Applying Learning Techniques to Oracle Synthesis

Facundo Molina
University of Río Cuarto
and CONICET, Argentina
fmolina@dc.exa.unrc.edu.ar

ABSTRACT

Software reliability is a primary concern in the construction of software, and thus a fundamental component in the definition of software quality. Analyzing software reliability requires a *specification* of the intended behavior of the software under analysis. Unfortunately, software many times lacks such specifications. This issue seriously diminishes the analyzability of software with respect to its reliability. Thus, finding novel techniques to capture the intended software behavior in the form of specifications would allow us to exploit them for automated reliability analysis.

Our research focuses on the application of learning techniques to automatically distinguish correct from incorrect software behavior. The aim here is to decrease the developer's effort in specifying oracles, and instead *generating* them from actual software behaviors.

ACM Reference Format:
Facundo Molina. 2020. Applying Learning Techniques to Oracle Synthesis. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3324884.3415287

1 PROBLEM STATEMENT

The remarkable advances in automated program analysis allow us to efficiently produce large sets of program inputs as well as examining very large sets of program executions, in a fully automated way. This poses new challenges on how to practically decide whether these program executions show correct (expected) software behavior, or they are the manifestation of a software defect, especially because executable software specifications, which can precisely distinguish between these two cases, are rarely found in practice, accompanying software. Thus, devising novel and effective mechanisms to distinguish valid from invalid software executions, i.e., to tackle the so-called oracle problem [4], would have an enormous impact on the ability of automated analyses to detect program defects.

The oracle problem [4] has received great attention, and numerous techniques have been proposed to tackle this problem, in particular for specification synthesis via either static analysis approaches like constraint solving, interpolation and abduction (e.g., [8]), or dynamic approaches that are able to approximate specifications, e.g., in the form of "likely" invariants, from program executions (e.g., [9]). The former, while powerful, suffer from scalability issues that prevent them from scaling to mid-size or large software projects; the latter attempts to identify specifications solely from valid software behaviors, thus still requiring a considerable human effort to distinguish general software properties from those that hold on valid observed executions. Among the various applications that these executable specifications have, we are particularly interested in their use in combination with program analysis techniques, for bug-finding, as well as in reducing developer's efforts when specifying the desired software behavior.

Our approach to the oracle problem focuses on the application of learning based techniques. Within this context, we mainly work in the problem of automatically generating software specifications that can be used for program analysis. We concentrate on two different approaches to the problem:

- the use of Neural Networks (NNs) [29] to capture class invariants as binary classifiers,
- the use of Genetic Algorithms (GAs) [16] for learning method postconditions in the form of assertions.

The first focuses on a general component specification, namely representation invariants [18] of object oriented classes. A class C both defines an abstract data type and provides an implementation for it. A *representation invariant* of class C is a property that is satisfied by all legitimate objects of the class, i.e., objects that represent valid abstract objects. More precisely, it is a predicate $inv : C \rightarrow Bool$ that is true of legitimate objects. Thus, our idea is to capture such representation invariants by learning a function that maps each valid object state to true, and each invalid to false.

The second technique, instead focuses on assertions capturing method's postconditions. The use of assertions as program specifications dates back to the works of Hoare [14] and Floyd [10]. Technically, an assertion is a statement predicating on program states, that can be used to capture assumed properties, as in the case of preconditions, or intended properties, as in the case of postconditions (and invariants). In particular, a *postcondition assertion* captures a property that is satisfied for every terminating state reached by executing a given program P under the assumed conditions. In this case, our approach aims to produce these postcondition assertions through an evolutionary algorithm.

