Testing an Intentional Naming Scheme Using Genetic Algorithms

Sarfraz Khurshid

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
khurshid@lcs.mit.edu

Abstract. Various attempts have been made to use genetic algorithms (GAs) for software testing, a problem that consumes a large amount of time and effort in software development. We demonstrate the use of GAs in automating testing of complex data structures and methods for manipulating them, which to our knowledge has not been successfully displayed before on non-trivial software structures. We evaluate the effectiveness of our GA-based test suite generation technique by applying it to test the design and implementation of the Intentional Naming System (INS), a new scheme for resource discovery and service location in a dynamic networked environment. Our analysis using GAs reveals serious problems with both the design of INS and its inventors' implementation.

1 Introduction

Genetic algorithms [7] are a family of computational models inspired by biological evolution. These algorithms encode a potential solution to a specific problem on a simple chromosome-like data structure and apply recombination operators to these structures so as to preserve critical information. Genetic algorithms are often viewed as function optimizers, although the range of problems to which they have been applied is quite broad [19].

There have been various attempts ([3], [6], [10], [14], [16], [18], [20]) to use genetic algorithms in software testing, a problem that is very labor intensive and expensive [2]. In this paper we explore the use of genetic algorithms in automating testing of complex data structures used in naming infrastructures for dynamic networks of computers and devices.

Naming is a fundamental issue in distributed systems that is growing in importance as the number of directly accessible systems and resources grows to the point that it is difficult to discover the (names of) objects of interest. The difference between a true confederation of computing services and a collection of networked centralized computing systems lies in the system's ability to provide a uniform and location independent way of accessing and naming resources.

Testing architectures that provide service location and resource discovery using location independent names in a worldwide internetwork is clearly a challenging task.

T. Margaria and W. Yi (Eds.): TACAS 2001, LNCS 2031, pp. 358–372, 2001.

1.1 Software Testing

Studies indicate that software testing consumes more than fifty percent of the cost of software development [2]. This percentage is even higher for critical software, such as that used for avionics systems. As software becomes more pervasive and is used more often to perform critical tasks, it will be required to be of higher quality. Unless we can find more efficient ways to perform effective testing, the percentage of development costs devoted to testing will increase significantly.

Generation of test data to satisfy testing requirements is a particularly labor-intensive component of the testing process. For a given testing requirement, test data generation techniques try to identify a program input that will satisfy the selected testing criteria. If the process of test data generation is automated, significant reductions in the cost of software development could be achieved.

Various test data generation techniques have been automated. *Goal-oriented* test data generators select inputs to execute the selected goal irrespective of the path taken (e.g. [13]). *Random* test data generators use some distribution to select random inputs (e.g. [15]). *Structural* or *path-oriented* test data generators make use of the program's control flow graph to select a particular path, and use a technique such as symbolic evaluation to generate test data for the selected path (e.g. [5], [17]). *Intelligent* test data generators typically guide the search for new test data using complex analyses of the code (e.g. [4], [14]).

In this paper we present a technique for automating test data generation using a genetic algorithm, that aims at testing structural properties of the data structures involved in the program and their associated methods. The genetic algorithm conducts its search by constructing new test data (*next generation*) from previously generated test data (*current generation*) that are evaluated as good candidates. The algorithm evaluates the candidate test data based on the code coverage achieved, the control points of interest executed or avoided, and the required properties satisfied.

Our GA-based testing technique has four essential components. The first part is to identify methods to test and global properties of interest concerning these methods. The second part is to determine a genetic encoding such that *each* test datum encodes a *sequence* of operations of interest and their parameters.

The third component involves computing the fitness of test data and has three subparts. First, we trivially modify the methods identified in the first part to reward test data that access them by incrementing their score per line of code executed in a method of interest. Second, we identify control points in code that are of particular interest and either add a bonus score or a penalty for executing that point. The rationale for doing so is explained in Section 3.2. Third, we award bonus points to test data that possess the properties identified in part one. This bonus or penalty is considerably greater than the score given per statement of execution in the first part.

The final component of our framework is to apply standard genetic operators of evaluation, crossover, and mutation on the genetic representation of test data in the current generation and move onto the next generation.

An advantage of our approach is that since the genetic representation of each test datum represents a *sequence* of operations of interest, it is straightforward to test the behavior of a program when such operations are interleaved.

Another benefit is due to the idea of using *barriers* (i.e. awarding a large negative penalty for executing certain control points) as this induces new test data to evolve and identify bugs that have not already been discovered, without having to fix the ones previously found.

