AlloyASG: Alloy Predicate Code Representation as a Compact Structurally Balanced Graph

Guanxuan Wu
The University of Texas at Arlington
Arlington, TX, USA
gxw6804@mavs.uta.edu

Allison Sullivan
The University of Texas at Arlington
Arlington, TX, USA
allison.sullivan@uta.edu

ABSTRACT

Writing declarative models has numerous benefits, ranging from automated reasoning and correction of design-level properties to automated testing and debugging of system implementations. Unfortunately, the model itself needs to be correct to gain these benefits. Alloy is a commonly used modeling language that has several existing efforts to repair faulty models automatically. Currently, these efforts are search-based methods that use an Abstract Syntax Tree (AST) representation of the model and do not scale, as ASTs suffer from exponential growth in their data size due to duplicate nodes. To address this issue, we introduce a novel code representation schema, Complex Structurally Balanced Abstract Semantic Graph (CSBASG), which represents code as a complex-weighted directed graph that lists a semantic element as a node in the graph and ensures its structural balance for almost finitely enumerable code segments. We evaluate the efficiency of our CSBASG representation for Alloy models in terms of it's compactness compared to ASTs, and we explore if a CSBASG can ease the process of comparing two Alloy predicates. Lastly, we identify several future applications of CSBASG, including Alloy code generation and automated repair.

CCS CONCEPTS

• **Software and its engineering → Formal software verification.**

ACM Reference Format:
Guanxuan Wu and Allison Sullivan. 2024. AlloyASG: Alloy Predicate Code Representation as a Compact Structurally Balanced Graph. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3640310.3674088

1 INTRODUCTION

Alloy [18] is a declarative, relational logic-based modeling language for describing the structural and dynamic properties of a system. Since Alloy is used to verify software system designs [5, 8, 41, 47], and to perform various forms of analyses over the corresponding implementation, including static checking [11, 19], systematic testing [25], repair [46], debugging [13] and to synthesize security attacks [2, 26, 34], there could be consequences if an Alloy model is faulty yet successfully compiles, as Alloy's robust enumeration of scenarios can place a high burden on the user to catch mistakes in the model [27], which could enable subtle but potentially dangerous design bugs to go undetected. Several iterative search methods based on ASTs have already been implemented for automated repair and partial generation of Alloy models [7, 16, 37, 49]. However, any iterative search methods for mutants or AST elemental changes that fix the Alloy code segments require expansive storage and have intractable search spaces due to the exponential growth of the code segment length, limiting their computational efficiencies.

In recent years, there have been several bodies of work that focus on automatically repairing code using machine learning techniques [3, 14, 17, 23, 32, 35, 45]. Such works provide valuable inspiration to us since we have a dataset of real-world fixing pairs consisting of faulty predicates written by students and their correct counterparts [24], effectively leading to a set of pairs of inputs and ground truths. Nonetheless, almost all the sequence-based algorithms treat the code segments as natural language sequences with a black-box function trained for faulty code as input and corrected code as output. These approaches ignore the logic of the target language, severely hindering the explainability of the process and limiting their applicability to modeling languages like Alloy.

To create a machine learning algorithm that is aware of the logic of the programming language we are generating code in, we need a logical structure-aware, vectorized, machine-and-human-readable, and computationally efficient code representation that improves upon the AST adjacency matrix representation that grows exponentially in size. Some of the representation learning methods [4, 29] train functions that output vectors from the AST, yet the nature of their dependency on the control flow paths in the AST of those general-purpose languages renders their methodologies implausible for a specification language such as Alloy, which is declarative and lacks control flow. Other methods [40, 42, 48] try to focus on the raw AST for a universal representation, yet those algorithms still suffer from rapid growth in the size of the AST. Therefore, this paper aims to create a structure that generates a compact representation of the code segment yet still gives the same complete information as the raw AST.

Recent work created an Abstract Semantic Graph (ASG) [10], which is close to meeting our needs. Still, the original designation lacks a numerical representation that provides an adjacency or Laplacian matrix that could be used in popular graph machine learning algorithms [12, 21, 28, 31, 36]. Therefore, we introduce a matrix representation of the ASG with the following objectives:

(1) One-on-one correspondence between the source code, the AST, and the graph matrix;
(2) Better performance in compactness and the potential to save space compared to the raw AST representation;

(3) Allow a direct comparison of the differences between code segments before and after applying a fix.

