

University of Nebraska - Lincoln
DigitalCommons@University of Nebraska - Lincoln

Computer Science and Engineering: Theses,
Dissertations, and Student Research

Computer Science and Engineering, Department of


Fall 11-30-2018

EvoAlloy: An Evolutionary Approach For Analyzing Alloy Specifications

Jianghao Wang

University of Nebraska - Lincoln, jianghao@huskers.unl.edu

Follow this and additional works at: <http://digitalcommons.unl.edu/computerscidiss>

 Part of the [Computer Engineering Commons](#), and the [Computer Sciences Commons](#)

Wang, Jianghao, "EvoAlloy: An Evolutionary Approach For Analyzing Alloy Specifications" (2018). *Computer Science and Engineering: Theses, Dissertations, and Student Research*. 162.

<http://digitalcommons.unl.edu/computerscidiss/162>

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in Computer Science and Engineering: Theses, Dissertations, and Student Research by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

EVOALLOY: AN EVOLUTIONARY APPROACH FOR ANALYZING
ALLOY SPECIFICATIONS

by

Jianghao Wang

A THESIS

Presented to the Faculty of
The Graduate College at the University of Nebraska
In Partial Fulfilment of Requirements
For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Hamid Bagheri

Lincoln, Nebraska

December, 2018

EVOALLOY: AN EVOLUTIONARY APPROACH FOR ANALYZING ALLOY SPECIFICATIONS

Jianghao Wang, M.S.

University of Nebraska, 2018

Adviser: Hamid Bagheri

Using mathematical notations and logical reasoning, formal methods precisely define a program's specifications, from which we can instantiate valid instances of a system. With these techniques, we can perform a variety of analysis tasks to verify system dependability and rigorously prove the correctness of system properties. While there exist well-designed automated verification tools including ones considered lightweight, they still lack a strong adoption in practice. The essence of the problem is that when applied to large real world applications, they are not scalable and applicable due to the expense of thorough verification process. In this thesis, I present a new approach and demonstrate how to relax the completeness guarantee without much loss, since soundness is maintained. I have extended a widely applied lightweight analysis, Alloy, with a genetic algorithm. Our new tool, EvoAlloy, works at the level of finite relations generated by Kodkod and evolves the chromosomes based on the feedback including failed constraints. Through a feasibility study, I prove that my approach can successfully find solutions to a set of specifications beyond the scope where traditional Alloy Analyzer fails. While EvoAlloy solves small size problems with longer time, its scalability provided by genetic extension shows its potential to handle larger specifications. My future vision is that when specifications are small I can maintain both soundness and completeness, but when this fails, EvoAlloy can switch to its genetic algorithm.

COPYRIGHT

© 2018, Jianghao Wang

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Dr. Hamid Bagheri, for his mentorship and encouragement. His enthusiasm and dedication on research always inspired me to move forward and investigate new possibilities. Without his guidance and inspiration this thesis would not have been possible. I deeply appreciate all the efforts he has put into helping me.

I would like to thank my committee members, Dr. Myra Cohen and Dr. Witty Srisa-an, for their guidance and helpful suggestions. Especially for the kind help from Dr. Myra Cohen, without her deep insight and knowledge of evolutionary algorithms, I could not finish this work successfully. Their suggestions greatly help me to improve this thesis and I owe them a deep sense of gratitude.

I thank all my friends at UNL, Yanlin Zhou, Parvez Rashid, Jonathan Saddler, Alireza Khodaei, Yan Xia, Tuyishime Yves, Zeynep Hakguder, Bruno Silva, Shideh Yavari, Pengfei Dong, Yalan Liang, Jared Soundy, James Drake, David Shriver, Mohammad-Ebrahim Mohammadi, Niloofar Mansoor, Zahre Sadri, Qi Xia, and Yang Liu who made my graduate studies a wonderful experience.

Last but not least, I would like to express my deepest gratitude to my Master, Mr. Li HongZhi, who teaches Falun Dafa, an advanced practice of Buddha school self-cultivation, to public and make practitioners including me and my mother greatly benefit from daily practicing. Through following his teaching "Truthfulness, Compassion, Forbearance", I become a better person everyday and gain significantly improvement both on mind and body. Human words are not enough to express my immense gratitude and appreciation to him. Without his teachings, I would not be able here. Thank you, Master Li!

Table of Contents

List of Figures	vii
List of Tables	viii
1 Introduction	1
2 Background	6
2.1 Alloy and Alloy Analyzer	6
2.2 Genetic Algorithm	12
2.2.1 Crossover	15
2.2.2 Mutation	16
2.2.3 Fitness	17
2.2.4 Selection	17
2.2.5 Termination Condition	19
2.2.6 Parameters Tuning	19
3 Related Work	21
3.1 Alloy extentions	21
3.2 Evolutionary algorithms	22
4 EvoAlloy	24
4.1 Motivation and Illustrative Example	25

4.2	Overview of EVOALLOY approach	28
4.3	Problem Representation	32
4.4	Fitness Function	33
4.5	Selection	34
4.6	Crossover	36
4.7	Mutation	37
5	Experimental Evaluation	38
5.1	Phase One	39
5.2	Phase two	43
5.3	Discussion	45
6	Conclusion and Future work	49
	Bibliography	51

List of Figures

1.1	High level view of lightweight formal methods	3
1.2	High level view of our approach	4
2.1	Overview of the main components of Alloy Analyzer	13
2.2	Overview of Genetic Algorithm	14
2.3	An illustration of one-point crossover	15
2.4	An illustrative example of mutation	16
2.5	An Overview of Tournament Selection	18
4.1	EvoAlloy’s (a) representation of a chromosome, (b) two produced chromosomes for the specification of Listing 2.1, (c) crossover step for creating a new chromosome, and (d) mutation step.	26
4.2	An Alloy model instance derived automatically from the chromosome shown in Fig. 4.1d.	28
4.3	Schematic view of EVOALLOY.	29
4.4	An example of unbiased tournament selection algorithm	35
4.5	An example of two point crossover algorithm	36
5.1	The population evolving diagrams of analysis on initial parameter settings	41
5.2	An example of unbiased tournament selection algorithm	43

List of Tables

5.1	The parameter configurations of Initial Settings and Final Settings . . .	40
5.2	The analysis time in second of tuning experiments over increasing mutation rate	42
5.3	The analysis time in second taken from Random (RD) over the increasing analysis scope across objects of study	44
5.4	The analysis time in second taken from EVOALLOY (EA) and Alloy Analyzer (AA) over the increasing analysis scope across objects of study . .	46
5.5	The number of iterations taken from EVOALLOY (EA) over the increasing analysis scope across objects of study	47

Chapter 1

Introduction

Software has become an intrinsic part of our daily life nowadays, and it has been successfully embedded in various devices with all kinds of purposes, ranging from transportation, communication, healthcare, and even home comfort. Yet at the same time, software including those considered life-critical, continues to fail. And software failures can be exploited by malicious users, consequently causing a variety of privacy and security issues [1]. Fifteen years ago, the National Institutes of Standards reported that an inadequate software quality infrastructure was costing the US upwards of \$59 Billion annually [2]. And similarly, another equally ominous report from Tricentis in 2017 estimated the annual financial loss due to software failures worldwide at \$1.7 Trillion [3]. These survey reports implied us that, as software products keep increasing their influence on almost every aspects of our life during the recent decades, the cost of software failures also follows this trend. To mitigate these ongoing threats, researchers in our software community have made great efforts to improve software engineering techniques, and to develop better software validation methods. Therefore over the last several decades, a large body of research works related to software verification and testing have been investigated, and new approaches such as machine learning have also been introduced to this area to ensure the quality of software products. However, the problems still persist. Recent highly publicized bugs

like the Toyota acceleration problem and the heartbleed bug as well as the explosion of Android exploits [4] show that there is no silver-bullet yet discovered — we are still lack of sufficient techniques to verify and validate our software.

One class of techniques that have been widely applied to tackle dependability problem, are those which fall into the category of formal methods. Leveraging mathematical concepts to rigorously model the entire system, formal method can precisely perform various verification tasks as well as proving the correctness of dependability properties. Most notably, lightweight formal methods, such as those based on bounded verification, have recently received a lot of attention due to their capabilities of conducting automated and formally precise analysis, which significantly reduce the burden on traditional formal techniques. Its applications spans a wide range of software engineering and security domains, including software design [5, 6, 7], code analysis [8], security analysis [4], test case generation [9, 10] and tradeoff synthesis and analysis [11, 12]. As shown in Figure 1.1, such kind of techniques often starts with a system specification and optional properties to be verified. By introducing user specified scope, bounded verification then transforms them into a finite satisfiability problem. Consequently, it delegates the task of solving the SAT problem to a constraint solver. And finally, the analysis is conducted by exhaustive enumeration over the bounded scope of specification instances.

Despite significant advances mentioned above, we still find ourselves lacking strong adoptions of formal techniques. The essential problem that prohibits them from being regularly applied in industry, lies in their scalability and applicability for large real-world systems. In other words, when the complexity of a software system increases to a certain extent, it is infeasible to thoroughly model and analyze the complete specifications of a entire system. Benefited from reduced analysis scope, bounded verification techniques are at once both sound and complete for the given bound.

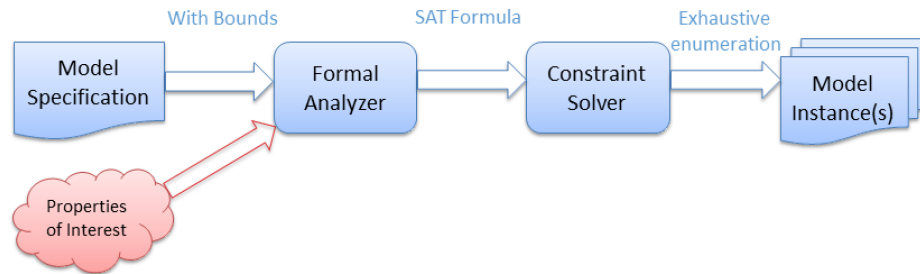


Figure 1.1: High level view of lightweight formal methods

However, the completeness still means that when performing analysis tasks on large systems, they either fail or need to be further reduced in scope. In software engineering community, an alternative approach to solving problems that grow exponentially has been to use search-based techniques, or more specifically evolutionary algorithms [13]. Basically, these algorithms successfully formulate different types of problems as optimization problems, therefore heuristically explore large complex solutions spaces and converge on single optimal solution, rendering them in sound but incomplete. There are numerous success stories for evolutionary algorithms. They are, for example, being used in test case generation[14], module clustering[15], and cost-effort prediction[16].

In this thesis, I present a novel tradeoff approach that provides a new path towards solving scalability issue. Our vision is that when the search space of a system’s specification is relatively small, we can keep using the full power of a constraint solver, and maintain both soundness and completeness. But when this approach fails as the scope exceeds its limitation, we switch on evolutionary algorithms [17], which in turn promise to scale to real-world large problems, and at the same time without sacrificing soundness.

To evaluate the feasibility of this new approach, I develop EvoAlloy, an extension to the existing lightweight formal analysis tool, Alloy Analyzer [18]. EvoAlloy skips exhaustive enumeration process conducted by the underlying SAT solver of traditional Alloy. Rather, it delegates the task of finding satisfiable instances to an alternative analysis engine using a genetic algorithm (GA), one of the most popular types of evolutionary algorithms, that have been demonstrated to be useful for pinpointing solutions in a large search space. At a very high level, as depicted in Figure 1.2, our genetic extension generates a population of candidate solutions as chromosomes derived from the bounded model. And it then search within the finite state space through iteratively evolving this population guided by the feedback about violated constraints, and eventually find a satisfiable solution. I have chosen the Alloy platform as an exemplar for our study, since it is a widely-used, open-source tool for modeling and analysis of software systems, and it has an active development community. Yet not surprisingly, it also suffers from exactly the scalability problems addressed by this work. The main contributions for this thesis are the following two aspects:

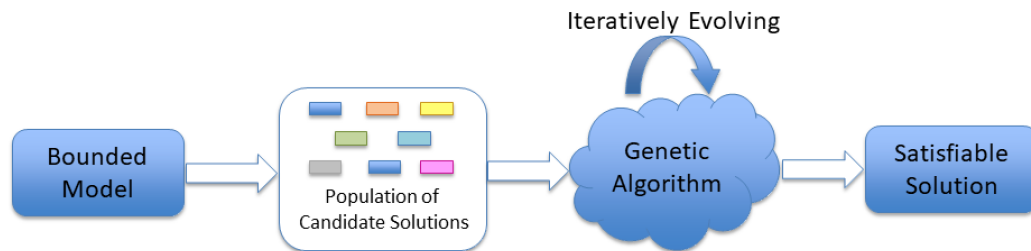


Figure 1.2: High level view of our approach

- I have implemented the prototype of our research artifact EvoAlloy, and make it publicly available to the research and education community [19].

