An evolutionary approach to translating operational specifications into declarative specifications

Facundo Molina[a,b,*], César Cornejo[a,b], Renzo Degiovanni[c], Germán Regis[a], Pablo F. Castro[a,b], Nazareno Aguirre[a,b], Marcelo F. Frias[d,b]

[a] *Department of Computer Science, FCEFQyN, University of Río Cuarto, Argentina*
[b] *National Council for Scientific and Technical Research (CONICET), Argentina*
[c] *SnT, University of Luxembourg, Luxembourg*
[d] *Department of Software Engineering, Buenos Aires Institute of Technology, Argentina*

Keywords:
Program specification
Program analysis
Relational logic
Evolutionary algorithms

A B S T R A C T

Various tools for program analysis, including run-time assertion checkers and static analyzers such as verification and test generation tools, require formal specifications of the programs being analyzed. Moreover, many of these tools and techniques require such specifications to be written in a particular style, or follow certain patterns, in order to obtain an acceptable performance from the corresponding analyses. Thus, having a formal specification sometimes is not enough for using a particular technique, since such specification may not be provided in the right formalism. In this paper, we deal with this problem in the increasingly common case of having an *operational* specification, while for analysis reasons requiring a *declarative* specification. We propose an evolutionary approach to translate an operational specification written in a sequential programming language, into a declarative specification, in relational logic. We perform experiments on a benchmark of data structure implementations, for which operational invariants are available, and show that our evolutionary computation based approach to translating specifications achieves very good precision in this context, and produces declarative specifications that are more amenable to analyses that demand specifications in this style. This is assessed in two contexts: bounded verification of data structure invariant preservation, and instance enumeration using symbolic execution aided by tight bounds.

1. Introduction

Many software validation and verification activities require a description of the software under analysis, since many analyses typically consist in checking compliance of the software against some prescribed intended behavior [18]. Formal specifications have gained an important relevance in such contexts, mainly due to their unambiguous interpretation and the increasing availability of technologies for their automated analysis, which are making them part of effective software analysis approaches. Indeed, automated analysis techniques that require some sort of formal specification have proved to be useful mechanisms to aid in many software development activities, including challenging tasks that are usually performed

* Corresponding author.
E-mail addresses: fmolina@dc.exa.unrc.edu.ar (F. Molina), ccornejo@dc.exa.unrc.edu.ar (C. Cornejo), renzo.degiovanni@uni.lu (R. Degiovanni), gregis@dc.exa.unrc.edu.ar (G. Regis), pcastro@dc.exa.unrc.edu.ar (P.F. Castro), naguirre@dc.exa.unrc.edu.ar (N. Aguirre), mfrias@itba.edu.ar (M.F. Frias).

manually, such as fault localization [25,46], test generation [4,38,6], bug finding [23,50,10,33], program verification [2,21] and program repair [20,43,40].

While all of the above-cited automated analysis techniques employ formal specifications, the specific notation and *style*, or specification paradigm involved, often differ across techniques. In particular for the context of program specification, two major specification styles can be identified, the *operational* and the *declarative*. In the operational style, specifications are captured through code, e.g., via a routine that checks whether the internal representation of a given object is consistent [32]. Examples of techniques using an operational style are those reported in [4,45]. On the other hand, the declarative style often uses a logical formalism for expressing the same kind of property. Some examples are the use of first-order logic complemented with closure operators, as put forward by notations such as JML [5] and Alloy's relational logic [22], and first-order logic with spatial operators, as in separation logic [2].

With the proliferation of notations and, more importantly, with the adoption of different specification styles by different tools, there is an emergence of techniques that profit from formal specifications only when these are expressed in particular notations or styles. For instance, the test generation tool Korat [4] requires a specification to be provided operationally (as a `repOK` routine) to automatically produce test inputs. It implements an effective and very efficient "perfect" symmetry-breaking mechanism, preventing the tool from generating isomorphic structures and contributing to its overall performance, that is particularly tied to the operational specification. This makes it very difficult and ineffective to use the technique to generate tests for, say, an object-oriented program equipped with a JML contract, or to profit from the symmetry-breaking technique in other contexts where non-operational specifications are available. On the other hand, tools for verification based on declarative notations, e.g., TACO [15], can exploit mechanisms such as tight bounds [14], whose computation are also strongly tied to declarative notations, and cannot straightforwardly nor effectively be computed from operational specifications. This situation is combined with the increasing need for cross-usage of automated analysis tools. A sample scenario arises with current techniques for fault localization and program repair, that require tests for their application; combining such tools with automated test generation is an obvious approach that integrates automated analysis technologies (see, e.g., [1]).

The need to take advantage of analysis techniques (and indirectly to profit from the optimizations inherent to their corresponding contexts) that are available in one style of specification to another, or combine techniques that apply to different specification styles, calls for mechanisms to allow us to *translate* specifications from one style to another. Moreover, it is often the case that even when semantics-preserving translations are available to translate between different formalisms, the existence of such translations is still unsatisfactory. Indeed, such translations are generally guided by the syntax and semantics of the related formalisms, but do not take into account automated analysis. That is, these translations can produce translated specifications that, although "correct" in the sense that they preserve the semantics of the original specifications, are ineffective for the analysis mechanisms of the target notations, due to the violation of (many times implicit) patterns for optimal exploitation of analysis. For instance, Korat requires `repOK` methods to "fail as soon as possible", in the sense that these methods should try to decide when a structure does not satisfy the predicate visiting the least possible elements of the structure, for test generation to be effective. Similarly, the efficiency of tools like Alloy are in many cases very dependent on how specifications are written; analyzing specifications with large numbers of (existential) quantification often fails during preprocessing (e.g., during translation to CNF to use SAT-based verification), while expressing equivalent specifications through simple transformations (e.g., skolemizations) can have a drastic impact in analysis efficiency. Appropriate translations should take these issues into account, which clearly are beyond the syntax and semantics of the involved languages.

In this paper, we present an evolutionary approach to solve a particular instance of the above-described translation problem, namely the translation from an *operational* specification of a representation invariant, written in an imperative sequential programming language, to a *declarative* invariant specification, in relational logic. In our context, the given operational specification is simply a routine that verifies whether a given object satisfies some properties or not, as used in [32,4,38] (where Java methods that check the internal consistent representation of Java objects are employed). The produced declarative specification, that should express the same properties as the operational one, is a logical formula written in Alloy's relational logic [22], a first-order logic complemented with closure operators, used, e.g., in [28,26,8,20]. Our translation approach is based on the use of a genetic algorithm [19] to look for a specification of an invariant in relational logic matching a given operational specification, in the sense that it accepts the same structures. Individuals in the genetic algorithm represent candidate declarative specifications. The original operational specification is used to define the *fitness* of individuals, essentially by counting the number of (bounded) structures that do not satisfy the candidate invariant but should satisfy it (according to the operational invariant), and those that do satisfy the candidate invariant but should not satisfy it (again, according to the operational invariant). Fitness also incorporates some additional criteria that allow the genetic algorithm to produce more compact translated specifications.

We assess our approach in various directions. Firstly, we evaluate our algorithm's effectiveness and efficiency on a benchmark of data structure implementations, translating their corresponding operational representation invariants. As our experiments show, our genetic algorithm is able to find with precision equivalent declarative invariants in a relatively small number of generations. We also analyze the precision of the obtained invariants, comparing them with invariants automatically inferred using two related tools, Daikon [12] and Deryaft [34]. Secondly, we analyze the impact of standard parameters of genetic algorithms, like population size and mutation rate, in the performance of our technique. Finally, we assess the use of our learned invariants in practical contexts: we show that verifying invariant preservation using our learned specifi-

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

Fig. 1. Java classes defining singly linked lists.

```
public boolean repOK() {
        if (header == null) return size == 0;
        Set<Node> visited = new java.util.HashSet<Node>();
        visited.add(header);
        Node current = header;
        while (true) {
            Node next = current.getNext();
            if (next == null) break;
            if (!visited.add(next)) return false;
            current = next;
        }
        if (visited.size() != size) return false;
        return true;
}
```

Fig. 2. Operational version of the representation invariant for singly linked lists.

cations outperforms invariant preservation verification with specifications obtained using an existing semantics-preserving translation, for the mentioned data structure implementations; and we show that instance enumeration (test input generation) using symbolic execution is improved when aided by tight bounds, where the latter are computed using our learned declarative specifications.

This paper is a journal extension of [36]. The additional contributions, with respect to the original conference submission, are the following: *(i)* an improved genetic algorithm, in particular in relation to how the fitness of candidate specifications is measured, *(ii)* additional data structures considered for the evaluation of efficiency, effectiveness and precision of our technique to translate operational specifications into declarative ones, *(iii)* an experimental evaluation of the impact of various parameters of the genetic algorithm, including our approach to producing the initial population, the population size, and the mutation rate, and *(iv)* an evaluation of the impact of our learned declarative specifications in aiding symbolic execution through tight bounds (whose computation requires declarative logical specifications). We have also provided further details on the mechanisms behind our genetic algorithm, including further precisions on how mutation and crossover are implemented.

