Dynamic Synthesis of Program Invariants using Genetic Programming

Luigi Cardamone
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Milano, Italy
Email: cardamone@elet.polimi.it

Andrea Mocci
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Milano, Italy
Email: mocci@elet.polimi.it

Carlo Ghezzi
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Milano, Italy
Email: carlo.ghezzi@polimi.it

***Abstract*—Symbolic program manipulation plays a key role in program comprehension and verification. Logic formulae are used to represent the program's state and transformation rules describe the effect of statement executions on the program's state. A well-known problem arises in the case of loops, since the number of iterations is generally unknown. The effect of a loop is therefore abstracted into a loop invariant, whose derivation cannot in general be automated and requires human ingenuity.**

In this paper, we present a preliminary approach that integrates genetic programming into the synthesis of invariant formula that describes the behavior of a loop. We present a specific representation of formulae that works well with loops manipulating arrays. The technique has been validated with a set of relevant examples with increasing complexity. The preliminary results are promising and show the feasibility of our approach.

I. Introduction

In many software development activities, descriptions of software behavior in terms of mathematical artifacts are critical to support or even enable the activities themselves. Such descriptions are usually called *specifications*; for example, a specification of a procedure behavior can be given in terms of pre- and post-conditions, each one expressed as a first-order logic formula. To test the correctness of the procedure, pre- and post-conditions can be effectively used as an oracle.

However, writing specifications is usually as time consuming as writing the code itself, and requires mathematical skills that developers might not possess. For this reasons, specifications are usually absent. To overcome specification absence, the software engineering community has proposed several techniques that are able to extract a description of a program's behavior.

A class of the proposed techniques uses *dynamic invariant inference*, which was introduced less than a decade ago by the pioneering Daikon method [1]. With the help of a test suite that exercises the functionality of a program, Daikon synthesizes program properties that hold at pre-selected program points (typically method entries and exits). Such properties are representation invariants for a class, and method pre- and post-conditions, collectively called invariants, expressed as first-order logic formulae. The extracted properties have no formal assurance of their correctness, but they do match the observed program executions and they are produced only when there is some statistical confidence that their occurrence is not incidental. A crucial aspect of the dynamic invariant inference process is that the invariants produced do not reflect only the behavior of the program, but also the assumptions and expectations of the test suite. This makes the approach doubly useful for software engineering purposes, by introducing the usage context of an application. Most dynamic invariant inference systems proposed by the software engineering community extract invariants using a predefined collection of patterns. This technique may lead to discover several invariants that are either irrelevant or false, that is, they hold accidentally.

A recent proposed method, implemented in the DySy tool [2], uses a dynamic symbolic execution technique to drastically improve the quality of inferred invariants (i.e., the percentage of relevant invariants) and reduce the number of test cases required to disqualify irrelevant invariants. The approach executes test cases, just like a traditional dynamic invariant inference tool, but simultaneously performs a symbolic execution of the program to avoid the need of a predefined set of invariants. The major drawback of DySy is that this approach is not able to deal effectively with loops. DySy detects each path executed in the cycle but it is not able to abstract over multiple execution to infer the general invariants that hold for that loop.

To overcome this limitation, we present a technique that can be used to synthesize invariants involving loops. Our approach uses an evolutionary algorithm-based methodology, *Genetic Programming* [3], to search for a logical formula that summarizes the computation of a loop. The results show that our approach is effective and can be integrated in a tool like DySy to improve its capabilities.

This paper is structured as follows. In Section II we introduce the approach by presenting dynamic detection of program invariants and the related state-of-the-art approaches. Thus, we present Genetic Programming and the particular tool implementing it that we used for our approach. In Section III we describe the details of our technique by illustrating the representation of logic formulae and the fitness function we used for the genetic programming. Thus, in Section IV we present the preliminary experimental results. Finally, Section V outlines the conclusions of this paper and illustrates the future

work.

II. Background

In this section, we introduce the area of dynamic detection of program invariants and some state-of-the-art approaches that deal with this problem. In particular, we present DySy, a methodology based on symbolic execution, which represents the reference for our contribution. Thus, we present Genetic Programming and the ECJ package which implements the genetic algorithms that we used in our approach.

