Evolutionary Analysis of Alloy Specifications with an Adaptive Fitness Function

Jianghao Wang[1][0009−0003−4609−2991], Clay Stevens[2][0000−0001−5399−9661], Brooke Kidmose[3][0000−0002−6673−6163], Myra B. Cohen[2][0000−0003−2443−2425], and Hamid Bagheri[1][0000−0001−6686−466X]

[1] University of Nebraska-Lincoln, Lincoln, NE 68588, USA
{jwang65,bagheri}@unl.edu
[2] Iowa State University, Ames, IA 50011, USA
{cdsteven,mcohen}@iastate.edu
[3] Technical University of Denmark, Kongens Lyngby, Denmark
blam@dtu.dk

Abstract. The use of formal methods in software engineering imparts a high degree of rigor and precision on the software development process. While formal methods are crucial for ensuring system dependability, their practical adoption has been limited in part due to scalability concerns, even though many automated analysis tools are available. In this paper, we address the scalability challenge in one type of formal analysis approach, model-finding. Prior work on EvoAlloy has demonstrated the potential for extending the Alloy Analyzer with an evolutionary algorithm by loosening the completeness guarantee while preserving soundness. However, that approach was evaluated on a small set of programs and failed to find many small-scope models that Alloy can find. In this work we introduce a new technique, called AdaptiveAlloy, which uses a novel adaptive fitness function for the analysis of Alloy relational logic specifications. Through our experiments, we illustrate that AdaptiveAlloy is capable of finding models of higher scope, and achieving greater scalability than both EvoAlloy and a state-of-the-art Alloy analyzer.

Keywords: Genetic Algorithm · Formal Analysis · Adaptive Fitness.

1 Introduction

Software engineers rely on a wide variety of tools to help them develop secure, efficient, and dependable software. In many software engineering domains—particularly for systems where reliable execution is paramount—developers employ *formal analysis* to verify their systems behave as expected. Researchers have developed and refined a variety of formal approaches to analyze software, applying their tools to validate software in domains such as autonomous vehicles [17], the Internet-of-Things [2], and database design [19, 27]. Unfortunately, despite great advances in these approaches, formal analysis techniques still face challenges regarding scalability when applied to large-scale software systems. Modern analysis tools—e.g., Alloy [12]—attempt to address these challenges by

defining *bounds* on the scope of the analysis; these bounded analysis approaches improve scalability for small applications, but larger scopes remain intractable even for state-of-the-art techniques.

These scalability challenges stem in part from a reliance on Boolean satisfiability (SAT) solvers, which many state-of-the-art formal techniques share. These approaches translate a formal specification into a satisfiability problem—typically represented in conjunctive normal form (CNF)—and provide that problem to an off-the-shelf SAT solver to solve. This translation from specification to SAT is a resource intensive process, requiring a great deal of system memory for large specifications. Once translated, the SAT solver systematically explores a vast search space which grows exponentially in the number of variables in the original specification—a total exploration of which may well be intractable.

In cases where the space of possible solutions is simply too large for systematic exploration, *search-based* techniques show promise as a sound alternative. In particular, EvoAlloy [23] proposed the use of a genetic algorithm to address the task of finding satisfying models within the Alloy context, yielding promising results. It represents potential assignments of relational variables in the Alloy specification as the genotype and utilizes methods such as mutation, crossover, and selection to navigate the solution space and ultimately reach an assignment that satisfies all the constraints. While EvoAlloy [23] represents a notable step forward, it primarily focuses on employing the GA to explore the solution space and converge on satisfactory assignments. However, its approach to fitness evaluation, which relies on "maxsat" and considers only the "top-level" subformulas of the specification, might not fully grasp the intricacies of Alloy's relational specifications. This approach could lead to early convergence on locally optimal solutions, potentially missing satisfying models of the specification.

In this paper, we present a novel approach that significantly enhances the capabilities of genetic algorithm-based analysis in the context of Alloy specifications. The key contribution lies in the refinement and depth of the fitness function employed. Unlike EvoAlloy's [23] approach, which operates on "top-level" subformulas, our proposed fitness function delves into the abstract syntax tree (AST) of the relational formula, offering a granular examination of the specification's structure. By traversing the AST and computing the number of genes that would require modification to satisfy the specification, our approach provides a more comprehensive and nuanced evaluation of candidate solutions. This nuanced evaluation enables our method, implemented in our custom tool AdaptiveAlloy, to effectively navigate the solution space and converge more quickly on satisfying solutions. Furthermore, we utilize adaptive fitness (e.g. [3]) in this domain, which dynamically adjusts the weighting of subformulas based on their complexity and difficulty in satisfying the specification. This adaptive approach allows AdaptiveAlloy to allocate resources more efficiently, focusing computational effort where it is most needed and enhancing the overall effectiveness of the GA-based analysis of Alloy specifications.

Our comparative analysis of AdaptiveAlloy with the Alloy Analyzer and EvoAlloy demonstrates scalability and efficiency improvements across various experimental subjects. Our approach shows enhancements in analysis time of up

to 182 times faster compared to the Alloy Analyzer and up to 172 times faster compared to EvoAlloy. Empowered by adaptive fitness functions and granular AST assessment, our approach demonstrates promising capabilities in addressing the scalability and performance challenges faced by current state-of-the-art tools.

2 Background and Running Example

In this section, we present a small, yet representative Alloy specification to illustrate our evolutionary search technique and thus motivate our research. An in-depth discussion of our adaptive approach is outlined in Section 3.

