗	∼	∼	∼

with respect to the properties in the ground truth (i.e., the *expected* properties in the real representation invariant). Table 3 summarizes the results. For each class, we manually analyzed the ground truth to extract the expected properties (Expected Property column), and analyzed whether the inferred invariants expressed such properties. We identified a total of 48 expected properties.

Daikon infers about one expected property for most of the classes, and these are properties typically related to checking for null values. This explains the very low precision exhibited by Daikon, and indicates that the predefined set of templates used by Daikon is not rich enough for it to find many important properties of complex representation invariants.

GPT-4 is able to infer many of the expected properties, inferring all of them for `AvlTree` and `TreeMap`. In the remaining subjects, GPT-4 misses some of the properties, which is also consistent with the precision results in Table 3. In total, GPT-4 infers 30 out of the 48 expected properties. It is important to recall that GPT-4 yields invariants with errors for `BinomialHeap` and `NCLL`, hence we cannot consider that it learns any useful property in those cases.

Surprisingly, Express is able to infer a larger number of expected properties compared to Daikon and GPT-4. It generates *all* the properties in two cases, and most of the properties in the remaining ones. In total, Express generates 40 out of the 48 expected properties, which is a significant improvement over the other approaches. Furthermore, the combined technique Express+GPT-4 is able to capture 46 out of the 48 expected properties. Some examples of the properties that Express cannot infer are balance properties of tree structures (`TreeMap`, `AvlTree` and `BinomialHeap`), and that the keys of `TreeMap`, `AvlTree` must be stored in sorted order. These are complex properties that would require the addition of more sophisticated mutators (and traversals) to Express's algorithm, and better mutation approaches for primitive-typed fields. These are all interesting lines of research for future work.

Overall, these results show that Express outperforms the related approaches Daikon and GPT-4 in terms of the number of (expected) properties inferred. Taken together with the precision numbers from Table 3 we can conclude that Express is the approach that learns the more complete invariants in this evaluation. In addition, combining the invariants inferred by Express with GPT-4's invariants can lead to more complete sets of properties.

4 Threats to Validity

Threats to external validity may arise from our experimental evaluation, which is focused on a limited number of classes, selected from previous work. Nevertheless, we believe that these classes include invariants that are representative of various types of properties, of varying complexity. As future work, we plan to extend the evaluation to classes from open-source projects, to assess the effectiveness of search-based techniques in more realistic settings. This will imply abandoning the effectiveness assessments with respect to ground truths, due to the effort

and expertise that would involve expressing target invariants for arbitrary open-source projects.

5 Related Work

Several approaches have been proposed for inferring class invariants. Deryaft infers representation invariants as imperative Java methods [24], similar to Express. Like Daikon, Deryaft relies on a fixed set of predefined templates to infer properties. However, this template-based approach limits its expressiveness, making it less practical for real-world applications. Additionally, since no replication package is publicly available, we could not include it in our evaluation.

Molina et al. [28] propose a machine learning approach based on training neural networks to classify valid and invalid instances based on test executions. While effective, neural networks introduce interpretability challenges, making debugging difficult. In our experiments, we use a state-of-the-art LLM as a representative of neural approaches.

Rather than focusing on class invariants, other approaches focus on inferring other properties, such as method postconditions. SpecFuzzer [27] uses grammar-based fuzzing to generate postconditions. Similarly, EvoSpex [30] defines a classic genetic algorithm for the same task. These approaches employ a declarative language for expressing the assertions, namely a first order logic based language extended with reachability operators. Though these search-based techniques could be adapted to learn representation invariants, they are not designed for this purpose, and thus we did not include them in our evaluation. GAssert [38] is another evolutionary approach for learning assertions, which focuses on relatively simple logical and arithmetic constraints. It does not allow for quantified expressions and reachability operators, which are required to express many of the properties of representation invariants. Precis [34] infers contracts (pre and postconditions) for methods. Its target language consists of basic logical and arithmetic operators (without quantification/reachability). Finally, Natural language processing (NLP) has also been used to infer specifications from comments. Jdoctor [6] extracts executable specifications for methods from Javadoc comments. nlp2postcondition [13] use LLMs for this task. To the best of our knowledge, LLMs have not been used to generate representation invariants.

