An Evolutionary Approach for Analyzing Alloy Specifications

Jianghao Wang
Department of Comp. Sci. & Eng.
University of Nebraska, Lincoln, USA
jianghao@huskers.unl.edu

Hamid Bagheri
Department of Comp. Sci. & Eng.
University of Nebraska, Lincoln, USA
bagheri@unl.edu

Myra B. Cohen
Department of Comp. Sci. & Eng.
University of Nebraska, Lincoln, USA
myra@cse.unl.edu

ABSTRACT

Formal methods use mathematical notations and logical reasoning to precisely define a program's specifications, from which we can instantiate valid instances of a system. With these techniques we can perform a multitude of tasks to check system dependability. Despite the existence of many automated tools including ones considered lightweight, they still lack a strong adoption in practice. At the crux of this problem, is scalability and applicability to large real world applications. In this paper we show how to relax the completeness guarantee without much loss, since soundness is maintained. We have extended a popular 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 failed constraints. In a feasibility study we demonstrate that we can find solutions to a set of specifications beyond the scope where traditional Alloy fails. While small specifications take longer with EvoAlloy, the scalability means we can handle larger specifications. Our future vision is that when specifications are small we can maintain both soundness and completeness, but when this fails, EvoAlloy can switch to its genetic algorithm.

CCS CONCEPTS

• **Software and its engineering → Formal methods**; **Software verification**;

KEYWORDS

Formal analysis, Evolutionary algorithms, Relational logic.

ACM Reference Format:
Jianghao Wang, Hamid Bagheri, and Myra B. Cohen. 2018. An Evolutionary Approach for Analyzing Alloy Specifications. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3238147.3240468

1 INTRODUCTION

Software has embedded itself in our daily lives, and is now essential for communication, healthcare, transportation, and even home comfort. Yet at the same time, software continues to fail, and malicious users exploit weaknesses of systems [1]. Fifteen years ago the National Institutes of Standards reported that a poor software quality infrastructure was costing the US upwards of $59 Billion annually [2], and an equally ominous report from Tricentis in 2017 estimated the annual financial loss due to software failures worldwide at $1.7 Trillion [3]. While many efforts have been made to improve our software engineering techniques, and to develop better software validation methods, these 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 we still lack sufficient techniques to verify and validate our software.

One class of techniques that have been used to tackle dependability are those which fall into the category of formal methods, with their strength residing in the mathematical concepts leveraged to prove 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 automated, yet formally precise analysis capabilities, which reduce the burden on traditional formal verification techniques. This spans a wide range of software engineering and security domains, including software design [5, 6], code analysis [7], security analysis [4], test case generation [8] and tradeoff synthesis and analysis [9, 10]. Such techniques often transform the system specification into a satisfiability problem, and delegate the task of solving it to a constraint solver. The analysis is then conducted by exhaustive enumeration over the bounded scope of specification instances.

Despite significant advances, we still find ourselves lacking strong adoption of formal techniques. At the crux of this problem, is scalability and applicability for large real-world applications. Bounded verification techniques are at once both sound and complete for the given analysis bound, but the completeness means that on large systems they either fail or need to be reduced in scope. An alternative approach to solving problems that grow exponentially has been to use search-based techniques or more specifically evolutionary algorithms [11]. These algorithms heuristically explore large complex solutions spaces and converge on single solution, rendering them sound but incomplete.

In this paper, we present a novel tradeoff that provides a new road towards scalability. Our vision is that when the search space of specifications are small, we can use the full power of a constraint solver and maintain both soundness and completeness. When this fails, we switch on evolutionary algorithms [12] promising to scale to real-world problems without sacrificing soundness.

To assess the feasibility of the approach, we develop EvoAlloy, an extension to the existing lightweight formal analysis tool, Alloy [13]. EvoAlloy delegates the task of finding satisfying models to an analysis engine using a genetic algorithm (GA), one of the most popular types of evolutionary algorithms. They have been shown to be useful for pinpointing solutions in a large search space.