Our framework applied to generate automated test data for the Intentional Naming System (INS) [1] (Section 2), a new scheme for resource discovery and service location in a dynamic networked environment, reveals serious flaws in both the design and implementation [21] of INS. These flaws, to our knowledge, were not previously known to the INS inventors. In particular, we establish that in the INS naming architecture, addition of a new service can cause a situation where the system makes valid services inaccessible to clients seeking them.

In the next section, the background on INS is given. Then, the genetic algorithm for test data generation is described. Following that, the results from testing the INS implementation are presented. Next, the technique presented is compared to related work. Finally, conclusions and future work are given.

2 INS Background

One particular service discovery solution in dynamic networked environments is the *Intentional Naming System* (INS) [1], which allows services to describe and refer to each other using names which are *intentional*. These names describe a set of properties that the services should have rather than specify a low-level network location. The idea is to allow applications to refer to *what* service they want rather than *where* in the network topology the service resides. It also allows applications to communicate seamlessly with end-nodes, despite changes in the mapping from name to end-node addresses during the session.

INS comprises applications and *intentional name resolvers* (INRs). Applications may be clients or services with services providing the functionality or data required by clients. Like IP routers or conventional name servers, INRs route requests from clients to appropriate locations, using a database that maps service descriptions to their physical network locations.

An INR provides a few fundamental operations. When a service wants to advertise itself – because, for example, it has come online after being down, or because its functionality has been extended – it calls the *Add-Name* operation to register the service against an advertisement describing it. Applications make queries by calling the resolvers *Lookup-Name* operation.

Intentional names are implemented in INS using *name-specifiers* that represent both queries and advertisements. A name-specifier (Figure 1) is an arrangement of alternating levels of *attributes* and *values* in a tree structure. In Figure 1, hollow circles identify attributes and filled circles identify values. Attributes represent categories in which an object can be classified. Each attribute has a

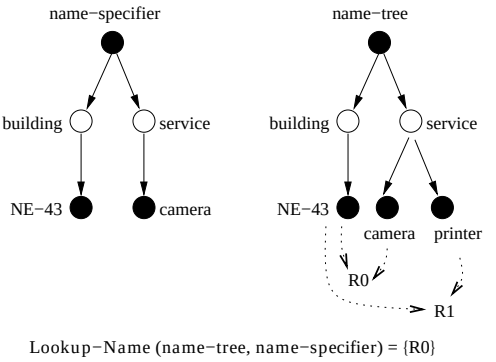

Fig. 1. Example of a *Lookup-Name* operation

corresponding value that is the objects classification within that category. A wild-card may be used in place of a value to show that any value is acceptable.

An attribute together with its value form an *av-pair*; each av-pair has a set of child av-pairs that specialize it to further describe the object. Orthogonal av-pairs specializing the same av-pair are siblings in the tree. The name-specifier in Figure 1 describes an object in building NE-43 that provides a camera service.

An INR stores its information in a database called a *name-tree* (Figure 1). A name-tree resembles a super-positioning of several name-specifiers, and stores the correspondence between name-specifiers and *name-records*, which include the IP addresses of services advertising the name.

A name-tree also has two fundamental building blocks, an *attribute-node* and a *value-node*. A value-node can have several attribute-nodes as children. Similarly, an attribute-node can have several value-nodes as children, each representing a distinct value the name-tree knows.

A value-node that corresponds to a leaf av-pair of an advertised name-specifier also contains a pointer to the relevant name-record. In Figure 1 this is represented by broken arrows, and the name-tree shown stores two objects, one (i.e. R0) that provides a camera service in NE-43 and the other one (i.e. R1) that provides a printer service in the same building.

The name-records for a name-specifier are retrieved from a name-tree using the *Lookup-Name* operation. An algorithm for this operation is given in pseudo-code in the published description of INS [1], and is replicated in Appendix A. When it is invoked on the name-specifier and name-tree in Figure 1, R0 is returned since the value of attribute 'service' sought by the client (i.e. camera) does not match that provided by R1 (i.e. printer).

An implementation of the naming architecture of INS appears in [21]. About 1400 lines of Java code implement *Lookup-Name* and *Add-Name* and the relevant data structures and methods, and another 900 lines constitute the testing code used by INS inventors.

3 The Stochastic Approach

A *genetic algorithm* is an optimization heuristic that emulates natural processes like selection and mutation in natural evolution. It evolves solutions to problems that have large solution spaces and are not amenable to traditional search or optimization techniques. Genetic algorithms have been applied to a broad range of learning and optimization problems [19] since their inception by Holland [7].