A. Dynamic Detection of Program Invariants

As we discussed in the introduction, the pioneering approach for invariant inference was the Daikon tool [1], which started a new subfield within software engineering that explicitly studies the problem of extracting logic formulae that synthesize a program's behavior. Such field has recently gathered significant attention in the software engineering community because such descriptions are really useful for program understanding and to support verification and validation activities.

The fundamental idea of Daikon consists in the analysis of a test suite that exercises the functionality of a program. The analysis is essentially an invariant inference system which synthesizes program properties that hold at pre-selected program points, typically method entries and exits.

In the case of Daikon, the logic formulae are synthesized from a set of predefined templates, which are not composable. A pattern involving two variables, for example, is $x > y$. As we already discussed, since this approach only observes a subset of possible program executions, the properties synthesized by Daikon have no assurance that they are correct. Instead, they are coherent with the observed program executions corresponding to the test suite used as inference basis. Some irrelevant properties are removed by using ad-hoc statistics, that assure that the inferred invariants have a relative high confidence, i.e., its occurrence is not incidental.

Dynamic invariant inference systems, like Daikon [1] or DIDUCE [4], consider a predefined collection of invariant templates. In these systems, each template is instantiated with program variables to produce the candidate invariants under examination. The user can expand the collection by adding more templates, but the number of possible instantiations for all combinations of program variables grows prohibitively fast. Therefore, dynamic invariant inference systems typically perform best by concentrating on a small set of simple candidate invariants. Furthermore, this kind of inference process will also produce several invariants that are either irrelevant or false (they may hold accidentally in a particular test-suite).

If the program behavior is outside the expressive power of predefined patterns, such approaches are not able to derive any specification. For this reason, a recent advancement has been proposed to overcome this limitation.

B. Integrating Symbolic Execution: DySy

The approach implemented in DySy [2] uses a dynamic symbolic execution technique to improve the quality of inferred invariants (i.e., the percentage of relevant invariants) and the ease of obtaining them (i.e., the number of test cases required to disqualify irrelevant invariants). In dynamic symbolic execution, test cases are executed, just like a traditional dynamic invariant inference tool, but simultaneously the technique also performs a symbolic execution of the program.

```
int testme(int x, int y) {
  int prod = x*y;
  if (prod < 0)
    throw new ArgumentException();
  if (x < y) {     // swap them
    int tmp = x;
    x = y;
    y = tmp;
  }
  int sqry = y*y;
  return prod*prod - sqry*sqry;
}
```

Fig. 1. A simple algorithm.

The symbolic execution results in the programs branch conditions being collected in an expression, called *path condition*. The path condition is always expressed in terms of program input variables. It gets refined while the test execution takes place, and symbolic values of the program variables are being updated. At the end of execution of all tests, the overall path condition corresponds to the precondition of the program entity under examination.

Symbolic values of externally observed variables provide the inferred postconditions, and symbolic conditions that are preconditions and postconditions for all methods of a class become the class state invariants.

As an example on how this technique works, consider the method of Figure 1. Appropriate unit tests for the method will probably exercise both the case $x < y$ and its complement, but are unlikely to exercise the code producing an exception, as this directly means illegal arguments. Consider the outcome of executing the code for input values x smaller than y (e.g., $x = 2$, $y = 5$), while also performing the execution in a symbolic domain with symbolic values x and y. In this case, variable names are overloaded to denote the respective symbolic values designating the original inputs.

The first symbolic condition observed is $x * y \geq 0$: the branch of the first if is not taken, and local variable prod has the value $x*y$ in the symbolic domain. The symbolic execution also accumulates the condition $x < y$ from the second if statement. At the end of execution the symbolic value of the returned expression is $y * x * y * x - x * x * x * x$. Note that this expression integrates the swapping of the original x and y values. If the process is repeated for more test inputs (also exercising the other valid path of the method) and collect together the symbolic conditions, then this approach yields:

- a precondition $x * y \geq 0$ for the method;
- a corresponding postcondition: $result == (x < y) \Rightarrow y * x * y * x - x * x * x * x$

else $x * y * x * y - y * y * y * y$

This captures the methods behavior quite accurately, while ensuring that the only symbolic conditions considered are those consistent with actual executions of the test suite. Thus the approach is symbolic, but at the same time dynamic: the symbolic execution is guided by actual program behavior on test inputs.