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3240468

```
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 1: An Alloy specification example describing a simple model of file system.

We 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, has an active development community, and suffers from exactly the scalability problems addressed by this work. We make research artifacts developed in this study and experimental data available to the research and education community [14]. The preliminary results corroborate the feasibility of the approach, and denote that this direction of research is promising.

The remainder of this paper is organized as follows. Section 2 uses an illustrative example to describe the intuition behind our technique as well as the necessary background. Section 3 overviews our approach towards achieving a more scalable analysis technique. Section 4 presents the preliminary results obtained in our experiments. Sections 5 and 6 outline related research and conclude.

2 ILLUSTRATIVE EXAMPLE

This section motivates our research and illustrates the EvoAlloy technique using a simple example. Section 3 presents a more detailed discussion of our approach.

Consider the Alloy specification for a simplified model of a file system, shown in Listing 1. Essential data types are specified in Alloy by their type signatures (sig), and the relationships between them are captured by the declarations of *field*s within the definition of each signature. The running example defines 4 signatures (lines 1–6): File system objects, FSObjects, are partitioned into Dir and File types, with Root defined as a singleton extending Dir. Each Dir may have a set of contents of type FSObject.

Facts (fact) are formulas that take no arguments, and define constraints that every instance of a specification must satisfy, thus confining 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. The Hierarchy fact paragraph (lines 8–15) states that the Root directory has no parent, and it cannot be a subdirectory for any other directory; that each single file and directory should be reachable from the Root directory; and that 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. In particular, to make the state space finite, certain scopes need to be specified that limit the number of instances of each type signature. The run specification (lines 16–19) then asks for 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 a relational specification bounded by the specified scope, both Alloy Analyzer and EvoAlloy then translate it into a corresponding finite relational model in a language called Kodkod [15]. Listing 2 partially shows a Kodkod translation of Listing 1. A model in Kodkod's relational logic is a triple consisting of a universe of elements (also called *atoms*), a set of relation declarations including their lower and upper bounds specified over the model's universe, and a relational formula, where the declared relations appear as free variables [15].

The first line of Listing 2 declares a universe of four uninterpreted atoms. In this section, we assume 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. Note that the abbreviated atom names are chosen for readability, and do not indicate type, as in Kodkod all relations are untyped.

Lines 3–6 state relational variables along with their lower and upper bounds and their size. Similar to Alloy, formulas in Kodkod are constraints defined over relational variables. Kodkod further allows specifying a scope over each relational variable from both above and below by two relational constants. In principle, a relational constant is a pre-specified set of tuples drawn from a universe of atoms. Consider the Root declaration (line 3), its upper and lower bounds both contain just one atom, R1, as it is defined as a singleton set in Listing 1. 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. 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 represents this form for the constraints specifications in our running example.

The Kodkod's model finder then leverages off-the-shelf SAT-solvers to explore within such upper and lower bounds defined for each relational variable to find instances of a formula, which are bindings of the formula's relational variables to relational constants in a way that makes the formula true. EvoAlloy, however, delegates the task of model finding currently performed by computationally-expensive constraint solvers to an analysis engine based on genetic algorithms.

Figure 1a delineates a genetic representation of the problem, where a candidate solution is represented as a *chromosome*, a.k.a. an *individual*, consisting of a vector of *genes*. Evolutionary algorithms are meta-heuristic optimization techniques that mimic the process of natural genetic variation and selection into a computational problem [12]. Each chromosome contains a gene for each relational variable within the specification under analysis. Each gene has a domain of values called *alleles*. Here, 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, lines 3–6).