Typically a genetic algorithm starts with a random population of solutions (*chromosomes*). Through a *recombination* process and *mutation* operators it evolves the population toward an optimal solution. Achieving an optimal solution is not guaranteed and the task is to design the process to maximize the likelihood of generating such a solution. The first step is the evaluation of *fitness* of solutions in the current population to act as parents in the next generation. Solutions are considered more fit than others if they are closer to an optimal.

Upon evaluation, several solutions are selected and solutions with a higher value of fitness are more likely to get selected. After selection, the parents are recombined and mutated to generate offsprings. The new population is thus formed and the cycle is repeated.

The processes of evaluation, selection, recombination and mutation are usually performed many times in a genetic algorithm. Selection, recombination, and mutation are generic operations in any genetic algorithm and have been throughly investigated in literature. On the other hand, evaluation is problem specific and relates directly to the structure of the solutions. Therefore, in a genetic algorithm a major issue is to design the structure of solutions and the method of evaluation. Among other issues are size of the population, portion of population taking part in recombination, and mutation rate.

Our GA-based testing technique has four essential components:

- identification of methods to test and their global properties of interest;
- framing a genetic encoding such that each chromosome represents a *sequence* of operations of interest and their parameters;
- formulation of the fitness function, which has three subparts:
  - (trivial) modification of the methods identified in part one to reward chromosomes that access them by incrementing the score of a chromosome per line of code it executes in such a method;
  - identification of control points of interest in code and addition of either a bonus score or a penalty for causing execution of that point;
  - awarding bonus points to chromosomes that possess the properties identified in part one. The bonus or penalty is considerably greater than the score given per statement of execution in the first part;
- application of standard evaluation, crossover, and mutation operators on the chromosomes in the current generation and move onto the next generation.

An optimal chromosome would therefore encode a test suite that invokes a sequence of operations of interest, and satisfies the desired properties with regards to that sequence, executes or avoids executing control points of interest as required, and gives maximal code coverage. The following sections explain these notions in detail.

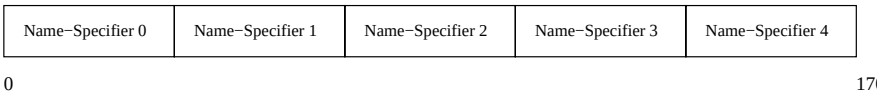

Fig. 2. Representation of a test suite

3.1 Genetic Encoding

As a very first step of our testing technique we identify methods that we want to test. The most interesting operations in the naming architecture of INS are *Lookup-Name* and *Add-Name*. These methods in turn determine the genetic encoding and fitness function that evaluates the chromosomes.

The most obvious way to test the behavior of operations is to have a chromosome represent which operation to perform along with its parameters. So if we were to test name resolution of INS, a chromosome could encode a *Lookup-Name* or *Add-Name* operation along with the name-tree and the name-specifier on which to perform that operation.

A problem with this representation is that it is not immediate how to observe the combined effect of a sequence of *dependent* operations. For example, in INS, if for a given name-tree we want to determine the effect of repeated additions on the resolution of a fixed name-specifier in the resulting name-trees, it would not be feasible to do so.

An alternative representation is to have a chromosome denote a sequence of operations with some parameter having an implicit representation. In the case of INS, a chromosome could then encode successive *Lookup-Name* and *Add-Name* operations with only *one* parameter. It represents that sequence of operations starting from an empty name-tree. So for example it could encode[1]

$$add\ \mathcal{N}_1,\ lookup\ \mathcal{N}_2,\ add\ \mathcal{N}_3,\ add\ \mathcal{N}_4,\ lookup\ \mathcal{N}_5$$

to represent a sequence of operations that starts with a new name-tree $\mathcal{T}_0$, adds $\mathcal{N}_1$ to $\mathcal{T}_0$ to result in name-tree $\mathcal{T}_1$, resolves $\mathcal{N}_2$ with respect to $\mathcal{T}_1$, and so on. This way we could evaluate how a chromosome performs based on the results generated by each of the *Add-Name* or *Lookup-Name* operation that it induces.

We use a slight modification of this representation in our framework for testing INS. In particular, a chromosome denotes five name-specifiers, last four of which are to be inserted in an empty name-tree one by one, and the first one is to be resolved following each insertion. So, for example, the chromosome in Figure 2 would start execution by creating a new name-tree, then $\mathcal{N}_1$ would be added to the name-tree, followed by resolution of $\mathcal{N}_0$, addition of $\mathcal{N}_2$, resolution of $\mathcal{N}_0$, and so on. Notice that this structure is particularly well suited for investigating the effect that addition has on resolution, and there is no need to have an explicit encoding for a name-tree.