The remainder of the paper is organized as follows. In Section 2, we motivate our approach by presenting an illustrating example, that in particular shows the need to translate across different specification styles. In Section 3 we present our evolutionary algorithm for learning declarative specifications from operational ones, including detailed descriptions of how candidate specifications are captured as chromosomes, and how these are evaluated during the genetic algorithm's search. In Section 4 we experimentally evaluate our approach's effectiveness and efficiency, on a benchmark composed of various data structure implementations. This section also contains an evaluation of the impact of parameters of the genetic algorithm in its performance, and the profit that the learned specifications provide in some practical analysis contexts. Section 5 compares our technique with related work, and finally, in Section 6, we present our conclusions and lines for further work.

2. A motivating example

In order to motivate our approach, let us consider an analysis scenario involving a simple data structure, *singly linked lists*. This data structure is captured through classes `SinglyLinkedList` and `Node`, as defined in Fig. 1. Assume, for instance, that we would need to verify that a routine manipulating such data structure, e.g., an insertion routine, preserves the representation invariant of lists, i.e., inserting an element in a *valid* list retrieves also a *valid* list. In order to proceed with this verification, we then need a specification of what it means for singly linked lists to be *valid*. A particular specification, with a style put forward in [32], consists in capturing the *representation invariant* of the structure (i.e., the intended *validity* condition for singly linked lists) through a boolean routine, that checks whether the condition holds or not for a given structure. An example of such a method, named `repOK()` as usual, indicating that singly linked lists must be acyclic and their number of nodes must coincide with the value in the `size` field, is shown in Fig. 2.

A substantially different approach to the *operational* style of using code to write specifications, is based on the use of some suitable logical formalism, for the same task. This alternative approach has been extensively used, from the seminal works of Hoare and Floyd, where first-order logic is used to express assertions regarding program states, to more modern languages such as JML [5] and Alloy [22], which due to further expressive power needs, have extended first-order logic with closure or reachability predicates. In particular, notice that first-order logic is not sufficiently expressive to capture the acyclicity on singly linked lists, in our example. A declarative predicate, expressed in Alloy's relational logic, and capturing

```
one sig Null { }

sig List { }

sig Node { }

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

Fig. 3. Declarative version of the representation invariant for singly linked lists, in Alloy's relational logic.

exactly the same property as method `repOK()` in Fig. 2, is shown in Fig. 3. A brief description of the Alloy notation is given in order to better understand this specification. Signatures declare sets of atoms (i.e., data domains), with different signatures describing disjoint sets. The `one` modifier for `Null` indicates this domain is a singleton; it is a standard way of declaring constants in Alloy. Predicates are "parameterized" formulas, i.e., formulas with free variables. Variables need to be typed, but types are not limited to domains (signatures), they can also be *relations*. For instance, `repOK` predicates over a list (`thiz`, since *this* is a reserved word in Alloy), a function `header` mapping list atoms to either a node or null, and a function `next` mapping nodes to either a node or null. The intuition here is that `header`, `next` and `size` are the relations that capture how lists relate to their head nodes, nodes are related to their "next" nodes, and lists are related to their sizes, in the heap. The body of the predicate is composed of a quantified formula (`all` is the universal quantifier), and can be understood directly, considering that operators `*` and `^` denote reflexive-transitive and transitive closures, respectively, `#` denotes cardinality, and propositional connectives are denoted by `and` (conjunction), `or` (disjunction), `!` (negation) and `implies` (implication); operator `!in` denotes nonmembership. The body of the formula states then that no node reachable from the header of the list can reach itself through `next`, and that the size of the list coincides with the number of nun-null nodes reachable from the header of the list (including the header). For further details regarding Alloy, we refer the reader to [22].

To illustrate the need for effective translations across different specification styles, suppose that we only count with the *operational* invariant, specified through method `repOK()` in Java. While this specification is suitable for generating test inputs using Korat [4] (in fact, this particular example is taken from Korat's set of case studies [30]), if we want to perform bounded verification using a tool like TACO [14,15], then this specification becomes unsuitable, since TACO expects a *logical* specification. However, it is possible to translate an operational specification into an equivalent declarative specification (equivalent in bounded contexts), e.g., using the translations embedded in tools like TACO [14,15] and CBMC [31]. The logical specification resulting from the translation of the `repOK()` method shown in Fig. 2 is shown in Fig. 4. This specification,[1] while correct with respect to the semantics of the original (again, for a particular bounded scope), is unsuitable for verification. For instance, verifying that method `insert` preserves the representation invariant for lists of size at most 12 takes 3839 seconds when using the invariant in Fig. 2, whereas it takes 1648 seconds when using the invariant in Fig. 3. As we will show later on in this paper, such a difference in efficiency becomes more notorious in more complex data structure invariants (see Section 4).

The above described problem is the motivation for our approach. As we explain in the following section, we have developed an evolutionary algorithm to translate from operational specifications into declarative ones, with the aim of obtaining better suited specifications, from the point of view of analysis. More precisely, our aim is to obtain, from operational specifications such as that in Fig. 2, declarative specifications closer to that in Fig. 3 (as opposed to that in Fig. 4), that would allow us to perform certain automated analyses more efficiently.

3. An evolutionary algorithm for learning declarative specifications

As we mentioned in previous sections, our objective is to compute a declarative specification Φ in relational logic, from an operational specification Φ_{op}, written in a sequential programming language. To do so, we design a genetic algorithm, that we describe below. Genetic algorithms [19] are non-exhaustive guided search algorithms, based on a hill climbing strategy [44]. The search space is composed of a generally very large set of individuals, that encode candidate solutions. The search objective is to find an individual with sought-for features, which should represent a solution to the problem. As opposed to classic search algorithms, genetic algorithms progress by maintaining a *population*, a set of candidate solutions, which is *evolved*, generating new individuals through recombination and mutation operators, and then selecting a number of individuals in the population that will be passed to the next generation. Genetic operators enable the algorithm to explore the search space, with selection reducing the search by prioritizing individuals of higher quality. The quality of an individual is measured by a *fitness function*, a heuristic function used to guide the search. This function applies to individuals, and its

[1] Basically, this specification corresponds to the pre-post relation of program `repOK`, for a maximum number of 3 iterations. Predicate parameters are the pre and post state variables (underscript zero is for pre states, while underscript one is for post states, with `result` being the variable representing the program output), and the predicate body includes a number of existentially quantified variables representing intermediate values for variables (we omitted the existential quantification for the sake of brevity).

```
pred repOK[thiz_0: List, header_0: List ->one (Node + Null),
    size_0: List ->one Int, next_0: Node ->one (Node + Null),
    result_0, result_1: boolean] {
nodesToVisit_1 = thiz_0.size_0 and
current_1 = thiz_0.header_0 and ((lt[thiz_0.size_0, 0] and
result_1 = false and current_1 = current_4 and
nodesToVisit_1 = nodesToVisit_4 ) or (not lt[thiz_0.size_0,0]
and  ((current_1 = current_4 and
nodesToVisit_1 = nodesToVisit_4 ) or
(gt[nodesToVisit_1, 0] and current_1 != Null and
nodesToVisit_2 = sub[nodesToVisit_1, 1] and
current_2 = current_1.next_0 and ((current_2 = current_4 and
nodesToVisit_2 = nodesToVisit_4 ) or (gt[nodesToVisit_2, 0]
and current_2 != Null and nodesToVisit_3 = sub[nodesToVisit_2,1] and
current_3 = current_2.next_0 and ((current_3 = current_4 and
nodesToVisit_3 = nodesToVisit_4 ) or (gt[nodesToVisit_3, 0]
and current_3 != Null and nodesToVisit_4 = sub[nodesToVisit_3, 1] and
current_4 = current_3.next_0)))))) and not (gt[nodesToVisit_4, 0] and
current_4 != Null) and ((eq[nodesToVisit_4, 0] and
current_4 = Null and result_1 = true) or
(not (eq[nodesToVisit_4, 0] and
current_4 = Null) and result_1 = false))))
}
```

Fig. 4. Declarative representation invariant for singly linked lists, obtained using a semantics-preserving translation from `repOK` in Fig. 2.

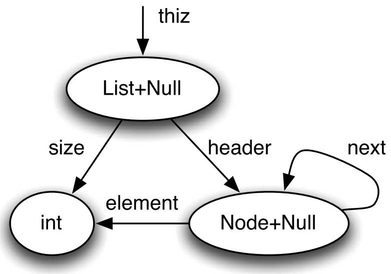

Fig. 5. Type graph for singly linked lists.

result is generalizable to a population (e.g., the fitness of a population may be taken as the fitness of its “fittest” individual). This function captures the features sought for in the search, and thus can be used as a halting criterion (e.g., the algorithm may stop after finding an individual with fitness above a certain threshold). Finally, individuals are often called *chromosomes*, and represented as vectors of *genes* that capture their characteristics. This idea is strongly related to how new individuals are constructed: by representing candidates as vectors of independent characteristics, one can build new candidates by combining part of the characteristics of an individual with part of the characteristics of another, or by randomly *changing* a characteristic of a given individual. These two operations are called *crossover* and *mutation*, respectively, and are the traditional mechanism to build new candidates out of existing ones in genetic algorithms. For further details, we refer the reader to [35].

3.1. Genes and chromosomes to represent candidate specifications

In order to capture candidate specifications, we start by taking the structure’s signature, i.e., its type description, and building a *type graph*. A type graph for a structure is automatically built from its fields and their types; nodes represent types, while arcs capture fields. As an example, consider the type graph for linked lists, as defined in Fig. 1, shown in Fig. 5.

Type graphs are used to form expressions, that will constitute the candidate specifications. Expressions are built out of paths in the graph. To make expressions finite, recursive fields are traversed at most once, and further “iteration” is represented through closure operators. For instance, from the type graph in Fig. 5, the following expressions are computed:

```
thiz
thiz.size
thiz.header
thiz.header.next
thiz.header.element
thiz.header.next.element
thiz.header.⋆next
thiz.header.⋆next.element
```

Moreover, in type graphs with multiple arcs connecting the same source and target nodes, their "union" is also considered for building expressions. Thus, for instance, for binary trees, there will be expressions of the form `thiz.root.left`, `thiz.root.right`, as well as `thiz.root.(left+right)`.

These expressions are complemented with constants, e.g., `Null`, `0`, `none` (empty set), to build expressions (integer expressions are also generated by applying the cardinality operator to non-singleton expressions). Also, the expressions cardinalities are taken into account (notice that the first 6 expressions above denote singletons, whereas the last two denote sets of any cardinality). Genes, the basic (independent) units that characterize chromosomes (in our case, representing candidate specifications) can be:

- the boolean constant `true`,
- an atomic formula built from the expressions originating in the type graph (including considered constants), respecting relational logic's grammar and taking into account types and cardinalities (e.g., `thiz.header != Null`, `thiz.header.*next = none`, etc.),
- a quantified formula, involving a (bound) variable `x`, and two expressions, one for `x`'s scope, the other for "predicating" in relation to `x` (e.g., `all n: thiz.header.*next.element | n != 0`, the two expressions here being `thiz.header.*next.element` and `0`); the first of these expressions is constrained to be a "set" expression, not a singleton.

Notice that, according to Alloy's grammar, the second item above includes, for every atomic formula α, its negation $\neg\alpha$. This is due to the fact that "boolean" operators in Alloy include their negated counterparts (e.g., `=` and `!=`, `in` and `!in`) [22].

A chromosome c is simply a vector of the previously described genes, and the specification represented by c is the *conjunction* of its genes:

$$c = \begin{bmatrix} g_0 & g_1 & g_2 & \dots & g_{n-1} \end{bmatrix} \Rightarrow spec(c) = g_0 \wedge g_1 \wedge g_2 \wedge \dots \wedge g_{n-1}$$

As opposed to what is common in genetic algorithms, our chromosomes have varying lengths, and genes' positions are disregarded. That is, if a gene belongs to a chromosome, it is part of the corresponding conjunction, independently of whether it is at the beginning of the conjunction, or in any other position; this is of course due to the well known associativity and commutativity properties of conjunction.

3.2. Initial population

The creation of the initial population consists of two steps. As a first step, we use the provided operational specification Φ_{op} to generate a set of instances that satisfy the specification (the *positive* instances), and a set of instances that do not satisfy the specification (the *negative* instances). These can be generated using any automated test input generation approach (requiring an operational specification). These are just samples of a few valid and invalid inputs, that will be used as part of the next step. They can be generated using, e.g., Korat, and collecting the first few valid instances generated, and invalid ones explored in the process. In our case, as we describe in the validation section, we translate the operational `repOK` into a relational logic predicate for a very small scope, using the semantics-preserving translation for bounded programs, and employ SAT solving to produce 2 valid and 2 invalid instances, i.e., 2 instances satisfying the obtained predicate, and 2 not satisfying it. As a second step, we take the set of expressions computed from the type graph (described in the previous section) and the instances collected in the first step, to create the initial chromosomes. The process works, for each expression `expr` and positive (resp. negative) instance o, as follows:

- If `expr` denotes a singleton, and evaluating it in the positive (resp. negative) instance o results in value `v`, we create a size 1 chromosome containing the gene `expr = v` (resp. `expr != v`). For instance, if the expression is `thiz.header`, and the header in the positive (resp. negative) instance o is `N0`, we generate a single gene chromosome for expression `thiz.header = N0` (resp. `thiz.header != N0`).
- If the expression `expr` denotes a set of any cardinality, of a non-basic type, then a size 1 chromosome with the expression `all n: expr | n != null` is generated. Moreover, for every field `f` such that `expr.f` is a legal expression, a size 1 chromosome with the expression `all n: expr | n != n.f` is generated. For instance, from the expression `thiz.header.*next`, we generate chromosomes with the expressions `all n: thiz.header.*next | n != null`, and `all n: thiz.header.*next | n != n.next`.
- If the expression `expr` denotes a numeric value, then for every expression `expr'` denoting a set, we create a size 1 chromosome containing the formula `expr = #(expr')`. For instance, from our motivating example we will generate a chromosome containing the formula `thiz.size = #(thiz.header.*next)`.

Notice that only the first of the above items involves the initial (positive and negative) instances. The other two items only depend on the expressions generated from the type graph.

All our initial chromosomes are size 1 chromosomes. These will serve as the basic units from which to generate more complex candidate specifications, as our genetic operators, defined below, show.

3.3. Genetic operators

As mentioned previously, genetic operators are used to explore the search space by generating new individuals from a population. One way to do this is by combining parts of existing chromosomes through the *crossover* operator. 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. If both chromosomes have size 1, then their crossover is the union of their genes.

The other mechanism to generate new individuals in a genetic algorithm is by randomly changing characteristics of existing individuals, a process called *mutation*. Our genetic algorithm supports the following rich set of mutations:

- **Gene deletion**: it is the simplest mutation, and it can be applied to any gene. It changes the gene to `true`, and is equivalent to removing the gene from the chromosome.

  $$c_i = \left[\, g_0 \;\; \texttt{g1} \;\; g_2 \,\right]$$
  $$c_i' = \left[\, g_0 \;\; \texttt{true} \;\; g_2 \,\right]$$

- **Operator replacement**: this mutation replaces an operator by another. Relational equality and relational containment are replaced by their negated counterparts, and vice versa. Integer comparison operators such as =, != , <, >, <= or >=, are replaced by other operators in this set. Quantified formulas are also mutated by changing the quantifier, e.g., `all` is replaced by `some`, and vice versa.

  $$c_i = \left[\, g_0 \;\; \texttt{thiz.size = 2} \;\; g_2 \,\right]$$
  $$c_i' = \left[\, g_0 \;\; \texttt{thiz.size != 2} \;\; g_2 \,\right]$$

- **Expression extension**: it uses the type graph and the join operator to extend a navigational expression with a new field.

  $$c_i = \left[\, g_0 \;\; \texttt{all n : ...| n != n.left} \;\; g_2 \,\right]$$
  $$c_i' = \left[\, g_0 \;\; \texttt{all n : ...| n != n.left.right} \;\; g_2 \,\right]$$

- **Value insertion**: it replaces a sub-expression of the gene with a constant of the corresponding type.

  $$c_i = \left[\, g_0 \;\; \texttt{all n: ...| n != n.next} \;\; g_2 \,\right]$$
  $$c_i' = \left[\, g_0 \;\; \texttt{all n: ...| n != N0} \;\; g_2 \,\right]$$

- **Integer addition/substraction**: it applies to genes that involve integers. It simply adds or substracts a constant k to an integer expression.

  $$c_i = \left[\, g_0 \;\; \texttt{thiz.size = \#(thiz.header.*next)} \;\; g_2 \,\right]$$
  $$c_i' = \left[\, g_0 \;\; \texttt{thiz.size = \#(thiz.header.*next) - k} \;\; g_2 \,\right]$$

- **Closure insertion/deletion**: it inserts or removes closure operators from expressions, taking into account the expressions types and arities.

  $$c_i = \left[\, g_0 \;\; \texttt{thiz.header.next} \;\; g_2 \,\right]$$
  $$c_i' = \left[\, g_0 \;\; \texttt{thiz.header.*next} \;\; g_2 \,\right]$$

As usual in genetic algorithms, these mutations are applied to randomly picked genes of chromosomes in the current population.

3.4. Fitness of candidate specifications

Our fitness function applies to chromosomes representing candidate specifications, and is meant to assess how close are the corresponding candidates to the desired specification. Of course, we do not have the desired specification (it is what we are trying to build), so a direct comparison is impossible. However, we do have the operational specification Φ_{op}, so we can (indirectly) compare candidate specifications against this one.