- I have conducted preliminary experiments to evaluate our approach and compare it with traditional Alloy Analyzer. The experimental results prove the feasibility of our approach and denote that this direction of research is promising.

The remainder of this thesis is organized as follows. Section 2 provides the background of this work and Section 3 puts it in context with related efforts. Section 4 overviews our approach towards achieving a more scalable analysis technique. Section 5 presents the preliminary results obtained in our experiments and lesson I learned. Finally and Section 6 concludes this thesis with a summary of our contributions and our vision of future research.

Chapter 2

Background

In this chapter, I first provide the background of light-weight formal methods, more specifically, Alloy and its corresponding analysis tool Alloy Analyzer. And then I describe the basic concepts of Genetic Algorithm, one of the most popular search-based techniques.

2.1 Alloy and Alloy Analyzer

Formal methods are techniques that leverage mathematical notions and logical reasoning to rigorously model a complex system [20]. This precisely defined mathematical model can then be used to design software products or perform a variety of verification tasks to improve system reliability in a thorough fashion. On one side, formal methods can promise both precision and discipline through mathematical proof over formal, rigorous specified descriptions. On the other side, since rigor involved, the expense of formal approaches always makes it prohibitive to be commonly applied in industry. In essence, the underlying problem is mainly twofold: (1) the steep learning curve that formal methods require, and (2) the infeasible cost of full verification of an entire real world system. The first problem can be potentially solved by designing simple and expressive formal language with automated analysis tool, which

implies less burden for developers. Yet finding the solution to the latter one is not that straightforward. Researchers are still attempting to develop effective alternative options that can alleviate the exponentially growing expense of thorough analysis.

One direction towards the scalability issue is lightweight formal methods, more specifically, bounded verification, which has received lots of attention in the software engineering community over the last decades. The intuition behind the bounded verification is simple and straightforward: since the expense of all-encompassing formal analysis over the entire system is prohibitive, as a tradeoff, emphasizing partiality can reduce the computational efforts and maintain soundness and completeness within a finite scope. The idea is supported by "small scope hypothesis", which implies a high proportion of problems can be found within some small scope[21]. As one of lightweight formal methods, Alloy is a first-order relational logic with transitive closure[22]. It uses a lightweight object modeling notation to abstract structural properties of a software system. Due to its expressive power, Alloy, and its corresponding analysis engine, Alloy Analyzer, has been used to solve numerous problems in software engineering domain, including software design, code analysis and test case generation. The Alloy Analyzer is an automatic analysis engine built on top of state-of-the-art SAT solvers. In general, given a specification of a software system in Alloy, the Alloy Analyzer automatically analyze the software system's properties over user defined scope, specified in the form of predicates and formulas.

Listing 2.1 is an alloy specification for a simplified model of a file system. This specification is adopted from [18] , and it is published with the Alloy Analyzer. A typical Alloy specification mainly consists of three components: data types, formulas that specify constraints over data types, and commands to run the analyzer. In Alloy, essential data types are specified by signatures (**sig**). Similar to the concept of inheritance in object oriented language, a signature can be extended (**extends**)


```

1 abstract sig FSObject {}
2 sig Dir extends FSObject {
3   contents: set FSObject
4 }
5 sig File extends FSObject {}
6 one sig Root extends Dir {}
7
8 fact Hierarchy {
9   // Root has no parent
10  no contents.Root
11  // All FSObjects are reachable from Root
12  FSObject in Root.*contents
13  // Each FSObject has at most one parent
14  all obj: FSObject | lone contents.obj
15 }
16 pred model {
17   some File
18 }
19 run model for 2 File, 2 Dir

```

Listing 2.1: An Alloy specification example describing a simple model of file system.

as a subsignature/extension, while an **abstract** signature has no elements of its own type except those belonging to its extensions. Singleton, defined by using the keyword (**one**), is a special data type that can only have exactly one element. The relationships between signatures are captured by the declaration of *fields* within the definition of each signature. The running example defines 4 signatures (line1-6): File system objects, **FSObject**, which are partitioned into **Dir** and **File** types, with **Root** defined as a singleton extending **Dir**. The declaration of field **contents** specifies the relation that each **Dir** may have a set of content objects of type **FSObject**.

Facts (**fact**) are formulas that does not take any argument, and define constraints that every model instance of a specification must satisfy, which means they are expressions that enforced to be true. Basically, they restrict the instance space of the specification. The formulas can be further structured using predicates (**pred**) and

functions (**fun**), which are parameterized formulas that can be invoked. Formulas in Alloy are hierarchical, which indicates they might contain other formulas. The **Hierarchy** fact paragraph (lines 8–15) states that for a satisfiable instance, (1) the **Root** directory should not have any parent, and it cannot be a subdirectory for any other directory; (2) each single file and directory should be reachable from the **Root** directory; and (3) each file and directory belongs to at most one parent directory.

Analysis of specifications written in Alloy is completely automated, but bounded up to user-specified scopes on the size of type signatures. Particularly, to make the search space finite, we need to specify certain scopes which can limit the number of instances of each signature. In general, an Alloy command **run/check** can be used to invoke **predicate/assertion** to analyze the given model, through requesting the analyzer to search for instances/counterexample. An optional keyword "**expect**" is provided for explicitly specifying the satisfiability and unsatisfiability of the predicate being invoked, respectively, with **expect 1** and **expect 0**. Here in our example, through invoking **model predicate**, the **run** command (lines 16–19) asks the analyzer for finding instances that contain at least one **File**, and specifies a scope that bounds the search for specification instances with at most two elements for both **File** and **Dir** top-level signatures.

In order to analyze such kind of relational specification bounded by the specified scope, the next step for Alloy Analyzer is translating the bounded specification in Alloy into a corresponding finite relational model in Kodkod language [23]. Listing 2.2 partially outlines a Kodkod translation of Listing 2.1. A model in Kodkod’s relational logic consists of three parts: (1) a universe of elements (also called *atoms*), (2) a set of relation declarations including their lower and upper bounds specified over the model’s universe, and (3) a relational formula, where the declared relations appear as free variables [23].

```

1 {F1,F2,R1,D1}
2
3 Root:      (1,1) :: {{R1},{R1}}
4 File:      (0,2) :: {{},{F1},{F2}}
5 Dir:       (0,1) :: {{},{D1}}
6 contents:  (0,8) :: {{},{R1,R1},{R1,D1},{R1,F1},{R1,F2},{D1,R1},{D1,D1
      },{D1,F1},{D1,F2}}
7
8 (all o: Root + Dir + File | !one (Dir.contents.o)) && ...

```

Listing 2.2: Kodkod representation of the Alloy module of Listing 2.1.

As shown in Listing 2.2, the first line declares a universe of four uninterpreted atoms (F1,F2,R1,D1). Here in this chapter, I arbitrarily choose an interpretation of atoms, where the first two (F1 and F2) represent **File** elements, the next one (R1) represents a **Root** element, and the last one (D1) represents a **Dir** element. To be noticed that, as I explain in the next paragraph, all relations in Kodkod are untyped, and the abbreviated atom names are just chosen for readability, but do not indicate type.

In general, each Kodkod relation declaration defines the arity of a relational variable and bounds on its value. In our example, lines 3-6 state relational variables (Root,File,Dir,contents). Comparable to Alloy, formulas in Kodkod are constraints that are specified over relational variables. One main difference between Alloy specification and Kodkod model with respect to relational variables is that, in Alloy they are divided into two types, *signatures* which represent unary relations and *fileds* that represent non-unary relations, yet in Kodkod all relations are untyped, which means there is no difference between unary and non-unary relational variables.

We can further specify a scope over each relational variable in Kodkod from both below and above by two *relational constants*, using upper and lower bounds, respectively. In principle, each relational constant is a pre-defined set of tuples drawn from a universe of atoms. Consider the declaration of signature **Root** in line 3, both its

```

1 //model instance 1
2 Root:      {{R1}}
3 File:      {{F1}}
4 Dir:       {{}}
5 contents:  {{R1,F1}}
6
7 //model instance 2
8 Root:      {{R1}}
9 File:      {{F1}}
10 Dir:      {{D1}}
11 contents:  {{R1,F1},{R1,D1}}

```

Listing 2.3: Two arbitrarily selected instances for the specification of Listing 2.1.

lower and upper bounds contain only one atom $R1$, as it is defined as a singleton in Listing 2.1, a special type of signature aforementioned. The upper bound for the variable $contents \subseteq Dir \times FSObject$ (line 6) is a product of the upper bound set for its corresponding domain and co-domain relations, taking every combination of an element from both and concatenating them. Essentially, for each relational variable, the lower bound contains the tuples that each relation in a model instance must include, whereas the upper bound holds the whole set of tuples which a relational variable's value may contain in an instance. An illustrative example is that, the lower bound for relational variable `File` is empty set $\{\}$, and its upper bound is $\{\{F1\},\{F2\}\}$. Therefore, the possible values that could be assigned to `File` are $\{\},\{F1\},\{F2\}$ and $\{\{F1\},\{F2\}\}$. In Kodkod, formula constraints are in the form of a conjunction of several sub-formulas, i.e., $F = \wedge_{subformulas}$. As an example, the formula at the last line of Listing 2.2 represents this form for the constraints specifications in our running example. The relation values of a valid solution to a Kodkod model must satisfy every sub-formula in F .

Generally, the Kodkod engine then translates kodkod relational model into Compact Boolean Circuit, which is a boolean logic. With this boolean formula, it further transforms the problem into a CNF SAT formula. And ultimately, Kodkod's model

finder leverages off-the-shelf SAT solvers to search satisfiable instances of the CNF formula. Figure 2.1 outlines the main steps that Alloy Analyzer takes when performing analysis on a given system's specification. In essence, the analyzer explores within upper and lower bounds that defined for each relational variable to find solutions to a formula, which are bindings of the formula's relational variables to relation constants that makes the formula true. Listing 2.3 shows two different instances for specification of Listing 2.1 found by Alloy Analyzer. Basically, a model instance can be viewed as an exact bound, where the lower and upper bounds are the same set of tuples.