Such goals could be impractical for general-purpose, high-level languages such as C or Java, as most languages have a complicated type system and data structures that are not finite or even integer-enumerable. Despite that, for a simple, declarative, almost finitely enumerable language such as Alloy, we could create a Complex-weighted Structurally Balanced Abstract Semantic Graph (CSBASG) that translates the code segments into compact graphs for the subset covering most of the possible combinations. One of the advantages of this graph representation is the uniqueness of the semantics. For two semantically different terms in an AST, e.g., a variable or a unary or binary operator, there will only be one corresponding vertex in the CSBASG. This intuitively represents the code elements and their relationships with a reduced space cost.

Therefore, in this paper, we make the following contributions:
CSBASG: We introduce the concept of a CSBASG that meets our three objects (one-to-one correspondence, more compact, and supports direct comparisons). We provide an algorithm to convert an AST into a CSBASG and convert a CSBASG back to an AST and corresponding code segments.
Application to Alloy: We tailor the generation of CSBASG to the Alloy modeling language.
Evaluation: We evaluate the improvement on compactness and density of a CSBASG compared to an AST and evaluate how to compare multiple CSBASGs.
Open Source: We release our source code at https://anonymous.4open.science/r/AlloyASG-Release-CF0F/.
Future Directions: We present a series of research directions now enabled due to the representation of Alloy models with a CSBASG.

2 BACKGROUND

This section describes key components of Abstract Semantic Graphs and the Alloy modeling language.

2.1 Abstract Syntax Tree and Abstract Semantic Graph

We use the common notation of the abstract syntax tree (AST) for the raw representation of the syntactic structure of the Alloy source code [6]. We follow the formal definitions from [22]:

Definition 1 (Context-Free Grammar). *[22] A context-free grammar (CFG) $G \equiv (N, \Sigma, R, s)$ is a tuple where N is the set of nonterminals, Σ is the set of terminals, R is the set of production rules $a \longrightarrow b_1 b_2 ... b_N$ where $a \in N$ is a nonterminal and $b_i \in N \cup \Sigma$, and $s \in N$ is the starting nonterminal of the grammar. The language of G is the set of strings from the starting non-terminal.*

Definition 2 (AST). *[22] Given a context-free grammar (CFG) $G = (N, \Sigma, R, s)$, an abstract syntax tree (AST) $T \equiv (G, X, r, \xi, \sigma)$ is a tree with a set of nodes X, a root node $r \in X$, a mapping $\xi : X \longrightarrow \mathcal{P}(X)$ from each of the nodes to a subset of nodes as its children and each child set $\forall x \in X : \xi(x)$ is either empty or well-ordered, and $\sigma : X \longrightarrow (N \cup \Sigma)$ maps each node to its corresponding label as a terminal or nonterminal word in the CFG.*

According to [10], an abstract semantic graph (ASG) resolves the branches defining the properties of a node in the AST to its original definitions. But here, to create a more compact form of an ASG specifically for Alloy, we use the following more radical definition that combines every syntactically and semantically equivalent node:

Definition 3 (ASG). *Given a CFG $G = (N, \Sigma, R, s)$, an abstract semantic graph (ASG) $\mathcal{G} \equiv (G, \mathcal{V}, \mathcal{E}, r)$ where $\mathcal{V} \subseteq N \cup \Sigma$ is a set of nonterminal and terminal words as nodes (vertices) of the graph, and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V} \times \mathbb{C}$ is the set of edges between the words with the edge weight between nodes $v_i, v_j \in \mathcal{V}$ be $w_{ij} \in \mathbb{C}$, and r is the root of the corresponding AST of the ASG, i.e. the first node being visited.*

Note that since $\mathcal{E}$ is an arbitrary set of edges in forms of (v_i, v_j, w_{ij}), an AST or empirical construction of ASG could have more than one set of $\{w_{ij}\}$ to represent. If the weight function $w : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{C}$ ensures that $w_{ij} \neq 0$ if and only if there exists an edge from v_i to v_j, and $w_{ij} \neq w_{ik}$ for any $j \neq k$ be children of i, either at the same order of execution (children under the same node in the corresponding AST) or at the different order (if a nonterminal presents more than once in the AST and each time comes with a set of children). There could also be multiple AST links connecting two ASG nodes, and we will discuss how we form a weight value for this in Section 3.