We will discuss later, in Section III, what are the issues and limitations of this approach, which are inherently related to the use of symbolic execution for programs containing loops. Before discussing these aspects, we briefly introduce the evolutionary techniques that we used within the proposed approach.

C. Genetic Programming

Genetic programming (GP) is a branch of Evolutionary Computation where evolutionary algorithms are applied to optimize computer programs or mathematical formulas. Thus, GP algorithms are a specialization of *Genetic Algorithms (GA)* [5] where each individual is a computer program or a formula. Because computer programs are variable in size, the representations used by GP are also variable in size. One of the most common form of GP employs trees since trees can be easily evaluated in a recursive manner. This representation was first proposed by Nichael Cramer[6], and then extended by John Koza [3]. The idea is that every tree node has an operator function and every terminal node has an operand. The representation of a GP is completely defined by the function set (specifying the arity of each function) and the terminal symbols.

Genetic Programming has been applied to a wide number of fields [7], [8]. Some examples of application are: evolving robot behaviors [9]; pattern classification [10]; mining DNA data [11]; synthesizing combinational logic circuits [12]; generating terrains for computer games [13]. For a wide survey on the applications of genetic programming please refer to [7], [8].

One problem with genetic programming, is that the size of the search space can be very big. Strongly typed genetic programming (STGP) introduced by Montana [14] try to reduce the search space of the problem by adding some structural constrains which limit the number of valid trees. In STGP every terminal symbol has a well defined type. The functions operators must define the type of each argument (domain) and the type of the return value (co-domain). Every tree built by the initialization operator or by the genetic operators must comply to the type constraints. Recently Strongly typed genetic programming has been applied in the field of software engineering for test cases generation [15], [16].

D. ECJ

ECJ is a research EC system, written in Java, developed at George Mason University's ECLab Evolutionary Computation Laboratory. It was designed to be highly flexible, with nearly all classes (and all of their settings) dynamically determined at runtime by a user-provided parameter file. All structures in the system are arranged to be easily modifiable. Even so, the system was designed with an eye toward efficiency.

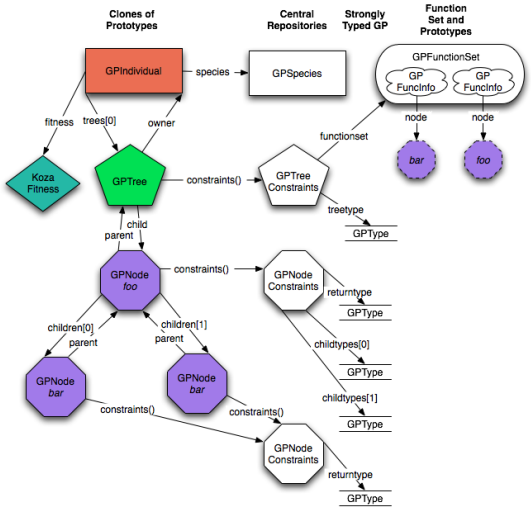

Fig. 2. ECJ class diagram for tree-based individuals.

ECJ provides several building blocks (initializers of the population, mutation and crossover operators, selection operators, niching) that can be assembled as in a pipeline to implement a big variety of genetic algorithms, evolutionary strategies and multi-objectives evolutionary algorithms. It also allows to use several genetic representation: fixed-length and variable-length linear genomes, trees and multiple tree forests. ECJ enables multi-threading and Master/Slave evaluation over multiple processors, with support for generational, asynchronous steady-state, and co-evolutionary distribution.

ECJ provides all the blocks to implement Strongly-Typed Genetic Programming (Koza-Style), which is an extended version of GP that support types and was used for the experiments of this work. In a Strongly-Typed GP both terminal symbols and the return values of the functions have a type. This implies that the domain of each function needs to be defined over types. In this way only the formulas that comply with the type-constraints are allowed and the search space can be drastically reduced.

To define a GP using ECJ it is necessary to define the syntax and the semantics. The syntax is defined using a configuration file where it is necessary to declare: the types used, the terminal symbols and their type, the function set with the type of the return value and the type and the arity of the domain. To define the semantic for each function it is necessary to add a new Java class that implements the behavior of the operator. Figure 2 depicts the classes used in ECJ to represent an individual: each Individual has a Fitness and a Tree; a Tree is composed of several Nodes that can be either a function or a terminal symbol.

