From Operational to Declarative Specifications using a Genetic Algorithm

Facundo Molina
Dept. of Computer Science,
University of Rio Cuarto, Argentina

Renzo Degiovanni
Dept. of Computer Science,
University of Rio Cuarto, Argentina

Germán Regis
Dept. of Computer Science,
University of Rio Cuarto, Argentina

Pablo Castro
Dept. of Computer Science,
University of Rio Cuarto, Argentina

Nazareno Aguirre
Dept. of Computer Science,
University of Rio Cuarto, Argentina

Marcelo Frias
Dept. of Software Engineering,
Buenos Aires Institute of Technology,
Argentina

ABSTRACT

In specification-based test generation, sometimes having a formal specification is not sufficient, since the specification may be in a different formalism from that required by the generation approach being used. In this paper, we deal with this problem specifically in the context in which, while having a formal specification in the form of an operational invariant written in a sequential programming language, one needs, for test generation, a declarative invariant in a logical formalism. We propose a genetic algorithm that given a catalog of common properties of invariants, such as acyclicity, sortedness and balance, attempts to evolve a conjunction of these that most accurately approximates an original operational specification. We present some details of the algorithm, and an experimental evaluation based on a benchmark of data structures, for which we evolve declarative logical invariants from operational ones.

ACM Reference Format:
Facundo Molina, Renzo Degiovanni, Germán Regis, Pablo Castro, Nazareno Aguirre, and Marcelo Frias. 2018. From Operational to Declarative Specifications using a Genetic Algorithm. In *SBST'18: IEEE/ACM 11th International Workshop on Search-Based Software Testing , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3194718.3194725

1 INTRODUCTION

Various approaches to automated test generation require a *specification* in order to automatically generate tests or test inputs. Examples of these are test generation tools based on symbolic or concolic execution [10, 17], tools that generate inputs from program invariants [2, 9], and tools that combinatorially generate inputs and use specifications for filtering [7]. However, not all tools use the same specification formalism; some tools require specifications to be given as *operational* predicates, i.e., as program routines in a programming language [2, 10] (e.g., so called `repOK` routines are class invariants captured operationally through a program); others support constraints in a logical formalism [5, 9]; and a few are able to combine different formalisms [4, 7, 15].

An issue that arises with the availability of multiple specification formalisms is that many times one does count with a specification, but this specification is not provided in the *right* language for the use of a given test generation approach. For instance, one may have a class invariant for a given Java class, written as a `repOK` routine, but in order to use, say a SAT-solving based generation mechanism, such invariant has to be somehow translated to an appropriate logical formalism. This is particularly relevant with the increasing growth of tools and techniques for program analysis, and the potential combined use of these tools, which may be inhibited by the "mismatch" in the specification styles required by the tools involved. A concrete example of this situation can be observed, for instance, in *bounded lazy initialization with SAT support* [16], where a combination of symbolic execution and SAT solving requires the user to provide *two* equivalent program invariants, one given as a `repOK` routine (used for lazy initialization), and the other as a logical specification (used for computing bounds and pruning symbolic execution).

This is exactly the problem we are interested in, in this paper. The problem is relevant because even in the case in which a translation from one formalism to the other is available, the "target" specifications resulting from the translations may be unsuitable for analysis reasons. For example, one *can* indeed translate a Java `repOK` routine into Alloy's relational logic [8], through the use of translations that capture programming language constructs in the logic, as those embedded in some program analysis tools [3, 5]. But the obtained logical formulas (that capture program executions) lead to unacceptable performances if these are used for test generation.

To deal with the above situation, we propose a genetic algorithm, that given a catalog of properties commonly used as part of invariants, such as acyclicity, sortedness and balance, appropriately specified in relational logic, attempts to evolve the conjunction of these that most accurately approximates an original operational specification, given as a `repOK` routine. We present some details of the algorithm, and an experimental evaluation based on a benchmark of data structures, for which we evolve declarative logical invariants from operational ones. The experiments show that declarative invariants that very precisely approximate provided operational ones can be efficiently produced.

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.
SBST'18, May 28–29, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5741-8/18/05…$15.00
https://doi.org/10.1145/3194718.3194725

2 MOTIVATION

To motivate our approach, let us consider an implementation of lists given as heap allocated doubly linked lists, as shown in Figure 1. Suppose that we are interested in testing that a routine that manipulates such data structure, for example the insertion routine, works as expected, or in particular, that it preserves the *representation invariant* of doubly linked lists. To perform automated testing in this context, we would need a specification establishing when a given list is *valid*, both to be used for assertions in tests, as for automatically producing test inputs satisfying such specification. As put forward in [12], one can specify the representation invariant as a boolean routine, the `repOK()`, that returns *true* iff the structure it is applied to satisfies the corresponding *representation invariant*. In the case of doubly linked lists, this routine, shown in Figure 2, must state two points: first, `header` holds a cyclic linked list, and second, the number of nodes in the list coincides with the value in field `size`.