2.2 Complex-weighted Graph and its Structural Balance

Since *Definition 3* gives a complex-weighted graph and our requirement for a vectorization of such ASG, the construction of the graph requires the edge values specifying the relationships between any of the two nodes. So we borrow the following structural balance concept from [43]:

Definition 4 (Complex-Weighted Structural Balanced Graph). *[43] For a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, let $a_{ij} = w_{ji}$ be the entries of the adjacency matrix A where $|a_{ij}| > 0$ indicates there exists an edge between v_j and v_i, then the graph $\mathcal{G}$ is said to be structurally balanced if all the entries of its adjacency matrix $A = [a_{ij}]^{N \times N}$ satisfies $a_{ij} \equiv |a_{ij}| \angle \theta_{ij} = |a_{ij}| \angle (\theta_i - \theta_j)$, where $\theta_1, ..., \theta_N \in (-\pi, \pi)$ is called the signatures of the nodes $v_1 ... v_N$, respectively.*

In the ASG context, intuitively, the signatures of the nodes $\theta_1, ..., \theta_n$ could encode the syntactic and semantic properties of each terminal or nonterminal node and the magnitudes of each edge $|a_{ij}|_{i,j \in \{1...N\}}$ encodes their relative properties, such as if a node is the left (1st) or right (2nd) child of a binary operator. We chose this concept for its mathematical solid properties. Let $d_i = \sum_{j \in 1}^{N} |a_{ij}|$ which is the in-degree of node v_i and let $D = diag(d_1, ..., d_N)$ be the diagonal matrix giving the in-degrees of each node on their corresponding rows, and define $L = D - A$ as the Laplacian matrix, then we have the lemma below [43]:

Lemma 1. *[43, 44]*
The following are equivalent:

(1) *Complex weighted graph $\mathcal{G}(A)$ is structurally balanced.*
(2) *Zero is a eigenvalue of L with eigenvector $\boldsymbol{\zeta} = [1\angle\theta_1, \ldots, 1\angle\theta_N]^T$.*
(3) *$D_\zeta := diag(\zeta)$ such that $\widehat{A} = D_\zeta^{-1} A D_\zeta$ is nonnegative and $\widehat{A} = [|a_{ij}|]_{N \times N}$.*
(4) *$D_\zeta := diag(\zeta)$ such that $\widehat{L} = D_\zeta^{-1} L D_\zeta$ has a zero eigenvalue with an eigenvector being $\mathbf{1}$, where $\widehat{L} = D - \widehat{A}$.*

Proof. See proof of Lemma 1 in [44]. □

(a)

```
1. sig Person  {
2.   Tutors : set Person,
3.   Teaches : set Class
4. }
5. sig Group
6. sig Class  { Groups : Person -> Group }
7. sig Teacher in Person  {}
8. sig Student in Person  {}
```

(b)

```
/* Assuming a universe of 3 persons, the tutoring
 * chain of every person eventually reaches a Teacher. */
 9. pred inv15oracle {
10.    all p:Person | some Teacher&(^Tutors).p
11. }
```

(c)

```
12. pred inv15 {
13.    all p : Person { some t : Teacher {
14.      t in p.Tutors
15.      or t in p.Tutors.Tutors
16.      or t in p.Tutors.Tutors.Tutors
17.    }}
18.}
```

Figure 1: Alloy Model of a Classroom Management System with an oracle and student submission for inv15

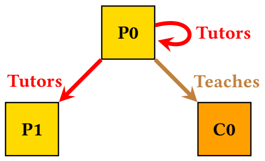


Figure 2: Counterexample highlighting difference between inv15 and inv15oracle

Notably, in the notation of [43, 44], each column specifies the outer edges of a node, and the row-based adjacency matrix must be transposed before acquiring the Laplacian.

2.3 Alloy

Alloy users write models that describe the properties of the system of interest. Then, the Analyzer helps the user understand their system by displaying the consequences of their properties, helping identify any missing or incorrect properties, and exploring the impact of modifications to those properties. To achieve this, the Analyzer uses off-the-shelf SAT solvers to search for scenarios, which are assignments to the sets and relations of the model such that all executed formulas hold. If no such scenario can be found, the Analyzer reports that the formulas are unsatisfiable.

To highlight how modeling in Alloy works, Figure 1 depicts the base model of a classroom management system. This model is from the Alloy4Fun dataset, which is a collection of submissions made by novice users learning Alloy [24]. Signature paragraphs introduce named sets and can define relations, which outline relationships between elements of sets. Line 1 introduces a named set `Person` and establishes that each `Person` atom connects to any number of (`set`) `Person` atoms through the `Tutor` relation and each `Person` atom connects to any number of (`set`) `Class` atoms through the `Teaches` relation. Line 5 introduces the named set `Group`, which contains no relations. Line 6 introduces the named set `Class` and states that each class has a set of people assigned to a group using the ternary relational `Groups`. Lines 7 and 8 introduce the named sets `Teacher` and `Student` as subsets (`in`) of `Person`.