2 MOTIVATION

As a motivating example, consider the Java class SortedList depicted in Figure 1. This class implements sorted (ascending) sequences of integers with an insertion operation. Let us assume that this implementation's intention is to represent such sequences by storing them over acyclic singly linked lists. We would like to check that the implementation behaves as expected. To do so, we can define the representation invariant [18] for class SortedList, and then check

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3415287

that it is preserved by every public method manipulating objects of the class. Representation invariants are properties of the whole class, or more precisely, of *all* public methods of the class. More specific properties involving single methods can be captured via the corresponding pre and postconditions. For instance, we may define a postcondition for method *insert*, to assert that its actual behavior corresponds to list insertion.

```
public class SortedList {

  private int x;
  private SortedList next;
  private static final int SENTINEL = Integer.MAX_VALUE;

  private SortedList(int x, SortedList next) {
    this.x = x;
    this.next = next;
  }

  public SortedList() {
    this(SENTINEL, null);
  }

  // Insert method maintining ascending order
  public void insert(int data) {
    if (data > this.x) {
      next.insert(data);
    } else {
      next = new SortedList(x, next);
      x = data;
    }
  }

}
```

Figure 1: Class SortedList implementing ordered sequences of integers

Figures 2 and 3 show an implementation of the representation invariant (repOk) of class SortedList and the postcondition of the insert method, respectively. The representation invariant checks that the list is acyclic and that the values appear arranged in an ascending order. The postcondition for the insertion operation, besides verifying that the representation invariant is preserved, also checks that the new element is added properly to the list. It is straightforward to notice that implementing the verification of such constraints is a non-trivial task, that will become increasingly more difficult as the software behavior increases in complexity. Moreover, notice that the postcondition specification (captured operationally) is actually incomplete, since it should also check that the elements that were previously in the list are still there. Having these oracles formally specified gives us the possibility of checking that they are indeed preserved, with the aid of some automated analysis tools. For example, some runtime assertion checkers, as that accompanying the JML toolset [7], or a test generation tool like Randoop [25], can profit from specifications for bug finding.

Specifying program behavior through these kinds of oracles, i.e., invariants and postconditions, can be, as we mentioned, very

```
public boolean repOk() {
  Set<SortedList> visited = new HashSet<SortedList>();
  SortedList curr = this;
  while (curr.x != SENTINEL) {
    // The list should be acyclic
    if (!visited.add(curr))
      return false;
    // The list should be sorted
    SortedList curr_next = curr.next;
    if (curr.x > curr_next.x)
      return false;

    curr = curr_next;
  }
  return true;
}
```

Figure 2: Java representation invariant for class SortedList

```
// The representation invariant should be preserved
assert(repOk());
// The item should have been inserted
assert(contains(data));
```

Figure 3: Java postcondition assertions for method SortedList.insert(int)

difficult and time consuming, even when the language used for specification is the same programming language used for implementation. Since having such oracles can have a huge impact in our ability of detecting program defects, our research objective is to assist engineers in producing oracles. The technique underlying our approach is machine learning, and the oracles that we aim at generating are oracles suitable for automated program analysis, such as automated bug finding tools.

3 ORACLE SYNTHESIS

As mentioned earlier in this paper, our approach to the oracle problem focuses on the application of learning based techniques. In this section, we describe two approaches to the problem, namely the use of Neural Networks (NNs) to capture class invariants as binary classifiers and the use of Genetic Algorithms (GAs) for learning method postconditions in the form of assertions.

3.1 Class Invariants as Binary Classifiers

Let us first briefly describe our proposed approach based on the use of machine learning models, more specifically neural networks, to capture class invariants as binary classifiers. The goal of this technique is to build and train neural networks to learn to classify class objects into valid and invalid, so that the obtained classifiers can later be used in place of the invariant when performing some program analysis task, such as bug finding.

The overview of the approach is shown in Figure 4. Three main steps can be identified:

Training set generation. Valid object states can be created from an assumed-correct set of object building routines. These methods allow us to build correct instances using any automated test generation tool such as Randoop [25]. Invalid instances can be produced by randomly "breaking" the collected valid objects, i.e., by mutating some part of the objects.

Neural network training. First, the objects of the training set are encoded as vectors by using the candidate vector format of the automated test generation tool Korat [6]. This format provides a canonical way to represent objects up to a given size. Then, a *feed-forward neural network* is built using a well-known mechanism for hyperparameter optimization called *Random search* [5]. Finally, the network is trained to learn to classify the objects as valid or invalid.

