An Evolutionary Approach to Translate Operational Specifications into Declarative Specifications

Facundo Molina[1], César Cornejo[1], Renzo Degiovanni[1,3], Germán Regis[1], Pablo F. Castro[1,3], Nazareno Aguirre[1,3], and Marcelo F. Frias[2,3]

[1] Department of Computer Science, FCEFQyN,
National University of Río Cuarto, Argentina,
{fmolina, ccornejo, rdegiovanni}@dc.exa.unrc.edu.ar
{gregis, pcastro, naguirre}@dc.exa.unrc.edu.ar
[2] Department of Software Engineering,
Buenos Aires Institute of Technology (ITBA), Argentina,
mfrias@itba.edu.ar
[3] National Council for Scientific and Technical Research (CONICET), Argentina

Abstract. 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, that show that translating representation invariants using our approach and verifying invariant preservation using the resulting specifications outperforms verification with specifications obtained using an existing semantics-preserving translation. Also, our evolutionary computation translation achieves very good precision in this context.

1 Introduction

Many software validation and verification activities, both formal and informal, require a description of the software under analysis, since many analyses typically consist in checking compliance of the software against some prescribed intended behavior [12]. In the last few decades, formal specifications have gained an important notoriety 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.

Among the broad variety of formal notations, some styles or specification paradigms can be identified. For instance, in the context of program specification via pre- and postconditions, representation invariants, and the like, two distinguishing styles are 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 [19]. On the other hand, the declarative style often uses a logical formalism for expressing the same kind of property. A well-established approach is based on using a first-order logic complemented with closure operators, as put forward by notations such as JML [3] and Alloy's relational logic [14].

A problem that arises with the proliferation of notations and, more importantly, with the above described different specification styles, is that different tools adopt different styles, and provide optimizations and enhancements that only become available for such particular notations or styles. For instance, the test generation tool Korat [2] requires a specification to be provided operationally (as a `repOK` routine) to automatically produce test inputs; it implements "perfect" symmetry-breaking and search pruning techniques that are particularly tied to such representation, and thus makes it very difficult (and ineffective) to generate tests for, say, an object-oriented program equipped with a JML contract. On the other hand, tools for verification based on declarative notations, e.g., TACO [10], can exploit mechanisms such as tight bounds [9], 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 combines automated analysis technologies. This problem leads to a clear demand to be able to translate specifications across different styles and notations.

Notice that even when semantics-preserving translations are available between different formalisms, in many cases these 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., in 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. Thus, in some cases the existence of semantics-preserving syntax-guided translations are still unsatisfactory.

In this paper, we deal with a particular instance of the above described situation, 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. While there exists a semantics preserving translation from one to the other, we show that the resulting specifications are inadequate for analysis. We then propose an evolutionary approach to produce relational logic specifications from imperative ones, based on a genetic algorithm especially designed for this purpose. We evaluate our approach on a benchmark of data structure implementations, translating their corresponding representation invariants for verification. As our experiments show, translating specifications using our approach and verifying invariant preservation using the resulting specifications outperforms invariant preservation verification directly with specifications obtained using the semantics-preserving translation, and our evolutionary computation translation achieves very good precision in this context.

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, on a benchmark composed of various data structure implementations. 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 Figure 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 [19], 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 method, named `repOK()` as is 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 Figure 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

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

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

Fig. 1. Java classes defining singly linked lists.

```
public boolean repOK() {
        Set<Entry> visited = new java.util.HashSet<Entry>();
        visited.add(header);
        Entry current = header;
        while (true) {
            Entry 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.

seminal work of Hoare and Floyd, where first-order logic is used to express assertions regarding program states, to more modern languages such as JML [3] and Alloy [14], 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 exactly the same property as method `repOK()` in Figure 2, is shown in Figure 3. Notice how reflexive-transitive and transitive closures (denoted by operators $*$ and $\hat{}$, respectively) are employed to capture reachability and acyclicity.

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 (in fact, this particular example is taken from Korat's set of case studies [17]), if we want to perform bounded verification using a tool like TACO [9, 10], 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

```
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 | n !in n.^next) and
    (# thiz.header.*next = thiz.size)
}
```

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

bounded contexts), e.g., using the translations embedded in tools like TACO [9, 10] and CBMC [18]. The logical specification resulting from the translation of the `repOK()` method shown in Figure 2 is shown in Figure 4. This specification, 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 Figure 2, whereas it takes 1648 seconds when using the invariant in Fig. 3. As we will show later on in this paper, such difference in efficiency becomes more notorious in more complex data structure invariants (see the Validation Section).

The above described problem is the motivation for our approach. As we explain in the following section, we will develop 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 [13] are non-exhaustive guided search algorithms, based on a hill climbing strategy [24]. The search space is composed of a generally very large set of individuals (the candidates), and the search objective is to find an individual with sought-for features. As opposed to classic