Predicate paragraphs introduce named first-order, linear temporal logic formulas that can be invoked elsewhere. Figure 1 (b) depicts the oracle for exercise `inv15`. The predicate `inv15oracle` uses universal quantification ('`all`'), set multiplicity ('`some`'), set intersection ('`&`'), transitive closure ('`^`'), and relational join ('`.`') to express that for every person, there is some intersection between the set Teacher and the set of Tutors reachable from person p. Figure 1 (c) displays a faulty student submission for this exercise. The student attempts to use nested quantification to explicitly outline that a teacher should be reachable in 1 to 3 traversals down the Tutor's relation. The student submission assumes incorrectly that a person's tutor relation captures the set of all people who tutor p, but the tutor relation captures who p themselves tutors.

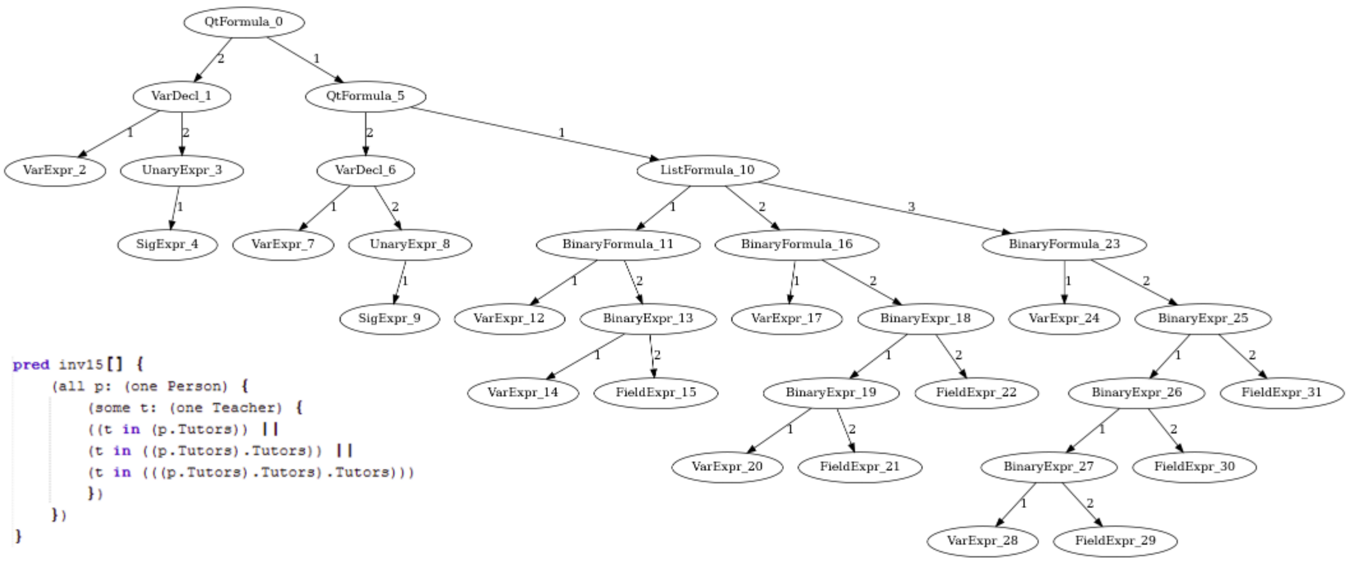
Figure 2 displays one of the counterexamples produced when the student submission is checked against the backend oracle in Figure 1 (b). The depicted scenario is valid for the oracle solution but not for student submission. To find this counterexample, an Alloy command is run that invokes the backend SAT solver. All commands require a scope, which places an upper bound of the universe of discourse. The default scope is 3, which means Alloy tries to find a counterexample highlighting that the two formulas are not equivalent using up to 3 Person, 3 Group, and 3 Class atoms.

2.4 The Alloy Grammar

We first highlight aspects of the Alloy grammar [18] that influence how we construct the graph representation. An Alloy model consists of a set of declarations with signatures as basic types, functions as processes, and predicates as logical arguments. Our dataset Alloy4Fun [24] covers only simple predicates without references to external libraries; therefore, creating representations for those code segments included in the dataset is an ideal starting point. Figure 3 gives an example of an Alloy predicate in the dataset that contains 32 AST nodes. From this representation, it is obvious that the raw AST representation is relatively huge and complicated, even with a short segment of code enveloped in a single predicate, mostly due to the redundancy of semantically identical nodes that appear multiple times in the AST since they appear multiple times in the original code. In the given example, there are three subset operators "in" and 6 relational join "." operators that are all treated as distinct nodes in the raw AST representation.

A predicate defines a scope that contains an expression while itself is a solid, invariant root for any fixing pair. We can easily enumerate all possible nodes under an expression (or formula): the finitely enumerable types of expressions or formulae, plus variable declarations that only occur under the quantifiers of first-order logic. In this paper, we assign them fixed signatures, yet an adaptive signature system could be used in the case of an extension of the CSBASG into either more types of nodes or extra optimizations.