6 Conclusion

We performed a comprehensive evaluation of Express, a recent search-based approach for automatically inferring class invariants in the form of `repOK` methods. We compare Express with Daikon, a well-known approach for invariant inference, and with GPT-4, a state-of-the-art large language model (LLM).

Our evaluation, based on a set of seven classes with known class invariants, shows that Express is able to infer very precise invariants, achieving perfect recall across all studied subjects, meaning that all valid instances are correctly accepted. Express outperforms related techniques in most of our subjects in

terms of precision and recall, being able to infer a larger number of properties that are expected for the target classes, and that related techniques are not able to infer. Finally, we show that combining EXPRESS's invariants with LLM-generated ones can lead to more robust and complete invariants, highlighting the potential of hybrid approaches that integrate search-based techniques with LLM-generated code. Overall, our results provide initial but strong evidence that search-based techniques can produce highly reliable class invariants.

Acknowledgments

This work is supported by the Ramón y Cajal fellowship RYC2020-030800-I, by the Spanish Government through grants TED2021-132464B-I00 (PRODIGY), PID2022-142290OB-I00 (ESPADA), and CEX2024-001471-M/funded by MICIU/AEI/10.13039/501100011033 and by the Comunidad de Madrid as part of the ASCEND project co-funded by FEDER Funds of the European Union.

References

1. Openai's chatgpt (2025), https://openai.com/index/chatgpt
2. Replication package of this paper (2025), https://zenodo.org/records/17300555
3. Abad, P., Aguirre, N., Bengolea, V.S., Ciolek, D.A., Frias, M.F., Galeotti, J.P., Maibaum, T., Moscato, M.M., Rosner, N., Vissani, I.: Improving test generation under rich contracts by tight bounds and incremental SAT solving. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013. pp. 21–30. IEEE Computer Society (2013). https://doi.org/10.1109/ICST.2013.46, https://doi.org/10.1109/ICST.2013.46
4. Astorga, A., Saha, S., Dinkins, A., Wang, F., Madhusudan, P., Xie, T.: Synthesizing contracts correct modulo a test generator. Proc. ACM Program. Lang. **5**(OOPSLA), 1–27 (2021). https://doi.org/10.1145/3485481, https://doi.org/10.1145/3485481
5. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. IEEE Trans. Software Eng. **41**(5), 507–525 (2015). https://doi.org/10.1109/TSE.2014.2372785, https://doi.org/10.1109/TSE.2014.2372785
6. Blasi, A., Goffi, A., Kuznetsov, K., Gorla, A., Ernst, M.D., Pezzè, M., Castellanos, S.D.: Translating code comments to procedure specifications. In: Tip, F., Bodden, E. (eds.) Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018. pp. 242–253. ACM (2018). https://doi.org/10.1145/3213846.3213872, https://doi.org/10.1145/3213846.3213872
7. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on java predicates. In: Frankl, P.G. (ed.) Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002. pp. 123–133. ACM (2002). https://doi.org/10.1145/566172.566191, https://doi.org/10.1145/566172.566191