2.2 Genetic Algorithm

During the last decades, searched-based techniques are increasingly applied to addressing a variety of software engineering problems including test case generation[14], cost-effort estimation[16], module clustering[15] and etc. For these large and complex problems that grow exponentially, traditional techniques such as model checking [24] are not suitable due to their scalability. As an alternative approach, searched-based techniques, more specifically evolutionary algorithms, provide researchers a more efficient method to tackle these hard problems. In general, evolutionary algorithms reformulate various type of problems as optimization problems, thus use heuristic search to explore large complex state spaces and converge on single or partial global optimal solution(s). Normally, several different types of evolutionary operators are adopted during the evolutionary process and the search is guided by a fitness function that can differentiate between better and worse candidate solutions. As a tradeoff approach, evolutionary algorithms can generate sound solutions to the problem, but completeness is often sacrificed. Due to their efficiency and effectiveness, evolutionary

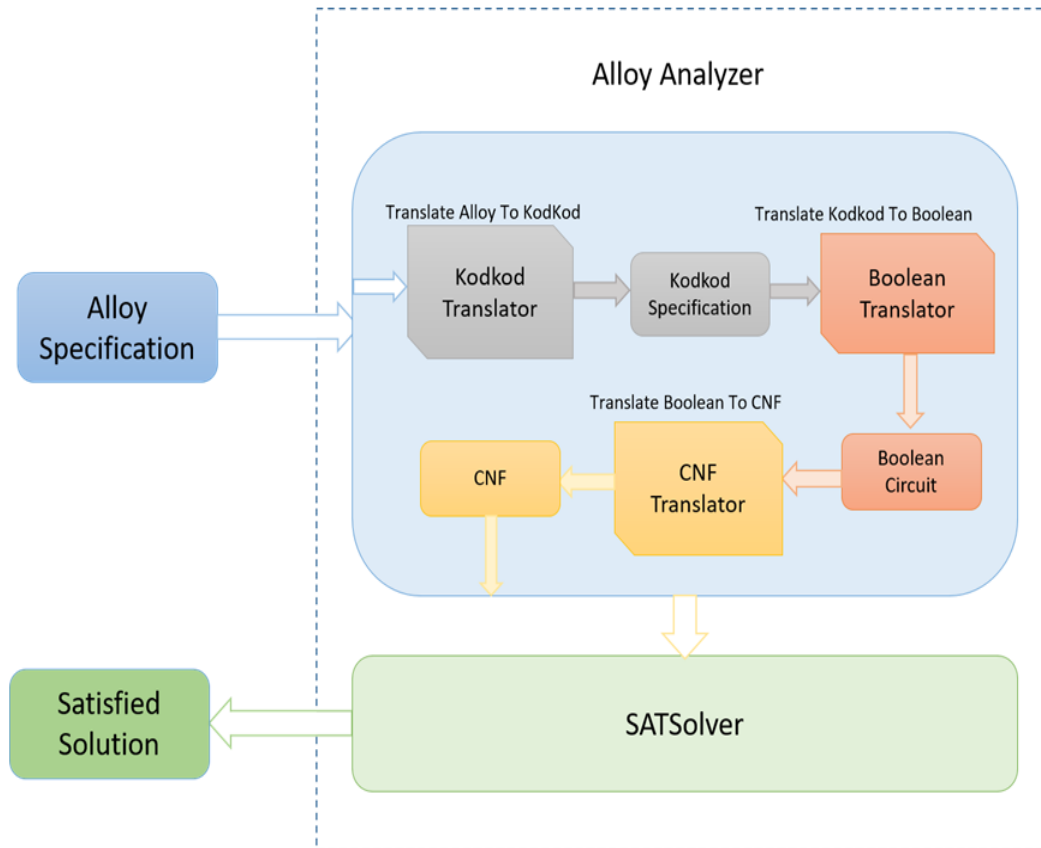


Figure 2.1: Overview of the main components of Alloy Analyzer

algorithms have been successfully applied for developing numerous research works in software engineering[14, 25, 26].

As one of the most widely used types of evolutionary algorithms, genetic algorithms are inspired by theory of biological evolution. In principle, they are meta-heuristic optimization techniques that emulate the process of natural genetic variation and selection into a computational problem [17]. Generally, a basic genetic algorithm starts with a randomly or manually created set of candidate solutions, where every one of them could be a potential instance to the problem we expect to solve[27]. Each candidate solution is represented as a *chromosome*, a.k.a. an *individual*, consisting of

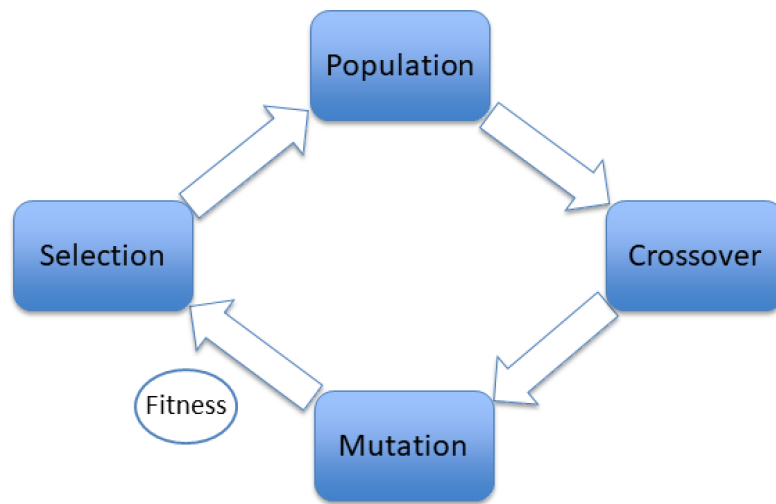


Figure 2.2: Overview of Genetic Algorithm

a set of *genes*. Here if the representations of genes are continuous, they are named with *vector*, otherwise, they are called bit strings. And each gene has a domain of values called *alleles*. Then there are two main genetic operators *crossover* and *mutation* that are often involved in the evolutionary process. Crossover operator combines two or more parent chromosomes to produce new offspring chromosomes. And mutation operator just simply mutate some randomly picked genes in the population. The better chromosomes that have been generated are then selected for the next generation based on fitness. The genetic algorithm will keep evolving this population of chromosomes through these processes, until one of the termination conditions is satisfied, which is normally the optimal solution has been found or the specified resources have been all consumed. Figure 2.2 outlines this iterative evolutionary process of a classic genetic algorithm.

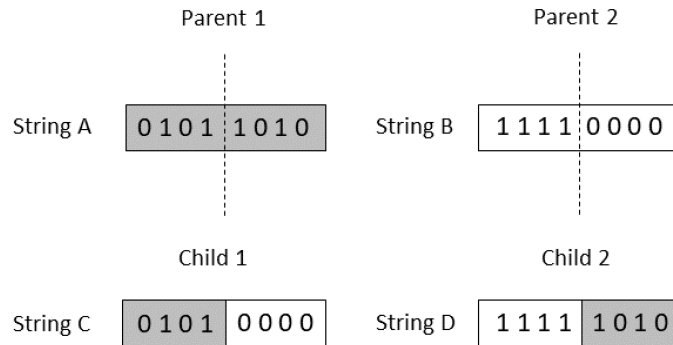


Figure 2.3: An illustration of one-point crossover

2.2.1 Crossover

Crossover operator implements a mechanism that generates new chromosomes by mixing the genetic makeup of two or more parent chromosomes. While in nature most species have only two parents, some variants of genetic algorithms extend the crossover to more than two parents. In general, the first step of crossover is picking pairs of chromosomes as parents. As mating in biology, the pairs of parents then combine their genes and inherit them to their offspring.

Figure 2.3 shows a simple example for one-point crossover. String A (01011010) and B(11110000) are two parents chromosomes we selected, each consisting of 8 bits. Using one-point crossover we can randomly pick position four as *crossover point*, cut them into four pieces, and recombine these four bit strings to produce two children String C(01010000) and D(11111010). For different optimization problems, we can easily extend it to two or more points crossover. Analogy to evolution in nature, the main benefit of applying crossover operator is to inherit parts of genetic makeup from successful parents such that it might get a higher chance to generate even better offspring.

2.2.2 Mutation

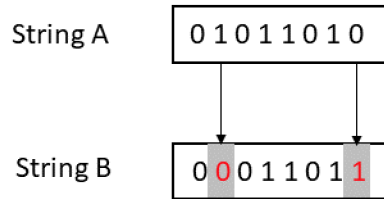


Figure 2.4: An illustrative example of mutation

Another commonly used genetic operator is mutation, which randomly change parts of the chromosome with a probability threshold called mutation rate. Analogy to genetic mutation in nature, mutation is normally applied to a small portion of genes within the entire population and the mutation rate varies from one problem to the other. The motivation of applying mutation operator is to keep a diverse population of chromosomes by generating new genes, such that it can get the opportunity to reach the parts of the solution space which never reached before. Theoretically, mutation is one of the most effective way to avoid getting stuck on local optima and plateaus. The three main principles for mutation operator are reachability, unbiasedness and scalability [27]. Reachability indicates that the mutation should provide the chance to reach every part of the search space. Unbiasedness means mutation should not guide the search to a particular direction, at least in the solution spaces that is unconstrained and without plateaus. And the third principle, scalability requires that mutation need to provide certain flexibility with respect to mutation rate.

A simple example of mutation is depicted in Figure 2.4. Here, it first selects chromosome String A (01011010) out of the entire population. Then it randomly chooses position two and eight and alters their values, thus a new chromosome String

B (01011010) is generated. To notice that, since in this example the two chromosomes are bit strings, which means that the alleles of each gene only contain 0 and 1, thus the mutation operator could only flip the values of each gene from 0 to 1 or the opposite way. In practice, the new values generated by mutation would have more choices when the range of alleles is larger.

2.2.3 Fitness

After finishing crossover and mutation, a new generation of population is produced and each chromosome must be evaluated in terms of its ability to solve the problem. In order to measure the quality of chromosomes, a fitness function need to be build. Designing a fitness function is a non-trivial task since the value of fitness is directly used to distinguish candidate solutions between better and worse, thus guide the search towards the global optimal solution(s). More specifically, we need to carefully choose both appropriate penalty function for the infeasible candidates solutions and the weights of multiple objectives, so that the values calculated by fitness function can accurately reflect the exact distance between candidates solutions and valid solutions. For multiple objectives problems, there are several well-known design techniques including Non-dominated Sorting, Crowding Distance [28] and etc. For the sake of simplicity, here I do not discuss them in detail.

2.2.4 Selection

By using selection operator, the best chromosomes are selected as parents in the next generation of population. This mechanism ensures the convergence of the search towards optimal solutions. For this purpose, the selection process generally picks the elitism of the population based on fitness values, keeping the optimal genes for potentially producing better offspring chromosomes in the next generation. The chro-

mosomes with higher fitness values are preferred in maximization problem, while it is the opposite case in minimization problem. One of the simplest selection algorithm is ranking selection/elitist selection that directly selects a certain number of elite candidate solutions after ranking the entire population based on fitness values.

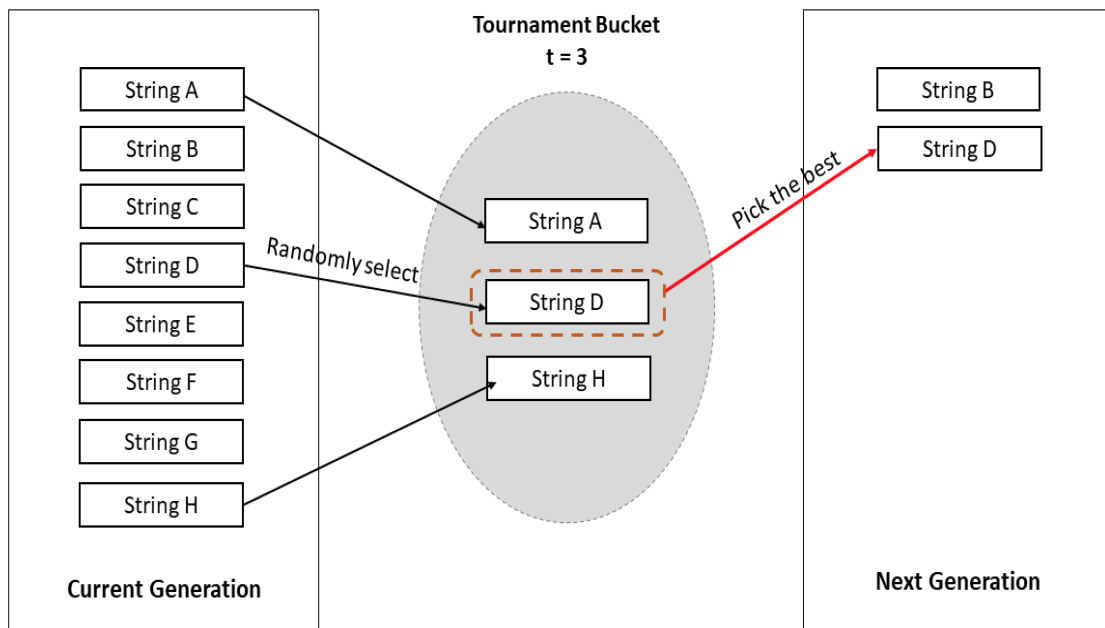


Figure 2.5: An Overview of Tournament Selection

To avoid the search easily converging to local optima and getting stuck at plateaus, selection algorithms that allow certain degree of randomness are commonly applied in practice. Figure 2.5 depicts the overview of the *tournament selection* [29]. Basically, tournament selection uses a mechanism called tournament bucket with a specific size t , and for each time t individuals are randomly chosen from the current generation of population as they are put into the bucket. The best chromosome in the tournament bucket is then selected to be the parent in the next generation. This process will be

iteratively executed until all the parent individuals for the next mating pool have been selected. In practice, the tournament bucket size is a key factor that can significantly affect the genes of next generation. Since when the size t is larger, there is a smaller chance for worse chromosomes to be chosen and vice versa. For certain optimization problems, there might be a need to keep some worse individual in the pool to achieve the diverse of the population. And another reason to randomly bring some bad chromosomes to the next generation is that they might contain some genes better than the others', that can contribute to producing better offspring.