Another property of Alloy is that almost all expression or formula nodes have a finite and fixed number of children, for instance, 1 for unary operators, 2 for binary operators, and 3 for the if-then-else sentences. In these cases, there could be a trivial, polynomial-based implementation of the $|w_{ij}|$ values for the edges between such a node v_i and one of its children v_j. Even so, there are some exceptions, such as a function call (having an indefinite number of parameters),



```
pred inv15[] {
    (all p: (one Person) {
        (some t: (one Teacher) {
        ((t in (p.Tutors)) ||
        (t in ((p.Tutors).Tutors)) ||
        (t in (((p.Tutors).Tutors).Tutors)))
        })
    })
}
```

Figure 3: An instance of an Alloy predicate in the dataset parsed with pretty string and its raw AST representation.

a list expression or formula (for example, the consecutive logical ANDs and ORs for a conjunctive or disjunctive normal form), or a set of variable declarations for the same quantifier. We will handle them on a case-by-case basis in Section 3.

3 CSBASG CONSTRUCTION

Intuitively, to create an ASG following *Definition 3*, we first need to do a pre-order traversal on the AST to identify and combine the semantically identical nodes. We construct an isomorphism between $\mathcal{V}$ and $N \cup \Sigma$, then rewrite each AST link into an ASG edge. The isomorphism is trivial since AST nodes with the same word are combined into the same node in the ASG. Nonetheless, constructing the ASG edge set by the AST relations ξ requires a cautious step: merging two AST nodes requires clarifying the orders of the visit to avoid confusion. For instance, in the example given in Figure 1, if we consider the binary operator "IN" as a single node without specifying the order of execution, we would not specify which of the following would be put as the right-hand operand: "`p.Tutors`", "`p.Tutors.Tutors`" or "`p.Tutors.Tutors.Tutors`". A naive approach that deviates a bit from the ASG definition is to use two numbers for the weight values w_{ij} instead of one, that is, for any of the existing link $x_2 \in \xi(x_1)$ in the AST, defining $v_1 = \sigma(x_1), v_2 = \sigma(x_2)$. Then, we create an edge between v_1, v_2 with a weight-pair $(\omega, t) \in \mathbb{N} \times \mathbb{N}$, specifying that x_2 is the ω-th child of x_1 in the order defined for $\xi(x_1)$, and it is the t-th time to visit v_1 in the program logic.

Nevertheless, we expect that for a matrix form of the ASG, we could take advantage of their linearity, especially for the potential application to machine learning operations. In addition, a matrix form of the graphs could be more easily compared with respect to multiple implementations of a predicate or other code segments of interest. With the naive construction above, it is challenging to proceed since the mapping from the pair given above to the edge value that serves as an entry in the adjacency matrix is nontrivial. Here, we could define $\mathcal{M} : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$ as mapping such that $\mathcal{M}(\omega, k)$ gives the entry in the matrix form of an ASG if it is the sole AST link in the ASG. The mapping could be defined arbitrarily, provided it maintains injectivity to avoid the simple, different-edge-same-value confusion (we will call it Type 1 Confusion later). Such forms the real part (magnitude) of the edges in the CSBASG.

Considering all the above, we formally define the complex-valued structurally balanced ASG as a subset of the general ASG constructions that maintain the structural balance.

Definition 5 (Complex-valued Structurally Balanced ASG (CSBASG)). *A Complex-valued Structurally Balanced ASG is a type of ASG $\mathcal{G} \equiv (G, \mathcal{V}, \mathcal{E}, r)$ where there exists a vector $\{\theta_1, \theta_2, ..., \theta_{|\mathcal{V}|}\}$, such that the weights in each of the edge $(v_i, v_j, w_{ij}) \in \mathcal{E}$ satisfy $w_{ij} = |w_{ij}|\angle(\theta_i - \theta_j)$. For each of the node $v_i \in \mathcal{V}$, $\theta_i \in (-\pi, \pi)$ is called the signature of v_i.*

Algorithm 1 gives the high-level process to convert an AST to a CSBASG. It follows the process of an abstract interpretation, where $\vec{t}$ in the recursive part gives a simplified interpretation state that tracks the times of node visitations in the pre-order traversal of the AST. We first initialize the nodemap and the list of angular signatures for each node, with the only element being the dummy overall root. Then, we assign each node an angular signature with regard to a predefined assignment, which outlines every unique signature. The number of unique angular signatures (semantically distinct nodes) is the number of rows and columns of the resulting adjacency matrix. After that, we initialize a state vector $\vec{t}$ to count the times of the order of visitation to each node. A recursive function then iterates over the AST beginning from the overall root, updates the adjacency matrix for each downlink from the root node according to the composition function β and the encoding method $\mathcal{M}$, while updating the counter of the given node with the corresponding entry of the state vector signaling the pre-order time of identical appearance in the AST before recursively visiting the subtree with the new root being a child of the local root.