8. Copia, J.M., Molina, F., Aguirre, N., Frias, M.F., Gorla, A., Ponzio, P.: Precise lazy initialization for programs with complex heap inputs. In: 34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023, Florence, Italy, October 9-12, 2023. pp. 752–762. IEEE (2023). https://doi.org/10.1109/ISSRE59848.2023.00080, https://doi.org/10.1109/ISSRE59848.2023.00080
9. Copia, J.M., Molina, F., Gorla, A., Ponzio, P., Aguirre, N.: Search-based inference of class invariants. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO, Málaga, Spain, July 14-18, 2025. ACM (2025)
10. Copia, J.M., Ponzio, P., Aguirre, N., Gorla, A., Frias, M.F.: LISSA: lazy initialization with specialized solver aid. In: 37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022. pp. 67:1–67:12. ACM (2022). https://doi.org/10.1145/3551349.3556965, https://doi.org/10.1145/3551349.3556965
11. d'Amorim, M., Pacheco, C., Xie, T., Marinov, D., Ernst, M.D.: An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan. pp. 59–68. IEEE Computer Society (2006). https://doi.org/10.1109/ASE.2006.13, https://doi.org/10.1109/ASE.2006.13
12. Dennis, G., Chang, F.S., Jackson, D.: Modular verification of code with SAT. In: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006. pp. 109–120. ACM (2006). https://doi.org/10.1145/1146238.1146251, https://doi.org/10.1145/1146238.1146251
13. Endres, M., Fakhoury, S., Chakraborty, S., Lahiri, S.K.: Can large language models transform natural language intent into formal method postconditions? Proc. ACM Softw. Eng. **1**(FSE), 1889–1912 (2024). https://doi.org/10.1145/3660791, https://doi.org/10.1145/3660791
14. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1-3), 35–45 (2007). https://doi.org/10.1016/j.scico.2007.01.015, https://doi.org/10.1016/j.scico.2007.01.015
15. Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: SIGSOFT FSE. pp. 416–419. ACM (2011)
16. Furia, C.A., Nordio, M., Polikarpova, N., Tschannen, J.: Autoproof: auto-active functional verification of object-oriented programs. Int. J. Softw. Tools Technol. Transf. **19**(6), 697–716 (2017). https://doi.org/10.1007/s10009-016-0419-0, https://doi.org/10.1007/s10009-016-0419-0
17. Galeotti, J.P., Rosner, N., Pombo, C.L., Frias, M.F.: Analysis of invariants for efficient bounded verification. In: Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010. pp. 25–36. ACM (2010). https://doi.org/10.1145/1831708.1831712, https://doi.org/10.1145/1831708.1831712
18. Garg, A., Degiovanni, R., Molina, F., Cordy, M., Aguirre, N., Papadakis, M., Traon, Y.L.: Enabling efficient assertion inference. In: 34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023, Florence, Italy, October 9-12, 2023. pp. 623–634. IEEE (2023). https://doi.org/10.1109/ISSRE59848.2023.00039, https://doi.org/10.1109/ISSRE59848.2023.00039

19. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edn. (2002)
20. Kirkpatrick, S., Jr., D.G., Vecchi, M.P.: Optimization by simmulated annealing. Sci. **220**(4598), 671–680 (1983). https://doi.org/10.1126/SCIENCE.220.4598.671, https://doi.org/10.1126/science.220.4598.671
21. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. Sci. Comput. Program. **55**(1-3), 185–208 (2005). https://doi.org/10.1016/j.scico.2004.05.015, https://doi.org/10.1016/j.scico.2004.05.015
22. Liskov, B., Guttag, J.V.: Program Development in Java - Abstraction, Specification, and Object-Oriented Design. Addison-Wesley (2001)
23. Liu, L.L., Meyer, B., Schoeller, B.: Using contracts and boolean queries to improve the quality of automatic test generation. In: Gurevich, Y., Meyer, B. (eds.) Tests and Proofs - 1st International Conference, TAP 2007, Zurich, Switzerland, February 12-13, 2007. Revised Papers. Lecture Notes in Computer Science, vol. 4454, pp. 114–130. Springer (2007). https://doi.org/10.1007/978-3-540-73770-4"˙7, https://doi.org/10.1007/978-3-540-73770-4_7
24. Malik, M.Z., Pervaiz, A., Uzuncaova, E., Khurshid, S.: Deryaft: a tool for generating representation invariants of structurally complex data. In: Schäfer, W., Dwyer, M.B., Gruhn, V. (eds.) 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008. pp. 859–862. ACM (2008). https://doi.org/10.1145/1368088.1368223, https://doi.org/10.1145/1368088.1368223
25. Meyer, B.: Object-Oriented Software Construction, 2nd Edition. Prentice-Hall (1997)
26. Molina, F., Cornejo, C., Degiovanni, R., Regis, G., Castro, P.F., Aguirre, N., Frias, M.F.: An evolutionary approach to translating operational specifications into declarative specifications. Sci. Comput. Program. **181**, 47–63 (2019). https://doi.org/10.1016/J.SCICO.2019.05.006, https://doi.org/10.1016/j.scico.2019.05.006
27. Molina, F., d'Amorim, M., Aguirre, N.: Fuzzing class specifications. In: 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022. pp. 1008–1020. ACM (2022). https://doi.org/10.1145/3510003.3510120, https://doi.org/10.1145/3510003.3510120
28. Molina, F., Degiovanni, R., Ponzio, P., Regis, G., Aguirre, N., Frias, M.F.: Training binary classifiers as data structure invariants. In: Atlee, J.M., Bultan, T., Whittle, J. (eds.) Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019. pp. 759–770. IEEE / ACM (2019). https://doi.org/10.1109/ICSE.2019.00084, https://doi.org/10.1109/ICSE.2019.00084
29. Molina, F., Gorla, A., d'Amorim, M.: Test oracle automation in the era of llms. ACM Trans. Softw. Eng. Methodol. (Jan 2025). https://doi.org/10.1145/3715107, https://doi.org/10.1145/3715107, just Accepted
30. Molina, F., Ponzio, P., Aguirre, N., Frias, M.F.: Evospex: An evolutionary algorithm for learning postconditions. In: 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021. pp. 1223–1235. IEEE (2021). https://doi.org/10.1109/ICSE43902.2021.00112, https://doi.org/10.1109/ICSE43902.2021.00112