Program analysis task. Once the network has been trained, the obtained classifier is used as an invariant to perform a bug finding task over the whole set of routines of the class, i.e., including methods that are not part of the assumed-correct building routines. Moreover, this classifier could be used in any program analysis task that aims at ensuring that operations manipulating and modifying the current class under analysis, do so preserving the representation invariant.

Further details of this approach can be found in [24]. Our experimental results show that our technique produces object classifiers that closely approximate class invariants, for classes with invariants of varying complexities. Indeed, the technique learns classifiers that achieve a very high precision and recall for invalid objects, and significantly better precision/recall for valid ones, compared to related techniques [9]. Additionally, the learned object classifiers improve bug detection by allowing a test generation tool to catch bugs, if classifiers are used in place of invariants, that the same tool cannot detect if the invariants produced by related techniques are directly used instead. This is evidenced by a set of experiments that were conducted on a benchmark of data structure implementations, of varying complexities.

3.2 Learning Postconditions with a Genetic Algorithm

In the context of learning method postconditions from program behaviors, we are currently studying a mechanism based on the use of a genetic algorithm to generate postconditions in the form of assertions. The goal of the technique is to produce assertions, similar to those of the JML language [7], that capture a specification characterizing the actual behavior of the method being analyzed. To do so, we propose an approach composed of the following two phases.

State generation phase. The objective of this phase is to generate valid (reachable) and invalid (unreachable) postcondition states, that will serve as the ground truth in the next phase. For the generation of valid postcondition states, any automated test generation mechanism that produces traces exercising the method under analysis can be applied. Once these tests have been executed and the valid states collected, the invalid ones can be produced by a mutation-based process, that randomly mutates valid states to produce (potentially) invalid states.

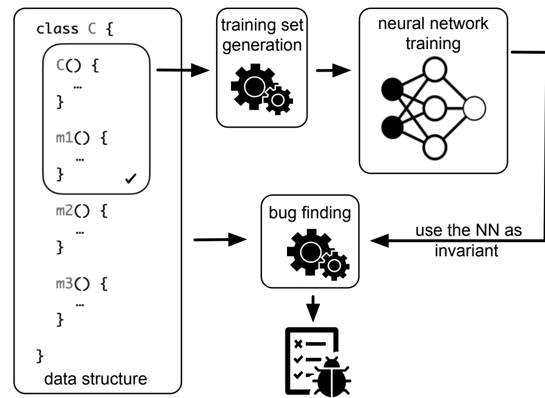

Figure 4: An overview of the NN based approach

Learning phase. The goal of this phase is to execute a genetic algorithm to evolve a set of executable state formulas, characterizing postcondition assertions that capture different method behaviors. These assertions attempt to capture relationships between pre and post states (the states before and after the execution of the method under analysis), return values of methods, and the relationship between method arguments and the state resulting from the method's execution. The valid and invalid post-states produced in the previous phase, are exploited in the fitness function to guide the algorithm towards formulas that hold for the valid post-states, and leaves out the invalid ones. More precisely, we aim at obtaining a postcondition assertion that:

- minimizes the number of valid (resp. invalid) states for which it is falsified (resp. satisfied),
- is as concise and simple as possible (in terms of its length and syntactic complexity), and
- maximizes the number of captured properties directly related to the method components (i.e., properties regarding the parameters, the result, or a relation between initial and final object states).

Figure 5 shows an overview of the described approach. Preliminary results of an implementation of this technique show that the approach allows us to produce post-conditions that involve more complex, stronger assertions, than those produced by tools like Daikon [9]. This is mainly due to the fact that our technique focuses on evolving assertions that target reference-based constraints, such as those of heap-allocated data representations. Furthermore, a significant number of the assertions inferred by our technique are *true positives*, as evaluated by an automated oracle assessment tool [15]. We have also compared our generated postcondition assertions to manually written postconditions in verified classes, confirming that our technique is able to generate an important part of the latter. Our evolutionary technique together with the experimental results just summarized are described in more detail in a recently submitted paper (under evaluation).