In order to compute the fitness $f(c)$, we employ both Φ_{op} and the candidate solution c, whose fitness we want to assess. We take advantage of a semantics-preserving translation that given a *scope* (a bound in the number of iterations or nested recursive calls performed by Φ_{op}), can translate Φ_{op} into an equivalent, for the given scope, declarative specification Φ'_{op} in relational logic. We use a very small scope for the translation, since as we mentioned (and as we show in the evaluation section), the obtained specifications very quickly become impractical for analysis. We then first check for the satisfiability of formula:

$$\Phi'_{op} \wedge \neg \Phi_c$$

for the same scope used for the translation of Φ_{op}. Recall that Φ'_{op} results from the translation of Φ_{op} into relational logic, for a very small scope (we use 3 in our experiments). Φ_c represents the specification corresponding to candidate c (the conjunction of its genes). If the above formula is satisfiable, it means that there exist cases, within the small scope considered, that *should* be accepted by c, but are not. In such a case, we assign a fitness value 0 to c, i.e., $f(c) = 0$. If instead the above formula is unsatisfiable, we consider the following formula:

$$\neg \Phi'_{op} \wedge \Phi_c$$

and we enumerate the instances that, for the considered scope, satisfy this formula. Here, any enumeration procedure would fit the purpose (although different approaches may significantly vary in performance). We use a SAT-based approach that performs a *field-exhaustive* enumeration [41]. Intuitively, this enumeration skips structures that cover the same values for fields as previously generated structures, and produces more variability with fewer inputs (cf. [41]). We define $f(c)$ as follows:

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

where MAX is a constant larger than any possible number of negative cases (can be calculated as all possible assignments to fields, within the given scope); $\mathsf{neg}(c)$ is the number of cases that satisfy Φ_c and do not satisfy Φ'_{op} (the result of the above enumeration); and $\mathsf{len}(c)$ is the length of c, i.e., its number of non-trivial genes (genes that are not the constant `true`).

The rationale for this definition of the fitness function has to do with the fact that we attempt to over approximate (in the sense that the useful candidates are weaker than the specification we are searching for) the sought-for specification. This motivates also how we capture candidate specifications. 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. Of course, this is a secondary issue, and this is why it contributes a fraction to the fitness value, as opposed to the actual driving acceptance criterion, namely, the number of counterexamples approaching to zero.

It is worth mentioning that, despite the fact that we use a semantics-preserving translation to take, for a very small scope, the operational specification Φ_{op} into the declarative context, this is not actually a requirement of our approach. We may, as we did for a preliminary version of our technique [36], use the operational specification Φ_{op} to produce a set of *positive* and *negative* examples, i.e., instances that satisfy and do not satisfy Φ_{op}, respectively, and use these to evaluate the fitness of a candidate c; this approach simply evaluates how many of the positive and negative instances satisfy the specification Φ_c, and defines the fitness of c as described above. To generate the instances, any test input generation mechanism that requires an operational specification, e.g. [4,49], could be used.

3.5. *Selection*

The selection operation determines which individuals are to be kept in the next generation. Our selection operation focuses in two aspects. Firstly, it maintains a predefined amount of the fittest individuals. To do that, the individuals are sorted by decreasing order according to their fitness values, and then the top individuals are selected. Secondly, the selection also keeps all the individuals of size 1 that represent valid properties, i.e., those whose fitness is greater than zero (accept all the instances that satisfy Φ_{op}). This last selection policy allows the algorithm to maintain all the discovered valid properties, that may be part of the best individuals approximating the sought-for specification.

3.6. *Overall structure of the genetic algorithm for learning specifications*

The previously described elements are the constituting parts of our genetic algorithm. These are put together following the general structure of a genetic algorithm, namely: producing the initial population, and then iteratively selecting individuals to expand the population generating the offspring (using crossover and mutation), and discard some individuals to control population size (using the selection mechanism), until a maximum number of generations is reached, or a suitable individual is produced. The initial population is generated by producing size 1 chromosomes, covering combinations of the previously described expressions. Both the initial population and the succeeding ones are limited in size to 100 individuals.

As described above, the selection of chromosomes for crossover is based on a “fittest-first” policy. We select the fittest 10% and randomly pick pairs from these for crossover. For mutation, high-fitness chromosomes have a mutation probability of 0.3, and low-fitness chromosomes have a mutation probability of 0.6. Moreover, for high-fitness chromosomes that are solutions to the problem (i.e., those that have no counterexamples) the only possible mutation is gene deletion. This is due to the fact that, since we already have a solution, we only want to find a more concise equivalent expression by removing redundant parts of the invariant.

Finally, the algorithm stops after 20 evolutions, or generations, have been produced. Whenever a satisfying specification is generated (i.e., one that has no counterexamples), it is stored and the time measured, but the algorithm is not stopped, in an attempt to produce shorter (i.e., more concise) specifications.

The rationale behind our selection of the above values for the genetic algorithm's parameters (population size, number of generations, probability for crossover and mutations, etc.) is not arbitrary. We learned adequate values for these parameters from trial-and-error runs of our genetic algorithm, on a single case study, namely singly linked lists. Trial-and-error is a common mechanism used, in the context of evolutionary computation, to appropriately set parameters of the evolutionary search. It is important to remark that, while we selected these values based on experimentation, a single case study was involved in the experiments leading to parameter selection, and the same selected values were employed on all cases of our experimental validation. We have also performed some *a posteriori* analysis regarding how some of these parameters affect the performance of the algorithm, in particular the population size, and the mutation probability. These analyses are described as part of the next section.

4. Validation

In this section we perform an experimental assessment of our evolutionary approach to learning declarative specifications from operational ones. All experiments were run on a workstation with Intel Core i7 2600, 3.40 GHz, and 16 GB of RAM. The genetic algorithm has been implemented using JGAP [24], running on Java OpenJDK 1.7, on an Ubuntu 16.04 LTS x86_64 operating system. The first part of our evaluation analyzes how fast our algorithm is able to learn a declarative specification from an operational one. We do so for data structure invariants, on a number of data structure implementations with increasingly complex invariants. These are implementations of

- 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.

All these structures and their corresponding operational invariants have been taken from Korat's set of accompanying examples, or are simple variants of these. Our genetic algorithm implementation, as well as all case studies considered in this paper, can be found and reproduced following the details in https://sites.google.com/view/alloy-learning/.

For each case study, we ran the algorithm 10 times, with a limit of 20 generations (evolutions of the genetic algorithm population). For generating the chromosomes in the initial population, we only use 2 positive and 2 negative instances (and the expressions produced from the type graph, of course), obtained by translating the corresponding `repOK` into a relational logic predicate (using the semantics-preserving translation with a scope of 3), and querying for satisfiability of the obtained predicate and its negation. We report the minimum, maximum and average runs, indicating the number of generations g that were necessary, and the time t in seconds required for learning the corresponding invariant. We report the cost of computing the first invariant (the time and generations required to get a suitable invariant), and the cost of computing the "best" invariant (the algorithm continues running after an invariant has been found, to try to optimize it, e.g., making it more concise). These results are summarized in Table 1.