Some test generation tools, notably those based on constraint-solving [5, 9], can profit from specifications written in a *logical* formalism, in contrast with the previously mentioned *operational* `repOK` routines. Indeed, some constraint-solving based approaches can more efficiently generate test inputs if invariants are given in a declarative formalism like JML or Alloy [8]. These languages offer a different specification "paradigm", and properties such as reachability, (a)cyclicity and the like, are typically captured through some transitive closure expressions. As an example, Figure 3 shows a declarative predicate, in Alloy's relational logic, that expresses an invariant equivalent to property `repOK()` of Figure 2.

```
public class DoublyLinkedList {
    private Node header;
    private int size;
    ...
}

public class Node {
    private int element;
    private Node next;
    private Node previous;
    ...
    //setters and getters
    //of the above fields
    ...
}
```

Figure 1: Java classes defining doubly linked lists.

Having the operational invariant specified through routine `repOK()` enables the use of various tools for test generation that expect this kind of property (e.g., Korat [2], where the doubly linked list example was taken from, in one such tool). But if we need to use a tool that expects the specification to be given declaratively (e.g., those in [1, 9]), our `repOK` is of little use. Even though one may employ a translation from operational specifications into declarative specifications (in bounded contexts), like those provided in tools like TACO [5, 6] and CBMC [11], the obtained specifications are in general unsuitable for analysis. In particular, the excessive use of quantifiers to capture program executions lead to specifications

```
public boolean repOK() {
    Set visited = new java.util.HashSet();
    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;
    if (visited.size() != size) return false;
    return true;
}
```

Figure 2: Operational version of the representation invariant for doubly linked lists.

```
one sig Null { }

sig List { }

sig Node { }

pred repOK[thiz: List, header: List->one Node+Null,
                  next: Node -> one Node+Null] {
(all n: thiz.header.*(next+prev) | n=n.prev.next)
and #(thiz.header.*(next+prev)-Null) = thiz.size
}
```

Figure 3: Declarative version of the representation invariant for doubly linked lists, in Alloy's relational logic.

that are very costly to compile, that many times do not pass, in the context of SAT-based test generation, the CNF generation phase.

The following section will describe our proposal to tackle this problem, which in essence consists of taking a *catalog* of properties that are commonly part of invariant specifications, appropriately characterized in the target formalism (in our case, Alloy's relational logic), and using a genetic algorithm to evolve an expression (a conjunction in our case) involving properties from the catalog, that more closely approximate a given operational invariant.

3 THE GENETIC ALGORITHM

As we mentioned in previous sections, our objective is to generate a declarative specification Φ that most accurately approximates an operational specification Φ_{op}, by combining common invariant properties taken from a catalog. Below we describe the main components of the genetic algorithm designed for this purpose.

3.1 Genes and Chromosomes for Candidate Specifications

In order to capture candidate specifications, we simply define chromosomes as vectors of integer genes. Each chromosome has as many genes as there are properties in the catalog, and the value of each gene can be 0, 1 or 2. If the *i-th* gene has value 0, then the *i-th* formula of the catalog is negated; if the gene has value 1, then

the *i-th* formula is considered positively; finally, if the gene has value 2, then the *i-th* formula is *disabled* (not part of the candidate specification). Thus, the candidate specification represented by a given chromosome is the *conjunction* of the formulas obtained from the interpretation of the genes, depending on their respective values. For instance, if the catalog is composed by the (ordered) set of formulas $\{f_0, f_1, f_2, f_3, f_4\}$, then chromosome $[0, 1, 2, 0, 2]$ will represent the specification $\neg f_0 \wedge f_1 \wedge \neg f_3$.

In our experiments, the initial population is produced by generating all individuals with exactly one positive gene (value 1), and all the others disabled (value 2). That is, we initially have as many individuals as specifications in the catalog. The maximum size for the population is set to 100.

3.2 Genetic Operators

Genetic operators are used to produce the search space, by generating new individuals from existing ones in a population. The main mechanism to achieve this is by combining parts of existing chromosomes through *crossover*. We use one-point crossover to build new chromosomes, by randomly selecting a point to "split" two chromosomes, and combining the initial (resp., final) part of one of them with the final (resp., initial) part of the other. In our experiments, we use a crossover rate of 35%.