```
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 | lone (Dir.contents.o)) && ...
```

Listing 2: Kodkod representation of the Alloy module of Listing 1.

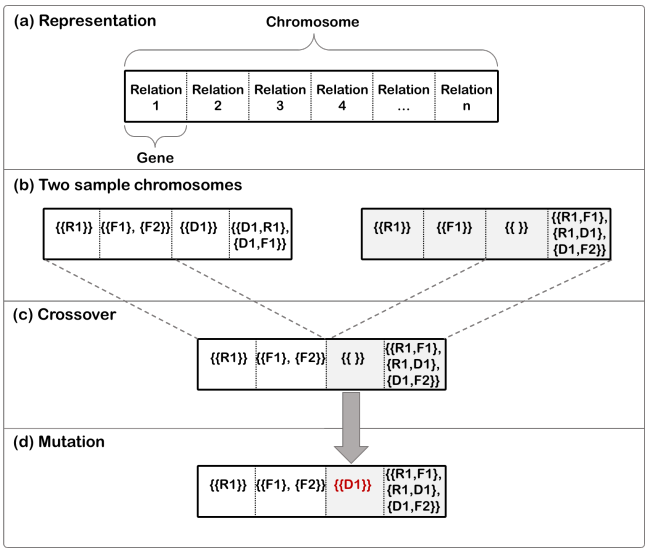

Figure 1: EvoAlloy's (a) representation of a chromosome, (b) two produced chromosomes for the specification of Listing 1, (c) crossover step for creating a new chromosome, and (d) mutation step.

A genetic algorithm starts with an initial *population* of randomly created chromosomes. Figure 1b demonstrates two sample chromosomes produced for our running example. Each chromosome in this case has 4 genes that correspond to the specification's relations, i.e., Root, File, Dir, and contents, from left to right, respectively. A solution is found by iteratively evolving population of chromosomes. Evolutionary search entails two types of operators, i.e., *crossover* and *mutation*. Crossover among two selected chromosome parents is carried out to breed new chromosomes. A crossover is often conducted by blending a subset of each parent's genetic makeup. Figure 1c represents EvoAlloy's crossover step for creating offspring.

In EvoAlloy the recombination of the two parents create two offspring. For the sake of simplicity and as it suffices to make the idea concrete, here we just demonstrate one offspring. The diagram shows a single-point crossover, i.e., a random point within the middle range of a chromosome; EvoAlloy yet effectively is capable of exploiting different types of crossover operators. Finally, some genes in the population will be *mutated* using a given mutation rate. Figure 1d illustrates applying a mutation operator to a chromosome that gives rise to a randomized change in the chromosome. In fact, mutation randomly selects a percentage of genes in the population and modifies each by assigning a different tuple from within that gene's domain.

The evolutionary search using genetic operators is carried on up to an identification of a satisfactory solution or an end criterion is met. EvoAlloy relies in part on the Kodkod analysis engine to get the relations that fail within each chromosome along with the number of failed subformulas to drive the search towards those which have no violations. Figure 2 illustrates a satisfying model instance produced by EvoAlloy from the chromosome shown in Fig. 1d.

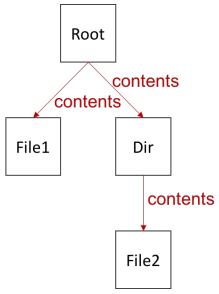

Figure 2: An Alloy model instance derived automatically from the chromosome shown in Fig. 1d.

The above example provides an intuitive description of model finding using both Alloy and EvoAlloy. However, in practice constraint solving, despite significant advances, continues to be a bottleneck in analyses relying on such technologies, including the traditional Alloy Analyzer. To gain further confidence in the correctness of their models, 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, preventing the 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. There is a need for mechanisms that facilitate efficient application of formal analyzers in rapidly growing domain of software systems.

The next section overviews our approach to extend Alloy with an evolutionary algorithm towards achieving a more scalable model finding technique.

3 EVOALLOY

Figure 3 shows an overview of EvoAlloy, and explains how it can bypass the computationally intractable part of the existing Alloy Analyzer. On the left, the Alloy Analyzer reads in an Alloy specification and translates it into a relational model, then passes that to Kodkod (a finite relational model analyzer) [15]. 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 [16]. Relational constraints are then captured as

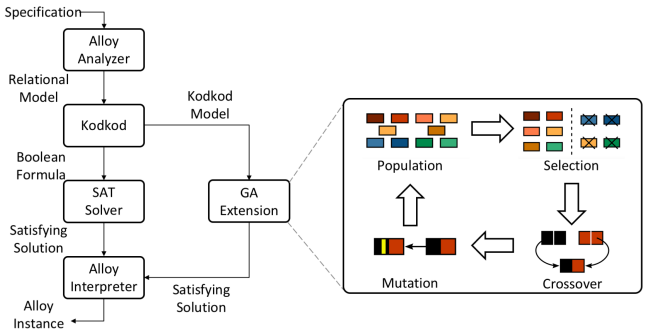

Figure 3: Schematic view of EvoAlloy

Boolean constraints over the translated Boolean variables. It then translates the resulting Boolean formula to CNF, and passes the CNF to an off-the-shelf SAT solver to obtain a solution. Last, the Alloy interpreter translates the SAT result into a solution instance.

Our insight is to utilize the bounded relational model to make the genetic algorithm scalable for two reasons. First, applying the genetic algorithm on the Kodkod level, rather than the higher Alloy level, is more efficient as both tight relational bounds and partial models limit the space of concrete instances that need to be explored by the search engine. Second, translating a Kodkod model to a propositional formula and then to CNF introduces many auxiliary variables [15, 17]. The explosion in the number of variables affects the scalability of the genetic algorithm approach.