Alloy specifications consist of a set of *relations*, defined in a syntax akin to object-oriented programming languages, and a set of *constraints*, expressed as first-order logic sentences. These constraints may include transitive closure over the defined relations. Additionally, specifications may contain one or more *commands*, which aim to find models satisfying the constraints or counterexamples, all within a specified *scope* defined on one or more of the relations.

```
1  abstract sig Node{}
2  sig Var extends Node {}
3  sig Expr extends Node{
4    connects: set Node
5  }
6  one sig Root extends Expr{}
7
8  fact Structure{
9    //Root Expression is accessible to every Node
10   Node in Root.*connects
11   //No parent for Root Expression
12   no connects.Root
13   //Each Expression has exact two child Nodes
14   all e: Expr | # e.connects = 2
15   //No Expression is its own predecessors
16   all e: Expr | e not in e.~
         connects
17   //Each Node has at most one parent
18   all n: Node | lone connects.n
19 }
20 pred model {
21     some Expr
22 }
23 run model for 2 Expr, 3 Var
```

Listing 1.1: An Alloy specification example describing a Binary Expression Tree.

Listing 1.1 depicts an Alloy specification of a Binary Expression Tree (BET). This specification outlines the BET's primary data types using four distinct signatures (lines 1-6) and enforces several key constraints (lines 8-19). Specifically, it ensures that each node possesses at most one parent, every expression possesses exactly two children, and no expression self-includes. Moreover, the Root expression is designated as exclusive and holds access to all other nodes.

The Alloy Analyzer translates specifications into a finite relational model using Kodkod [22]. This process involves defining bounds for each relation, which encompass possible tuples. Kodkod then converts these relations, bounds, and constraints into a Boolean formula, which is solved by SAT solvers to identify valid instances. However, Alloy's scalability is limited by its reliance on SAT solvers, which employ an exhaustive enumeration approach, hindering its application to real-world systems. In contrast, our approach, AdaptiveAlloy, replaces SAT solvers with a genetic algorithm, offering improved scalability and

performance. The next section discusses how our approach successfully achieves a more economical and scalable model-finding technique by using a novel adaptive genetic algorithm in detail.

3 AdaptiveAlloy

Figure 1 presents an overview of AdaptiveAlloy and elucidates its ability to circumvent the computationally intensive aspects of the current Alloy Analyzer. On the top, the Alloy Analyzer first reads an Alloy specification and converts it into a relational model. This model is then forwarded to Kodkod. Using the scopes and signature bounds provided by Alloy, Kodkod concretizes these parameters to define the specification boundaries. To represent this finite relational model as a Boolean logic formula, Kodkod maps each relation to a Boolean matrix. Within this matrix, every tuple within the bounds of the given relation corresponds to a unique Boolean variable. The relational constraints are then transformed into Boolean constraints over these translated variables. Subsequently, Kodkod translates the resulting Boolean formula into Conjunctive Normal Form (CNF), which is then passed to an off-the-shelf SAT solver to derive a solution. Finally, the Alloy interpreter interprets the SAT solver's output, translating it into a solution instance.

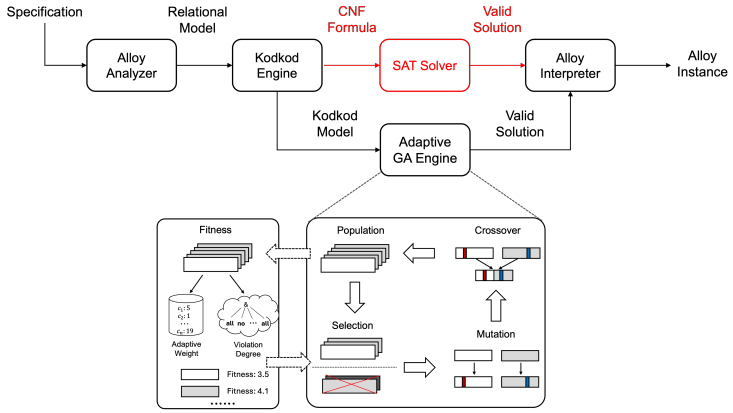

Fig. 1: AdaptiveAlloy overview

Both EvoAlloy and AdaptiveAlloy modify the process of finding satisfying models of a given specification by circumventing the traditional SAT solver-based approach, as depicted in Figure 1. AdaptiveAlloy entails the following steps: (1) It begins by converting the Alloy specification into a bounded relational model, akin to the process employed by Kodkod in traditional analysis methods. (2) Next, it constructs a genotype representation of candidate solutions. This representation encapsulates the assignments of tuples to the relations within the model. (3) The crux of our approach lies in executing a genetic algorithm-based search of the solution space. This search employs crossover, mutation, and selection techniques, backed by our adaptive Alloy-specific fitness function, to iteratively explore and refine potential solutions.

The following subsections detail our approach, focusing on key components: *Genotypic Representation (Sec.3.1)* elucidates the methodology behind encoding Alloy specifications, required for enabling our genetic algorithm to effectively navigate the solution space. *Genetic Algorithm Processes (Sec.3.2)* delve into the processes of crossover, mutation, and selection employed by our genetic algorithm, instrumental in iteratively refining and improving candidate solutions. Following this, our fitness function, which incorporates two core innovations, is presented: *Degree of Violation Computation (Sec.3.3)* introduces a method for computing the "degree" to which a given assignment violates the Alloy specification. This granular analysis provides valuable insights into the quality of candidate solutions. Lastly, *Dynamic Weight Adjustments (Sec.3.4)* explains how our dynamic weight adjustments enhance our approach's efficacy in navigating complex solution spaces by allocating more weight to challenging subformulas.