The second part of our assessment is concerned with evaluating the impact of different parameters, in the performance of our algorithm. Let us start with the generation of the initial population. Recall that, as described in the previous section, our genetic algorithm's initial population is composed of size 1 chromosomes only. In order to measure the impact of this decision, we analyzed two different versions of our algorithm, one that starts with randomly generated chromosomes of varying sizes (using the same approach to generate genes described in the previous section), and another version where the initial population is composed only of size 1 chromosomes. Tables 2 and 3 report the average costs of computing the first and "best" invariant using an initial population of chromosomes with varying sizes, and when using size 1 chromosomes as the initial population, (cf. Section 3.5), respectively. In both cases, the selection policy for the analysis is the "fittest first" policy, disregarding the size of the chromosomes.

Continuing with the analysis of the impact of different parameters on the performance of our genetic algorithm, we executed the algorithm with different parameter configurations and measured the corresponding *likelihood of optimality* $Lopt(k)$. The $Lopt(k)$ is defined as the estimated probability that the algorithm will reach an optimal solution in k generations. It is calculated from n executions of k generations of the algorithm, as m/n, where m is the number of runs that produced an optimal solution [47]. This measure allows us to analyze the effectiveness of the algorithm when selecting different parameter values. All the executions of this performance measure were performed in three case studies of different complexities,

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

Data Structure	First Invariant Found						Best Invariant Found					
	Min		Max		Avg		Min		Max		Avg	
	g	*t*	g	*t*	g	*t*	g	*t*	g	*t*	g	*t*
s. linked lists	1	*1*	1	*3*	1	*2*	1	*1*	3	*6*	2	*3*
s. linked sort. lists	1	*1*	3	*6*	1	*4*	2	*2*	7	*18*	4	*11*
s. circular lists	1	*2*	1	*3*	1	*2*	1	*2*	3	*4*	1	*3*
doubly linked lists	2	*5*	4	*19*	3	*11*	2	*7*	5	*39*	5	*23*
binary trees	1	*10*	1	*15*	1	*12*	3	*61*	9	*211*	7	*156*
binary search trees	1	*4*	2	*8*	1	*6*	4	*16*	7	*42*	4	*32*
heaps	1	*6*	3	*22*	2	*11*	2	*19*	8	*110*	6	*70*
binary DAGs	1	*3*	1	*5*	1	*5*	1	*3*	4	*25*	2	*16*
red-black trees	2	*10*	3	*17*	2	*14*	9	*71*	18	*406*	11	*234*

Table 2
Experimental Results corresponding to learning declarative invariants from operational ones, using "fittest first" selection policy and initial population of chromosomes of varying sizes.

Data Structure	First Invariant Found		Best Invariant Found		Times learned
	Avg		Avg		
	g	*t*	g	*t*	
s. linked lists	5	*10*	6	*12*	1/10
s. linked sort. lists	10	*18*	11	*21*	9/10
s. circular lists	1	*3*	2	*9*	10/10
doubly linked lists	3	*17*	6	*29*	10/10
binary trees	13	*128*	16	*311*	8/10
binary search trees	–	–	–	–	0/10
heaps	15	*139*	17	*186*	4/10
binary DAGs	8	*56*	9	*58*	5/10
red-black trees	–	–	–	–	0/10

Table 3
Experimental Results corresponding to learning declarative invariants from operational ones, using "fittest first" selection policy and initial population of size 1 chromosomes.

Data Structure	First Invariant Found		Best Invariant Found		Times learned
	Avg		Avg		
	g	*t*	g	*t*	
s. linked lists	1	*2*	3	*6*	10/10
s. linked sort. lists	2	*6*	5	*19*	10/10
s. circular lists	1	*2*	2	*7*	10/10
doubly linked lists	3	*10*	5	*24*	10/10
binary trees	2	*20*	8	*280*	10/10
binary search trees	1	*10*	5	*60*	10/10
heaps	2	*15*	7	*80*	10/10
binary DAGs	1	*6*	3	*19*	10/10
red-black trees	2	*29*	12	*301*	9/10

Table 4
Comparison of operational invariants vs our computed declarative invariants, verifying invariant preservation in bounded scenarios.

Data Structure	Rel.	Op.	Rel.	Op.	Rel.	Op.	Rel.	Op.
Scopes	**5**		**12**		**15**		**20**	
s. linked lists	$<$ 00:01	$<$ 00:01	00:01	00:03	00:07	00:10	01:46	01:38
s. linked sorted lists	$<$ 00:01	$<$ 00:01	00:30	01:54	03:25	10:16	21:51	TO
doubly linked lists	$<$ 00:01	$<$ 00:01	00:01	00:03	00:03	00:08	00:22	01:23
s. circular lists	$<$ 00:01	$<$ 00:01	00:02	00:04	00:10	00:22	01:37	02:18
Scopes	**5**		**7**		**8**		**9**	
binary trees	$<$ 00:01	00:01	00:01	01:05	00:10	28:06	01:25	TO
binary search trees	00:01	00:09	01:32	01:13	49:11	TO	TO	TO
heaps	00:01	00:03	00:48	02:45	01:54	49:52	06:54	TO
binary DAGs	$<$ 00:01	00:03	00:01	00:54	00:06	07:14	00:43	50:15
red-black trees	$<$ 00:01	00:01	00:01	01:40	00:13	36:22	01:16	TO

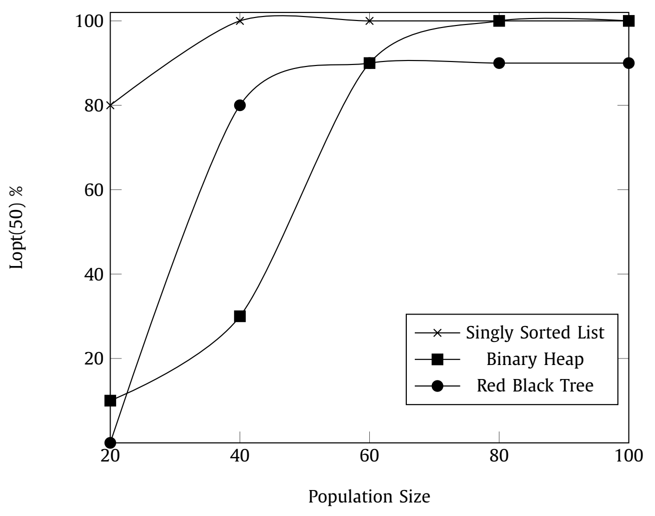

Fig. 6. Lopt(50) varying the population size.

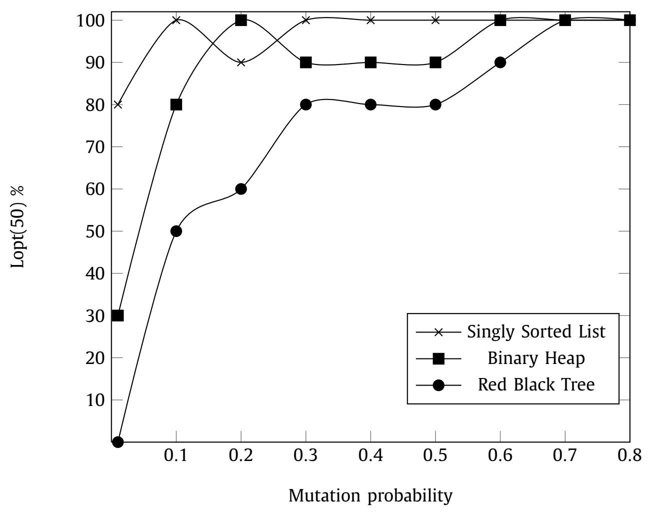

Fig. 7. Lopt(50) varying the mutation probability with a population size value of 100.

namely, Singly Sorted List, Binary Heap and Red Black Tree. Fig. 6 shows how *Lopt*(50) varies, for different population sizes, from 20 to 100. As expected, the effectiveness of the algorithm increases as the population size is greater, showing substantial effectiveness for a population size of about 60 (we used 100 in our experiments).

We also measured *Lopt*(50) when varying the mutation probability, with a fixed population size of 100 and also with a population size of 40. Fig. 7 shows the impact of different mutation probabilities with a population size of 100. As it can be seen from the table, in all the cases the effectiveness of the algorithm increases with higher mutation probabilities. While typically high mutation probabilities have a negative impact in the effectiveness of genetic algorithms, our algorithm performs rather well with high probabilities. This, in our opinion, has to do with the following. On one hand, our initial population starts with size one chromosomes; a high mutation probability allows us to be likely to explore many mutations of size one chromosomes, whose "valid" cases we maintain across generations. Secondly, our state space exploration is highly driven by mutation; crossover simply "builds" conjunctions, but is the mutation what generates the "conjuncts". Generating the conjuncts is what the initial population and mutation are concerned with; our "keep the valid conjuncts" policy is what enables crossover to build appropriate conjunctions.

The second part of our analysis evaluates our learned specifications in verification contexts. We first compare our approach with a semantics preserving translation from operational specifications into declarative ones, in a specific verification scenario. We verify, for increasingly larger scopes (i.e., maximum sizes of the corresponding structures), that the insertion routines of the data structures considered for analysis, preserve the corresponding structure's representation invariant. We use DynAlloy [13] for this task, using the original operational specification translated into relational logic as described in [13,14], and our learned declarative specification. Running times are reported in minutes:seconds, in Table 4 (we set a time-

Table 5
Comparison of our learned invariants with automatically inferred ones using Daikon.

Our approach	**Daikon**
s. linked lists	
(all n: thiz.header.*next | not (n in n.^ next)) and eq[#(thiz.header.*next - Null), thiz.size]	thiz.header != Null and gte[thiz.size,0]
s. linked sort. lists	
(all n: thiz.header.*next | not (n in n.^next)) eq[#(thiz.header.*next - Null),thiz.size] (all n: thiz.header.*next-Null | (n.next != Null) => lte[n.element,n.next.element])	thiz.header!=Null and gte[thiz.size,0] and eq[thiz.header.element,0]
s. circular lists	
(all n: thiz.header.*next | (n in n.^next)) and eq[#(thiz.header.*next), thiz.size]	thiz.header=Null and gte[thiz.size,0]
doubly linked lists	
(all n: thiz.header.*(next + prev) | n = n.prev.next) and eq[#(#(thiz.header.*(next + prev) - Null)), thiz.size]	this.header.prev = Null and gte[thiz.size,0]
binary trees	
(all n : thiz.root.*(left + right) | (n.left.*(left + right)) & (n.right.*(left + right)) in Null) and (eq[thiz.size,#(thiz.root.*(left + right) - Null)]) and (all n : thiz.root.*(left + right) | n !in n .^(left+right))	#(thiz.root^(left+right))>=0 and gte[thiz.size, 0] and (all n:Node | #(n.^(left+right)) >= 0) and (all n:Node | #(n.left.^(left+right)) >= 0) and (all n:Node | #(n.right.^(left+right)) >= 0) and (all n:Node | #(n.left.^(left+right)) <= #(n.^(left+right))) and (all n:Node | #(n.right.^(left+right)) <= #(n.^(left+right)))
binary search trees	
(eq[thiz.size,#(thiz.root.*(left + right) - Null)]) and (all n : thiz.root.*(left + right) | n !in n .^(left+right)) and (all n: thiz.root.*(left+right) | (all x: n.left.*(left+right) - Null| lt[x.element,n.element]) and (all x: n.right.*(left+right) - Null| gte[x.element,n.element]))	(all n:Node | n.left.element < n.element) and (all n:Node | n.right.element > n.element) and gte[thiz.size,0]
heaps	