2.2.5 Termination Condition

As aforementioned, the evolutionary algorithm is a iterative process that explores the solutions space, thus certain end criterion need to be specified to ensure that the search will be eventually terminated. These end criterion are called termination condition in genetic algorithm. Normally, limited resources including, running time, number of generations are reasonable predefined conditions to restrict the length of search process. Another commonly used termination condition is convergence of the evolutionary process, which implies the search stops when approximating the optimal solution and no significant fitness improvements are further made. And for certain types of problems, the most straightforward criteria is the global optimal solution is found.

2.2.6 Parameters Tuning

The choice of parameter settings is one of the most important foundations that decides if evolutionary algorithm can be successfully applied to certain problems. In fact, finding the optimal parameter choices itself is also a challenging optimization problem. Normally, parameter tuning is the initial step in research problem for ge-

netic algorithm. While there are many static settings were proposed, i.e. mutation rate = 0.1 in bit flip mutation, no optimal configurations are found that can generally solve all problems [27]. This means every time before applying genetic algorithm to a new problem, the process of searching appropriate parameter configurations has to be developed. There are several different kinds of parameter tuning methods, including latin hypercube sampling, simple grid search and etc [27]. To applying proper tuning strategies, experts with domain knowledge, good sense of guesses and estimations can also contribute to finding the optimal parameters settings efficiently.

Chapter 3

Related Work

In this chapter, I provide a discussion of the most related research efforts in light of my study from two areas: Alloy extensions and evolutionary algorithms.

3.1 Alloy extensions

The widespread use of Alloy has led to a significant number of extensions to its underlying automated analyzer [30, 31, 32, 33]. Many research works have been conducted for exploring model instances from Alloy's relational logic constraints [34, 35, 36, 37]. Nelson et al. present Aluminum [32], an extension to Alloy Analyzer, that defines a relational model instance as *scenario*, thus generate the minimal scenarios through selection and partitioning. It relies on a procedure in which tuples are iteratively removed from the tuple set of found scenarios until a minimal scenario is reached. Macedo et al. [35] explores the space of scenario exploration operations by formulating them using relational logic. Rather than facilitating the exploration of the space of solutions for evolving models, their work focuses on the order of exploring model instances. Montaghani and Rayside [31] developed an extension to Alloy that explicitly supports specification of partial models.

Other relevant works target on improving Alloy's performance from various di-

rections. Uzuncaova and Khurshid [38] divide a specification into base and derived slices, in which a solution to the base slice can be extended to produce a solution for the entire specification. Rosner et al. [36] present a technique, Ranger, that leverages a linear ordering of the solution space to support parallel analysis of first-order logic specifications. These techniques rely on leveraging multiplicity of computing to improve the efficiency of the Alloy analyzer. Bagheri and Malek present Titanium [33], that leverages the results from previous analysis narrow the state space of the revised specifications, thus significantly reducing the required computational efforts when applied to analysis tasks of evolving systems. Ghazi et al. [39] propose their approach that by replacing the underlying SAT Solver with SMT Solver, it can virtually prove the correctness a property without the restriction of a bounded scope. They develop a mechanism to translate alloy to SMT-LIB their idea of using SMT Solver also extends the Alloy’s capability of handling of arithmetic expressions and supporting for numerical constraints. Different with all the works above mentioned, EVOALLOY is geared towards the application of genetic algorithms to foster exploration of large, complex solution spaces.

3.2 Evolutionary algorithms

There is a large body of research works on using evolutionary algorithms to solve software engineering problems [13]. EVOALLOY falls within this class of solutions. Much work is about applying evolutionary algorithms to software testing and test case generation[40, 41, 14, 42]. Among others, One of the exemplar approaches, EvoSuite, uses a genetic algorithm to generate JUnit test cases for Java classes [14]. K. Inkumsah and T. Xie [42] proposed Evacon framework that integrates symbolic execution and evolutionary testing to improve coverage of structural testing. There are

also other techniques that have been developed for solving other software engineering problems based on evolutionary algorithms. Targeting on automatic software repair, C. Le Goues et al. [43] presents Genprog, that iteratively evolves a program variant by using genetic programming to repair a particular defect without loss of functionality. Through restricting the operation at statement level and reusing existing program statements, their technique succeeds to automatically repair large scale programs with both quality and efficiency. Thomé et al. develops ACO-Solver to solve large complex string constraints. Their approach employs a hybrid constraint solving procedure based on the Ant Colony Optimization, that achieves significantly improvement in terms of vulnerability detection when combining with two other off-the-shelf constraint solver[25]. Dings and Agha designed Concolic Walk algorithm, that combines linear constraint solving with tabu search, another popular evolutionary algorithm, to solve complex arithmetic path conditions[26]. Through mixing heuristic search and symbolic reasoning, their approach successfully generates tests with higher coverage and efficiency. The work conducted by Godefroid and Khurshid [44] is perhaps the most closely related work to ours. It uses a genetic algorithm to guide a search in the analysis of concurrent reactive systems towards errors like deadlocks and assertion violations. In contrast with all of this prior work, the problem addressed in this thesis is bounded analysis of large-scale solution spaces specified in relational logic. Among other things, it requires the development of both original chromosome encodings and fitness functions appropriate for models specified in Alloy's relational logic. To the best of our knowledge, EVOALLOY is the first evolutionary technique for automated analysis of bounded relational logic specifications.

Chapter 4

EvoAlloy

In this chapter ¹, I present our novel approach, EVOALLOY. The specific goal of designing this mechanism is to prove that by leveraging the power of evolutionary algorithms, it is feasible to improve the scalability of formal method tool, i.e. Alloy, to handle various analysis tasks for large complex software systems. To this end, I develop EVOALLOY, an extension to traditional Alloy Analyzer, that delegates the model finding process currently performed by computationally expensive constraint solvers to an efficient analysis engine based on genetic algorithm. Working at the level of finite relations generated by Kodkod, our EVOALLOY engine can efficiently create and evolve a population of candidate solutions iteratively, and eventually converges to single satisfiable solution, in the meantime only consumes limited resources.

This chapter is organized as follows. I first provide motivation of my study and an illustrative example. I then overview the high-level idea of my approach in Section 4.2. In Section 4.3, I formally define the genetic representation of the problem. Section 4.4 presents the fitness function. Finally, Section 4.5, 4.6 and 4.7, describes the main evolutionary operators I implemented in this work, including selection, crossover and mutation, respectively.

¹The approach described in this chapter has been presented in my published paper "An Evolutionary Approach for Analyzing Alloy Specifications" [45]

4.1 Motivation and Illustrative Example

In Section 2.1, I have introduced the principles of Alloy and its corresponding analysis tool Alloy Analyzer using an example that describes the model finding process for a simple piece of Alloy specification. As aforementioned, because of its significant advances, Alloy has been a popular analysis tool for solving a variety of software problems and it has an active development community. However, in practice constraint-solving techniques that traditional Alloy Analyzer relies on, continues to be a bottleneck when applied to various analysis tasks. To gain further confidence in the correctness of their system’s specification, Alloy users must re-analyze them with larger and larger scopes. Yet, the cost of the constraint-solving technologies underlying Alloy is exponential in those bounds, thus prevents further analysis beyond only trivial bounds. The magnitude of formulas tends to increase exponentially in the size of the system to be analyzed, making it less practicable to employ constraint solving in analyzing realistic complex systems. An open problem to us as software engineering researchers is that, we need to develop certain mechanisms that can facilitate efficient application of formal analyzers in rapidly growing domain of software systems. Therefore, our EVOALLOY, an analysis engine that bypass the computational heavy constraint solver based on genetic algorithm, is inspired by this ongoing demand.

Utilizing the bounded relational model in Kodkod to generate a population of candidate solutions, EVOALLOY successfully reformulates the Alloy model finding problem as optimization problem, such that it heuristically explores the entire search space to find a satisfiable solution through iterative evolving the population. Figure 4.1a delineates a genetic representation of the problem. In general, a candidate solution to a system’s specification is represented as a chromosome. Each chromosome

contains a gene for each relational variable within the specification under analysis. The domain of values of each gene, namely, the alleles, are defined as a set of tuples drawn from a universe of uninterpreted atoms within the upper and lower bounds defined for each relation (Listing 2.2, lines 3-6).

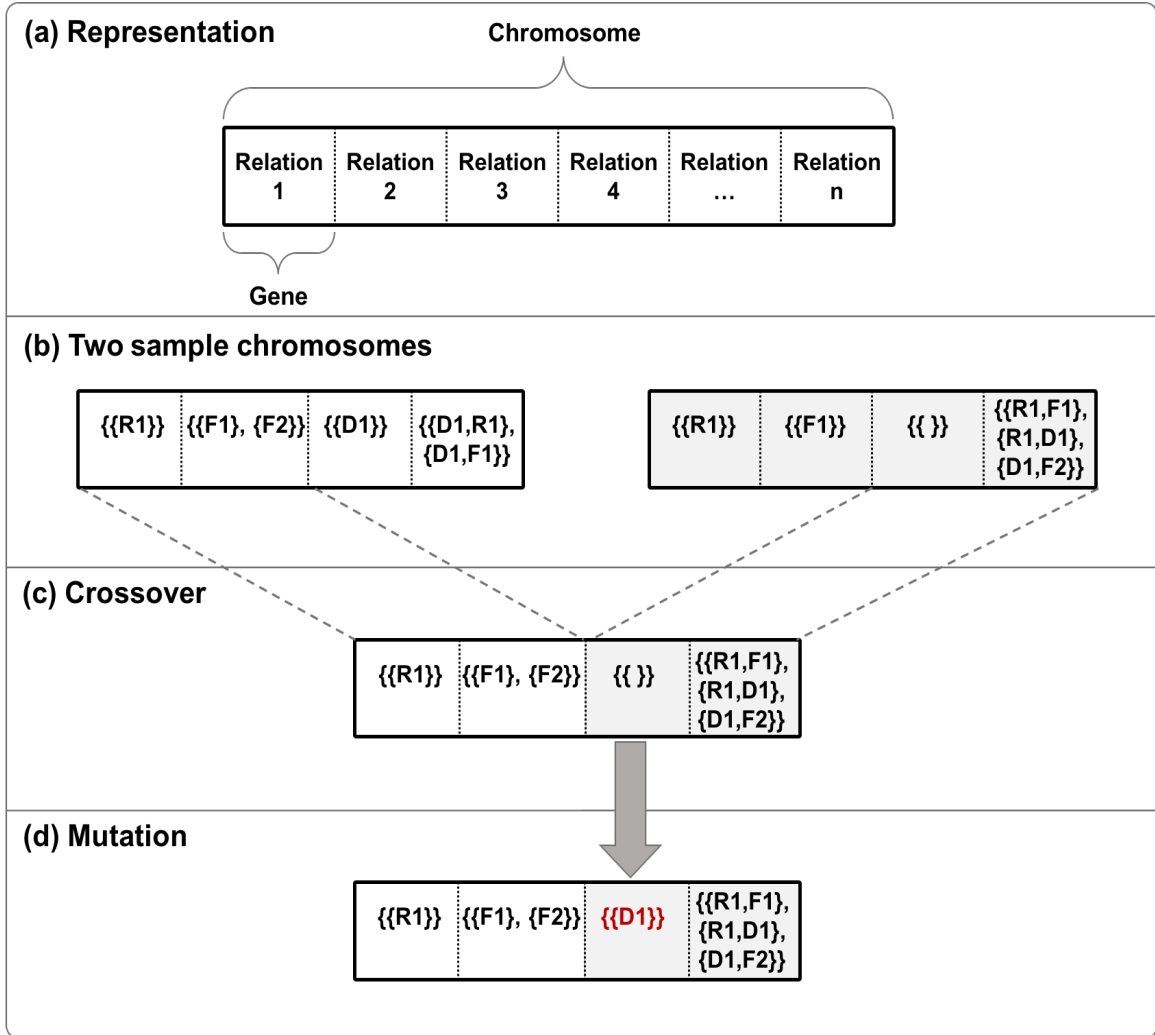


Figure 4.1: EvoAlloy's (a) representation of a chromosome, (b) two produced chromosomes for the specification of Listing 2.1, (c) crossover step for creating a new chromosome, and (d) mutation step.