The second mechanism to generate new chromosomes is *mutation*, i.e., the generation of a new individual by randomly changing characteristics of an existing one. Since in our case genes only have three possible values (0,1 or 2), our mutation operator simply randomly sets a value in the range $[0, 2]$ with a probability of $1/12$, of each gene to be mutated.

3.3 Fitness of Candidate Specifications

Our fitness function is meant to assess how close are the corresponding candidates to the desired specification, and is the most important part of our algorithm. We exploit the operational specification Φ_{op}, to (indirectly) compare candidate specifications against this one. In order to do so, we automatically generate from Φ_{op} a set of *positive* and *negative* examples. These are instances that satisfy and do not satisfy Φ_{op}, respectively. These instances can be generated using any test input generation mechanism that requires an operational specification, e.g., [2]. We use an ad hoc variant of Korat, that generates inputs using that "cover" different values for object fields. The number of generated positive and negative cases is limited to a provided bound k.

Fitness $f(c)$ for a chromosome c is computed as follows. First, we build the specification Φ_c corresponding to c, and evaluate whether the positive and negative cases satisfy Φ_c. If any positive case fails with Φ_c, meaning that there are cases that should be accepted but our specification rejects them, then $f(c) = 0$. Instead, if the candidate has only negative cases (cases that should not pass the specification but do so), fitness is defined as follows:

$$f(c) = (\mathsf{MAX} - \mathsf{neg}(c)) + \left(\frac{1}{\mathsf{len}(spec(c)) + 1}\right)$$

where MAX is a constant larger than k, the total number of negative cases; $\mathsf{neg}(c)$ is the number of negative cases that satisfy Φ_c; and $\mathsf{len}(spec(c))$ is the length of c, i.e., the number of formulas from the catalog present in the conjuction.

The rationale for this definition of the fitness function has to do with the fact that we attempt to over approximate the sought-for specification. Thus, when a positive case is not accepted by a candidate, we will simply consider it unfit. Fitness for other candidates has two parts. First, the fewer the "counterexamples", the better; second, the smaller the specification, the better. This last part can be thought of as a penalty related to formula length, that will make the genetic algorithm tend towards producing smaller formulas.

3.4 Selection

The selection operation determines which individuals are to be kept in the next generation. Our selection operation is very simple. It maintains a predefined amount of the fittest individuals by sorting all the chromosomes by decreasing order according to their fitness values, and then selecting the top individuals. This simple selection mechanism results useful in our problem since the algorithm will tend to keep the chromosomes representing specifications containing formulas with less negative cases that do not satisfy Φ_{op} (recall that the higher the fitness value, the fewer counterexamples the formula has).

4 EVALUATION

Our evaluation is based on invariants of the following data structures taken from Korat's case studies: singly linked lists; sorted singly linked lists; circular linked lists; doubly linked lists; binary trees; binary search trees; heaps; (binary) directed acyclic graphs; and red-black trees. The genetic algorithm has been implemented using JGAP, and the experiments were run on a workstation with Intel Core i7 2600, 3.40 Ghz, and 16Gb of RAM. The catalog for our genetic algorithm is composed of properties commonly found in invariants, with distinguishing cases for linear structures (structures with a single reference field per node) and n-ary (tree-like) structures (e.g., binary trees). More precisely, for linear structures we considered 23 properties, including (a)cyclicity, circularity, etc., the relationship between number of reachable nodes and integer-typed fields, and ordering constraints. For n-ary structures, on the other hand, we considered 28 properties, including all of the linear cases, and other properties such as disjointness across different fields, balance, etc.

For each case study, we ran the algorithm 10 times, with a limit of 20 generations (i.e., evolutions of the genetic algorithm population). Table 1 reports the minimum, maximum and average runs, indicating the number of generations (g) and the time (t) in seconds, that were necessary to obtain declarative invariants. We distinguish between the cost of computing the first suitable invariant, and the cost of computing the "best" invariant (the algorithm continues running, trying to make it more concise). In all these cases the obtained invariants were indeed *equivalent* to their corresponding operational ones. Some results were surprising, e.g., an acyclicity property indirectly captured via cardinality constraints:

```
(thiz.size = # thiz.head.*next - Null) and
not (thiz.size = # thiz.head.*next)
```

5 RELATED WORK AND CONCLUSION

The problem of automatically producing specifications has been extensively studied. In the context of Alloy, the approach in [13]

Table 1: Experimental Results corresponding to learning declarative invariants from operational ones, using our evolutionary algorithm.