3.1 Genotypic representation

The bounded relational models used by Kodkod are typically converted into Boolean variables in the underlying SAT problem by creating a unique variable to represent each possible tuple assignment to each relation based on the bounds defined for that relation. If a given variable is true in a given candidate solution, the corresponding tuple *is* assigned to the corresponding relation in that candidate; if the variable is false, that tuple is *not* assigned to the relation. In AdaptiveAlloy, we use a similar mapping to represent the genotype for each individual as a set of chromosomes corresponding to the set of relations, where each gene in each chromosome is a single bit value (1 or 0) representing the assignment/non-assignment of a specific tuple to that relation, respectively. Thus, each individual can be defined genetically as a bitstring of genes indicating the assignment/non-assignment of each relation-tuple pair that falls within the bounds of the specification. Figure 2 depicts examples of bit chromosomes created for the Var relation in the example from Sec. 2.

$$\begin{array}{l|l} \varnothing \subseteq \mathsf{Var} \subseteq \{ V1, V2, V3 \} & \\ \hline \mathsf{Var}_{s0} = \{ \} & \Longrightarrow 0\ 0\ 0 \\ \mathsf{Var}_{s1} = \{ V2 \} & \Longrightarrow 0\ 1\ 0 \\ \mathsf{Var}_{s2} = \{ V1, V3 \} & \Longrightarrow 1\ 0\ 1 \\ \multicolumn{2}{c}{\ldots} \end{array}$$

Fig. 2: Example (bit-string) chromosome representation of tuple assignments to relation Var.

3.2 Genetic algorithm

For AdaptiveAlloy's initial generation, we employ a combination of random gene assignment for the majority of individuals and a domain-specific strategy. This strategy generates two special chromosomes: one composed of all 1s in a bit-string format, representing an instance incorporating all tuples for each relation, and another composed of all 0s, representing an empty tuple set instance, thus providing a diversity of alleles in the population. Figure 3 provides an illustration of various aspects of AdaptiveAlloy, including its (a) chromosome representation, (b) two arbitrarily selected chromosomes corresponding to Listing 1.1, (c) transformation from tuple-sets form into bit-string chromosome, (d) crossover step for generating a new bit string, and (e) mutation process over the bit-string.

Selection: AdaptiveAlloy's selector employs a combination of *elitism* and *tournament selection* strategies to determine the population for the subsequent generation. Initially, the selector retains the e most-fit chromosomes from the current population, adding them unchanged to the next generation (*elitism*). Subsequently, it randomly selects t individuals from the remaining population and iteratively picks the most-fit individual among those t, repeating this process until all individuals are chosen (*tournament selection*). The next generation consists of a set of *survivors*, comprising the elites (e) and a portion of the winners from the tournament selection, along with a set of *offspring* generated through crossover and mutation. The number of survivors and offspring is determined based on the ratio $rate_s$, with the total population size (p) being composed of these individuals for the subsequent generation.

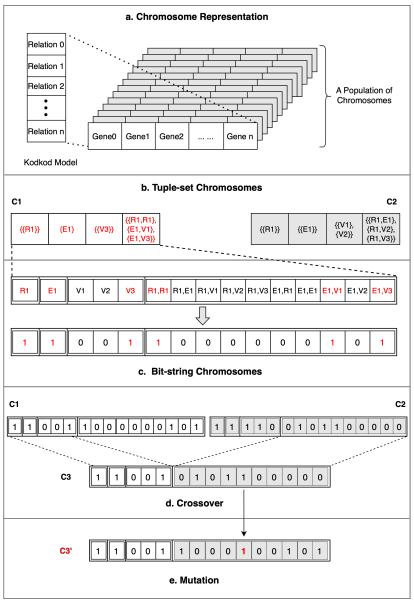

Fig. 3: AdaptiveAlloy's (a) chromosome representation, (b) two arbitrarily picked chromosomes for Listing 1.1, (c) transformation from tuple-sets form into bit-string chromosome,(d)crossover step for creating a new bit string, and (d) mutation over bit-string.

Crossover: After the mating pool is established, AdaptiveAlloy employs the *crossover* operation to produce offspring for the subsequent generation. The crossover process starts by randomly choosing two parent genotypes from the mating pool. Subsequently, a random bit index ranging from 0 to the length of the shorter of the two bit-string genotypes is selected as the *cut point* for *one-point* crossover. Using this cut point, the crossover operator generates two new individuals by exchanging the bits to the right of the cut point between each of the parents.

Mutation: To maintain genetic diversity and prevent premature convergence to local optima, AdaptiveAlloy employs mutation as a crucial genetic operator. The mutation process in AdaptiveAlloy involves a strategic combination of configurable mutation rates and a probability-based selection of mutation operators. The mutation process operates at two levels: chromosomes and genes within those chromosomes. The mutation rate for chromosomes, denoted as $p_{\text{individual}}$, determines the likelihood that a chromosome will be selected for mutation, while the gene mutation rate, denoted as p_{gene}, determines the likelihood that a gene within the selected chromosome will be altered. More specifically, it can be represented mathematically as follows:

$$P(\text{mutation}) = p_{\text{individual}} \times p_{\text{gene}}$$

where $P(\text{mutation})$ represents the probability of mutation occurring.

Moreover, the mutation operator is determined based on a probability distribution. The choice of mutation operator is made from a set of options each with its associated probability. These operators include:

- **Chromosome Creation:** If the selected chromosome has only 0s assigned to each individual, the creation operator generates a new equal-length bit chromosome by randomly assigning 0 or 1 for each gene in the chromosome.
- **Chromosome Removal:** Each gene in the selected chromosome is replaced with a 0, effectively altering the chromosome's composition.
- **Chromosome Transformation:** The original value of the selected chromosome is replaced with a newly generated bit chromosome.
- **Bit Transformation:** This operator focuses on altering individual genes within the selected chromosome. It randomly selects a gene and flips its value, thereby introducing localized changes.