Similar to classic genetic algorithm, our genetic extension starts with an initial population of randomly created chromosomes. Figure 4.1b demonstrates two sample

chromosomes generated for our Alloy example described in Section 2.1. Each chromosome in this case consists of four genes that correspond to the specification's relations, i.e., `Root`, `File`, `Dir`, and `contents`, from left to right, respectively. Our evolutionary search mainly employs two types of operators, i.e. crossover and mutation. Generally, crossover between two or more selected parent chromosomes is carried out to breed new chromosomes. Figure 4.1c represents EVOALLOY's crossover step for producing offspring. In EVOALLOY the recombination of the two parent chromosomes creates two new chromosomes. For the sake of simplicity and as it suffices to make the idea concrete, here I just demonstrate one offspring chromosome. The diagram shows a single-point crossover operator. Generally, it just randomly picks certain relation within the middle range of a chromosome as crossover point, then splits each parent individual into two pieces and mix them to produce the children. Here in the example, the crossover operator selects the cutting point before relation `Dir`, and generate the offspring shown in the diagram by combining the first two genes from *Parent 1* and the last two genes from *Parent 2*. EVOALLOY also provides configurable options for crossover operator including two-points crossover and all-points crossover, and it is possible for us to effectively exploit other types of crossover operators. For mutation in EVOALLOY, some genes in the population will be mutated using a given mutation rate. Figure 4.1d illustrates applying a mutation operator to a chromosome that gives rise to a randomized change in the chromosome. As shown in the example, mutation randomly selects the gene `Dir` as one of the mutation point and alter its value from an empty tuple set to `D1`. In practice, mutation randomly selects a percentage of genes in the population and modifies each by assigning a different tuple set from within that gene's domain.

For the purpose of calculating the fitness, EVOALLOY relies in part on the Kodkod analysis engine to get the relations that fail within each chromosome along with the

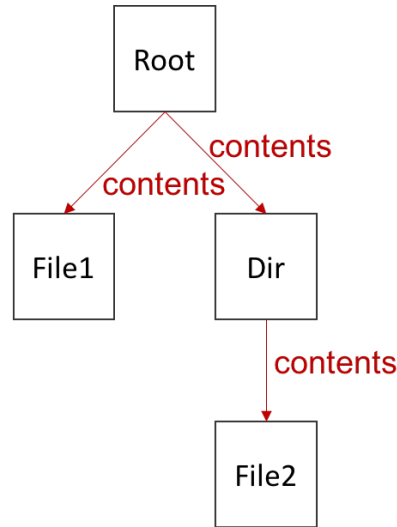


Figure 4.2: An Alloy model instance derived automatically from the chromosome shown in Fig. 4.1d.

number of failed subformulas. Based on the fitness values, `EVOALLOY` chooses the best candidate solutions for the next generation with a variant of tournament selection called unbiased tournament selection, guiding the search towards those solutions that have no violations of the constraint formula. In principle, the evolutionary search using genetic operators above mentioned is iteratively carried on up to an identification of a satisfying solution or an termination condition is encountered. Figure 4.2 illustrates a satisfiable model instance automatically derived from the chromosome shown in Fig. 4.1d using `EVOALLOY`. Basically, it describes a valid file system instance that, it has a directory `Dir` and a file `File1` under the `Root` directory. And `Dir` also contains a content file, which is `File2`.

4.2 Overview of EvoAlloy approach

The key idea of our approach that towards achieving better scalability is to somehow replace the intractable part of the existing Alloy Analyzer. For this purpose, I choose genetic algorithm as an alternative technique since they are exceptionally success-

ful in pinpointing solutions in a large search space. Figure 4.3 shows an overview of EVOALLOY, and explains how it bypass the computationally-heavy SAT solver underlying the traditional Alloy Analyzer.

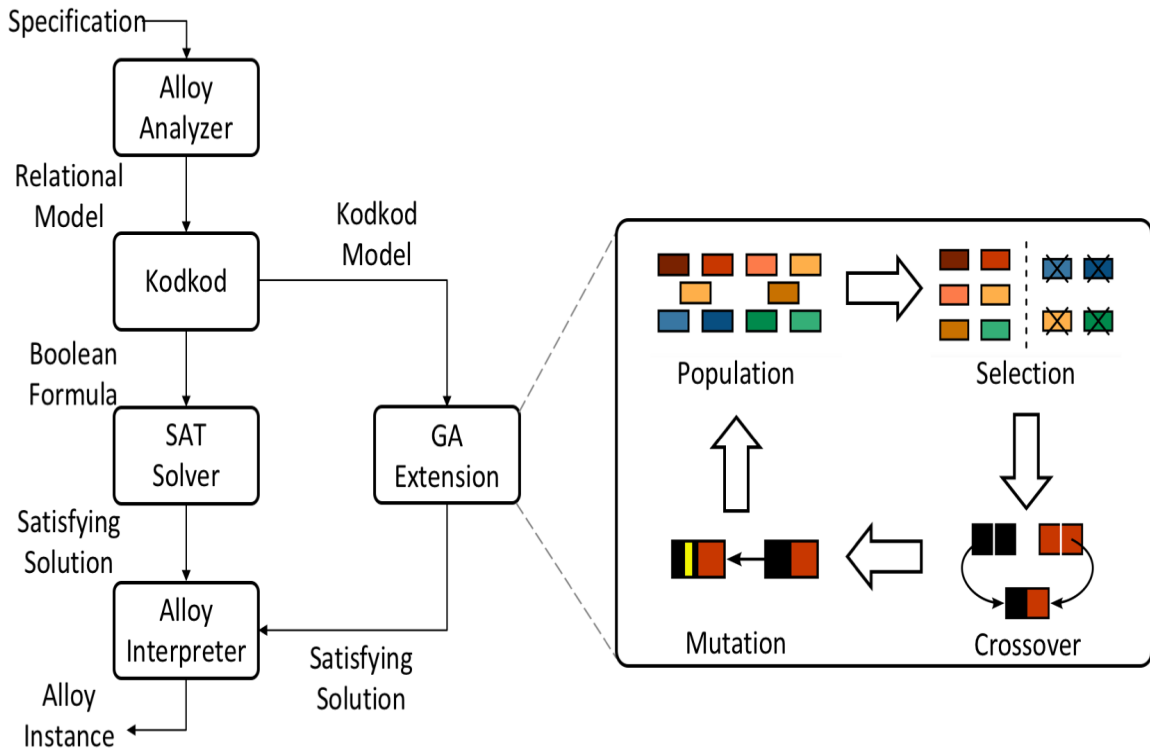


Figure 4.3: Schematic view of EVOALLOY.

The left side of Figure 4.3 depicts the flowchart of Alloy Analyzer at very high level. Basically, the Alloy Analyzer first reads in an Alloy specification and translates it into a relational model. Then it passes that model through Kodkod engine (a finite relational model analyzer) [23]. For each relation, Kodkod uses the scopes and signature bounds from Alloy, and concretizes these to bound the problem specification. The use of Kodkod in Alloy has already provided scalability beyond its original implementation, because it can help reason about partial models. To transform such a

finite relational model into a Boolean logic formula, Kodkod renders each relation as a Boolean matrix, in which any tuple within the bounds of the given relation maps to a unique Boolean variable [37]. Relational constraints are then captured as Boolean constraints over the translated Boolean variables. It then translates the resulting Boolean formula to CNF (Conjunctive Normal Form), and passes the CNF formula to an off-the-shelf SAT solver to obtain a solution. Last, the result produced by the SAT solver is translated by the Alloy interpreter into a solution instance.

I decide to utilize the bounded relational model in Kodkod to make the genetic algorithm scalable mainly for two reasons. First, applying the genetic algorithm on the Kodkod level, rather than the higher Alloy level, is more efficient. Since Kodkod relational model is both tightly bounded and partial represented, the space of concrete instances that need to be explored by the search engine is thus limited. Second, translating a Kodkod model to a propositional formula and then to CNF formula introduces many auxiliary variables [23, 32], which will increase extra computational cost due to the expensive transformation process. In fact during the experiment, other than the underlying SAT Solver, I find the transformation from boolean formula to CNF is another bottleneck that prevents the Alloy Analyzer from analyzing complex specifications. The explosion in the number of variables can also largely affect the scalability of the genetic algorithm approach.

Therefore, our GA extensions is inserted between Kodkod and the Alloy interpreter, as depicted in Figure 4.3. At the highest level, EVOALLOY’s GA extension takes in the bounded relational model generated by Kodkod, and outputs a satisfying solution to the Alloy interpreter. The box at right describes the steps EVOALLOY follows to do this.

Algorithm 1 delineates the GA employed in our work. The *initial population* of our chromosomes is made up of random assignment of values to each relational variable,

Algorithm 1 The genetic algorithm applied in EVOALLOY

```

1:  $Pop_{current} \leftarrow$  generate random population
2: repeat
3:    $Pop_{new} \leftarrow elite(Pop_{current}, e)$ 
4:    $P', P'' \leftarrow permute(Pop_{current})$ 
5:    $i \leftarrow 0$ 
6:   while  $|Pop_{new}| \neq |Pop_{current}|/2$  do
7:      $Pop_{new} \leftarrow Pop_{new} \cup select(P'[i], P''[i])$ 
8:      $i \leftarrow i + 1$ 
9:   end while
10:  while  $|Pop_{new}| \neq |Pop_{current}|$  do
11:     $p1, p2 \leftarrow pickParents(Pop_{new})$ 
12:     $\langle c1, c2 \rangle \leftarrow crossover(p1, p2, prob_{crossover})$ 
13:     $Pop_{new} \leftarrow Pop_{new} \cup \{c1, c2\}$ 
14:  end while
15:   $j \leftarrow 0$ 
16:  while  $j \neq |Pop_{new}| * rate$  do
17:     $c_{old} \leftarrow pickChromosome(Pop_{new})$ 
18:     $c_{new} \leftarrow mutation(c_{old}, Prob_{mutation})$ 
19:     $replaceChromosome(Pop_{new}, c_{old}, c_{new})$ 
20:     $j \leftarrow j + 1$ 
21:  end while
22:   $Pop_{current} \leftarrow Pop_{new}$ 
23: until solution found OR maximum resources spent

```

from within the legal relations and their bounds. As mentioned in Section 2.1, The scope of each relational variable in Kodkod is defined by two relational constants, called *upper* and *lower* bounds, respectively. The upper bound represents the whole set of tuples that a relational variable may contain, and a lower bound represents a partial solution for a given model. Every relation in a satisfying solution, thus, must contain all tuples in the lower bound, and no tuple that is not in the upper bound. In the initial population, I randomly assign a value to each relation from within the specified bounds. To be noticed that, here by assigning a concrete value to a specific

relation, it means that its upper and lower bounds are set to be equal. In practice, a random relational value generator is implemented to create the equal bound by arbitrarily selecting a set of tuples from the upper bounds of this relation. Essentially, each chromosome within the population represents a *potential* Alloy instance with exact bound for each relation.

Fitness is measured by assessing the chromosome and monitoring how close it gets to satisfying constraints of the target specification. To verify each individual, I employ the APIs provided by the Kodkod model finder; it also has a built-in ability to identify a minimal unsatisfiable core when the individual does not satisfy the specification constraints. Essentially, if any constraint is omitted from the identified core, the resulting set of constraints would be satisfiable. Thus it provides us a good measurement to distinguish between better and worse individuals, i.e. a candidate solution with less violated constraints and less relation variable involved should be closer to a satisfiable solution. With each subsequent iteration, I generate new offspring chromosomes through combining chromosomes selected with a likelihood proportional to their fitness value as parent, and then altering the resulting ones with mutation operators (e.g., arbitrarily change some of its tuples). And eventually when satisfiable solution is found or certain termination criterion is encountered (e.g., maximum resources all spent), the evolutionary cycle stops and the Alloy interpreter render the result as Alloy instance.

I describe the details of EVOALLOY in the followings sections.

4.3 Problem Representation

The initial step in developing any evolutionary algorithm is to decide on a genetic representation of a candidate solution to the problem. This involves defining a chro-

mosome and the mapping from it to the original problem context. As I illustrate in Section 4.1 with an example shown in Figure 4.1, in EVOALLOY, a potential model instance to the problem, namely a chromosome, is represented as a vector, where each index in the vector denotes a gene. It can be seen as a tuple-string of length n , where n is the number of relations within the problem specification. Each single gene then refers to the value assignment of exactly one relational variable. Given an Alloy specification S , I formally define a function $f_S : Relation(S) \rightarrow \mathbb{N}$ that maps each relation r of the specification S into a vector index assigned to that relation. Similarly, I define $f_S^{-1} : \mathbb{N} \rightarrow Relation(S)$ as a function that maps a given vector index to the relation it represents. To be noticed that, this representation of a chromosome has a fixed size for a given problem, which is determined by the number of relations within the problem specification under analysis. And it further influences variation operators, i.e., crossover and mutation.