III. Our Approach

In this section we describe the issues that involve discovering program invariants with DySy and in particular we present our approach which focuses on finding the logical formula

Algorithm 1 Linear Search in a Vector

```
1: procedure SEARCH(V,x)
2:     for (int i=0; i<V.length; i++) do
3:         if (x==V(i)) return i;
4:     end for
5:     return -1;
6: end procedure
```

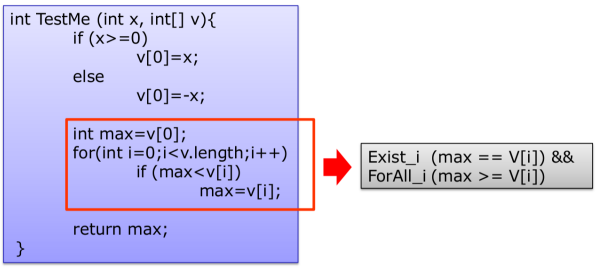

Fig. 3. Overview of the proposed approach. The body loop can be replaced with a logical formula.

that synthesize the behavior of a loop. Then we introduce the representation used for the formula and how we applied genetic programming to search the space of the solutions. Finally we present the overall architecture of the system that implements the proposed approach and the experimental results.

A. Problem Definition

Dynamic Detection of Program Invariants has the goal of discovering some properties that holds during the execution of a given program. These properties are useful for debugging the code or for understanding the correctness of the program.

The approach implemented in the tool DySy [2] resulted to be very effective for the problem of invariants inference. The main idea of DySy is to use the concrete execution of test cases to generate the path conditions coupled with the symbolic execution to infers abstract conditions. In this way it is possible to reduce the number of useless invariants and to avoid the use a pre-defined set of patterns as in Daikon [1]. However, the DySy approach has a major drawback: it is not able to properly handle loops. When the source code presents a loop, DySy is able to extract invariants only through ad-hoc heuristics.

To better understand the problem of dealing with loops it is possible to consider an example. Algorithm 1 contains a simple loop that performs linear search of an element in a vector. If we apply the symbolic execution using the test case $x = 8$ and $V = [4, 5, 8]$ the result is:

$$(x \neq V[0] \ \wedge \ x \neq V[1] \ \wedge \ x = V[2]) \Rightarrow return = 2$$

Otherwise, if we apply the symbolic execution using the test case $x = 4$ and $V = [4, 5, 8]$ the result is:

$$(x = V[0]) \Rightarrow return = 0$$

It is clear to see that the inference process of DySy is not able to find a general property that holds for the execution of this loop (for example in one case we have $x \neq V[0]$ and in the other one $x = V[0]$).

To overcome this problem, the approach that we propose in this work consists in replacing the loop body with a logical formula, that summarizes the loop execution (Figure 3). After the replacement of body loops, several techniques, like DySy, can be applied more effectively to discover the invariant of the overall program. To summarize the computation of a loop we developed a technique that use Genetic Programming to find a logical formula that can replace the computation of the loop.

B. Representation of Logical Formulae

The representation of the logical formula is based on a subset of First Order Logic. In Genetic Programming, the representation of a formula is defined by the function set and the terminal symbols. In particular, in Strongly Typed Genetic Program [14] it is also necessary to define the types used: the type of the terminal symbols and domain and co-domain of the functional operators.

C. Terminal Symbols

The logical formula we are looking for is a formula which expresses a relation among the variables used in a piece of code (in this case a loop). In particular, the formula should take into account only those variables that are visible before and after the loop execution. Thus, we discard any variable that is created inside the body loop and is not visible outside. If the value of a variable changes during the execution of the loop, the formula only takes into account the value before entering the loop (e.g., x') and the value after executing the loop (e.g., x''). If the value of a variable does not change, we refer to the variable simply using its name (e.g., x). Therefore, in the representation of the formula there is a terminal symbol for every variable that is visible before or after the execution of the loop. If the variable is modified inside the loop, an additional terminal symbol is added to hold the final value of the variable. If the loop manipulates also constants, they can be considered as variables whose value is always the same. The used terminal symbols are not fixed but depend on the structure of the loop to be analyzed. Terminal symbols are computed from a simple syntactical analysis of the piece of code. For example, in Algorithm 1, the terminal symbols that are necessary for the formula are three: V, x and $ReturnValue$. Since V and x are not modified inside the body loop only one terminal symbol is used. $ReturnValue$ is a special variable which represents the value returned by the function.