Our GA extension is thus inserted between Kodkod and the Alloy interpreter, as depicted in Figure 3. At the highest level, EvoAlloy's GA extension takes in the model from Kodkod and outputs a satisfying solution to the Alloy interpreter. The box at right shows the steps EvoAlloy follows to do this.

The GA employed in this work is delineated in Algorithm 1. The *initial population* of our individuals is made up of random assignment of values to each relational variable, from within the legal relations and their bounds. The scope of each relational variable 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, we randomly assign a value to each relation from within the specified bounds. In essence, each chromosome within the population represents a *potential* Alloy solution.

Fitness is measured by assessing the chromosome and monitoring how close it gets to satisfying constraints of the target specification. To verify each individual, we 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. With each subsequent iteration, we breed new chromosomes through combining chromosomes selected with a likelihood proportional to their fitness value, and then mutating the resulting ones (e.g., arbitrarily change some of its tuples).

Algorithm 1 The genetic algorithm applied in EvoAlloy

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

The remainder of this section describes the details of EvoAlloy.

3.1 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 entails defining a chromosome and the mapping from it to the original problem context. In our case, a chromosome is represented as a vector shown in Figure 1a, 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 refers to the value assignment of exactly one relation. Given an Alloy specification S, we 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. Analogously, we define $f_S^{-1} : \mathbb{N} \rightarrow Relation(S)$ as a function that maps a given vector index to the relation it represents. Note that the chosen representation has a fixed size for a given problem, determined by the number of relations within the problem specification under analysis. This representation influences variation operators, i.e., crossover and mutation, discussed below.

3.2 Fitness Function

The fitness function is a decisive factor of evolutionary algorithms. 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. EvoAlloy considers two factors in assessing the fitness of chromosomes: Formula constraints (c_i) and relations (r_i). The fitness of a chromosome *chrom* is determined as follows:

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

where $T_c(c_i, chrom)$ 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 not satisfied by *chrom*; it evaluates to zero otherwise. When a chromosome for a given specification satisfies all its constraints defined over its relational variables, we 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.

3.3 Selection

The Algorithm on lines 3–9 explains the process by which EvoAlloy selects chromosome variants to pass to the next generation. It leverages both elitism and unbiased tournament selection strategies [18] to select half of population members in a new generation from the current generation. The select group of chromosomes establishes the next mating pool. Specifically, it first picks a configurable number (e) of chromosomes with best fitness values. The use of elitism prevents the loss of the current fittest members of the population. The new generation is then half-filled with chromosomes produced by the unbiased tournament selection, which forms two distinct permutations of the population and conducts a pairwise comparison to select one chromosome from each pair of compared chromosomes. The use of unbiased tournament selection promises to eliminate the loss of diversity due to chromosomes not being sampled, typically occurred in the traditional tournament selection.

3.4 Crossover

The initial step in producing new chromosomes for the next generation is crossover. It picks two chromosomes from the population, and produces two new chromosomes by mixing their genetic makeup. The employed crossover operator in EvoAlloy is essentially the well-known two-point crossover. Because the lengths of the two chromosomes are the same, the cut points are uniformly chosen within the chromosomes' length. The crossover creates two offspring, where it swaps every tuple assigned to the genes between the two points of the parent chromosomes.

3.5 Mutation

To counter genetic drift [19] and recover lost genes, crossover is often used along with mutation to achieve a diverse population of chromosomes. Mutation simply alters parts of the genetic makeup of a chromosome with a probability threshold that is configurable. EvoAlloy mutates genes with various *creation*, *transformation* and *removal* operators.