	First Invariant Found						Best Invariant Found					
Data	Min		Max		Avg		Min		Max		Avg	
Structure	Gen	*Sec.*	Gen	*Sec.*	Gen	*Sec.*	Gen	*Sec.*	Gen	*Sec.*	Gen	*Sec.*
s. linked lists	2	*4*	5	*12*	4	*8*	3	*5*	6	*27*	4	*12*
s. linked sort. lists	2	*8*	7	*23*	5	*16*	3	*10*	7	*32*	5	*19*
s. circular lists	2	*6*	3	*16*	2	*10*	2	*6*	4	*20*	3	*13*
doubly linked lists	1	*8*	5	*35*	3	*24*	1	*8*	5	*35*	3	*25*
binary trees	3	*35*	6	*146*	5	*102*	3	*35*	8	*296*	6	*148*
binary search trees	3	*14*	6	*75*	5	*42*	3	*14*	6	*75*	5	*44*
heaps	6	*39*	18	*90*	9	*58*	6	*39*	19	*116*	13	*83*
binary DAGs	2	*3*	4	*10*	3	*7*	2	*3*	4	*10*	3	*7*
red-black trees	7	*95*	15	*242*	12	*171*	7	*112*	20	*303*	15	*225*

is based solely on positive examples, as opposed to our case. The work in [14] also uses genetic algorithms, but attempts to evolve navigational expressions, as opposed to our (simpler) case based on a specification catalog. Our approach is easier to extend to support new properties, a limitation of [14].

In summary, we have presented an approach to compute a declarative specification in Alloy's relational logic from an operational one in Java, based on a genetic algorithm. The approach considers a catalog of common invariant properties and tries to achieve a conjunction of these that approximates the original invariant. It produces valid and invalid cases from the operational specification, which are then used as part of the fitness function driving the algorithm, to "grade" specification candidates. Our preliminary experimental evaluation shows promising results.

We plan to further develop our approach, and in particular to search for more general specification patterns to consider in specification catalogs. We also plan to incorporate our operational-to-declarative translation mechanism in the context of BLISS [16], to simplify the requirements for users of the technique to only providing an operational invariant.

REFERENCES

[1] 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. 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*, pages 21–30. IEEE Computer Society, 2013.

[2] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In Phyllis G. Frankl, editor, *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, pages 123–133. ACM, 2002.

[3] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. Modular verification of code with SAT. In Lori L. Pollock and Mauro Pezzè, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 109–120. ACM, 2006.

[4] Kyle Dewey, Lawton Nichols, and Ben Hardekopf. Automated data structure generation: Refuting common wisdom. In Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 32–43. IEEE Computer Society, 2015.

[5] Juan P. Galeotti, Nicolás Rosner, Carlos Gustavo López Pombo, and Marcelo F. Frias. TACO: efficient sat-based bounded verification using symmetry breaking and tight bounds. *IEEE Trans. Software Eng.*, 39(9):1283–1307, 2013.

[6] Juan P. Galeotti, Nicolás Rosner, Carlos López Pombo, and Marcelo F. Frias. Analysis of invariants for efficient bounded verification. In Paolo Tonella and Alessandro Orso, editors, *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, pages 25–36. ACM, 2010.

[7] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in UDITA. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 225–234. ACM, 2010.

[8] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis.* MIT Press, 2006.

[9] Shadi Abdul Khalek, Guowei Yang, Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. Testera: A tool for testing java programs using alloy specifications. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 608–611. IEEE Computer Society, 2011.

[10] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.

[11] Daniel Kroening and Michael Tautschnig. Cbmc – c bounded model checker. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[12] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2000.

[13] Muhammad Zubair Malik, Aman Pervaiz, Engin Uzuncaova, and Sarfraz Khurshid. Deryaft: a tool for generating representation invariants of structurally complex data. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 859–862. ACM, 2008.

[14] Facundo Molina, César Cornejo, Renzo Degiovanni, Germán Regis, Pablo F. Castro, Nazareno Aguirre, and Marcelo F. Frias. An evolutionary approach to translate operational specifications into declarative specifications. In Leila Ribeiro and Thierry Lecomte, editors, *Formal Methods: Foundations and Applications - 19th Brazilian Symposium, SBMF 2016, Natal, Brazil, November 23-25, 2016, Proceedings*, volume 10090 of *Lecture Notes in Computer Science*, pages 145–160, 2016.

[15] Nicolás Rosner, Valeria S. Bengolea, Pablo Ponzio, Shadi Abdul Khalek, Nazareno Aguirre, Marcelo F. Frias, and Sarfraz Khurshid. Bounded exhaustive test input generation from hybrid invariants. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 655–674. ACM, 2014.

[16] Nicolás Rosner, Jaco Geldenhuys, Nazareno Aguirre, Willem Visser, and Marcelo F. Frias. BLISS: improved symbolic execution by bounded lazy initialization with SAT support. *IEEE Trans. Software Eng.*, 41(7):639–660, 2015.

[17] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.