4 EXPECTED CONTRIBUTIONS

At this point, we have made the following contributions:

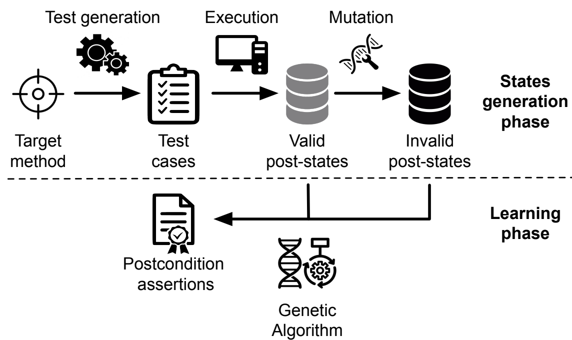

Figure 5: An overview of the GA based approach

- A mechanism for encoding objects as vectors, making the object classification problem suitable for other machine learning based approaches.
- The development of a NN based technique for inferring object classifiers with very high accuracy compared to related approaches. This contribution is also confirmed by [33].
- An analysis of the use of object classifiers in place of class invariants in a bug finding context, showing that bug detection is improved, as evidenced by experiments on a benchmark of data structure implementations of varying complexities.

These mentioned contributions are reported in [24], and all the related data is publicly available at [1]. Additionally, the following contributions are currently under evaluation:

- The development of an automated mechanism for producing reachable and unreachable states through methods executions.
- The definition of a GA that explores the state space of candidate postcondition assertions that includes rich constraints involving method parameters, return values, internal object states, and the relationship between pre and post method execution states, etc.

By continuing this line of work, we plan to extend our techniques in various directions. Regarding the NN based approach, we would like to scale the technique to coarser grained components, in particular modules with RESTFul APIs, which may require exploring more sophisticated methods for the construction and training of the NNs, such as *feature engineering* [13]. In the case of our second technique based on the use of a GA, the test generation mechanism used to produce the valid post-states plays a very important role. Given that there exists a variety of approaches to automated test generation, performing an analysis of the adequacy of different test suites from which assertions may be generated would have a considerable impact. Finally, the existence of false negatives for our produced postcondition assertions also opens lines of improvement for our inference mechanism.

5 RELATED WORK

Many tools for automated program analysis can profit from oracles describing intended software behavior. Some tools use them for run-time checking, notably the Eiffel programming language, that incorporates contracts as part of the programming language [21]; the runtime assertion checker and static verifiers that use JML [7]; Code Contracts for .NET [3]; among others. Some techniques for automated test case generation exploit these oracles for run-time checking too, converting the corresponding techniques into bug finding approaches, such as Randoop [25] and AutoTest [22].

The oracle problem has been extensively studied, and techniques to tackling it in different ways, have been proposed [4]. Our research is more closely related to techniques for *oracle derivation* [4], more precisely, for *specification inference*. Within this category, tools that perform specification inference from executions, like ours, include Daikon [9] and JWalk [32]. Both these tools attempt to infer oracles from *positive* executions, as opposed to our approaches that also integrate mechanisms to produce invalid situations. We have compared both of our approaches with Daikon, since JWalk tends to infer properties that are more scenario-specific. The use of neural networks for inferring specifications has been proposed before [30, 31]; these works, however, attempt to learn postcondition relations from "golden versions" of programs, i.e., assumed correct programs. While the approach is useful, e.g., in regression/differential testing scenarios, using it in our case would mean to learn the I/O relation for a "golden" representation invariant. This would require having the repOk in the first place, being a simpler problem compared to the ones we are tackling. Similarly, genetic algorithms has been applied to oracle related issues, e.g., in generating test-specific oracles for automatically generated tests [11], and in assessing oracles [15]. Genetic algorithms for evolving state formulas is specifically reported in [28], an in our own previous work [23], although targeting different problems.