3.3 Granular Fitness Analysis: Assessing Degree of Violation

Unlike EvoAlloy [23], which relies on high-level assessments of solution quality, our fitness function delves into the details of the relational formula structure, offering a nuanced assessment of candidate chromosomes. Specifically, our fitness function goes beyond simply counting violated constraints, aiming to capture the diversity and complexity of constraint violations.

To facilitate our fitness function's analysis, we categorize relational formulas into two main classes: (1) Elementary Formulas: These include multiplicity, comparison, and int comparison formulas. They represent basic building blocks of the relational formula and can be evaluated directly. (2) Composite Formulas: These connect multiple elementary formulas with logical operators and often depend on the truth values of more than one subformula. Examples include n-ary, binary, and quantified formulas.

We conceptualize the relational formula as a large abstract syntax tree (AST), with the global root symbolizing the conjunction of all subformulas. Figure 4 demonstrates the AST of the relational constraints of the running example. Each leaf node

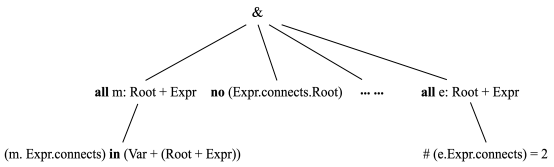

Fig. 4: The Abstract Syntax Tree of the Constraint Formulas

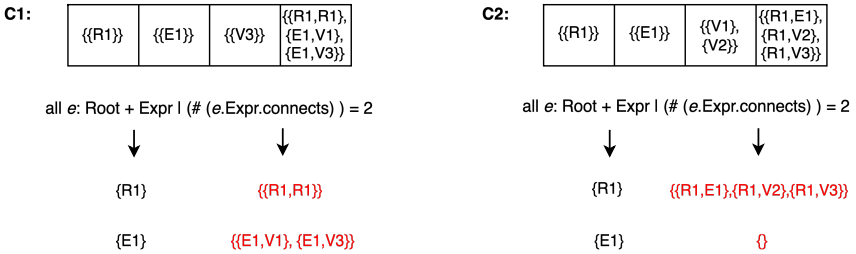

Fig. 5: Two chromosomes with distinct genetic makeup exhibit varying degrees of violation for the same constraint

corresponds to an elementary formula, while composite formulas serve as intermediate nodes, connecting smaller subtrees and leaf nodes. Our fitness function conducts a detailed examination of the AST of the relational formula to identify unsatisfied subformulas when a chromosome fails to meet the system constraints. This process provides a better understanding of the violated constraints, including the specific subformulas involved and the extent of their violation. For instance, consider a constraint ensuring each node in a graph has exactly two outgoing edges. If a chromosome violates this constraint, we pinpoint the specific subformulas responsible, such as the multiplicity formula expressing the number of outgoing edges for each node.

Chromosomes with different genetic makeup produce varying degrees of violation for the same constraints. Essentially, when a common constraint is evaluated as unsatisfied by distinct chromosomes, the dissimilarity in their genetic makeup often results in the violated tuples varying in both quantity and composition. By analyzing the specific tuples involved in constraint violations, we quantify the degree of violation for each chromosome. An example of this is illustrated in Figure 5, where chromosomes *C1* and *C2* both violate the same constraint, requiring different degrees of modification to satisfy it. This constraint stipulates that for all types of Expr, each one should connect to exactly two child Nodes. Upon evaluating their relational values assigned to Root, Expr, and connects, *C1* requires only one additional tuple for connects to satisfy this constraint. In contrast, *C2* has one extra connected node for R1 and is missing two for E1, resulting in a total of three tuples needing revision to meet the constraint.

Based on the detailed analysis of constraint violations and the tuple-wise changes needed for satisfaction, our fitness function computes a fitness score for each chromosome. This score represents the accumulated number of tuple-wise changes required to satisfy all constraints, offering a precise measure of each chromosome's proximity to a valid solution, as represented by the following formula:

$$\sum_{c_i \in Consts} F_t(c_i, ch)$$

where $F_t(c_i, ch)$ indicates the number of violating tuples when evaluating chromosome ch against the ith constraint. This approach precisely quantifies the distance of a specific formula from being satisfied by a chromosome in terms of the number of tuples needed to be altered, in contrast to abstractly counting how many relations are involved in the violation, as considered by prior work. It also supports our design choice of representing the problem using a bit-string chromosome at the tuple level.

3.4 Dynamic Weight in Fitness Computation

In addition to tracking the number of tuple revisions required to satisfy each subformula, our approach incorporates a dynamic weight in fitness value computation to tackle the challenge posed by exceptionally difficult constraints. These constraints can often lead to the search converging to local optima, hindering the effectiveness of the genetic algorithm. The adaptive fitness function is defined as follows:

$$w_i = w_i + \Delta w, \quad \Delta w = F_c(c_i, ch*)$$

$$f\,(ch) = \sum_{c_i \in Consts} w_i \times F_t(c_i, ch)$$

Here, w_i represents the dynamic weight of the ith constraint, which accumulates the number of failed tuples for chromosome ch when the ith constraint is unsatisfied. Initially, all weights are set to 1 and are subsequently updated every certain number of generations by adding given values Δw. $ch*$ denotes the best chromosome in the population found during the period between two consecutive updates of the weights, denoted as P.