The creation operator basically generates a new tuple-string from within the upper and lower bounds specified for the relation associated with a given gene currently containing no tuple. The number of added tuples is random with a minimum of one and a configurable upper threshold. 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.

4 EXPERIMENTAL EVALUATION

We have implemented EvoAlloy as an open-source extension to the Alloy analysis engine. To realize the genetic algorithms discussed in the prior sections, EvoAlloy modifies both the Alloy Analyzer and its underlying finite relational model finder, Kodkod [15]. The modifications lie in realizing the facility to producing the initial population of chromosomes and next generations, assessing satisfiability of each chromosome within the population, collecting the information necessary in measuring fitness values, and transforming chromosome-level model instances into high-level Alloy models. The EvoAlloy prototype is available at the project website [14].

To assess the effectiveness of EvoAlloy, we compare it with the state-of-the-art Alloy Analyzer (version 4.2). In addition, we consider a random exploration approach, RD, that neither applies a GA nor leverages constraint solvers. Rather, it randomly generates candidate solutions following the rules implied by the bounds of specifications relations. We set RD to generate 10,000 candidates.

Objects of Analysis. Our objects of analysis are specifications that vary in terms of size and complexity and are distributed with the Alloy Analyzer (cf. Table 1). 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 comparison experiments, we gradually increased the scope of analysis on each of our object specifications.

Experimental Setup. For our GA parameters we ran some initial experiments to heuristically tune these to work across more than one subject. We leave a full evaluation of tuning as future work. We use 32 as the population size. We configured the algorithm to perform a two-point crossover with a crossover probability of 50%, and set the mutation rate to 80%. For mutation, we 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, we ran the technique three times, and report the average. We 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. We used two stopping criteria: reaching a (1) a satisfying solution or (2) exceeding the given maximum memory.

Results and Interpretation. Table 1 reports the analysis time in second taken from EvoAlloy, the Alloy Analyzer (AA), and Random (RD) over the increasing analysis scope across object specifications. The scope of analysis is specified on the horizontal axis.

As Table 1 shows, for each 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. The random approach, except in one case, i.e., the sync specification with 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. We also see that for smaller scopes Alloy often outperforms EvoAlloy, but 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 memory. We believe that better tuning and a more compact way to store finite Kodkod models will allow us to keep improving the analysis.

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

Table 1: The analysis time in second taken from EvoAlloy (EA), Alloy Analyzer (AA), and Random (RD) over the increasing analysis scope across objects of study; dashes indicate the approach terminates without finding a solution.

Spec	Analysis Scope																				
	5			25			45			65			85			105			125		
	RD	AA	EA	RD	AA	EA	RD	AA	EA	RD	AA	EA	RD	AA	EA	RD	AA	EA	RD	AA	EA
com	—	11	4	—	—	313	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
sync	1	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

5 RELATED WORK

There is a large body of research on using evolutionary algorithms to solve software engineering problems [11]. EvoAlloy falls within this class of solutions. Concolic Walk combines linear constraint solving with tabu search to solve complex arithmetic path conditions [20]. ACO-Solver uses the Ant Colony Optimization to solve complex string constraints [21]. The work of Godefroid and Khurshid [22] 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 paper addresses 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, EvoAlloyis the first evolutionary technique for automated analysis of bounded relational logic specifications.

The widespread use of Alloy has lead to a number of extensions to its underlying analyzer [23, 24]. Among others, Uzuncaova and Khurshid [25] partition 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. [26] 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, whereas EvoAlloy is geared towards the application of genetic algorithms to foster exploration of large, complex solution spaces.