[1] For convenience, we write $\mathcal{N}_i$ for Name-Specifier i

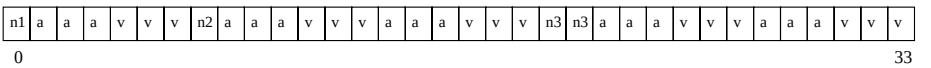

Fig. 3. Genetic representation of a name-specifier

In order to represent a name-specifier we need to determine a suitable number of bits that capture the behavior of methods under scrutiny. Due to the recursive nature of *Lookup-Name* and *Add-Name* it is necessary to have a representation that induces some recursive calls. We use two way branching at the top level and allow one of the children to branch two way, while the other child may only have one further child.

The name-specifier in the top right corner of Figure 5 depicts a *full* name-specifier that can be encoded like this. Moreover, we select the attributes and values from a pool of 8 attributes, {a0,. . .,a7}, and 8 values, {v0,. . .,v7}. This gives us sufficient freedom to perform our testing using diverse test cases.

We use 34 bits (Figure 3) to represent a name-specifier and thus, a chromosome can be represented using 170 bits as is shown in Figure 2. In Figure 3, 'n1' represents 1 or 2 way branching at top level, 'n2' determines whether the first child has a child, 'n3n3' determines whether the second child has 0,1, or 2 children. The sequence 'aaa' contains an attribute and 'vvv' contains a value.

Figure 4 illustrates a sample chromosome and presents the results of the test sequence it would induce. Name-specifiers1–4 are inserted one by one into an empty name-tree to get the name-tree shown in the bottom right corner. Name-specifier0 is resolved after each addition and the resulting name-records are displayed in the bottom line. Notice that during this execution as more advertisements are added to the name-tree, resolution returns more name-records[2].

3.2 Fitness Function

To evaluate the performance of a chromosome, we define our fitness function to have two components. The first component, $\mathcal{F}_s$, only computes the number of statements that are executed while simulating the sequence of operations encoded in a chromosome. In the case of INS, we add a statement of the form `score++;` with every statement of the *Lookup-Name* procedure. This step can easily be automated. Notice, that based solely on this fitness function we can start our experimentation and the fittest chromosomes would try to maximize code coverage of this method.

However, simply achieving maximal code coverage is not our goal. The second component of our fitness function, $\mathcal{F}_d$, is determined by the kind of tests we would like to perform. It uses two simple ideas.

Firstly, in order to induce chromosomes to explore certain aspects of the system being tested, we award *bonus* reward to chromosomes that do so. This could

[2] The result of *Lookup-Name* is treated empty if it is {} or {*}

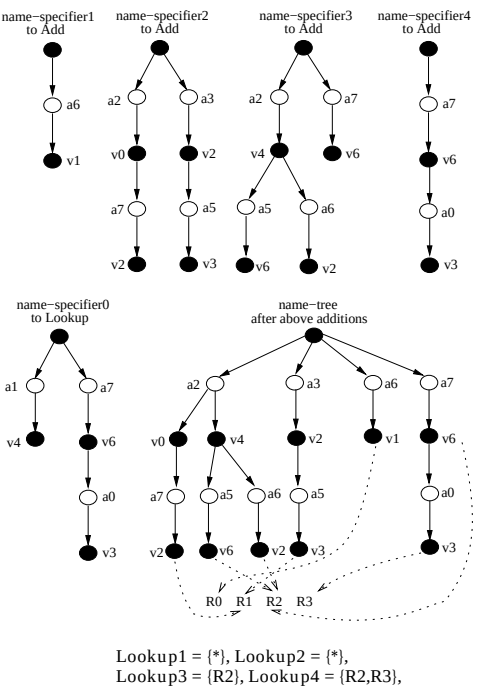

Fig. 4. Visualisation of a sample chromosome to test INS. Lookupi denotes the result of ith call to *Lookup-Name* .