4.4 Fitness Function

The fitness function is a decisive factor of evolutionary algorithms. Essentially, it measures the solution-quality of a chromosome, and acts as a means to differentiate chromosomes in proportion to the extent of their contribution to a solution. Specifically in EVOALLOY, when designing the fitness of chromosome, I mainly consider two main factors: Formula constraints (c_i), i.e., subformulas, and relations (r_i). Accordingly, the fitness function of a chromosome $chrom$ is expressed as follows:

$$f(chrom) = \sum_{c_i \in Consts} T_c(c_i, chrom) + \sum_{r_i \in Rels} T_r(r_i, chrom)$$

In this formula, $T_c(c_i, chrom)$ equals one if c_i is not satisfied by $chrom$; and it evaluates to zero otherwise. Similarly, $T_r(r_i, chrom)$ equals one if r_i is violated by

chrom; it is assigned to be zero otherwise. This representation of fitness function implies that I mapped the Alloy model finding problem as a minimization problem. The better a candidate solution is, the more constraints and relations are satisfied, and the lower the fitness value will be. When a chromosome for a given specification satisfies all its constraints defined over its relational variables, I identify it as an ideal chromosome with a fitness score of 0. The fitness function establishes truth-invariance, as the Alloy specification is satisfied provided that all the relations and formulas thereof are satisfied.

4.5 Selection

The Algorithm on lines 3–9 explains the process by which EVOALLOY selects chromosome variants to pass to the next generation. Basically, I implement a selection mechanism that leverages both elitism and unbiased tournament selection algorithm [46] to select half of population members in a new generation from the current generation. The selected group of chromosomes establishes the next mating pool. Specifically, the selection algorithm first picks a configurable number (e) of chromosomes with best fitness values. The goal of keeping elitism is to prevent the loss of the current fittest members of population. Then the new generation is half-filled with chromosomes chosen by the unbiased tournament selection, a variant of the traditional tournament selection that aims to reduce the biased sampling.

An example that illustrates how unbiased tournament selection works is depicted in Figure 4.4. It first starts with lines up the population as two distinct permutations of the current population, $P1$ and $P2$, respectively. Then it conducts pairwise comparison, where normally the chromosome with better fitness value is selected from each pair, as demonstrated by comparison of *Pair 1* in the example. Note that *Pair*

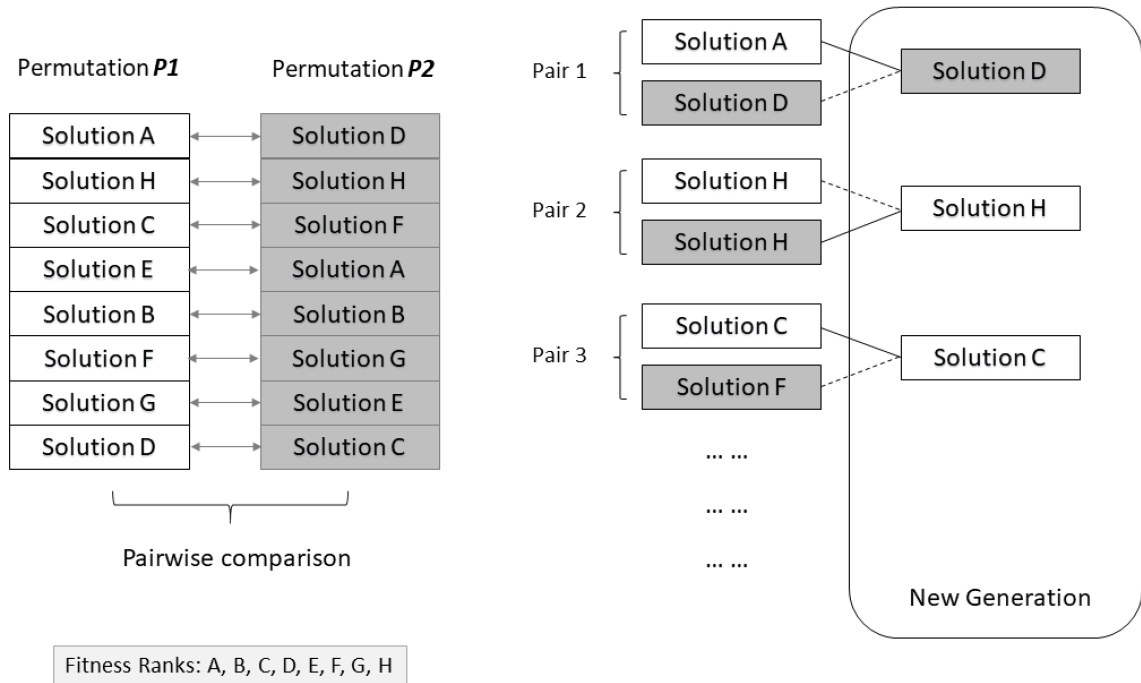


Figure 4.4: An example of unbiased tournament selection algorithm

\mathcal{Q} are accidentally formed with the same chromosome *Solution H*. In this case, I just simply pick any one of them for the next generation. The selection process ends once the new generation of population is half-filled. In EVOALLOY, I also implemented traditional tournament selection as an optional selection operator. Yet, our experiments prove that the use of unbiased tournament selection can successfully eliminate the loss of diversity due to chromosomes being sampled, such that it can help avoid the search getting stuck to local optima and plateaus, thus outperforming the traditional tournament selection in terms of efficiency. The detail of the comparison experiment is presented in the next chapter.

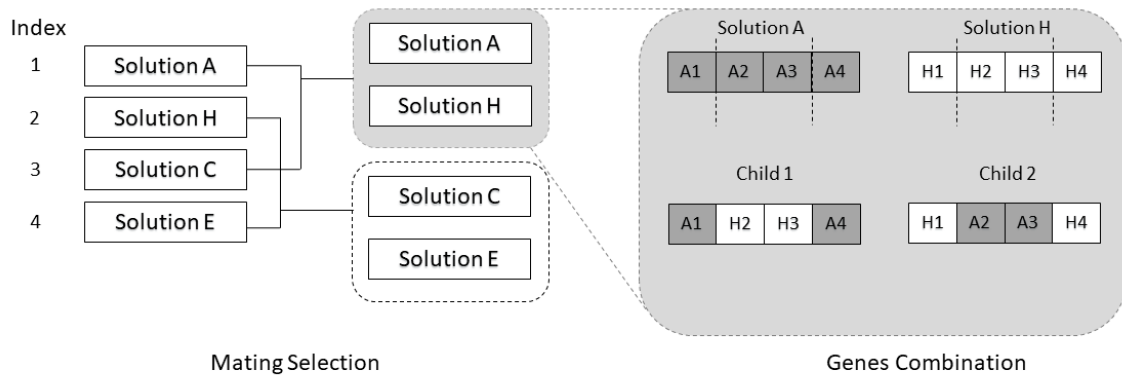


Figure 4.5: An example of two point crossover algorithm

4.6 Crossover

The initial step in producing new chromosomes for the next generation is crossover. In EVOALLOY, crossover operator starts with selecting two chromosomes from the mating pool. While other mating selections that pair two chromosomes can be applied, here I just simply picked parents based on their indices as shown in Figure 4.5. Basically, a chromosome is paired with another one when the difference between their indices is two. Then to produce new offspring chromosomes, the well-known two point crossover is essentially employed with a configurable probability, that decides if the offspring are generated by recombining parents genes or simply replicating their genes. When the decision is made to mix the two parent chromosomes, two random cut points for both parent chromosomes are uniformly chosen, since the lengths of the two chromosomes are the same. The crossover operator then swaps every tuple assigned to the genes between the two points of the parent chromosomes, eventually creates two offspring. To be noticed that, during the tuning process of developing EVOALLOY, I realized two other crossover algorithms i.e. one point crossover and all points crossover. However, both of them can not guarantee to converge when applying

to alloy specifications solved by two point crossover. Therefore, the EVOALLOY tool do not provide any options for crossover operator.

4.7 Mutation

To counter genetic drift [47] and recover lost genes, crossover is often used along with mutation, another commonly used evolutionary operator. In general, Mutation simply alters parts of the genetic material of a chromosome with a configurable mutation rate. The specific aim of applying mutation is to achieve a diverse population of chromosomes. To this end, I develop three different mutation operators in EVOALLOY, i.e. *creation*, *transformation* and *removal* .

The Algorithm on lines 16-21 describes the mutation process in EVOALLOY. It picks a chromosome from the population, and a likelihood ratio is introduced, which decides if this chromosome will be mutated or not. Once decided, one of the three mutation operators might be applied to an randomly selected gene within the chromosome. Basically, given a gene, namely a relation, currently contains no tuple, creation operator generates a new tuple-string from within the upper and lower bounds specified for this relation. But if the chosen gene is not empty set of tuples, then both transformation and removal operator can potentially be applied. Transformation operators include changing one tuple to another and inserting a new tuple-string at a random index. The removal operator omits the tuple-string assigned to a gene. In other words, the gene becomes empty, if permitted by its given lower bound.

Chapter 5

Experimental Evaluation

To evaluate our approach, I implement EVOALLOY as an open-source extension to the Alloy automated analysis engine. To implement the genetic algorithms discussed in the prior chapters, EVOALLOY modifies both the Alloy Analyzer and its underlying finite relational model finder, Kodkod [23]. The modifications mainly lie in realizing facilities for producing the initial population of chromosomes and next generations, assessing satisfiability of each chromosome within the population, collecting the feedback information necessary for measuring fitness values, and transforming chromosome-level model instances into high-level Alloy models. And the implementation of all types of evolutionary operators that facilitate the search is also designed by leveraging the bounded relational model at Kodkod level. The EVOALLOY prototype is publicly available at the project website [19]. In this chapter, I present an empirical study that evaluates EVOALLOY. Our evaluation mainly addresses two research problems:

- **RQ1** What is the performance of EVOALLOY when applied to small specifications compared to Alloy Analyzer?
- **RQ1** Can EVOALLOY solve problems that their scope of specification are too large for Alloy Analyzer to solve ?

I design a two phases evaluation. In phase one through heuristic parameters tuning experiments, I obtain the optimal settings of all the optional parameters for our GA extension that best suits our five object specifications. In phase two, I compare our EVOALLOY tool to both random exploration approach and the state-of-the-art Alloy Analyzer (version 4.2) in terms of their ability to solve problems with large search space and the running time for each experiment.

5.1 Phase One

As mentioned in Chapter 2.2.6, the initial step for genetic algorithm problems is to decide the parameter settings through necessary tuning process. Our objects of analysis for both parameter tuning experiments in Phase one and comparison experiments in Phase two are the same set of five specifications. These specifications varies in terms of both size and complexity, and they are all distributed with the Alloy Analyzer as samples (cf. Table 5.2). **Chord** models the chord distributed hash table lookup protocol; **com** specifies Microsoft component object model query interface and aggregation mechanism; **sync** is a model of a generic file synchronizer; **fileSystem** specifies a generic file system; and **life** specification models John Conway's game of life. To perform the experiments of adjusting the configurations, I just choose a small set of scopes, i.e. 5, 10, and 15, as specified scopes to conduct analysis on each of five object specifications.

We used a Mac computer with an Intel Core i7 2.3GHz CPU processor and 8 GB of main memory to conduct all the experiments in Phase One. Based on our expert knowledge, I initially configured the GA parameters as described in column *Initial Settings* of Table 5.1. I then performed a series of tuning experiments that every time only one parameter was selected for heuristically altering its value, in the meantime

all other parameters were kept unchanged. To control for variance, for each of the five objects I ran our technique three times with the same configurations and record the average. Through around 1200 runs, eventually I obtained an optimal GA parameter settings, as shown in column *Final Settings* of Table 5.1, that was specifically adjusted for solving our five object specifications .