Algorithm 1 Convert AST to CSBASG

```
1: Input: an AST $T \equiv (G, X, r, \xi, \sigma)$ parsed from the original code of an Alloy predicate, with root node $r$; an injective function $\varphi : N \cup \Sigma \to (-\pi, \pi)$ to map the words to their unique signature, and another injective function $\mathcal{M} : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$ outputs the magnitude of an AST edge given the positional argument of a link and the time has the sourcing node been visited. For the entries, we use $\beta$ as a recursive encoder that calculates the matrix entries iteratively.
2: Output: a CSBASG $\mathcal{G}(A)$ representing $T$ with its adjacency matrix $A$, and the signatures of the nodes $\{\theta_0, \theta_1, \theta_2, ..., \theta_{|\mathcal{V}|}\}$, each corresponds to a column of $A$.
3: $\vec{\theta} \leftarrow [0]$
4: $nodemap \leftarrow [(r, 0)]$
5: $\mathcal{V} \leftarrow \{r\}$
6: for $i \leftarrow 2, ..., |X|$ do
7:     $node \leftarrow X_i$
8:     $\theta_i \leftarrow \varphi(\sigma(node))$
9:     if $\theta_i \notin \vec{\theta}$ then
10:        $\vec{\theta} \leftarrow \vec{\theta} :: \theta_i$
11:        $\mathcal{V} \leftarrow \mathcal{V} :: node$
12:    end if
13:    $nodemap \leftarrow nodemap :: (node, \theta_i)$
14: end for
15: $A_{\vec{0}} \leftarrow [0]_{|\mathcal{V}| \times |\mathcal{V}|}$
16: $A \leftarrow$ ENCODE-RECURSIVE$(r, \vec{0}_{|\mathcal{V}|}, A_{\vec{0}})$
17: return $A, \vec{\theta}$
18: function ENCODE-RECURSIVE$(r_t, \vec{t}, A_{\vec{t}})$
19:    $\theta_t \leftarrow nodemap$.GET$(r_t)$
20:    $i \leftarrow \vec{\theta}$.GET-INDEX$(\theta_t)$
21:    $t \leftarrow \vec{t}$.GET$(i)$
22:    $\vec{t}' \leftarrow \vec{t}$
23:    $\vec{t}'[i] \leftarrow t + 1$
24:    $A_{\vec{t}'} \leftarrow A_{\vec{t}}$
25:    for $\omega \leftarrow 1, ..., |\xi(r_t)|$ do
26:        $c \leftarrow \xi(r_t)$.GET$(\omega)$
27:        $\theta_c \leftarrow nodemap$.GET$(c)$
28:        $j \leftarrow \vec{\theta}$.GET-INDEX$(\theta_c)$
29:        $A_{\vec{t}'}[j, i] \leftarrow \beta(\mathcal{M}(\omega, t), A_{\vec{t}}[j, i]) \cdot \angle(\theta_i - \theta_j)$
30:        $A_{\vec{t}'} \leftarrow$ ENCODE-RECURSIVE$(c, \vec{t}', A_{\vec{t}'})$
31:    end for
32:    return $A_{\vec{t}'}$
33: end function
```

In general, with a static signature assignment φ, the algorithm terminates in $O(|X|)$ since each of the nodes in the AST is visited exactly once and $|\mathcal{V}| < |X|$ strictly. We say an ASG is *generated* by an AST if the AST was converted to the same ASG with the same signature assignment and magnitude mapping.

By *Definition 5*, for a given Alloy predicate as an input, we define a fixed unique root $r \equiv v_1$, which is the predicate itself, and such root node is unique in each predicate. We can intuitively assign $\theta_1 = \varphi(\sigma(r)) \equiv 0$ as a starting point. Besides, we expect the same input parameters for the pairs of predicates to be directly compared, so the overall root node of each predicate only has one child, the local root expression or formula. Therefore, the only work left is to define the signature assignment function φ on the subset that could be present under a tree rooted by an expression or formula.