The value of Δw is 1 if the best chromosome $ch*$ does not satisfy constraint c_i, and 0 when it is satisfied. This implies that the weights of unsatisfied constraints are increased by 1 periodically. As the population evolves, constraints that persistently remain unsatisfied over extended periods are penalized with higher weights.

When the genetic algorithm encounters a plateau caused by resistant constraints, the adaptive fitness function assigns much lower fitness values to chromosomes that satisfy these constraints. This favors the genes of chromosomes that do not satisfy these constraints to propagate to the next generation, helping to navigate the search out of local optima more efficiently.

This fitness function ensures truth-invariance by requiring the satisfaction of the Alloy specification, which necessitates satisfaction of all its relations and formulas. Our ablation study demonstrates that adaptive fitness outperforms plain fitness significantly, as discussed in detail in Section 4.

4 Experimental Evaluation

This section presents the experimental evaluation of AdaptiveAlloy. We have implemented AdaptiveAlloy's genetic algorithms engine on top of the Alloy Analyzer, its underlying finite relational model finder, Kodkod [21], and the Jenetics framework [26]. AdaptiveAlloy consists of two main components: the *Adaptive Evaluator* and the *GA Generator*. The *Adaptive Evaluator* assesses chromosome satisfiability, measures error degrees, and computes adaptive weights for the fitness function. The *GA Generator* produces initial populations, implements mutation operators for effective solution exploration, and facilitates chromosome conversion between Kodkod and bit-string representations. Additionally, it includes a component for transforming chromosome-level model instances into high-level Alloy models at the final stage of the evolutionary search. We used the AdaptiveAlloy apparatus for carrying out the experiments. The AdaptiveAlloy prototype and data is available on the project website [24]. Our evaluation addresses the following research questions:

- **RQ1.** How does AdaptiveAlloy compare to Alloy and EvoAlloy in terms of both effectiveness and efficiency?
- **RQ2.** What is the impact of Adaptive AST-Based Fitness compared to Non-Adaptive Fitness in terms of performance improvement?

Experimental subjects Our experimental subjects consist of publicly available Alloy specifications with varying sizes and complexities. More specifically, we use a list of twelve Alloy specifications modeling prominent algorithms (i.e., Chord models chord protocol for a peer-to-peer distributed hash table) or ubiquitous systems (i.e., Railway models a simplified railway system that declares

safety policies for trains) that are distributed with the Alloy Analyzer [1]. When performing the comparison experiments on this collection of specifications, we gradually increased the scope of analysis for each specification. Figure 6 displays the counts of variables and clauses in propositional formulas for each subject system. The data reflects a notable escalation in both variables and clauses as the analysis scope progresses from 5 to 25, highlighting the considerable rise in complexity and computational demands for broader analyses.

Fig. 6: Size Comparison of Variables and Clauses in Propositional Formulas for Subject Systems Across Two Analysis Scopes of 5 and 25.

Spec	Analysis Scope			
	5		25	
	Vars	Clauses	Vars	Clauses
abstract-Memory	2,622	4,604	639,497	1,240,899
birthday	2,503	4,245	179,148	335,100
ceilings	952	1,568	44,002	83,578
chord	13,582	32,551	11,643,597	43,930,456
com	9,031	16,447	2,046,548	4,093,018
dijkstra	578	512	63,878	62,552
fileSystem	2,037	3,965	149,946	409,604
grandpa	1,810	2,985	133,594	255,397
handshake	757	1,335	45,214	99,532
life	2,893	8,307	335,508	1,159,923
lists	2,948	8,613	205,940	756,769
railway	2,941	6,078	242,976	922,713

Experimental Setup We conducted all experiments on a PC equipped with a 64-core 4.3 GHz AMD Ryzen Threadripper 3990X processor boasting 128 threads and 64GB of RAM. To maintain consistency, each experiment was allocated 8 cores (16 threads) and 16GB RAM. Consequently, a maximum of three jobs could run concurrently on the system. Following parameter-tuning, we heuristically settled on the following parameters for all experiments: a population size of 32, with primary GA configurations initialized as follows: 60% offspring fraction, an overall gene mutation rate of 80%, and a one-point crossover with a 60% probability. Additionally, the likelihoods for each mutation operator were configured as follows: for a selected gene represented by a string of 0s, a 50% chance for bit-string creation and 50% for single-bit creation; for a non-empty selected gene, a 20% chance of deletion, 30% for bit-string transformation, and 50% for single-bit transformation. Regarding the hyper-parameters of our adaptive algorithm, the incremental adaptive weight was initially set to 1, and the adaptive step was set to 100 iterations.

4.1 Results for RQ1: Comparison against State-of-the-art

We conducted a comparative analysis of AdaptiveAlloy with two state-of-the-art tools: Alloy Analyzer (version 5.1) [1] and EvoAlloy [23]. This comparison aimed to assess how well AdaptiveAlloy scales and performs in terms of both effectiveness and efficiency across a range of experimental subjects. We evaluated the effectiveness of AdaptiveAlloy by comparing its scalability with Alloy Analyzer and EvoAlloy over increasing analysis scopes. Each technique was subjected to three stopping criteria: reaching a satisfying solution, exceeding the maximum memory allocation, or surpassing a 24-hour time limit. To reduce variance, we performed each analysis five times and recorded the analysis time.