6 CONCLUSIONS

Software specification plays a central role in various stages of software development, such as requirements engineering, software verification and maintenance. In the context of program analysis, there is an increasing availability of powerful techniques, including test generation [2, 22, 25], bug finding [12, 19], fault localization [35, 36] and program fixing [17, 26, 34], for which the need for program specifications becomes crucial. While many of these tools resort to tests as specifications, they would in general benefit from the availability of stronger, more general specifications, such as those that class invariants and postconditions provide. These types of oracles are becoming more common in program development, with methodologies that incorporate these [18, 20], and tools that can significantly exploit them when available for useful analyses.

The oracle problem has become a very important problem in software engineering, and within this context, oracle derivation or inference is particularly challenging [4]. In this paper, we described our learning based techniques for oracle inference, in particular for capturing class invariants as binary classifiers and for generating method assertions in the form of *postconditions*. Our experimental evaluations show that our approaches are able to produce more accurate oracles (stronger contracts in the sense of [27], with the associated benefits described therein), compared to the ones produced by related techniques, especially for reference-based implementations of components with implicit strong invariants.

REFERENCES

[1] 2019. Object classification approach site. https://sites.google.com/site/learninginvariants.

[2] Pablo Abad, Nazareno Aguirre, Valeria S. Bengolea, Daniel 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 *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. IEEE Computer Society, 21–30. https://doi.org/10.1109/ICST.2013.46

[3] Mike Barnett. 2010. Code Contracts for .NET: Runtime Verification and So Much More. In *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings (Lecture Notes in Computer Science)*, Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann (Eds.), Vol. 6418. Springer, 16–17. https://doi.org/10.1007/978-3-642-16612-9_2

[4] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[5] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-parameter Optimization. *J. Mach. Learn. Res.* 13 (Feb. 2012), 281–305. http://dl.acm.org/citation.cfm?id=2188385.2188395

[6] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, Phyllis G. Frankl (Ed.). ACM, 123–133. https://doi.org/10.1145/566172.566191

[7] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. 2005. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures (Lecture Notes in Computer Science)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.), Vol. 4111. Springer, 342–363. https://doi.org/10.1007/11804192_16

[8] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. 2013. Inductive invariant generation via abductive inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 443–456. https://doi.org/10.1145/2509136.2509511

[9] 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), 35–45. https://doi.org/10.1016/j.scico.2007.01.015

[10] R. W. Floyd. 1967. Assigning meanings to programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, J. T. Schwartz (Ed.). American Mathematical Society, Providence, 19–32.

[11] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, Tibor Gyimóthy and Andreas Zeller (Eds.). ACM, 416–419. https://doi.org/10.1145/2025113.2025179

[12] Juan P. Galeotti, Nicolás Rosner, Carlos López Pombo, and Marcelo F. Frias. 2010. 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*, Paolo Tonella and Alessandro Orso (Eds.). ACM, 25–36. https://doi.org/10.1145/1831708.1831712

[13] Isabelle Guyon and André Elisseeff. 2003. An Introduction to Variable and Feature Selection. *Journal of Machine Learning Research* 3 (2003), 1157–1182. http://www.jmlr.org/papers/v3/guyon03a.html

[14] C. A. R. Hoare. 2004. Towards the Verifying Compiler. In *From Object-Orientation to Formal Methods, Essays in Memory of Ole-Johan Dahl (Lecture Notes in Computer Science)*, Olaf Owe, Stein Krogdahl, and Tom Lyche (Eds.), Vol. 2635. Springer, 124–136. https://doi.org/10.1007/978-3-540-39993-3_8

[15] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2016. Test oracle assessment and improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 247–258. https://doi.org/10.1145/2931037.2931062

[16] Kenneth A. De Jong. 2006. *Evolutionary computation - a unified approach.* MIT Press.

[17] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[18] Barbara Liskov and John V. Guttag. 2001. *Program Development in Java - Abstraction, Specification, and Object-Oriented Design.* Addison-Wesley.

[19] Florian Merz, Stephan Falke, and Carsten Sinz. 2012. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings (Lecture Notes in Computer Science)*, Rajeev Joshi, Peter Müller, and Andreas Podelski (Eds.), Vol. 7152. Springer, 146–161. https://doi.org/10.1007/978-3-642-27705-4_12

[20] Bertrand Meyer. 1997. *Object-Oriented Software Construction, 2nd Edition.* Prentice-Hall.