The function, $\mathcal{M}$, could be defined in multiple ways if it maintains a one-on-one correspondence between the syntactic positions and the execution or tree-walking order. However, for an AST with multiple syntactically and semantically identical nodes, there can be multiple edges that connect them, but those edges could have different positions ω and thus different $\mathcal{M}$ values. So, given an AST, we can define the three types of confusion that could happen in an ASG construction:

Definition 6 (Completeness of CSBASG). *Given a class of AST $T \equiv (G, X, r, \xi, \sigma)$, we say a CSBASG construction is **complete** if the parser L outputs the AST correctly by the generated CSBASG in Algorithm 1; that is, none of the confusions formed in the construction of $\mathcal{M}$ and β mappings:*

- *If $\mathcal{M}$ is not injective, i.e., for two edges with positional values $\omega_1 \neq \omega_2$ or tree-walking orders $t_1 \neq t_2$, $\mathcal{M}(\omega_1, t_1) = \mathcal{M}(\omega_2, t_2)$. This situation is a Type 1 Confusion, or Incomplete Edge Mapping Confusion.*
- *If an ASG parser L cannot parse a multi-edge in an ASG that corresponds to two AST links that have the same parent and different but syntactically and semantically equal children with different positions, e.g., $(a + b) - (c + d)$ where the central binary operator $-$ has two syntactically and semantically equal children of binary operator nodes $+$, then the situation causes a Type 2 Confusion or Twin Children Confusion.*
- *If an ASG parser L cannot parse a multi-edge in an ASG that corresponds to two AST links connecting semantically and syntactically equivalent pairs of nodes, but the parent nodes are at different positions in the AST, e.g., $(a \wedge b) \vee (a \wedge c)$, the ASG node "$\wedge$" has two left children of terminal node a, then the situation causes a Type 3 Confusion or Twin Parent Confusion.*

Here, the goal becomes creating the mappings $\mathcal{M}$ and β that ensure the parser L is free of confusion.

Polynomial-based Static CSBASG Encoding

By *Definition 6*, intuitively, the signature assignment φ is not relevant to the completeness of a CSBASG; that function could be arbitrarily defined or even *learned*. However, the processes directly related to the fidelity of the representation scheme must be predefined to ensure the completeness of the CSBASG. A naive implementation could consider the ω positional values as sequential numbers 1, 2, ... and add them together for the representation; however, if there is a ternary operator tor, then $tor(a, a, c)$ will have its children-position map be defined as $\{a : 3, c : 3\}$, causes confusion that has two children with a position 3 (and therefore, missing positions 1 and 2), causes a Type 2 confusion.

Here, we introduce a polynomial-based, integer-magnitude static CSBASG encoding for a theoretical language: each AST generated from its CFG has a maximum number of children p. Then, let T be the maximum number of syntactic and semantically identical nodes in the code segment. For the AST link drawn from a node to its $(\omega + 1)$-th child, let $\mathcal{M}(\omega, t) = 2^{(t-1)p+\omega}$ and β be a simple summation of the $\mathcal{M}$ values. Our construction of the scheme was defined in the order of $O(2^{pT})$. It could be despairing at a glance, yet consider that in many languages without a list expression, $p \leq 5$ could be achieved, and T only increases with the density and size of the code chunk, and raw AST representations also suffers such a polynomial growth.

Lemma 2. *The polynomial-based CSBASG magnitude encoding is complete and optimal; there is no complete encoding with the maximum magnitude of the ASG matrix entry less than $O(2^{pT})$.*

Proof. Completeness follows since $1 \leq \omega \leq p$ always holds, for each integer $k \in \mathbb{Z}$ such that $\mathcal{M}(\omega, t) = 2^k$, there is a unique pair of ω and t, there is no Type 1 Confusion. Consider multiple AST links from the syntactically and semantically same parent node to the syntactically and semantically same child, each with $\omega_1, \omega_2, ..., \omega_n$ as their positions, at $t_1, t_2, ..., t_n$-th visit of the parent node. Then, $\sum_{j=1}^{n} 2^{(t_j-1)p+\omega_j} = \sum_{k=0}^{pT} \mathbb{I}\{\exists j = 1, ..., n | k = (t_j - 1)p + \omega_j\} 2^k$ is a unique integer representation of the magnitude of an adjacency matrix entry in the ASG, so it is also free of Type 2 or Type 3 Confusion, gives the completeness of the representation. Optimality follows since each positive integer has a binary form $k = \sum_{j=0}^{\infty} c_j 2^j$, $c_j \in \{0, 1\}$, which each $c_j = 1$ corresponds to an edge with $j = (t - 1)p + \omega$. □