Box plots in Figure 7 illustrate the analysis time (in logarithmic scale) in milliseconds (ms) obtained from Alloy Analyzer, EvoAlloy, AdaptiveAlloy, and AdaptiveAlloy without dynamic weight across increasing analysis scopes for various study objects. The notations “M” and “T” in the diagram denote that the

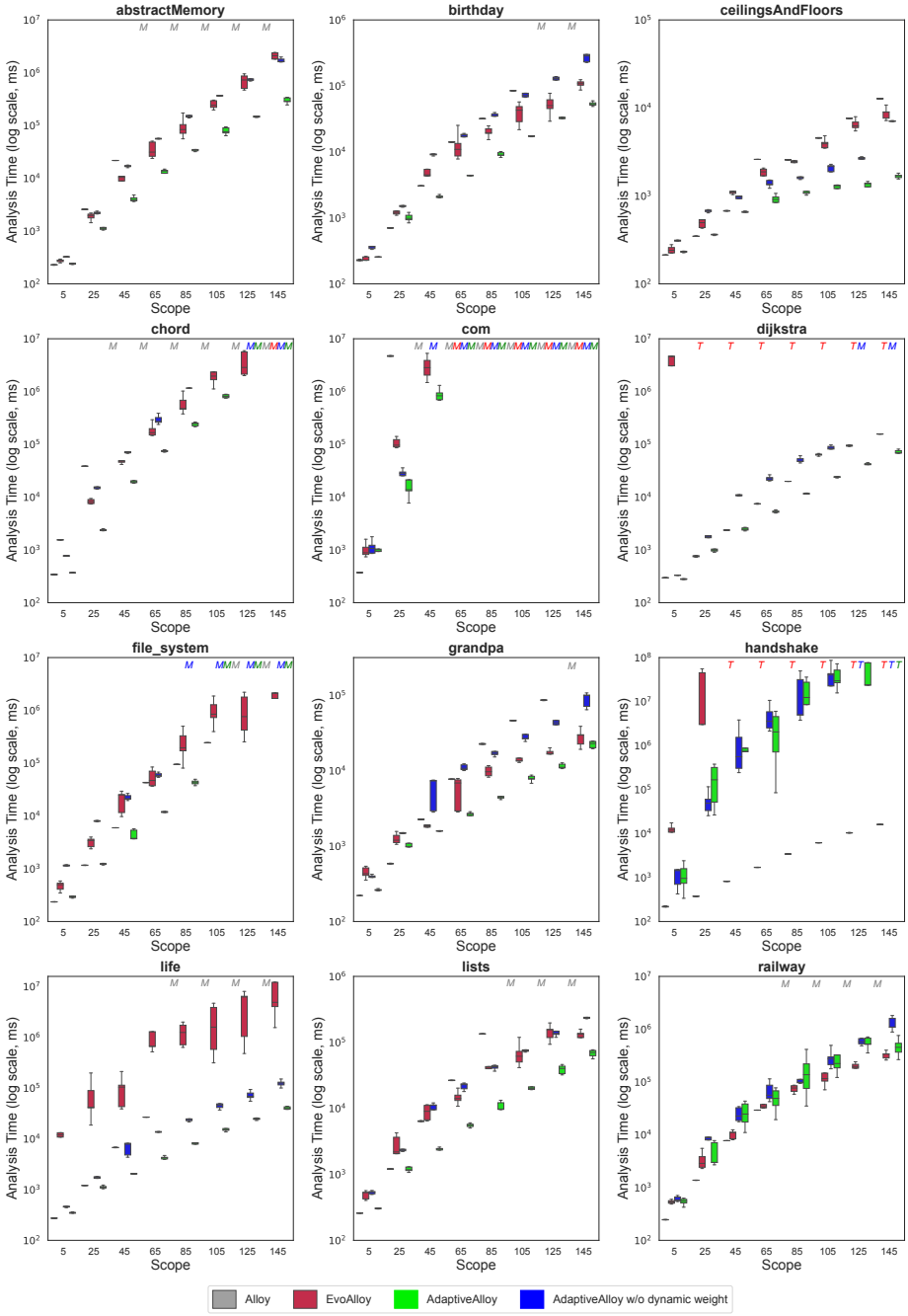

Fig. 7: Box plots depict the analysis time (in logarithmic scale) in milliseconds (ms) taken from **Alloy Analyzer**, **EvoAlloy**, **AdaptiveAlloy**, **and AdaptiveAlloy without dynamic weight** over the increasing analysis scope across objects of study. **M** denotes exceeding the maximum memory allocation, and **T** indicates surpassing a 24-hour time limit.

corresponding technique cannot identify a valid solution given the available memory and time resources, respectively; "M" signifies the technique exceeded the maximum memory allocation and "T" indicates the technique surpassed the 24-hour time limit. From the experimental results, several observations emerged. First, for more than half of the specifications, both EvoAlloy and AdaptiveAlloy could scale to larger analysis scopes compared to Alloy Analyzer, which frequently encountered memory limitations. This trend was particularly evident in specifications such as abstractMemory, birthday, and railway. Second, for smaller scopes where all three techniques performed well, EvoAlloy generally exhibited running times comparable to, or worse than, Alloy Analyzer. However, AdaptiveAlloy outperformed both the Alloy Analyzer and EvoAlloy in terms of the analysis time required to find solutions. Finally, while EvoAlloy's genetic algorithm (GA) struggled to efficiently solve certain problems, often hitting the time limit even for small scopes, AdaptiveAlloy's advanced adaptive GA approach proved effective. For instance, in the case of Dijkstra, AdaptiveAlloy achieved superior performance compared to both Alloy Analyzer and EvoAlloy.