(all n: thiz.root.*(left+right)| n !in n.^(left+right)) and eq[thiz.size, #(thiz.root.*(left + right) - Null)] and (all n : thiz.root.*(left+right) | n.left.*(left+right) & n.right.*(left+right) in Null) and (all n:thiz.root.*(left+right) | ((n.left!=Null) => gte[n.element,n.left.element]) and ((n.right!=Null) => gte[n.element,n.right.element]))	thiz.root^(left+right)>=0 and gte[thiz.size, 0] and (all n:Node | #(n.^(left+right)) >= 0) and (all n:Node | #(n.left.^(left+right)) >= 0) and (all n:Node | #(n.right.^(left+right)) >= 0) and (all n:Node | #(n.left.^(left+right)) <= #(n.^(left+right))) and (all n:Node | #(n.right.^(left+right)) <= #(n.^(left+right)))
binary DAGs	
(all n: thiz.root.*(left+right)-Null| n !in (n.^next)) and eq[thiz.size,#(thiz.root.*(left+right) - Null)]	-
red-black trees	
all n: thiz.root.*(left+right)| n !in n.^(left+right)) and eq[thiz.size, #(thiz.root.*(left+right)-Null)] and (thiz.root.color != Red) and (all n : thiz.root.*(left+right) | n.left.*(left+right) & n.right.*(left + right) in Null) and (all n : thiz.root.*(left+right)-Null | n.color=Red => ((n.left.color!=Red) and (n.right.color!=Red)))	(thiz.root.color = Black) and gte[thiz.size, 0]

out of 1 hour, and marked those analyses exceeding the timeout as TO). Notice that we used different scopes for different kinds of structures. In particular, linear data structures admit larger scopes for analysis, compared to tree-like structures.

As an additional analysis context, we consider *generalized symbolic execution* [27]. An important mechanism for extending the application of symbolic execution to programs manipulating heap-allocated data is *lazy initialization*, put forward in [27]. This mechanism can even profit from operational preconditions or invariants, to prune paths during symbolic execution analysis. More recently, in [17,42], the lazy initialization mechanism has been enhanced by the use of precomputed *tight bounds*. Essentially, during lazy initialization, a partially symbolic heap is incrementally concretized as a program is symbolically executed, concretizing symbolic data as soon as it is accessed by the program (hence the “lazy” adjective). During this concretization, all possible concrete values for a symbolic value are considered, leading to different paths in the symbolic execution. What pre-computed tight bounds provide is a *reduction* in the number of cases to consider, based on how the data is constrained. For instance, if a list is assumed to be acyclic, and we are concretizing the “next” pointer of a given node in the list, lazy initialization would try all possible concretizations, including those that point to previous nodes; tight bounds reduce these cases to only two possible ones: it can either be null, or point to a new node (with all its attributes symbolic), since these are the only two cases that do not violate acyclicity). This improves the performance of

Table 6
Comparison of our evolutionary approach with Deryaft, in generating data structure invariants.

Properties	Our approach	Deryaft
s. linked lists		
Acyclicity	✓	✓
Size consistency	✓	✗
s. linked sort. lists		
Acyclicity	✓	✓
Size consistency	✓	✗
Order	✓	✓
s. circular lists		
Cyclicity	✓	✗
Size consistency	✓	✗
doubly linked lists		
Next-Previous relation	✓	✗
Size consistency	✓	✗
binary trees		
Acyclicity	✓	✓
Disjoint subtrees	✓	✓
Size consistency	✓	✗
binary search trees		
Acyclicity	✓	✓
Order	✓	✓(incomplete)
Size consistency	✓	✗
heaps		
Acyclicity	✓	✓
Disjoint subtrees	✓	✓
Order	✓	✓
Size consistency	✓	✗
binary DAGs		
Acyclicity	✓	✓(raises an error)
Size consistency	✓	✗
red-black trees		
Acyclicity	✓	✓
Root color	✓	✗
Disjoint subtrees	✓	✓
Color rule	✓	✗
Black height	✗	✗
Size consistency	✓	✗

Table 7
Analysis time for class BinTree.

Method	Technique	S04	S05	S06	S07	S08	S09	S10	S11	S12
bfs	LI	**00:01**	**00:01**	**00:02**	00:03	00:07	00:18	01:02	04:23	15:06
	BLI	**00:01**	**00:01**	**00:02**	00:03	00:05	00:14	00:48	03:08	11:10
	RBLI	**00:01**	**00:01**	**00:02**	**00:02**	**00:04**	**00:08**	**00:21**	**01:09**	**03:38**
dfs	LI	**00:01**	**00:01**	00:02	00:03	00:05	00:16	00:56	03:43	14:38
	BLI	**00:01**	**00:01**	00:02	**00:02**	00:05	00:12	00:43	02:31	11:15
	RBLI	**00:01**	**00:01**	**00:01**	**00:02**	**00:04**	**00:10**	**00:35**	**02:12**	**08:55**
repOK	LI	**00:01**	**00:02**	**00:03**	**00:06**	00:17	00:54	03:09	11:47	41:35
	BLI	**00:01**	**00:02**	**00:03**	**00:06**	00:16	00:51	02:59	10:12	37:04
	RBLI	**00:01**	**00:02**	**00:03**	**00:06**	**00:15**	**00:44**	**02:31**	**08:11**	**29:31**

lazy initialization, but with an additional cost: bounds are computed from *declarative* specifications, and thus the developer needs to provide two versions of the same specification: the operational one typical of the symbolic execution context, and the declarative one used for computing the tight bounds. We then compare standard lazy initialization, with lazy initialization aided by tight bounds (with two different techniques put forward in [42]), where the required declarative specification is automatically learned from the operational specification, using our evolutionary technique. Tables 7 and 8 compare the symbolic execution analysis times of standard lazy initialization (LI), with two techniques that use tight bounds computed with our learned specifications, bounded lazy initialization (BLI) and refined bounded lazy initialization (RBLI, which reduces cases, based on the bounds, as the partially symbolic heap gets concretized), for various routines of two data structures, binary trees and red-black trees. Again, times are given in minutes:seconds, experiments were run with a set timeout of 1

Table 8
Analysis time for class TreeSet.

Method	Technique	S04	S05	S06	S07	S08	S09	S10	S11	S12
bfs	LI	**00:01**	**00:02**	**00:04**	00:09	00:29	01:44	06:32	23:51	1:27:29
	BLI	**00:01**	**00:02**	**00:04**	00:09	00:28	00:40	05:39	23:09	1:21:16
	RBLI	**00:01**	**00:02**	**00:04**	**00:08**	**00:26**	**01:26**	**05:05**	**19:20**	**1:10:02**
dfs	LI	**00:01**	**00:01**	**00:02**	00:03	00:07	00:21	01:20	05:20	21:44
	BLI	**00:01**	**00:01**	**00:02**	00:03	**00:05**	00:16	00:57	04:14	15:53
	RBLI	**00:01**	**00:01**	**00:02**	**00:02**	**00:05**	**00:13**	**00:52**	**03:09**	**12:42**
repOK	LI	**00:03**	00:13	01:14	07:44	50:54	TO			
	BLI	**00:03**	**00:12**	01:13	07:35	51:13	TO			
	RBLI	**00:03**	**00:12**	**01:09**	**07:24**	**48:36**	TO			

hour, and those cases exceeding the timeout are marked as TO. These results show that, by employing our learned specifications to compute tight bounds, symbolic execution over these data structures is improved, by some significant margin in some cases.

Finally, we analyze the precision of the obtained invariants, in comparison with related techniques. We compare our learned invariants with automatically inferred ones using Daikon [12] and Deryaft [34]. Daikon computes likely invariants from run-time information, and thus requires tests to exercise the program under analysis, and perform the inference. We fed Daikon with randomly produced tests, computed using Randoop [38]. Derayft is a tool for generating constraints of complex data structures from instances; it takes a handful of concrete data structures and generates a predicate that represents the invariant. Since Daikon computes invariants for all involved classes, when an invariant refers to an auxiliary class, e.g., `Node`, we report the inferred invariant as being a property of all nodes of the structure. Invariants inferred by Daikon are JML expressions. We show these as relational logic expressions for easier comparison. The comparison with the invariants obtained with Daikon is summarized in Table 5. In the case of Deryaft, the output is a Java predicate that represents the invariant, i.e., a boolean routine that takes an input structure and returns true if and only if it satisfies the invariant. While such output is *operational*, the produced (likely) invariant is composed of a set of “catalogue” properties (e.g., acyclicity, sortedness, etc.), which have known declarative counterparts. Table 6 compares the generated properties for each case study using our approach, with the properties generated by Deryaft for the same data structures.

4.1. Assessment

Let us now evaluate our experimental results. First, consider the running times for our genetic algorithm. For most structures and in most runs, we are able to compute invariants in a few seconds. Our most complex data structure considered, red-black trees, takes in some cases a few minutes (about 2.5 minutes in the worst case) to compute an invariant. In general, our algorithm runs very efficiently.