31. Molinelli, D., Martin-Lopez, A., Zackrone, E., Eken, B., Ernst, M.D., Pezzè, M.: Tratto: A neuro-symbolic approach to deriving axiomatic test oracles. In: Proceedings of the 2025 International Symposium on Software Testing and Analysis, ISSTA 2025, Trondheim, Norway, June 25-28, 2025. ACM (2025)
32. OpenAI: Gpt-4 technical report (2024), https://arxiv.org/abs/2303.08774
33. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007. pp. 75–84. IEEE Computer Society (2007). https://doi.org/10.1109/ICSE.2007.37, https://doi.org/10.1109/ICSE.2007.37
34. Pasareanu, C.S.: Symbolic Execution and Quantitative Reasoning: Applications to Software Safety and Security. Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers (2020). https://doi.org/10.2200/S01010ED2V01Y202005SWE006, https://doi.org/10.2200/S01010ED2V01Y202005SWE006
35. Pasareanu, C.S., Visser, W., Bushnell, D.H., Geldenhuys, J., Mehlitz, P.C., Rungta, N.: Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. Autom. Softw. Eng. **20**(3), 391–425 (2013). https://doi.org/10.1007/S10515-013-0122-2, https://doi.org/10.1007/s10515-013-0122-2
36. Pei, K., Bieber, D., Shi, K., Sutton, C., Yin, P.: Can large language models reason about program invariants? In: Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., Scarlett, J. (eds.) International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA. Proceedings of Machine Learning Research, vol. 202, pp. 27496–27520. PMLR (2023), https://proceedings.mlr.press/v202/pei23a.html
37. Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C.C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., Synnaeve, G.: Code llama: Open foundation models for code (2024), https://arxiv.org/abs/2308.12950
38. Terragni, V., Jahangirova, G., Tonella, P., Pezzè, M.: Evolutionary improvement of assertion oracles. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 1178–1189. ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3368089.3409758, https://doi.org/10.1145/3368089.3409758
39. Usman, M., Wang, W., Vasic, M., Wang, K., Vikalo, H., Khurshid, S.: A study of the learnability of relational properties: model counting meets machine learning (MCML). In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 1098–1111. ACM (2020). https://doi.org/10.1145/3385412.3386015, https://doi.org/10.1145/3385412.3386015
40. Wei, Y., Furia, C.A., Kazmin, N., Meyer, B.: Inferring better contracts. In: Taylor, R.N., Gall, H.C., Medvidovic, N. (eds.) Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011. pp. 191–200. ACM (2011). https://doi.org/10.1145/1985793.1985820, https://doi.org/10.1145/1985793.1985820