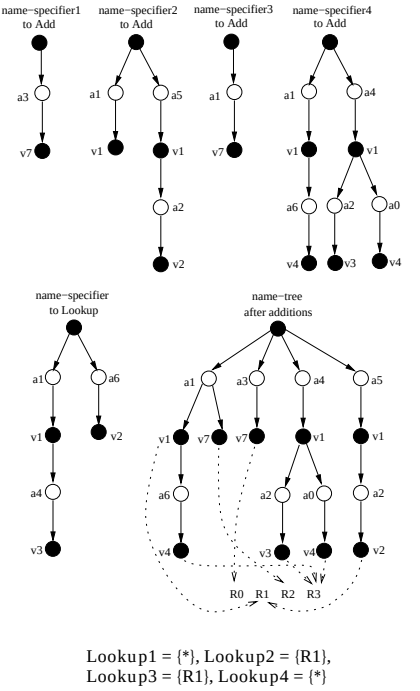

Fig. 5. Revealing a flaw in the INS implementation

involve control points in the code that are more susceptible to lead to run-time errors, or global properties of the test sequence represented by a chromosome like for example, rewarding chromosomes that result in differing results of *Lookup-Name* operation. The chromosome presented in Figure 4 was in fact produced by rewarding 10 extra points per pairwise different *Lookup-Name* results that it produced.

Secondly, we introduce *barriers* in the form of penalty points for chromosomes that execute parts of the code that we have already determined no longer to be interesting from the point of view of further testing. This concept of using barrier functions turns out to be a very powerful idea as we demonstrate in the next section.

The fact that we can use it to evolve chromosomes that do *not* visit certain parts of the code means that once we discover a bug, we do not have to fix it immediately in order to proceed with our testing. Instead we can just introduce a *negative* score at that control point and the testing system would evolve to test other parts of the code that can be executed independently of this buggy point.

These two components together form our fitness function

$$\mathcal{F} = \mathcal{F}_s + \mathcal{F}_d$$

and we demonstrate its utility in the next section where we test INS.

4 Analysis

In this section we use our testing technique to analyze the Java implementation of INS that is given in [21]. There are various properties that we can test of a naming scheme. For example, whether the name resolution mechanism ever returns objects that have functionality conflicting what an application seeks, or whether it returns all objects that conform to a request.

We test a fundamental property that we believe is essential for the correctness of a naming scheme. In particular, we see if addition behaves monotonically in INS, i.e. performing the *Lookup-Name* operation after an addition results in at least the name-records that result if the same name-specifier is resolved before that addition.

In order to test this property we modify our fitness function to *reward* chromosomes that are able to *violate* it. This is achieved by comparing the elements of the sequence $\mathcal{R}_1, \ldots, \mathcal{R}_4$ of results produced by the *Lookup-Name* operations performed after each *Add-Name* operation that the chromosome induces[3]. If this sequence is found to have elements $\mathcal{R}_i$ and $\mathcal{R}_j$ for $i > j$ such that $\mathcal{R}_i \subset \mathcal{R}_j$, we reward such a chromosome with an additional score of 100 points for each such pair.

After incorporating this change we execute our system and let it evolve to see if any of the chromosomes can actually result in such behavior. The evolution stabilizes in about 110 generations and the highest scoring chromosome represents the name-specifiers illustrated in Figure 5. As we see from the results of *Lookup-Name* operations, the last *Lookup-Name* operation produces no name-records, despite the fact that before the final addition was performed, a valid service, namely R1, was returned.

Careful examination of the INS code reveals that the inventors of INS use a (boolean) flag to indicate whether a set contains all elements of a domain, instead of actually inserting those elements into it at the time of its creation (line 1 of pseudo-code in the Appendix). This flag representation later causes problems when set unions are performed, and results in loss of information. In Figure 5 this happens after the final addition when an attribute corresponding to a4 in the name-specifier to lookup is searched in the name-tree. This is an extremely subtle flaw, and our system quickly evolves to detect it and generates our first counterexample.

We next explore the question if this violation of a fundamental property is solely due to the use of this flag based representation. Our task now is to induce

[3] We write $\mathcal{R}_i$ for Lookupi, to represent a set of name-records resulting by an execution of *Lookup-Name* after the i^{th} addition