Regarding the efficiency of our computed invariants as opposed to the operational ones for bounded verification, our declarative invariants show a substantial profit in analysis, with the sole exception of our simplest case study, singly linked lists. In this case study, and for our largest considered scope, the invariant obtained by a semantics-preserving translation from the operational one is actually better than the one produced with our approach, in verification time (although very slightly). In all other cases, verification with our produced invariants outperforms verification with those directly translated from the operational ones. Notice that learning pays off exceedingly, comparing the time taken in learning and the speed up achieved when replacing the “translated” invariant with the “learned” one. In the context of symbolic execution, the improvement provided by tight bounds computed with our learned invariants over lazy initialization is also significant in some cases. For instance, for `bfs` from `BinTree`, RBLI (that uses bounds computed with the learned invariant) is more than 4 times faster than LI for scope 12. In other cases the margin is smaller, but it is worth mentioning that the bounds computation is amortized across all methods from the same class, and thus learning the declarative invariant and computing the bounds pays off.

Of course, neither of the first two parts of our analysis is meaningful if our invariants are imprecise. Our third part of the analysis confirms that our learned invariants are rather precise, compared to the expected outcome. Indeed, in all cases except red-black trees, we learn an invariant that is actually *equivalent* to the `repOK`. In order to check equivalence, besides manually inspecting the obtained invariants, we bounded-exhaustively enumerated instances satisfying `repOK` using Korat, for various selected bounds, and compared the number of obtained instances with the number of bounded instances satisfying our obtained Alloy specification, for the corresponding bounds. In the case of red-black trees, we are able to learn most of the expected invariant, except for the “black height” portion of it. This part of the invariant states that *“the number of black nodes in all paths from the root to a leaf is the same”*. Such constraint is not expressible with the expressions that our genetic algorithm considers, and thus constitutes a limitation of our approach.

In relation to the alternative mechanisms to generate invariants that we considered for comparison, if we compare with Daikon, our approach computes more precise specifications. Indeed, as our third table shows, Daikon is able to compute weaker invariants, sometimes erroneous ones, resulting from properties that consistently hold for the tests used for inference, but are not true in the general case, compared to our computed specifications. For instance, for doubly linked lists and binary search trees, Daikon computes some likely invariants that fail to hold in the general case; as a sample case, property `this.header.previous == this.last.next`, that Daikon computes, only holds for non-empty doubly linked lists (the implementation is acyclic, without sentinel). Regarding Deryaft, although the tool showed some precision in a number of case studies, it failed to produce a suitable invariant in several cases. This precision problem, in our opinion, has to do with the fact that Deryaft tries to build an invariant only from *positive* examples, as opposed to our case. In our experiments, we provided Deryaft the same set of positive examples that we used to compute the fitness in our algorithm, which are generated from the operational specification ϕ_{op} (the input of our algorithm).

5. Related work

Translating between formal languages has a long tradition both in Logic and in Computer Science. There exist translations and mappings between logical systems that have been used for automated analysis purposes, as well as for complexity and decidability arguments (see, e.g., [7]). This kind of approach has been borrowed by formal methods, in particular heavyweight ones, whose associated analysis mechanism is in general deductive verification, with the aim of using a proof system for a given formalism to reason about specifications in a different one (see, e.g., [3]). In general, the emphasis has been in sound, many times partial, syntactic mechanisms to define semantics-preserving translations, that enable *conservative* analyses of the source specifications in the target formalism. With the advent of lightweight formal methods, the conservativeness requirement can sometimes be dropped, as is the case e.g., with the (incomplete) SAT-based checking of Alloy specifications [22]. In these works the use of imprecise search based techniques such as the one presented in this paper is not observed, as far as we are aware of. However, learning techniques associated with formal specification has been applied in the past. Some examples are the use of the L* algorithm to assist assume-guarantee reasoning [39] and the inference of loop invariants through a combination of mutation (as in genetic programming) and static checking [16]. The first attempts to learn specifications of a routine from calls it receives from the environment, while the second applies specifically to loop invariants, thus differing from our presented work. Other related works are, of course, the preliminary version of our technique presented in [36], the works on specification inference as put forward in techniques like Daikon [12] and Deryaft [34] (that we compared with in this paper), and our recent work in [37]. As opposed to the work presented in this paper, the approach in [37], while it also uses a genetic algorithm, is less powerful since it only learns invariants composed of properties from a given catalog, in the style of [34] but considering both positive and negative instances. Our current approach is thus more general, and it does not require a provided catalog.

Model synthesis is also an active line of research related to our work. In the general case, synthesis techniques assume a specification, and work on synthesizing operational models that satisfy it (cf. [48,11,29,9]), thus working on a different direction compared to our presented work.

6. Conclusions and future work

The increasing availability of automated technologies based on formal methods is evidencing a lack of formal specifications accompanying software systems, while at the same time contributes to showing their necessity. Indeed, many tools for program analysis, including run time assertion checkers, and static analysis tools for verification, fault localization, test generation and bug finding, require formal specifications. In this paper, we argued about the fact that, even in cases in which one has a formal specification available, many times this specification is unsuitable for the kind of analysis, tool or technique, one is interested in. We studied this situation in the particular case in which an operational specification, represented as code, is available, but one requires such specification to be provided in a logical setting. We proposed an evolutionary algorithm that produces such declarative specifications from operational ones, and showed that, for a benchmark composed of data structures of varying complexities, the algorithm is able to learn adequate declarative representation invariants, from their operational counterparts. Moreover, we showed that these learned invariants are better suited for analysis, in particular bounded verification, than performing an existing semantics preserving translation of the operational ones and using those for the same analysis. We also showed that the learned invariants can contribute to improving other automated analysis settings, in particular lazy initialization, and that our algorithm produces, for the analyzed case studies, specifications that are significantly more precise than those generated by related specification inference tools.

The presented work opens several lines for future work. As we explained in the paper, we have concentrated on properties of linked structures, and the whole design of our algorithm and the expressions it supports makes it infeasible to learn some relevant properties (the black height invariant for red-black trees is an example of this situation illustrated in the paper). An obvious line of research is work on a generalization of our approach, to enable learning a richer set of specifications. Our case studies are so far limited to data structure representation invariants, so analyzing our approach on other kinds of programs, is also part of our plans. In particular, in attempting to learn specifications from larger programs we will come into scalability issues, that will need to be tackled. Finally, our operational-to-declarative approach enables interconnecting analysis techniques and tools, some of which we have mentioned in the paper. We plan to take advantage of our evolutionary algorithm to implement such tool cross usages.

Acknowledgements

Renzo Degiovanni is on leave from the Departments of Computer Science and Mathematics, University of Río Cuarto, Argentina. This work was partially supported by Argentina's National Agency for Scientific and Technological Promotion (ANPCyT) through grants 2015-0586, 2016-1384, 2017-1979, 2017-2622; and by the Agence Nationale de la Recherche (France) INTER/ANR/18/12632675/SATOCROSS project.