Our approach focuses on loops which manipulates simple data structures. So, we use only two types of variables: single values and one-dimensional arrays. To build operators that works in the same way with both types we represented also the single value as arrays of length one. Thus, the type of every terminal symbols is $array$.

D. Function Set

The operators used in the function set are a subset of the operators of First Order Logic plus some comparison operators. We divided the operators in groups which have the same domain and co-domain.

The first group includes comparison operators:

- $=$ (equals)
- $\geq$ (greater or equals)

The operators for comparison take as input two arrays and return boolean values. These operators are defined as $(array, array) \rightarrow (logical)$ where logical is a particular array that contains only boolean values. Given two input arrays, these operators compare every element i of the first array with every element j in the second one. The result is a logical array with m columns and n rows where m and n are respectively the length of the first and the second array.

The second group includes the quantifiers:

- $\exists$ (existential quantification)
- $\forall$ (universal quantification)

The quantifiers have the usual meaning and are defined as $(logical) \rightarrow (logical)$. The domain is represented by arrays of boolean values with length greater than one. The quantifiers are always applied to every element of an array. So, for example the formula $\forall i(V[i] = 0)$ is an abbreviation for $\forall i \in \{0, 1, \ldots, V.lenght - 1\}(V[i] = 0)$.

The third group includes the logical connectives:

- $\wedge$ (and)
- $\Rightarrow$ (implication)

The logical connectives have the usual meaning and are defined as $(logical, logical) \rightarrow (logical)$. The operator $\Rightarrow$ is an abbreviation for $(A \Rightarrow B \textbf{ else } ReturnValue = -1)$ in order to capture the behavior of programs that returns a default value when the condition A is not verified. The logical connectives are applied to two arrays with the same cardinality.

Finally, there is a special operator $V[\,]$ which takes as input a single value and outputs the element of the vector V in that position.

E. Formula Truth Value

In this section, we define how the truth value of a valid logical formula is computed. The truth of each formula depends on the values of the terminal symbols. Once the values of the terminals are instantiated, the evaluation of the formula is computed recursively from the root node to the lower nodes of the tree that represents the formula. To describe how the truth value of a logical formula is computed we present a short example. Consider the following formula:

$$\forall i(max \geq V[i])$$

which is the typical invariant of a piece of code which finds the maximum value of a vector. The syntactic tree of the formula and all the evaluation steps are reported in Figure 4. To evaluate a formula it is necessary to fix all the values

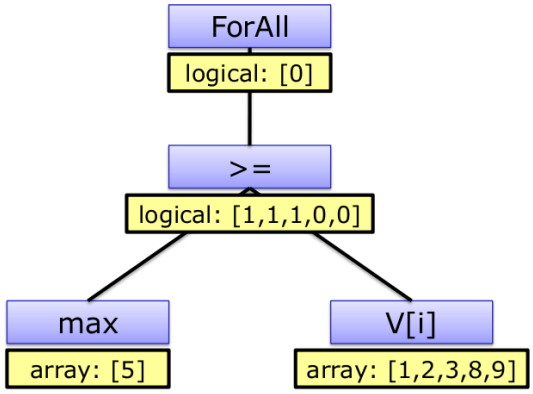

Fig. 4. Evaluation tree of the formula $\forall i(max \geq V[i])$

of the terminal symbols. In this example we use the following assignment:

$$\langle max = [5], V = [1, 2, 3, 8, 9] \rangle$$

The first operator to evaluate is $\geq$ that takes as input two numerical arrays (max and $V[\,]$) and returns an array of logical values. This operator compares max with every element i in $V[\,]$. The result is a logical array where each element i is 1 if $max \geq V[i]$ and 0 otherwise. The resulting array has the same cardinality of $V[\,]$. The last operator to evaluate is $\forall i$. This operator takes as input a logical array. The outputs is 1 only if all the elements in the input array are equal to 1, and 0 otherwise. In this case the output is 0. Since this operator is the root, also the truth value of the overall formula is 0, i.e. false.