AdaptiveAlloy achieves analysis time improvements of up to 181.62 times faster (with an average enhancement of 20.56 times) compared to the Alloy Analyzer and up to 172.10 times faster (with an average improvement of 33.04 times) compared to EvoAlloy across various specifications.

4.2 Results for RQ2: Ablation Study on Dynamic Weight

To investigate the impact of Adaptive AST-Based Fitness compared to Non-Adaptive Fitness, we conducted an ablation study on dynamic weight. Our GA-boosted approach was developed in two phases: initially incorporating advanced AST-based granular assessment of constraint violation degrees for fitness evaluation, and subsequently adding dynamic weights as indicators of the difficulty level in satisfying specific subformulas alongside fitness based on constraint violation degrees. Figure 7 outlines the runtime performance of both versions as the analysis scope increases across the objects of study. The complete form of AdaptiveAlloy(involving both advanced AST-based granular assessment of constraint violation degree and dynamic weights in fitness analysis) outperforms the version without introducing dynamic weight for almost every specification under analysis, with handshake as the only exception, on which both versions exhibit similar performance. It is noteworthy that, for most specifications, the efficiency gained from using adaptive fitness becomes increasingly significant as the analysis scope increases. Notably, for certain specifications like chord and file system, the non-adaptive version ran out of memory at smaller scopes compared to the one using adaptive fitness.

The Ablation Study compared Adaptive AST-Based Fitness with and without dynamic weights. Results show dynamic weights significantly outperformed static weights, with improvements ranging up to 5.78 times (with an average improvement of 3.62 times).

5 Discussion

The experimental results generally indicate that AdaptiveAlloy's GA notably improves the scalability of the state-of-the-art without experiencing the runtime efficiency degradation observed in EvoAlloy. However, a few drawbacks of AdaptiveAlloy are worth discussing. During the preliminary hyperparameter tuning experiments, we discovered that no universally optimal configuration achieves the best performance across all experimental objects, leaving us a tradeoff option that keeps significantly better running time on the majority of the specs, while maintaining acceptable performance for the rest. This results in AdaptiveAlloy having a larger variance in running time over handshake and railway, and performs equally or slightly less efficiently when compared to EvoAlloy. A dynamic parameter tuning technique can potentially further enhance AdaptiveAlloy's performance.

AdaptiveAlloy enhances the precision of its search guidance by evaluating constraint violation through AST traversal. While this strategy notably boosts fitness function accuracy and overall performance, it comes with increased memory consumption, potentially limiting scalability. Jenetics' memory management shortcomings exacerbate this issue by retaining allocated memory post-iteration. Our ablation study revealed that the non-adaptive AdaptiveAlloy variant faced memory constraints due to prolonged search iterations and residual memory accumulation. Implementing a memory-efficient GA engine and imposing a threshold on AST tracking depth could mitigate this challenge.

6 Related Work

Numerous extensions to Alloy and its automated Analyzer have been developed to enhance its performance and address scalability challenges [2, 4, 6, 7, 9, 13–15]. Notable among these are Titanium [5], which optimizes analysis time by generating a complete solution set for original specifications to inform revised ones, and Platinum [28], which partitions constraints into independent subclauses for more efficient analyses. Similarly, iAlloy [25] and SoRBoT [18] leverage solution reuse techniques to enhance efficiency. Aluminum [15] extends the Alloy Analyzer to generate minimal model instances by iteratively removing tuples from found model instances until a minimal instance is reached. Unlike our approach, Aluminum does not incorporate search-based solutions.

EvoAlloy [23] stands out for its focus on scalability, employing evolutionary algorithms to address Alloy Analyzer's limitations. However, its oversimplified problem representation and fitness design hinder its effectiveness. In contrast, our approach, AdaptiveAlloy, introduces a sophisticated GA with a novel fitness function and adaptive weight optimization to overcome these limitations.

PLEDGE [16] employs a hybrid metaheuristic search and SMT approach for improving constraint solving, particularly in system testing. While promising, PLEDGE's relies on UML models and OCL constraints. Additionally, PLEDGE lacks significant scalability improvements over Alloy due to its approach of delegating subformulas to an SMT solver, which can be a scalability bottleneck. In contrast, AdaptiveAlloy focuses on improving scalability and efficiency through a Genetic algorithm approach, bypassing intensive solvers.

There is extensive research on using evolutionary algorithms in software engineering [11]. Allmula and Gay [3] propose the use of adaptive fitness functions, however, their focus is on traditional program code coverage. Godefroid and Khurshid apply a genetic algorithm to analyze concurrent reactive systems for errors [10]. ACO-Solver utilizes Ant Colony Optimization for solving intricate string constraints [20]. Concolic Walk combines linear constraint solving with tabu search for complex arithmetic path conditions [8]. In contrast, our work focuses on bounded analysis of large-scale solution spaces specified in relational logic, requiring original chromosome encodings and fitness functions suitable for Alloy's relational logic.

7 Conclusion

In this paper, we introduce a novel approach that enhances genetic algorithm-based analysis, particularly within Alloy specifications. Our key contribution lies in the depth of the fitness function, which offers a granular examination of the specification's structure by traversing the abstract syntax tree. This nuanced evaluation, implemented in our tool, AdaptiveAlloy, enables effective navigation of the solution space, leading to globally optimal solutions. Additionally, we introduced an adaptive fitness, dynamically adjusting subformula weighting based on complexity. This optimizes resource allocation, enhancing GA-based analysis efficiency. Our comparative analysis with state-of-the-art Alloy Analyzer and EvoAlloy underscores significant scalability and efficiency improvements, with AdaptiveAlloy achieving analysis times up to 181.62 times faster than Alloy Analyzer and up to 172.10 times faster than EvoAlloy.