References

[1] S. Artzi, J. Dolby, F. Tip, M. Pistoia, Directed test generation for effective fault localization, in: Proceedings of the Nineteenth International Symposium on Software Testing and Analysis ISSTA 2010, ACM, 2010.
[2] J. Berdine, C. Calcagno, P. O'Hearn, Symbolic execution with separation logic, in: Proceedings of the Third Asian Conference on Programming Languages and Systems, APLAS 2005, Springer, 2005.
[3] J. Bicarregui, M. Bishop, T. Dimitrakos, K. Lano, T. Maibaum, B. Matthews, B. Ritchie, Supporting Co-use of VDM and B by translation, in: Proceedings of VDM in 2000! (2nd VDM Workshop), 2000.
[4] C. Boyapati, S. Khurshid, D. Marinov, Korat: automated testing based on Java predicates, in: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2002, ACM, 2002.
[5] L. Burdy, Y. Cheon, D.R. Cok, M.D. Ernst, J.R. Kiniry, G.T. Leavens, K. Rustan, M. Leino, E. Poll, An overview of JML tools and applications, Int. J. Softw. Tools Technol. Transf. 7 (3) (2005), Springer.
[6] A. Çelik, S. Pai, S. Khurshid, M. Gligoric, Bounded exhaustive test-input generation on GPUs, in: PACMPL 1 (OOPSLA), ACM, 2017.
[7] Sjoerd Cranen, Jan Friso Groote, Michel Reniers, A linear translation from CTL* to the first-order modal mu-calculus, Theor. Comput. Sci. 412 (28) (2011), Elsevier.
[8] G. Dennis, F. Chang, D. Jackson, 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, ACM, 2006.
[9] Ramiro Demasi, Pablo F. Castro, T.S.E. Maibaum, Nazareno Aguirre, Synthesizing masking fault-tolerant systems from deontic specifications, in: Proceedings of International Symposium on Automated Technology for Verification and Analysis, ATVA 2013, in: LNCS, Springer, 2013.
[10] Julian Dolby, Mandana Vaziri, Frank Tip, Finding bugs efficiently with a SAT solver, in: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Dubrovnik, Croatia, 2007.
[11] E. Allen Emerson, Roopsha Samanta, An algorithmic framework for synthesis of concurrent programs, in: Proceedings of International Symposium on Automated Technology for Verification and Analysis, ATVA 2011, in: LNCS, Springer, 2011.
[12] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, Chen Xiao, The Daikon system for dynamic detection of likely invariants, Sci. Comput. Program. 69 (1–3) (2007), Elsevier.
[13] Marcelo F. Frias, Juan P. Galeotti, Carlos López Pombo, Nazareno Aguirre, DynAlloy: upgrading alloy with actions, in: Proceedings of International Conference on Software Engineering, ICSE 2005, ACM, 2015.
[14] J.P. Galeotti, N. Rosner, C. López Pombo, M.F. Frias, Analysis of invariants for efficient bounded verification, in: Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA 2010, ACM, 2010.
[15] J.P. Galeotti, N. Rosner, C. López Pombo, M. Frias, TACO: Efficient SAT-based bounded verification using symmetry breaking and tight bounds, IEEE Trans. Softw. Eng. 39 (9) (2013), IEEE.
[16] Juan P. Galeotti, Carlo A. Furia, Eva May, Gordon Fraser, Andreas Zeller, Inferring loop invariants by mutation, dynamic analysis, and static checking, IEEE Trans. Softw. Eng. 41 (10) (2015), IEEE.
[17] J. Geldenhuys, N. Aguirre, M.F. Frias, W. Visser, Bounded lazy initialization, in: Proc. of NFM 2013, in: LNCS, vol. 7871, Springer, 2013.
[18] C. Ghezzi, M. Jazayeri, D. Mandiroli, Fundamentals of Software Engineering, second edition, Prentice-Hall, 2003.
[19] D. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley, 1989.
[20] D. Gopinath, M.Z. Malik, S. Khurshid, Specification-based program repair using SAT, in: Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2011, in: LNCS, Springer, 2011.
[21] C. Gordon, M. Ernst, D. Grossman, M. Parkinson, Verifying invariants of lock-free data structures with rely-guarantee and refinement types, ACM Trans. Program. Lang. Syst. 39 (3) (2017), ACM.
[22] D. Jackson, Software Abstractions: Logic, Language, and Analysis, MIT Press, 2006.
[23] D. Jackson, M. Vaziri, Finding bugs with a constraint solver, in: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2000, ACM, 2000.
[24] Web site of the Java Genetic Algorithms Package (JGAP), http://jgap.sourceforge.net.
[25] J. Jones, M.J. Harrold, J. Stasko, Visualization of test information to assist fault localization, in: Proceedings of the International Conference on Software Engineering ICSE 2002, Orlando, Florida, ACM, 2002, pp. 467–477.
[26] S. Khalek, G. Yang, L. Zhang, D. Marinov, S. Khurshid, TestEra: a tool for testing Java programs using alloy specifications, in: ASE 2011, IEEE, 2011.
[27] S. Khurshid, C. Pasareanu, W. Visser, Generalized symbolic execution for model checking and testing, in: Proceedings of TACAS 2003, in: LNCS, vol. 2619, Springer, 2003.
[28] S. Khurshid, D. Marinov, TestEra: specification-Based testing of Java programs using SAT, Autom. Softw. Eng. 11 (4) (2004), Springer.
[29] Uri Klein, Nir Piterman, Amir Pnueli, Effective synthesis of asynchronous systems from GR(1) specifications, in: Proceedings of the 13th International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI 2012, in: LNCS, Springer, 2012.
[30] Home Page of the Korat test generation tool, http://korat.sourceforge.net.
[31] D. Kroening, M. Tautschnig, CBMC – C bounded model checker, in: Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2014, in: LNCS, vol. 8413, Springer, 2014.
[32] B. Liskov, J. Guttag, Program Development in Java: Abstraction, Specification, and Object-Oriented Design, Addison-Wesley, 2000.
[33] Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O'Neil Meredith, T. Florin Serbanuta, G. Rosu, RV-monitor: efficient parametric runtime verification with simultaneous properties, in: Proceedings of the 14th International Conference on Runtime Verification, RV 2014, in: LNCS, Springer, 2014.
[34] Muhammad Zubair Malik, Aman Pervaiz, Sarfraz Khurshid, Generating representation invariants of structurally complex data, in: Proceedings of the 11th Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS, 2007.
[35] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, Springer, 1996.
[36] F. Molina, C. Cornejo, R. Degiovanni, G. Regis, P. Castro, N. Aguirre, M. Frias, An evolutionary approach to translate operational specifications into declarative specifications, in: Proceedings of 19th Brazilian Symposium on Formal Methods, SBMF 2016, in: LNCS, Springer, 2016.
[37] F. Molina, R. Degiovanni, G. Regis, P. Castro, N. Aguirre, M. Frias, From operational to declarative specifications using a genetic algorithm, in: Proceedings of 11th International Workshop on Search-Based Software Testing, SBST 2018, ACM, 2018.

[38] C. Pacheco, S.K. Lahiri, M.D. Ernst, T. Ball, Feedback-directed random test generation, in: Proceedings of the 29th International Conference on Software Engineering, ICSE 2007, IEEE, 2007.
[39] Corina S. Pasareanu, Dimitra Giannakopoulou, Mihaela Gheorghiu Bobaru, Jamieson M. Cobleigh, Howard Barringer, Learning to divide and conquer: applying the L^* algorithm to automate assume-guarantee reasoning, Form. Methods Syst. Des. 32 (3) (2008), Springer.
[40] Y. Pei, C. Furia, M. Nordio, Y. Wei, B. Meyer, A. Zeller, Automated fixing of programs with contracts, IEEE Trans. Softw. Eng. 40 (5) (2014), IEEE.
[41] P. Ponzio, N. Aguirre, M. Frias, W. Visser, Field-exhaustive testing, in: Proceedings of International Symposium on the Foundations of Software Engineering, FSE 2016, Seattle (WA), USA, ACM, 2016.
[42] N. Rosner, J. Geldenhuys, N. Aguirre, W. Visser, M. Frias, BLISS: improved symbolic execution by bounded lazy initialization with SAT support, IEEE Trans. Softw. Eng. (2015).
[43] Fan Long, Martin Rinard, Staged program repair with condition synthesis, in: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, FSE 2015, Bergamo, Italy, 2015.
[44] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, second edition, Prentice-Hall, 2003.
[45] V. Senni, F. Fioravanti, Generation of test data structures using constraint logic programming, in: Proceedings of the 6th International Conference on Tests and Proofs, TAP 2012, in: LNCS, Springer, 2012.
[46] K. Shchekotykhin, T. Schmitz, D. Jannach, Efficient sequential model-based fault-localization with partial diagnoses, in: Proceedings of the 25th International Joint Conference on Artificial Intelligence, IJCAI 2016, IJCAI/AAAI Press, 2016.
[47] K. Sugihara, Measures for performance evaluation of genetic algorithms, in: Proceedings of the 3rd Joint Conference on Information Sciences, JCIS '97, vol. I, 1997, pp. 172–175.
[48] Sebastian Uchitel, Greg Brunet, Marsha Chechik, Synthesis of partial behavior models from properties and scenarios, IEEE Trans. Softw. Eng. 35 (3) (2009), IEEE.
[49] W. Visser, C.S. Pasareanu, S. Khurshid, Test input generation with Java PathFinder, in: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis ISSTA 2004, ACM, 2004.
[50] Yichen Xie, Alex Aiken, Saturn: a SAT-based tool for bug detection, in: Proceedings of the 17th International Conference on Computer Aided Verification, Edinburgh, Scotland, UK, 2005.