Parameter	Initial Settings	Final Settings
Population Size	100	32
Selection Algorithm	Tournament Selection with bucket size 2	Unbiased Tournament Selection & Elitism Selection of 4
Crossover Algorithm	One Point Crossover with 100% probability	Two Points Crossover with 50% probability
Mutation Algorithm	Transformation 100%	Transformation 60% Creation 30% Removal 10%
Mutation Rate	0.1	0.8

Table 5.1: The parameter configurations of Initial Settings and Final Settings

For the sake of simplicity, I just review the process that I adjusted the mutation rate through analyzing the experimental results. I first configure all the parameters with initial settings and run each of the five object specifications over scope of 10 for three times. Figure 5.1 only shows the search processes for *com* and *chord*. Basically, In each diagram, the x axis and the y axis represent fitness value and number of generations, respectively. The curve with blue color shows the best chromosome within the population and the yellow one delineate the median. These diagrams mainly reveal that the search process under current configuration often quickly converges to local optima and can not efficiently get out of stuck. I observe the similar results when running the same experiments over scope 5 and 15. Based on our experience,

I can potentially improve the search by applying a different mutation settings, as better mutation operator can effectively increase the diversity of population, thus get higher chances to reach previously unexplored states. As it is just the first attempt for optimizing the configurations, I decide to simply increase the mutation rate rather than revising the mutation algorithm.

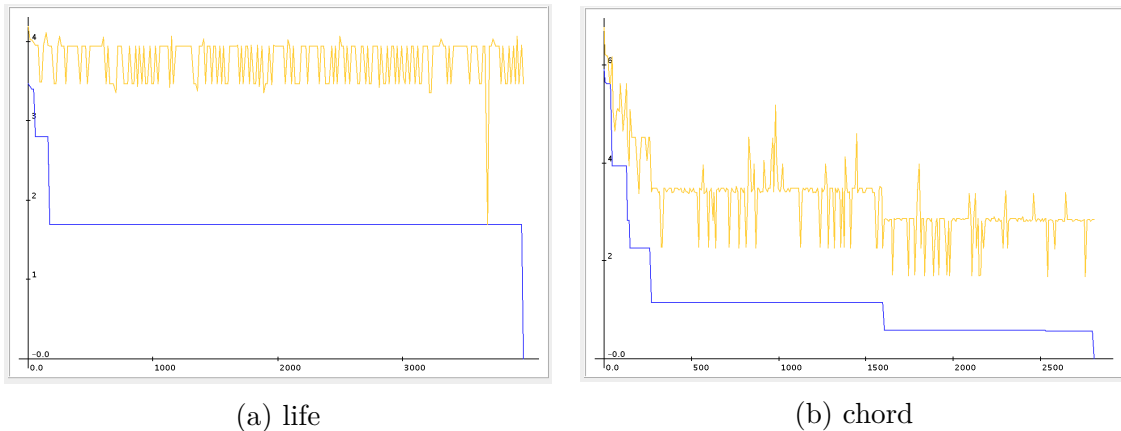


Figure 5.1: The population evolving diagrams of analysis on initial parameter settings

To find the optimal mutation rate, I perform the experiments by increasing the mutation rate step by step from 0.1 to 0.5, 0.8 and 1.2. The results shown in Table 5.2 are consistent with our guess that higher mutation rate can keep a diverse population, thus improve the efficiency of exploring satisfiable solution. Figure 5.2 clearly reveals the trend that as mutation rate increased from 0.1 to 0.5, the analysis time for each object specification decreased as expected. This trend continues when the rate is increased to 0.8, only except for sync where the performance improvement stops. For all the specifications, the performance gains from increasing the number of genes being mutated are entirely reversed when the mutation rate is changed to 1.2. This is reasonable according to our prior experience that if there is a large number of genes within the population are about to revised, (1) the quality of the chromosomes might deteriorate, and (2) the mutation itself might become much more time-consuming.

Therefore, I heuristically estimate that the best mutation rate for our objects of analysis might varies within the range from 0.8 to 1.2.

Spec	Mutation Rate			
	10%	50%	80%	120%
com	169.7	59.7	44	76
sync	8.3	3.7	4.3	8
fileSys	24.3	9.7	8.3	15.7
chord	143.3	56	40.7	75
life	158.3	40	32	39.7

Table 5.2: The analysis time in second of tuning experiments over increasing mutation rate

After roughly getting the estimation, I continue to optimize the performance of EVOALLOY through conducting experiments on gradually revising other parameters i.e crossover algorithm, crossover probability, selection algorithm, elitism number, mutation algorithm and population size. Since our ultimate goal is to apply our approach to large analysis scope, therefore when tuning the parameters, the main principle I followed is that the optimal configuration should be customized towards obtaining efficiency not only from minimizing the number of generations it takes to find the solution, but also from reducing the calculation effort of GA itself. For example, if I realized that with mutation rate 0.8 EVOALLOY can solve the five specifications by running the evolutionary process with similar number of iterations as with 1.2, then I should select 0.8 for saving extra computational effort. At last, I obtained the optimal parameter settings for EVOALLOY to perform comparison experiment in Phase two.

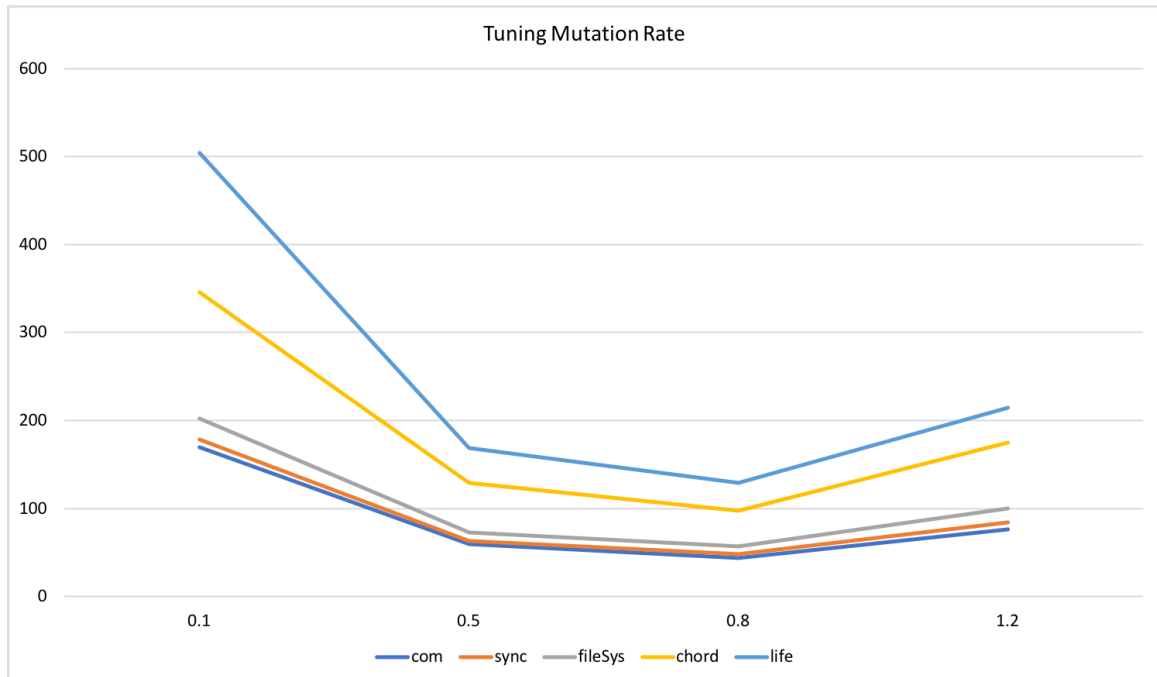


Figure 5.2: An example of unbiased tournament selection algorithm

5.2 Phase two

In Phase two ¹, I evaluate the effectiveness of EVOALLOY by comparing it with the state-of-the-art Alloy Analyzer (version 4.2). Prior to the comparison experiment, I also consider a random exploration approach, RD, that neither applies a GA nor leverages constraint solvers. It just randomly generates candidate solutions following the rules implied by the bounds of specifications relations. I set RD to generate 10,000 candidates. This is a general process for genetic algorithm research work, that it provides the ground truth about the complexity of the problem. In other words, if the target problem can be solved by random exploration, then there is no need for using genetic algorithm or other alternative approach.

The objects of analysis are the same specifications as I used in Phase one. And I configured my EVOALLOY with the optimal settings obtained in Phase one. Basically,

¹The experimental results described in this section has been presented in my published paper "An Evolutionary Approach for Analyzing Alloy Specifications" [45]

I use 32 as the population size. And I configured the algorithm to perform a two-point crossover with a crossover probability of 50%, and set the mutation rate to 80%. For mutation, I use the addition operator 10% of the time, the transformation operators 60% of the time, and the creation operator 30% of the time. To control for variance, I ran the technique three times. I did this separately on each of the five specifications under consideration. All of the experiments were conducted on an 8-core 2.0 GHz AMD Opteron 6128 system, with an 8 GB of memory was dedicated to the running technique to keep extraneous variables constant. I used two stopping criteria: (1) reaching a satisfying model, (2) exceeding the given maximum memory.

Spec	Analysis Scope						
	5	25	45	65	85	105	125
com	—	—	—	—	—	—	—
sync	1	—	—	—	—	—	—
fileSys	—	—	—	—	—	—	—
chord	—	—	—	—	—	—	—
life	—	—	—	—	—	—	—

Table 5.3: The analysis time in second taken from Random (RD) over the increasing analysis scope across objects of study

The analysis time of randomly generating solutions for all five objects over the increasing analysis scope is recorded in Table 5.3. The scope of analysis is specified on the horizontal axis. Note that the dashes simply indicate that the approach terminates without finding a solution. The result of this experiment demonstrates that, the random approach, except in one case, i.e., the `sync` specification over the analysis scope of 5, was not able to find any satisfying solution. This confirms that one has almost no chance to come up with a valid Alloy solution with a pure random search, which also proves that the need of applying genetic algorithm as an alternative approach actually exists.

I then conduct the comparison experiment between EVOALLOY and Alloy Analyzer, to answer the two research questions aforementioned. Table 5.4 reports the analysis time taken from both approaches over the increasing analysis scope across object specifications. Firstly, I observe that for the small scopes that less than 45, EVOALLOY is almost as efficient as Alloy Analyzer. The only exception is *life*, where Alloy outperforms our approach. Therefore according to the experimental results, my answer to **RQ1** is that for the selected five objects, the performance of our EVOALLOY is no worse than Alloy tool over the relatively small scope. I then observe that as the scope of analysis increases, EVOALLOY is more effective than the Alloy Analyzer. For instance, for *chord*, Alloy fails at scope 45, but EVOALLOY finds a solution up to a scope of 125. Indeed, higher analysis scope is accompanied by a larger search space, which can amplify the relative effectiveness of a GA-based approach, like EVOALLOY. With *com*, EVOALLOY goes beyond Alloy and solves scope 25, but fails afterwards due to out of allocated time. The reasoning about results of the two outliers, i.e. *life* and *com* is discussed in the following section in detail. Overall, the experimental result answers **RQ2** that for each of five specification, EVOALLOY outperforms the state-of-the-art Alloy Analyzer in terms of scalability, and the difference in the analysis capability is more pronounced for the larger analysis scopes.

5.3 Discussion

The results of Phase two experiment generally prove that, EVOALLOY gains significantly improvement on scalability of analysis over larger scopes compared to traditional Alloy Analyzer. However, in Table 5.4, there exist inconsistency in the results of experiments with *com* and *life* that need to be carefully investigate.