```
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.

search algorithms, genetic algorithms maintain a *set* of individuals, called the population, and search progresses by iteratively selecting a number of individuals in the population, using these for evolution (building new individuals out of these), and leaving out some individuals of the whole set (the "old" ones and the "new" ones). Selection of individuals for population evolution, as well as individuals' removal, are guided by a *fitness function*, the heuristic function used to guide the search. This function applies to individuals, and its result is generalizable to the population too (e.g., the fitness of the 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., algorithm stops 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 arbitrarily *changing* a characteristic of a given individual. These two forms of

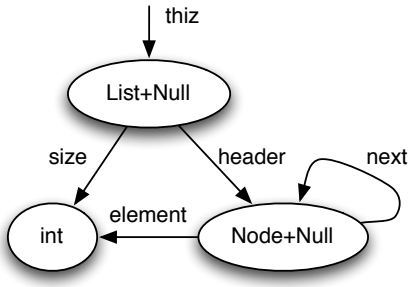

Fig. 5. Type graph for singly linked lists.

evolution 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 [20].

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:

- 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`) [14].

Chromosomes are simply vectors of the previously described genes, and represent *conjunctions* of the corresponding genes. As opposed to what is common in genetic algorithms, our chromosomes have varying lengths, and genes' positions are disregarded (i.e., 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). Genes' positions do play a role in crossover; we use one-point crossover to build new chromosomes, by randomly selecting points 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.

Our genetic algorithm has a very rich set of mutations. The simplest changes a randomly picked gene to `true` (equivalent to removing the gene). The others include changing an operator by another (e.g., `=` replaced by `!=`), changing a quantifier (e.g., `all n: ...` changed into `some n: ...`), adding or substracting from an integer in an expression (e.g., changing `#thiz.header.*next` by `#thiz.header.*next+1` in an expression), and inserting/removing closure operators from expressions (e.g., changing `thiz.header.next` to `thiz.header.*next` and vice versa).

3.2 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 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, 26]. We use an ad hoc variant of Korat, that generates inputs using a field-exhaustive approach [23]. Intuitively, this generation skips structures that cover the same values for fields than previously generated structures, and produces more variability with fewer inputs (cf. [23]). 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 (conjunction of its genes), and evaluate whether the positive and negative cases (recall that these are positive or negative according to Φ_{op}) 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:

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

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 $\mathsf{len}(c)$ 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 to 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.

3.3 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 select individuals for evolution (crossover/mutation), produce the ampled population, and discard some individuals to control population size, until a maximum number of evolutions 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.

The selection of chromosomes for crossover and mutation is based on a "fittest-first" policy. We select the fittest 10% for crossover and mutation, and randomly pick pairs from these for crossover; a small proportion of these, less than 10% (i.e., about 1% of the size of the population), are selected for mutation.

Finally, the algorithm stops after 20 evolutions, or generations, have been produced. Whenever a satisfying specification is generated (i.e., one that thas 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, percentage of individuals used for evolution, 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.

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 [15], 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;
- binary 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. For each case study, we ran the algorithm 10 times, with a limit of 20 generations (evolutions of the genetic algorithm population). We report the minimum,

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	0	*1*	2	*8*	1	*4*	0	*1*	2	*10*	1	*4*
s. linked sort. lists	1	*10*	4	*27*	2	*15*	2	*13*	5	*35*	3	*23*
s. circular lists	0	*1*	1	*7*	0	*3*	0	*1*	2	*11*	0	*3*
binary trees	1	*10*	4	*31*	2	*18*	1	*10*	4	*31*	2	*19*
heaps	2	*27*	7	*73*	4	*44*	2	*27*	11	*105*	5	*55*
binary DAGs	0	*2*	2	*15*	1	*7*	0	*2*	2	*15*	1	*7*
red-black trees	4	*56*	8	*112*	6	*85*	4	*82*	12	*165*	8	*119*

maximum and average runs, indicating the number of generations that were necessary, and the time 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 the experiments compares our approach with a semantics preserving translation from operational specifications into declarative ones, in verification scenarios. More precisely, we verify, for increasingly larger scopes (i.e., maximum sizes of the corresponding structures), that the insertion routine of the corresponding structure preserves the structure's representation invariant. We use DynAlloy [8] for this task, using the original operational specification translated into relational logic as described in [8, 9], and our learned declarative specification. Running times are reported in minutes:seconds, in Table 2. 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.