F. Fitness Function

To apply Genetic Programming for discovering the formula that can synthesize the computation of a loop it is necessary to define a proper fitness function. The first way to define the fitness function is to count the number of times in which a formula is true after the execution of a certain number of test cases (i.e. assignment of the terminal symbols). For example, after a loop that computes the maximum value of a vector the formula $\forall i(max \geq V[i])$ will be true for every test case generated (assignment of max and V). However, this fitness function has a major drawback: also all the tautologies are true at the end of the loop and the genetic algorithm will return formulas like $max = max$ (tautologies) and will not search effectively for other formulas.

Our idea to overcame this problem consists in adding to the test suite also incorrect test cases, that is, test cases that describe behaviors that are not expressed by the loop. For example, consider a loop that computes the maximum value of a vector; correct test cases can be $(V = [1, 2, 3, 8, 9], max = 9)$ and $(V = [1, 2, 3, 5, 4], max = 5)$, while incorrect test cases can be: $(V = [1, 2, 3, 8, 9], max = 3)$ and $(V = [6, 2, 3, 5, 4], max = 1)$. In this way, a valid formula must be true for the correct test cases and false for the incorrect

ones. So the fitness function can be defined as the number of matches over all the test cases:

$$fitness = \frac{\#matches}{\#testcases}$$

There is a match only when the formula is true for a correct test cases or the formula is false for an incorrect test cases. With the fitness defined above it is easy to verify that the formula $\forall i(max \geq V\,[i])$ applied to previous test cases has a fitness equal to $4/4$, i.e. it is the optimal solution. It is also easy to see as any tautology has a fitness that is always equal to $2/4$, far away from the optimum. In fact, any tautology is true for the two correct cases (2 matches), but it is also true for the two incorrect cases (0 matches).

G. Test Cases Generation

To compute the fitness of a given formula, our approach needs both correct and incorrect test cases. To generate test cases it is useful to represent the loop as a function F which takes some variable as inputs and uses some variables to output the result. In this way, a test case can be defined as a tuple of input variables and output variables:

$$TestCase = \langle InputVars, OutVars \rangle$$

To generate *correct test cases* it is only necessary to generate the input variables and then the output variables are directly computed by applying F:

$$OutVars = F(InputVars)$$

The input variables can be generated by applying many methods for automated testing. In this work, for simplicity, we generate the inputs using a random sampling from a predefined set of values. To generate *incorrect test cases* it is necessary to apply a different process. After the generation of the input values, it is necessary to generate the outputs values using a random sampling. If $OutVars = F(InputVars)$ the tuple is discarded otherwise the tuple is added to the set of the incorrect test cases. The process is iterated until $OutVars \neq F(InputVars)$.

In the implemented approach, for the fitness evaluation of the formula, we automatically generate 40 test cases: 20 correct test cases and 20 incorrect test cases. This number was determined empirically, in order to achieve a robust evaluation of the fitness.

IV. Experimental Results

To validate our approach we tested 6 algorithms of increasing complexity:

- **Max Value**: takes as input a vector; returns the highest value of the vector;
- **Index of Min**: takes as input a vector; returns the index of the element with the lowest value in the vector;
- **Exist Value**: takes as input a single value X and a vector; returns 1 if the value X is present in the given vector;
- **Linear Search**: takes as input a single value X and a vector; search the element X in the given vector and if the element is found the index is returned;
- **Is Sorted**: takes as input a vector; returns 1 if the given array is sorted, -1 otherwise;
- **Sort**: takes as input a vector; sorts the input vector.

TABLE II
Experimental Results

Algorithm	Generations	Fitness	Optimum Found
Max Value	20.77 ± 21.75	97.90 ± 3.56	66 %
Index of Min	1.16 ± 0.50	100.00 ± 0.00	100 %
Exist Value	36.82 ± 21.22	94.52 ± 5.04	31 %
Linear Search	30.72 ± 22.19	96.65 ± 4.04	46 %
Is Sorted	30.72 ± 22.19	96.66 ± 3.34	37 %
Sort	37.39 ± 20.17	93.14 ± 6.09	32 %

For each of these algorithms our approach is able to find the proper logical formula that synthesizes the computation. Table I reports the best formula found for each algorithm, where X is the input value, V is the input vector, $ReturnValue$ is the output value and RV is the output vector. It is possible to see that our approach is able to find the correct formula also in the case of the sort algorithm. This formula is complex as involve two distinct parts: a first part which say that the returned vector is sorted in ascending order; a second part which say that all the elements in RV are in V, i.e. RV is a permutation of V.