Spec	Analysis Scope													
	5		25		45		65		85		105		125	
	AA	EA	AA	EA	AA	EA	AA	EA	AA	EA	AA	EA	AA	EA
com	11	4	—	313	—	31311	—	—	—	—	—	—	—	—
sync	2	2	4	3	13	6	31	11	55	30	235	43	294	74
fileSys	1	3	8	8	23	26	63	176	203	333	363	680	—	1501
chord	3	2	94	16	—	241	—	299	—	391	—	705	—	1496
life	3	3	7	80	26	624	93	1000	205	3412	—	4389	—	6850

Table 5.4: The analysis time in second taken from EVOALLOY (EA) and Alloy Analyzer (AA) over the increasing analysis scope across objects of study

The first object that has inconsistent results is `com`. As mentioned before, the dashes in the first row of Table 5.4 indicate that when the scope is increased beyond 25, both EVOALLOY and Alloy Analyzer fail to generate any result for `com`, which implies both techniques are terminated before reaching a valid solution. By checking the output files and log files in detail, I realize that the termination of Alloy Analyzer experiments with `com` was caused by exceeding the memory limits, whereas EVOALLOY could keep exploring the search space by using the genetic algorithm until 24 hours time limit was passed. On one side, it confirms that my GA extension could scale for complex specifications in terms of memory consumption. On the other side, for each iteration GA consumes much more time than it supposed to, which impinge the effectiveness of EVOALLOY for exploring large complex state space. To figure out the reason for this abnormal result, I profile each stage of GA to check which procedure takes most of the time within each iteration, or if every GA processes consumes unacceptable amount of time. The profiling results reveal that when the scope is increased to 45 or more, after generating the first generation of population, EVOALLOY keep executing the examination process of each constraint and collecting number of violated constraint for each chromosome until it is terminated 24 hours later, which means it has not finished one iteration within the time limit. Given that signatures in `com` have more complicated relationships that generate exceptionally large search

space for each relation compared to other four objects, it is reasonable that its calculation effort of transformation from Kodkod to boolean logic increased dramatically faster than other specifications. Thus the main challenge to improve the scalability of EVOALLOY further relies on developing a more efficient mechanism that supports evaluating the quality of chromosomes without Kodkod engine. Moreover, a more compact way to store finite Kodkod models that previously has been explored will also contribute to improving the analysis.

Spec	Analysis Scope						
	5	25	45	65	85	105	125
com	25	34	69	—	—	—	—
sync	2	3	3	3	4	4	3
fileSys	31	12	15	13	12	16	12
chord	26	26	24	33	25	29	26
life	64	1710	2069	2150	2099	2149	2113

Table 5.5: The number of iterations taken from EVOALLOY (EA) over the increasing analysis scope across objects of study

The second outlier is `life`. According to Phase two experiment, while the complexity of `life` specification is not significantly higher than the other three objects, i.e. `sync`, `fileSys`, `chord`, it takes EVOALLOY unexpected large amount of time to solve `life` problem. Only by comparing the Kodkod models generated by each of the five objects, I am not able to reach the conclusion that `life`'s state space grows much faster than other three objects like `com` does, which means the running time of `life` for each iteration should not increase dramatically. This assumption is supported by the data recorded in Table 5.5, which represents the number of generations taken from EVOALLOY to reach the satisfying solution over increasing scope across all five objects. Then in order to figure out the origin of this large number of generations,

I checked the population of chromosomes from `life` in detail, including their genes, fitness, failed constraints and failed relations. I observe a pattern that no matter on which analysis scope, my GA extension can rapidly evolve the entire population to local optima that the best 20% chromosomes have relatively low fitness, only taking less than 30 generations on average. Yet, after 30 iterations, even the population can keep a certain degree of diversity due to my composite mutation operator, (1) The fitness of best chromosomes could not be improved over more than 1000 iterations, (2) chromosomes with different genes share similar or even the same fitness, (3) certain constraints have never been satisfied before the model instance is found. These three observations hint that for effectively analyzing diverse Alloy specifications including `life`, I need to further develop fitness that can accurately differentiate chromosomes, and tune the genetic algorithm towards better solving constraints that are hard to satisfy.

In summary, while certain limitations exist, the preliminary results provide the evidence that the line of this research on exploring the synergy between evolutionary algorithms and lightweight formal analyzers is worth pursuing.

Chapter 6

Conclusion and Future work

Leveraging mathematical concepts, lightweight formal methods can rigorously model the software system and precisely perform a multitude of analysis tasks over a bounded scope. As one of the most popular lightweight analysis, Alloy and its automated analyzer, has been applied to a variety of software engineering problems, including program analysis[8, 48, 49], test case generation[9, 10], software design[5, 6, 7] and tradeoff synthesis and analysis[11, 12]. Yet, the underlying constraint solving technique that Alloy currently relies on still prohibits it from being commonly adopted in real world applications. Therefore developing mechanisms that facilitate efficient application of formal analyzer is still an open problem.

To this end, I presented a novel approach and an accompanying tool, namely EVOALLOY, that can find solutions to a set of specifications beyond the scope in which traditional Alloy Analyzer fails. EVOALLOY extends Alloy with an efficient analysis engine based on genetic algorithm. To bypass the computational heavy model finding process performed by off-the-shelf SAT Solver underlying traditional Alloy Analyzer, EVOALLOY first randomly generates a population of candidate solutions based on Kodkod relational model. It then explores the bounded search space by iteratively evolving this population until a satisfying solution is found. Taking advantage of genetic algorithm, EVOALLOY is feasible to analyze even larger specifications where

Alloy fails, and in the meantime the result is maintained to be sound without much loss. The result of comparison experiments between my approach and Alloy corroborate the significant improvement on scalability of EVOALLOY.

While the results obtained so far are promising, EVOALLOY is still early in its development and it suffers from some limitations. First, The fitness function provides strong guidance early in the search, but needs refinement when the solution gets close. I plan to experiment with more sophisticated fitness functions and to consider an adaptive approach that has been used in prior work on evolutionary algorithms for constraint based problems. Second, I have found that the parameter tuning (e.g. mutation, crossover) is sensitive to the specific specification being solved. I plan to explore this issue further; recent work on self-tuning and hyperheuristic algorithms may help us in this context. Last, I still depend on loading the entire Kodkod model for each chromosome, which may limit us as we scale to even larger systems. I plan to examine ways to store it in a more efficient way.

I have made the prototype of EVOALLOY and the experimental results publicly available to the research and education community.

Bibliography

- [1] Symantec Corp. 2012 norton study: Consumer cybercrime estimated at \$110 billion annually. http://www.symantec.com/about/news/release/article.jsp?prid=20120905_02, September 2012. [Online].
- [2] RTI. The economic impacts of inadequate infrastructure for software testing. Technical Report 7007.011, National Institute of Standards & Technology, 2002.
- [3] Tricentis Corp. 2017 tricentis software fail watch report. <https://www.tricentis.com/software-fail-watch/>, 2017. [Online].
- [4] H. Bagheri, A. Sadeghi, R. Jabbarvand, and Sam Malek. Practical, formal synthesis and automatic enforcement of security policies for android. In *Proceedings of DSN*, pages 514–525, 2016.
- [5] H. Bagheri and K. Sullivan. Model-driven synthesis of formally precise stylized software architectures. *Formal Aspects of Computing*, 28(3):441–467, 2016.
- [6] H. Bagheri and K. Sullivan. Bottom-up model-driven development. In *Proceedings of ICSE*, pages 1221–1224, 2013.
- [7] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of*

- Software Engineering*, ESEC/FSE-9, pages 62–73, New York, NY, USA, 2001. ACM.
- [8] M. Taghdiri and D. Jackson. Inferring specifications to detect errors in code. *Automated Software Engineering*, 14(1):87–121, 2007.
- [9] D. Marinov and S. Khurshid. Testera: A novel framework for automated testing of java programs. In *Proceedings of the IEEE International Conference on Automated Software Engineering*, ASE'01, 2001.
- [10] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek. Reducing combinatorics in gui testing of android applications. In *Proceedings of ICSE*, pages 559–570, 2016.
- [11] H. Bagheri, C. Tang, and Kevin Sullivan. Trademaker: Automated dynamic analysis of synthesized tradespaces. In *Proceedings of ICSE*, 2014.
- [12] H. Bagheri, C. Tang, and K. Sullivan. Automated synthesis and dynamic analysis of tradeoff spaces for object-relational mapping. *IEEE Transactions on Software Engineering*, 43(2):145–163, February 2017.
- [13] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):11:1–11:61, December 2012.
- [14] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, feb 2013.
- [15] A. C. Kumari, K. Srinivas, and M. P. Gupta. Software module clustering using a hyper-heuristic based multi-objective genetic algorithm. *Advance Computing Conference*, pages 813–818, February 2013.

- [16] S. J. Huang and N. H. Chiu. Optimization of analogy weights by genetic algorithm for software effort estimation. *Information and Software Technology*, 48(11):1034–1045, November 2006.
- [17] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. SpringerVerlag, 2003.
- [18] D. Jackson. *Software Abstractions, 2nd ed. MIT Press, 2012*. MIT Press, 2012.
- [19] Evoalloy web page. <https://sites.google.com/site/evoalloy2018/>, 2018. [Online].
- [20] M. Collins. Formal methods. carnegie mellon university. 18-849b dependable embedded systems. https://users.ece.cmu.edu/~koopman/des_s99/formal_methods/, Spring 1998. [Online].
- [21] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov. Evaluating the "small-scope hypothesis". Technical report, MIT CSAIL, 2002.
- [22] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.
- [23] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *Proceedings of TACAS*, pages 632–647, 2007.
- [24] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.
- [25] J. Thomé, L. K. Shar, D. Bianculli, , and L. C. Briand. Search-driven string constraint solving for vulnerability detection. In *Proceedings of ICSE*, pages 198–208, 2017.

- [26] P. Dings and G. A. Agha. Solving complex path conditions through heuristic search on induced polytopes. In *Proceedings of FSE*, pages 425–436, 2014.
- [27] O. Kramer. *Genetic Algorithm Essentials*. Springer, 2017. Volume 679 of Studies in Computational Intelligence.
- [28] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.
- [29] B. Miller and D. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995.
- [30] E. Torlak, M. Taghdiri, G. Dennis, and J. P. Near. Applications and extensions of alloy: past, present and future. *Mathematical Structures in Computer Science*, 23(4):915–933, 2013.
- [31] V. Montaghani and D. Rayside. Extending alloy with partial instances. In *Proc. of ABZ*, pages 122–135, 2012.
- [32] T Nelson, S Saghafi, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Aluminum: Principled scenario exploration through minimality. In *Proceedings of ICSE*, pages 232–241, 2013.
- [33] H. Bagheri and S. Malek. Titanium: Efficient Analysis of Evolving Alloy Specifications. In *Proceedings of FSE*, pages 27–38, 2016.
- [34] A. Cunha, N. Macedo, and T. Guimaraes. Target oriented relational model finding. In *Proc. of International Conference on Fundamental Approaches to Software Engineering, FASE’14*, pages 17–31, 2014.

- [35] N. Macedo, A. Cunha, , and T. Guimaraes. Exploring scenario exploration. In *Proc. of International Conference on Fundamental Approaches to Software Engineering*, FASE'15, pages 301–315, 2015.
- [36] N. Rosner, J. H. Siddiqui, N. Aguirre, S. Khurshid, and M. F. Frias. Ranger: Parallel analysis of alloy models by range partitioning. In *Proceedings of ASE*, pages 147–157, 2013.
- [37] E. Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications*. PhD thesis, MIT, February 2009.
- [38] E Uzuncaova and S. Khurshid. Constraint prioritization for efficient analysis of declarative models. In *Proc. of International Symposium on Formal Methods*, FM'08, pages 310–325, 2008.
- [39] A.A.E. Ghazi and M. Taghdiri. Relational reasoning via smt solving. *Proceedings of FM*, pages 133–148, 2011. Springer, Berlin.
- [40] Stefan Wappler and Frank Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, pages 1053–1060, New York, NY, USA, 2005. ACM.
- [41] Phil McMinn, Mark Harman, David Binkley, and Paolo Tonella. The species per path approach to searchbased test data generation. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 13–24, New York, NY, USA, 2006. ACM.
- [42] K. Inkumsah and Tao Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proceedings*

- of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08, pages 297–306, Washington, DC, USA, 2008. IEEE Computer Society.
- [43] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, 2012.
- [44] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. *Int. J. Softw. Tools Technol. Transf.*, 6(2):117–207, March 2004.
- [45] J. Wang, H. Bagheri, and M. B. Cohen. An evolutionary approach for analyzing alloy specifications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 820–825, New York, NY, USA, 2018. ACM.
- [46] A. Sokolov and D. Whitley. Unbiased tournament selection. In *Proceedings of GECCO*, pages 1131–1138, 2005.
- [47] A. Rogers and A. Pruegel-Bennett. Genetic drift in genetic algorithm selection schemes. *IEEE Transactions on Evolutionary Computation*, 1999.
- [48] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek. Covert: Compositional analysis of android inter-app permission leakage. *IEEE Transactions on Software Engineering*, 41(9):866–886, 2015.
- [49] Joseph P. Near and Daniel Jackson. Derailer: Interactive security analysis for web applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 587–598, New York, NY, USA, 2014. ACM.