Finally, we analyze the precision of the obtained invariants. We compare our learned invariants with automatically inferred ones using Daikon [7]. 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 [21]. 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 obtained invariants are summarized in Table 3.

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

Table 2. Comparison of operational invariants vs our computed declarative invariants, verifying invariant preservation in bounded scenarios.

Data Structure	Rel.Spec.	Op.Spec.	Rel.Spec.	Op.Spec.	Rel.Spec.	Op.Spec.	Rel.Spec.	Op.Spec.
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
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
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

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, 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 operational invariant is actually better than the declarative one, in verification time (although very slightly). In all other cases, verification with the declarative invariant outperforms verification with the operational one. Notice that learning pays off exceedingly, comparing the time taken in learning and the speed up achieved when replacing the operational invariant with the declarative one.

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 mechanism to learn invariants that we considered for comparison, namely the Daikon approach, 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.

Table 3. 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 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]
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)))
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 (thiz.size >= 0)

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., [4]). 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., [1]). 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 [14]. 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 [22] and the inference of loop invariants through a combination of mutation (as in genetic programming) and static checking [11]. 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. 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. [25, 6, 16, 5]), 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 perfoming an existing semantics preserving translation of the operational ones and using those for the same analysis. We also showed 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 color 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

The authors would like to thank the anonymous referees for their helpful comments. This work was partially supported by the Argentinian Agency for Scientific and Technological Promotion (ANPCyT), through grants PICT 2012 No. 1298, PICT 2013 No. 2624 and PICT 2013 No. 0080.

References

1. 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.
2. C. Boyapati, S. Khurshid and 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.
3. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. Rustan M. Leino and E. Poll, *An overview of JML tools and applications*, in STTT 7(3), Springer, 2005.
4. Sjoerd Cranen, Jan Friso Groote and Michel Reniers, *A linear translation from CTL* to the first-order modal mu-calculus*, Theoretical Computer Science Volume 412, Issue 28, Elsevier, 2011.
5. 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, LNCS, Springer, 2013.
6. 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, LNCS, Springer, 2011.
7. Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao, *The Daikon system for dynamic detection of likely invariants*, Science of Computer Programming, vol. 69, no. 1–3, Elsevier, 2007.

8. 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.
9. J. P. Galeotti, N. Rosner, C. López Pombo and 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.
10. J.P. Galeotti, N. Rosner, C. López Pombo, M. Frias, *TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds*. IEEE Transactions on Software Engineering 39(9), IEEE, 2013.
11. Juan P. Galeotti, Carlo A. Furia, Eva May, Gordon Fraser, Andreas Zeller, *Inferring Loop Invariants by Mutation, Dynamic Analysis, and Static Checking*, IEEE Trans. Software Eng. 41(10), IEEE, 2015.
12. C. Ghezzi, M. Jazayeri and D. Mandiroli, *Fundamentals of Software Engineering*, Second Edition, Prentice-Hall, 2003.
13. D. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, 1989.
14. D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, MIT Press, 2006.
15. Web site of the Java Genetic Algorithms Package (JGAP): `http://jgap.sourceforge.net`
16. 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, LNCS, Springer, 2012.
17. Home Page of the Korat test generation tool: `http://korat.sourceforge.net`
18. D. Kroening and 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, LNCS 8413, Springer, 2014.
19. B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*, Addison-Wesley, 2000.
20. Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer, 1996.
21. C. Pacheco, S. K. Lahiri, M. D. Ernst and T. Ball, *Feedback-Directed Random Test Generation*, in Proceedings of the 29th international conference on Software Engineering ICSE 2007, IEEE, 2007.
22. 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*, Formal Methods in System Design 32(3), Springer, 2008.
23. P. Ponzio, N. Aguirre, M. Frias and W. Visser, *Field-Exhaustive Testing*, to appear in Proceedings of International Symposium on the Foundations of Sofware Engineering FSE 2016, Seattle (WA), USA, ACM, 2016.
24. S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, second edition, Prentice-Hall, 2003.
25. Sebastian Uchitel, Greg Brunet, Marsha Chechik, *Synthesis of Partial Behavior Models from Properties and Scenarios*, IEEE Trans. Software Eng. 35(3), IEEE, 2009.
26. W. Visser, C. S. Pasareanu and 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.