For future work, we plan to optimize the memory overhead introduced by the AST traversal tracking procedure as aforementioned. A potential tradeoff can be restricting the maximum depth of the AST for constraints being exploited. Our preliminary parameter tuning results reveal that no single global optimal configuration achieves the best performance for all experimental objects, thus we would also seek to explore incorporating a learning-based technique to dynamically tune the hyperparameters to enhance the performance when analyzing a diversity of specifications.

Acknowledgment

We thank the anonymous reviewers for their valuable comments. This work was supported in part by National Science Foundation awards CCF-1618132, CCF-1755890, CCF-1909688, CCF-2139845, and CCF-2124116.

References

1. Alloy toolset. `https://alloytools.org`, accessed: April 2024
2. Alhanahnah, M., Stevens, C., Bagheri, H.: Scalable analysis of interaction threats in IoT systems. In: ISSTA. pp. 272–285 (2020)
3. Almulla, H., Gay, G.: Learning how to search: generating effective test cases through adaptive fitness function selection. Empir. Softw. Eng. **27**(2), 38 (2022). `https://doi.org/10.1007/s10664-021-10048-8`, `https://doi.org/10.1007/s10664-021-10048-8`

4. Bagheri, H., Kang, E., Malek, S., Jackson, D.: A formal approach for detection of security flaws in the Android permission system. FAOC **30**, 525–544 (2018)
5. Bagheri, H., Malek, S.: Titanium: efficient analysis of evolving Alloy specifications. In: FSE. pp. 27–38 (2016)
6. Bagheri, H., Wang, J., Aerts, J., Malek, S.: Efficient, evolutionary security analysis of interacting Android apps. In: ICSME. pp. 357–368. IEEE (2018)
7. Brunel, J., Chemouil, D., Cunha, A., Macedo, N.: The Electrum analyzer: model checking relational first-order temporal specifications. In: ASE (2018)
8. Dinges, P., Agha, G.A.: Solving complex path conditions through heuristic search on induced polytopes. In: Proceedings of FSE. pp. 425–436 (2014)
9. Galeotti, J.P., Rosner, N., López Pombo, C.G., Frias, M.F.: Analysis of invariants for efficient bounded verification. In: ISSTA. pp. 25–36 (2010)
10. Godefroid, P., Khurshid, S.: Exploring very large state spaces using genetic algorithms. Int. J. Softw. Tools Technol. Transf. **6**(2), 117–127 (2004)
11. Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: Trends, techniques and applications. ACM Comput. Surv. **45**(1), 11:1–11:61 (2012)
12. Jackson, D.: Software Abstractions. MIT Press, 2nd edn. (2012)
13. Milicevic, A., Near, J.P., Kang, E., Jackson, D.: Alloy*: A general-purpose higher-order relational constraint solver. FMSD **55**, 1–32 (2019)
14. Mirzaei, N., Garcia, J., Bagheri, H., Sadeghi, A., Malek, S.: Reducing combinatorics in GUI testing of Android applications. In: ICSE. pp. 559–570 (2016)
15. Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: principled scenario exploration through minimality. In: ICSE. pp. 232–241 (2013)
16. Soltana, G., Sabetzadeh, M., Briand, L.C.: Practical constraint solving for generating system test data. TOSEM **29**(2), 11:1–11:48 (2020). `https://doi.org/10.1145/3381032`, `https://doi.org/10.1145/3381032`
17. Stevens, C., Bagheri, H.: Reducing run-time adaptation space via analysis of possible utility bounds. In: ICSE. pp. 1522–1534. ACM (2020). `https://doi.org/10.1145/3377811.3380365`, `https://doi.org/10.1145/3377811.3380365`
18. Stevens, C., Bagheri, H.: Combining solution reuse and bound tightening for efficient analysis of evolving systems. In: ISSTA. pp. 89–100 (2022)
19. Stevens, C., Bagheri, H.: Parasol: efficient parallel synthesis of large model spaces. In: ESEC/FSE. pp. 620–632. ACM (2022). `https://doi.org/10.1145/3540250.3549157`, `https://doi.org/10.1145/3540250.3549157`
20. Thomé, J., Shar, L.K., Bianculli, D., Briand, L.C.: Search-driven string constraint solving for vulnerability detection. In: ICSE. pp. 198–208 (2017)
21. Torlak, E.: A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications. PhD thesis, MIT (Feb 2009)
22. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: TACAS. pp. 632–647. Springer (2007)
23. Wang, J., Bagheri, H., Cohen, M.B.: An evolutionary approach for analyzing Alloy specifications. In: ASE. pp. 820–825 (2018)
24. Wang, J., Stevens, C., Kidmose, B., Cohen, M.B., Bagheri, H.: AdaptiveAlloy webpage. `https://sites.google.com/view/adaptivealloy`, May 2024
25. Wang, W., Wang, K., Gligoric, M., Khurshid, S.: Incremental analysis of evolving Alloy models. In: TACAS 2019. pp. 174–191. Springer (2019)
26. Wilhelmstötter, F.: Jenetics. URL: http://jenetics. io (2021)
27. Wu, N., Simpson, A.C.: Formal relational database design: an exercise in extending the formal template language. FAOC **26**(6), 1231–1269 (2014). `https://doi.org/10.1007/S00165-014-0299-6`, `https://doi.org/10.1007/s00165-014-0299-6`
28. Zheng, G., Bagheri, H., Rothermel, G., Wang, J.: Platinum: Reusing constraint solutions in bounded analysis of relational logic. In: FASE. pp. 29–52 (2020)