[21] Bertrand Meyer. 1998. Design by Contract: The Eiffel Method. In *TOOLS 1998: 26th International Conference on Technology of Object-Oriented Languages and Systems, 3-7 August 1998, Santa Barbara, CA, USA*. IEEE Computer Society, 446. https://doi.org/10.1109/TOOLS.1998.711043

[22] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa Ling Liu. 2007. Automatic Testing of Object-Oriented Software. In *SOFSEM 2007: Theory and Practice of Computer Science, 33rd Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 20-26, 2007, Proceedings (Lecture Notes in Computer Science)*, Jan van Leeuwen, Giuseppe F. Italiano, Wiebe van der Hoek, Christoph Meinel, Harald Sack, and Frantisek Plasil (Eds.), Vol. 4362. Springer, 114–129. https://doi.org/10.1007/978-3-540-69507-3_9

[23] 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.

[24] Facundo Molina, Renzo Degiovanni, Pablo Ponzio, Germán Regis, Nazareno Aguirre, and Marcelo F. Frias. 2019. Training binary classifiers as data structure invariants. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 759–770. https://doi.org/10.1109/ICSE.2019.00084

[25] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 75–84. https://doi.org/10.1109/ICSE.2007.37

[26] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. 2014. Automated Fixing of Programs with Contracts. *IEEE Trans. Software Eng.* 40, 5 (2014), 427–449. https://doi.org/10.1109/TSE.2014.2312918

[27] Nadia Polikarpova, Carlo A. Furia, Yu Pei, Yi Wei, and Bertrand Meyer. 2013. What good are strong specifications?. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 262–271. https://doi.org/10.1109/ICSE.2013.6606572

[28] Sam Ratcliff, David Robert White, and John A. Clark. 2011. Searching for invariants using genetic programming and mutation testing. In *13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Proceedings, Dublin, Ireland, July 12-16, 2011*, Natalio Krasnogor and Pier Luca Lanzi (Eds.). ACM, 1907–1914. https://doi.org/10.1145/2001576.2001832

[29] Stuart J. Russell and Peter Norvig. 2010. *Artificial Intelligence - A Modern Approach (3. internat. ed.).* Pearson Education. http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0136042597,00.html

[30] Seyed Reza Shahamiri, Wan Mohd Nasir Wan-Kadir, Suhaimi Ibrahim, and Siti Zaiton Mohd Hashim. 2011. An automated framework for software test oracle. *Information & Software Technology* 53, 7 (2011), 774–788. https://doi.org/10.1016/j.infsof.2011.02.006

[31] Seyed Reza Shahamiri, Wan M. N. Wan-Kadir, Suhaimi Ibrahim, and Siti Zaiton Mohd Hashim. 2012. Artificial neural networks as multi-networks automated test oracle. *Autom. Softw. Eng.* 19, 3 (2012), 303–334. https://doi.org/10.1007/s10515-011-0094-z

[32] Anthony J. H. Simons. 2007. JWalk: a tool for lazy, systematic testing of java classes by design introspection and user interaction. *Autom. Softw. Eng.* 14, 4 (2007), 369–418. https://doi.org/10.1007/s10515-007-0015-3

[33] Muhammad Usman, Wenxi Wang, Kaiyuan Wang, Cagdas Yelen, Nima Dini, and Sarfraz Khurshid. 2019. A Study of Learning Data Structure Invariants Using Off-the-shelf Tools. In *Model Checking Software - 26th International Symposium, SPIN 2019, Beijing, China, July 15-16, 2019, Proceedings (Lecture Notes in Computer Science)*, Fabrizio Biondi, Thomas Given-Wilson, and Axel Legay (Eds.), Vol. 11636. Springer, 226–243. https://doi.org/10.1007/978-3-030-30923-7_13

[34] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[35] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Trans. Software Eng.* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[36] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa (Eds.). ACM, 272–281. https://doi.org/10.1145/1134324