6 CONCLUSIONS AND FUTURE WORK

In this paper we have provided a proof-of-concept for EvoAlloy to demonstrate its potential benefit and power. However, it 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. We plan to experiment with additional fitness functions and to consider an adaptive approach that has been used in prior work on evolutionary algorithms for constraint based problems. Second, we have found that the parameter tuning (e.g., mutation, crossover) is sensitive to the specific specification being solved. We plan to explore this issue further; recent work on self-tuning and hyperheuristic algorithms may help us in this context. Last, we still depend on loading the entire Kodkod model which may limit us as we scale to even larger systems. We plan to examine ways to store in a more efficient way.

ACKNOWLEDGEMENT

This work was supported in part by an NSF EPSCoR FIRST award, and awards CCF-1755890, CCF-1618132 and CCF-1745775 from the National Science Foundation.

REFERENCES

[1] Symantec Corp., "2012 norton study: Consumer cybercrime estimated at $110 billion annually," Sep. 2012. [Online]. Available: $http://www.symantec.com/about/news/release/article.jsp?prid=20120905_02$
[2] RTI, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards & Technology, Technical Report 7007.011, 2002.
[3] Tricentis Corp., "2017 tricentis software fail watch report," 2017. [Online]. Available: https://www.tricentis.com/software-fail-watch/
[4] H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek, "Practical, formal synthesis and automatic enforcement of security policies for android," in *Proceedings of DSN*, 2016, pp. 514–525.
[5] H. Bagheri and K. Sullivan, "Model-driven synthesis of formally precise stylized software architectures," *Form. Asp. of Comput.*, vol. 28, no. 3, pp. 441–467, 2016.
[6] ——, "Bottom-up model-driven development," in *Proceedings of ICSE*, 2013, pp. 1221–1224.
[7] M. Taghdiri and D. Jackson, "Inferring specifications to detect errors in code," *Automated Software Engineering*, vol. 14, no. 1, pp. 87–121, 2007.
[8] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing Combinatorics in GUI Testing of Android Applications," in *Proceedings of ICSE*, 2016, pp. 559–570.
[9] H. Bagheri, C. Tang, and K. Sullivan, "Trademaker: Automated dynamic analysis of synthesized tradespaces," in *Proceedings of ICSE*, 2014, pp. 106–116.
[10] ——, "Automated Synthesis and Dynamic Analysis of Tradeoff Spaces for Object-Relational Mapping," *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 145–163, Feb. 2017.
[11] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 11:1–11:61, Dec. 2012.
[12] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. SpringerVerlag, 2003.
[13] *D. Jackson, Software Abstractions, 2nd ed. MIT Press, 2012.* MIT Press, 2012.
[14] "EvoAlloy web page," 2018. [Online]. Available: https://sites.google.com/site/evoalloy2018/
[15] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *Proceedings of TACAS*, 2007, pp. 632–647.
[16] E. Torlak, "A constraint solver for software engineering: Finding models and cores of large relational specifications," PhD Thesis, MIT, Feb. 2009.
[17] T. Nelson, S. Saghafi, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "Aluminum: Principled scenario exploration through minimality," in *Proceedings of ICSE*, 2013, pp. 232–241.
[18] A. Sokolov and D. Whitley, "Unbiased tournament selection," in *Proceedings of GECCO*, 2005, pp. 1131–1138.
[19] A. Rogers and A. Pruegel-Bennett, "Genetic drift in genetic algorithm selection schemes," *IEEE Transactions on Evolutionary Computation*, 1999.
[20] P. Dinges and G. A. Agha, "Solving complex path conditions through heuristic search on induced polytopes," in *Proceedings of FSE*, 2014, pp. 425–436.
[21] J. Thomé, L. K. Shar, D. Bianculli, and L. C. Briand, "Search-driven string constraint solving for vulnerability detection," in *Proceedings of ICSE*, 2017, pp. 198–208.
[22] P. Godefroid and S. Khurshid, "Exploring Very Large State Spaces Using Genetic Algorithms," *Int. J. Softw. Tools Technol. Transf.*, vol. 6, no. 2, pp. 117–127, 2004.
[23] E. Torlak, M. Taghdiri, G. Dennis, and J. P. Near, "Applications and extensions of alloy: past, present and future," *Mathematical Structures in Computer Science*, vol. 23, no. 4, pp. 915–933, 2013.
[24] H. Bagheri and S. Malek, "Titanium: Efficient Analysis of Evolving Alloy Specifications," in *Proceedings of FSE*, 2016, pp. 27–38.
[25] E. Uzuncaova and S. Khurshid, "Constraint prioritization for efficient analysis of declarative models," in *Proceedings of FM*, 2008, pp. 310–325.
[26] 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*, 2013, pp. 147–157.