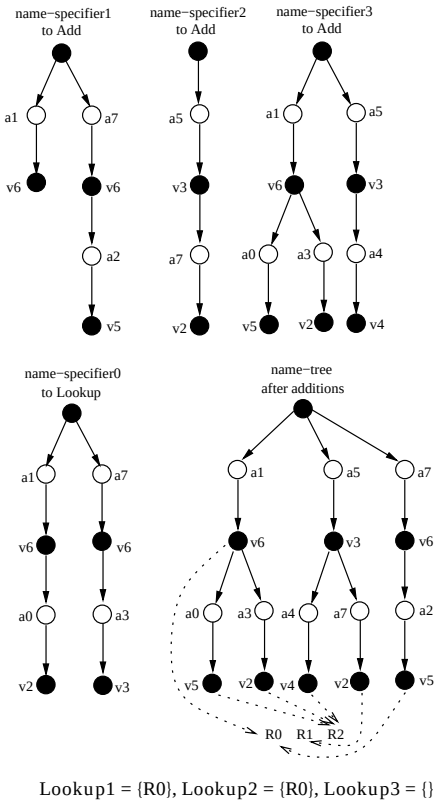

Fig. 6. Another flaw in the INS implementation

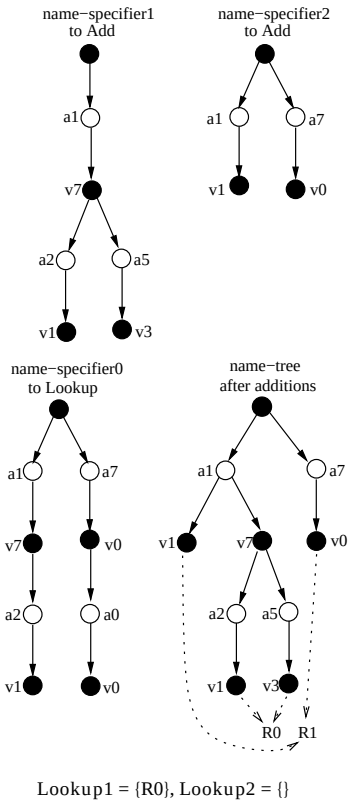

Fig. 7. Non-monotonicity of the Add-Name operation in INS

our system to evolve *away* from using any union operation that leads to this effect. To achieve this, we introduce a *negative* reward for any chromosome that causes an execution of the union operation when exactly one of the sets involved has its flag set and the other one is non-empty. We subtract 200 points from the score of such a chromosome.

Having set the parameters this way, we restart the evolution of our testing system and observe the behavior of the highest ranked chromosome. Around generation number 98 the system stabilizes and the best chromosome in that state is presented in Figure 6. We only illustrate the first four name-specifiers since the desired effect is observed then. Notice that the third addition contradicts the monotonicity property.

An analysis of the behavior of *Lookup-Name* on this test suite reveals that the INS implementation does not handle a value mismatch correctly. When attributes match at a certain level but no corresponding value matches, the imple-

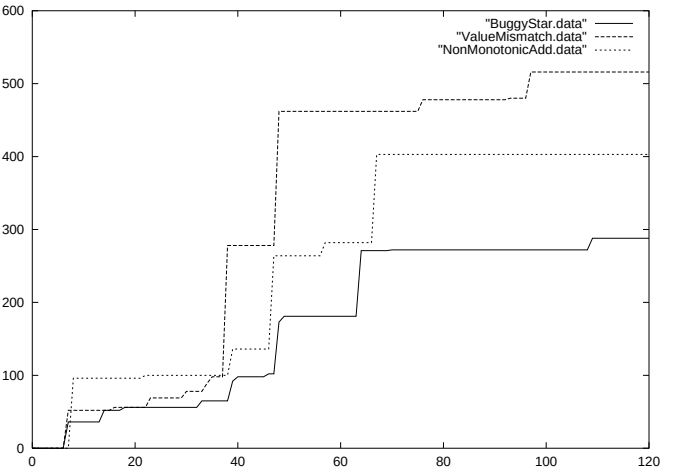

Fig. 8. Performance

mentation behaves in a fashion that once again leads to this erratic behavior. In Figure 6 this happens after the third addition, when a value corresponding to v2 in name-specifier0 is searched in the name-tree among the children value-nodes of the attribute-node a0. It should be noted here that this behavior is *not* due to the bug discovered above.

Having identified another cause of failure of the fundamental property of monotonicity in INS we once again use the idea of introducing a penalty function to discourage chromosomes from causing execution paths that lead to already discovered bugs. We now add an additional penalty of 100 points at the control point in *Lookup-Name* that handles a value mismatch.

This time our system evolves to a stable state in about 65 generations. Figure 7 displays part of the highest scoring chromosome in that generation. The second addition triggers off the required effect. It is interesting to note that this behavior is independent of the bugs discovered above with the INS inventors implementation.

In fact, this problem is due to a flaw in the semantics of INS. INS inventors defined missing attributes to act as wild-cards [1] and the *Lookup-Name* algorithm tries to incorporate that feature. However, this leads to INS displaying this highly undesirable behavior and there is no consistent notion of what it means for a name-record to conform to a name-specifier.

Figure 8 shows the performance of our testing system in producing each of the three counterexamples discussed in this section. We plot the score of the best test suite in a generation (on the vertical axis) against the generation number. "BuggyStar.data" shows the results of experiment that resulted in the chromosome in Figure 5, "ValueMismatch.data" for that in Figure 6, and "NonMonotonicAdd.data" for the chromosome in Figure 7.