Since we are dealing with a stochastic algorithm there is no guarantee that the correct formula will be discovered in each run of the algorithm. To evaluate the performance of our approach, for each formula, we measured the number of times over 100 runs in which the optimum is found (the formula with fitness 100 %). The results are shown in Table II which also reports the average number of generations to find the optimum and the average fitness of the best solution found in every run. For the experiments we used all the default parameters present in the ECJ package: in particular the population size is 1024, the maximum number of generations is 50, tournament selection is used and the algorithm stops when the optimum is found. The results reported in Table II shows that the performance of our approach are good. In the worst case the optimum is found in the 32 % of the cases which means that, on average, one run over three may return the correct formula. Since each run takes an average time that is below 10 seconds (on a computer with an Intel duo processor of 2.5 GHz) our approach is feasible.

The biggest bottleneck that affects the performance of our approach is the size of the search space, that is basically represented by the number of symbols used in our GP. To analyze how much the search space affects the performance, we repeated the previous experiments removing for each case 2 or 3 function operators that are not used by the corresponding algorithm. For example, in the Linear Search algorithm we removed the function $\geq$ that does not appear in the code (see Algorithm 1). The results are reported in Table III. It is possible to see that the performance improvement is remarkable: the smallest improvement is 15 % while the biggest is 51 % and for every formula the final performance is always greater than the 50 %. This result means that selecting the smallest set of symbols is the key to improve performance. This suggests

TABLE I
EXAMPLES OF LOGICAL FORMULAS FOUND WITH THE PROPOSED APPROACH

Algorithm	Formula
Max Value	$\forall i(ReturnValue \geq V[i]) \ \wedge \ \exists i(V[i] = ReturnValue)$
Index of Min	$\forall i(V[i] \geq V[ReturnValue])$
Exist Value	$\exists i(V[i] = X) \Rightarrow (ReturnValue = 1)$
Linear Search	$\exists i(V[i] = X) \Rightarrow (V[ReturnValue] = X)$
Is Sorted	$\forall i(V[i+1] \geq V[i]) \Rightarrow (ReturnValue = 1)$
Sort	$\forall j(RV[j+1] \geq RV[j]) \ \wedge \ \forall i(\exists j(V[i] = RV[j]))$

TABLE III
COMPARISON BETWEEN A GP THAT USES ALL SYMBOLS AND A GP WITH A REDUCED SET OF SYMBOLS

Algorithm	All Symbols	Reduced Symbols
Max Value	66 %	97 %
Index of Min	100 %	100 %
Exist Value	31 %	82 %
Linear Search	46 %	63 %
Is Sorted	37 %	52 %
Sort	32 %	71 %

that our final approach can use a first passage of Symbolic Execution to extract the symbols used by the algorithm and then apply GP using only the discovered symbols plus the logical connectives.

V. CONCLUSIONS AND FUTURE WORKS

In this work, we presented our approach that can be used to improve a state-of-the-art technique for dynamic and symbolic detection of program invariants called DySy. The technique used by DySy mainly consists of combining symbolic and concrete execution to a given program to derive program invariants that describe its behavior. The main drawback is that DySy is not able to deal effectively with loops.

We presented a dynamic analysis approach that focuses on loops in order to improve the capabilities of DySy. We leverage genetic programming to discover a logical formula that synthesizes the behavior of a loop. In order to do so, we designed a specific representation of a class of logical formulae describing loop behaviors that works well with genetic programming. The representation proposed is flexible an can be extended with new symbols if necessary. We also designed a specific fitness function that guide the search towards correct invariants.

In a preliminary assessment of the proposed technique, we validated our approach using 6 different algorithms of increasing complexity. The results show that our approach is able to find the logical formula that summarizes the computation of the loop in all the 6 algorithms. Since genetic programming is a stochastic algorithm, it is not guaranteed that the invariant is found in every run of the algorithm. Thus we computed the results over multiple runs of the algorithm. The results show that the percentage of discovering the correct invariant range from 31 % (worst case) up to 100 % (best case). This means that, on average, in the worst case three runs of the algorithm can be enough to find the correct formula. Since each run of the algorithm is relatively fast (on average between 1 and 10 seconds) our approach is feasible and can be a useful tool for invariant detection for loops.