All the tests took less than 1 minute on an Intel Celeron 400 MHz processor. Throughout the experiments our genetic algorithm used a population size of 200 chromosomes with the fittest 100 parenting offsprings, mutation rate of 0.05 and a single-point crossover. It took fewer than 120 generations for the system to stabilize in each testing scenario.

5 Related Work

Other researchers have investigated the use of genetic algorithms for automating test data generation, but most work has focused on achieving maximal code or branch coverage.

McGraw et al. [14] explore their use in dynamic test data generation where the problem of test data generation is reduced to one of minimizing a function. They provide an implementation of Korel's function minimization approach to test data generation using a genetic algorithm. A stated goal of their approach is to cover all branches in a program.

Pargas et al. [16] present a goal-oriented technique for automatic test data generation using a genetic algorithm that is guided by the control dependencies in the program. They aim at achieving statement and branch coverage.

The GA-based framework of Roper et al. [18] tests C programs by instrumenting them with probes that provide feedback on the coverage achieved.

Jones et al. [10] have used genetic algorithms to generate test sets automatically that satisfy the requirements for test data set adequacy of structural testing. A recent paper by Bueno et al. [3] builds on their work and presents a tool for the automation of test data generation and infeasible path identification. Their focus is also to perform structural software testing.

Grob [6] argues that genetic algorithms make Dynamic Timing Analysis of systems feasible, and give accurate predictions of a system's run-time behavior through their analysis of the interactions of the program's input parameters.

Schultz et al. [20] apply GA-based machine learning techniques to the general problem of evaluating an intelligent controller for an autonomous vehicle. Their approach subjects a vehicle controller to an adaptively chosen set of fault scenarios within a vehicle simulator, and searches for combination of faults, using genetic algorithms, that produce noteworthy performance by the controller.

Our approach contrasts with these in several ways. First, we aim to test complex data structures and methods for manipulating them, and our primary concern is not to get the maximal code or branch coverage. Second, we use the idea of barrier functions (negative reward) which allows us to identify new bugs without having to fix the ones that we have already discovered. Third, we are able to test properties concerning interleaving of operations in a real world system.

Recently [12], we created an object model of the naming infrastructure of INS in Alloy [8] and analyzed it with the Alloy Analyzer [9] to disprove a published claim made by the inventors of INS about the equivalence of wild-cards and missing attributes. Using that model, we also discovered that the published

Lookup-Name algorithm [1] failed to handle certain boundary conditions and gave erratic results. Private communication with the INS inventors revealed that those boundary cases were fixed in their Java implementation given in [21].

The most important advance over our work in [12] is that the analysis presented here discovers bugs in the proposed fixes of the inventors, and, moreover, identifies a major flaw in the design of the naming semantics of INS and its name resolution algorithm. In [11] we extend our original Alloy model to reveal the flaws discussed here using the Alloy Analyzer.

Using a model checker to verify properties about a structure of the complexity of INS requires a thorough understanding of the algorithms involved and changing them necessitates remodeling. A model checker, however, typically guarantees to find a bug if one exists in (small) finite scope, provided the model is sound. Also, a model can be constructed without an actual implementation.

Our analysis using genetic algorithms only needs elementary knowledge of the implementation details of INS. Moreover, since our GA-based framework manipulates the implementation code directly, the same framework can be used to incorporate any future changes to the code being tested.

6 Conclusions

We have presented and successfully demonstrated an automated test data generation framework based on genetic algorithms that can be adapted to test complicated software structures and methods for manipulating them. Our approach is especially well suited to evaluating other naming schemes in which the correspondence between names and objects is non-trivial.

Care, however, needs to be taken in order to adjust the parameters, especially the fitness function, so as to induce the chromosomes to evolve to test the desired features. We decided to set the bonus or penalty points two orders of magnitude more than the reward for executing a statement of the code, after some experimentation.

Designing a suitable genetic representation of the test data required some care. A cursory examination of the description of the data structures involved would lead to an inefficient encoding. The use of a representation that never encodes a name-tree directly makes it more versatile.

We believe that the use of genetic algorithms in testing has great benefits, as they not only generate quality test data quickly but also can identify structural flaws that are particularly hard to detect otherwise. We view them as complementary to other standard testing tools. A static analysis tool, for example, might be used to assist in computing a suitable fitness function.

It is our goal to identify a set of properties that encapsulates the correctness of a general naming scheme. This would be a first step in creating a framework for testing an arbitrary naming scheme using our GA-based testing technique.

We would also like to explore the possibility of using these ideas in program slicing and detection of infeasible program paths. The concept of using barriers while evaluating fitness seems especially promising.

Acknowledgement

This research was funded by an Information Technology Research grant from the National Science Foundation (#0086154), and by a grant from the NTT Corporation. We would like to thank William Adjie-Winoto for discussions on INS. We are also grateful to Daniel Jackson, Viktor Kuncak, and Alexandru Salcianu for detailed comments on earlier drafts of this paper. Ahmad Kamal, Mehreen Naseem, and Maimoon Nasim kindly helped with the final draft while the author was in Lahore.

References

1. W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *17th ACM Symposium on Operating Systems (SOSP 99)*, Kiawah Island, December 1999.
2. B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.
3. P. M. S. Bueno and M. Jino. Identification of potentially infeasible program paths by monitoring the search for test data. In *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering (ASE 2000)*, Grenoble, France, September 2000.
4. K-H. Chang, J. H. Cross, W. H. Carlisle, and D. B. Brown. A framework for intelligent test data generators. In *Journal of Intelligent and Robotic Systems – Theory and Applications*, July 1991.
5. L. A. Clarke. A system to generate test data and symbolically execute programs. In *IEEE Transactions on Software Engineering*, September 1976.
6. H. Grob. Intelligent timing analysis of real-time software. Internal Report, University of Glamorgan, U.K.
7. J. Holland. Adaption in natural and artificial systems. In *Ann Arbor: The University of Michigan Press*, 1975.
8. D. Jackson. Alloy: A lightweight object modelling notation. Technical Report 797, MIT Laboratory for Computer Science, Cambridge, MA, February 2000.
9. D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. In *Proceedings of the International Conference on Software Engineering*, Limerick, Ireland, June 2000.
10. B. F. Jones, H. H. Sthamer, and D. E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, pages 299–306, Sep 1996.
11. S. Khurshid. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. Masters Thesis, MIT Laboratory for Computer Science, Cambridge, MA, May 2000.
12. S. Khurshid and D. Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE 2000)*, Grenoble, France, September 2000.
13. B. Korel. Automated software test data generation. In *IEEE Transactions on Software Engineering 16(8):870 – 879*, August 1990.
14. G. McGraw, C. Michael, and M. Schatz. Generating software test data by evolution. Technical Report RSTR-018-97-01, RST Corporation, Sterling, VA, February 1998.

15. H. D. Mills, M. D. Dyer, and R. C. Linger. Cleanroom software engineering. In *IEEE Software 4(5): 19-25*, September 1999.
16. R. P. Pargas, M. J. Harrold, and R. P. Peck. Test-data generation using genetic algortihms. In *Journal of Software Testing, Verification, and Reliability*. Wiley, 1999.
17. C. V. Ramamoorthy, S. F. Ho, and W. T. Chen. On the automated generation of program test data. In *IEEE Transactions on Software Engineering 2(4):293 – 300*, December 1976.
18. M. Roper, I. Maclean, A. Brooks, J. Miller, and M. Wood. Genetic algorithms and the automatic generation of test data. Technical Report RR/95/195 [EFoCS-19-95], University of Strathclyde, Glasgow, U.K., 1995.
19. P. Ross and D. Brown. Applications of genetic algorithms. *AISB Quaterly on Evolutionary Computation*, (89):23–30, Autumn 1994.
20. A. C. Schultz, J. J. Grefenstette, and K. A. De Jong. Learning to break things: adaptive testing of intelligent controllers. In *Handbook of Evolutionary Computing*. IOP Publishing Ltd and Oxford University Press, 1995.
21. E. Schwartz. Design and implementation of intentional names. Masters Thesis, MIT Laboratory for Computer Science, Cambridge, MA, May 1999.

A Pseudo-code for *Lookup-Name*

The following pseudo-code description of *Lookup-Name* is taken from [1].

```
Lookup-Name(T,n)
   S <- the set of all possible name-records
   for each av-pair p := (na, nv) in n
      Ta <- the child of T such that
            Ta's attribute = na's attribute
      if Ta = null
         continue
      if nv = *         // wild card matching
          S' <- empty-set
          for each Tv which is a child of Ta
             S' <- S' union (all of the name-records in the
                             subtree rooted at Tv)
          S <- S intersection S'
      else              // normal matching
          Tv <- the child of Ta such that
                Tv's value = nv's value
          if Tv is a leaf node or p is a leaf node
             S <- S intersection (the name-records of Tv)
          else
             S <- S intersection Lookup-Name(Tv, p)
   return S union (the name-records of T)
```

Fig. 9. *Lookup-Name* algorithm