The research presented in this paper is the first one that apply genetic programming to address the issue of program invariants detection. Thus, there are many aspects of the approach that can be extended and further investigated. First, a possible extension consists in expanding the number of operators used in the formula and testing the approach with more complex algorithms. Another extension deals with reducing the search space: our results confirms that performance drastically improves when the search space is reduced. Thus, it is necessary to investigate how, for a given loop, it is possible to remove some operators that are not useful, for example by exploiting a preliminary symbolic execution of the loop body to discover the useful operators. An additional future work consists in investigating more robust approaches for generating the test cases that are needed for computing the fitness of each formula. Finally, another important future work aims to integrate our approach in a existing tool like DySy.

ACKNOWLEDGMENTS

This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

REFERENCES

[1] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 35–45, 2007.

[2] C. Csallner, N. Tillmann, and Y. Smaragdakis, "Dysy: dynamic symbolic execution for invariant inference," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 281–290.

[3] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.

[4] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA: ACM, 2002, pp. 291–301.

[5] D. Goldberg, *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley, 1989.

[6] N. L. Cramer, "A representation for the adaptive generation of simple sequential programs," in *Proceedings of the 1st International Conference on Genetic Algorithms*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1985, pp. 183–187.

[7] R. Poli, W. B. Langdon, N. F. Mcphee, and J. R. Koza, "Genetic programming an introductory tutorial and a survey of techniques and applications," Tech. Rep., 2007.

[8] M.-J. Willis, H. Hiden, P. Marenbach, B. McKay, and G. Montague, "Genetic programming: an introduction and survey of applications," in *Genetic Algorithms in Engineering Systems: Innovations and Applications, 1997. GALESIA 97. Second International Conference On (Conf. Publ. No. 446)*, Sept. 1997, pp. 314 –319.

[9] W. po Lee, J. Hallam, H. H. Lund, and E. U. K, "Applying genetic programming to evolve behavior primitives and arbitrators for mobile robots," in *In Proceedings of IEEE 4th International Conference on Evolutionary Computation*. IEEE Press, 1997, pp. 495–499.

[10] J. Kishore, L. Patnaik, V. Mani, and V. Agrawal, "Application of genetic programming for multicategory pattern classification," *Evolutionary Computation, IEEE Transactions on*, vol. 4, no. 3, pp. 242 –258, Sept. 2000.

[11] W. B. Langdon and B. F. Buxton, "Genetic programming for mining dna chip data from cancer patients," *Genetic Programming and Evolvable Machines*, vol. 5, pp. 251–257, 2004, 10.1023/B:GENP.0000030196.55525.f7. [Online]. Available: http://dx.doi.org/10.1023/B:GENP.0000030196.55525.f7

[12] S. M. Cheang, K. H. Lee, and K. S. Leung, "Applying genetic parallel programming to synthesize combinational logic circuits," *Evolutionary Computation, IEEE Transactions on*, vol. 11, no. 4, pp. 503 –520, 2007.

[13] M. Frade, F. F. de Vega, and C. Cotta, "Breeding terrains with genetic terrain programming - the evolution of terrain generators," *International Journal for Computer Games Technology*, vol. 2009, no. Article ID 125714, p. 13, 2009.

[14] D. J. Montana, "Strongly typed genetic programming," *Evol. Comput.*, vol. 3, pp. 199–230, June 1995. [Online]. Available: http://dx.doi.org/10.1162/evco.1995.3.2.199

[15] J. C. B. Ribeiro, "Search-based test case generation for object-oriented java software using strongly-typed genetic programming," in *Proceedings of the 2008 GECCO conference companion on Genetic and evolutionary computation*, ser. GECCO '08. New York, NY, USA: ACM, 2008, pp. 1819–1822. [Online]. Available: http://doi.acm.org/10.1145/1388969.1388979

[16] S. Wappler and J. Wegener, "Evolutionary unit testing of object-oriented software using strongly-typed genetic programming," in *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, ser. GECCO '06. New York, NY, USA: ACM, 2006, pp. 1925–1932. [Online]. Available: http://doi.acm.org/10.1145/1143997.1144317