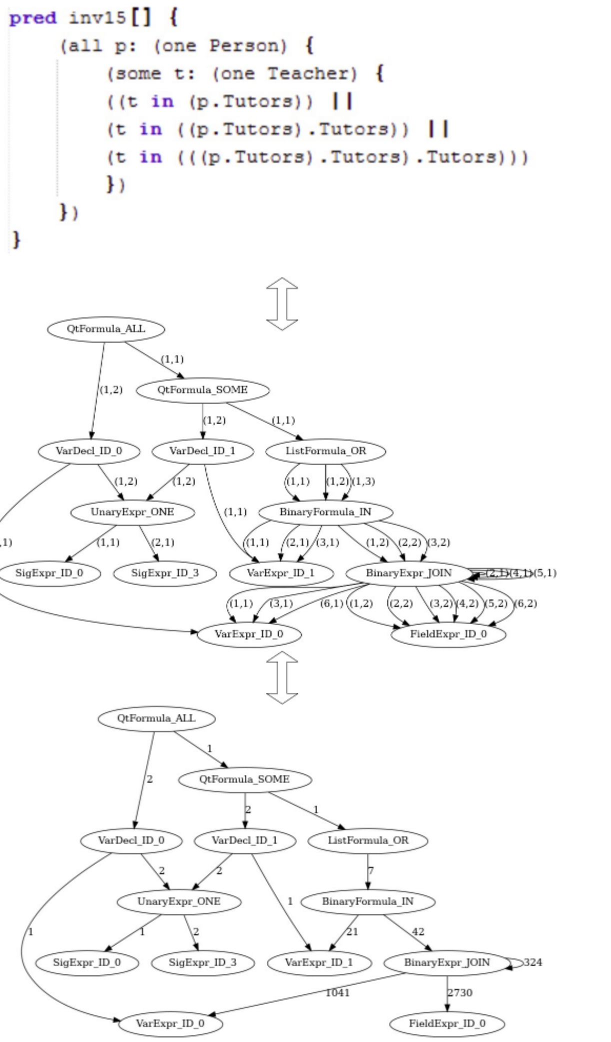
By *Lemma 2*, the summation-of-exponentials gives the most compact and accurate solution for the complete enumeration of the graph, and the creation of such a graph also depends on the enumerability of the grammar. However, a representation within the 32- or 64-bit bound could be achieved for a less-scaled language like Alloy. A heuristic generated by representation learning could also be achieved by learning the logarithmic parameters for a more general-purpose language with more syntactic elements and a larger AST.

The Decoding Algorithm

The decoding process is straightforward in theory. Words of the code are stored in correspondence to rows and columns of the matrix; each row or column corresponds to an individual word, and vice versa. Formally, we designate $L : \mathcal{G} \rightarrow \mathcal{T}$ as the parser that translates an ASG back to its uniquely corresponding AST. L is therefore a counterpart of *Algorithm 1* and bounded by the constructions of $\mathcal{M}$ and β functions. For each of the nonzero entries in the ASG adjacency matrix, we can break it into concrete links by reversing $\mathcal{M}$ and β. For the binary polynomial-based magnitude encoding given above, we take the integer value of the magnitude $|a_{ij}|$ and break it with regard to its binary form: each digit signing 1 in the binary form corresponds to a concrete AST link. In this case, an ASG node is broken into a set of AST nodes, each with its corresponding outer edges by the visit order. Since the possible edges are all examined once and the size of the entries determines the binary forms, we have a complexity of $O(|\mathcal{E}| \log(\max(w_{i,j})))$ or $O(|\mathcal{V}|^2 \log(\max(w_{i,j})))$ for the decoding process.

4 APPLICATION OF CSBASG ON ALLOY

There are two main features of Alloy that make for an ideal starting point to apply our CSBASG representation. First, as a first-order logic-based declarative language, we can easily construct a finite set



```
pred inv15[] {
    (all p: (one Person) {
        (some t: (one Teacher) {
            ((t in (p.Tutors)) ||
            (t in ((p.Tutors).Tutors)) ||
            (t in (((p.Tutors).Tutors).Tutors)))
            })
    })
}
```

Figure 4: An example illustrating the compactness and simplicity of CSBASG; the original code (upper) could be expressed as an equivalent multigraph (middle) with the AST in *Figure 1*, and turns into the adjacency relationship (lower) with the polynomial-based encoding. With the same predicate in Figure 1, the number of nodes reduced significantly from 31 to 12. All edges are now assigned an integer weight value, which forms an adjacency matrix.

of symbols given a model's source code. Second, while there are a few infinitely extensible structures in its grammar, which does limit the usage of empirical estimates for the maximum nodes under a given structure, even the infinitely extensible structures could be assumed to have a relatively low count of children.

To begin, we utilize a parser that classifies the syntactic properties of the code keywords into a limited number of categories. After that, we need to set a p value corresponding to the maximum number of children for each category, for instance, 1 for unary operators, 2 for binary operators, 3 for if-then-else, and a more

Table 1: Overview of problem set complexity.

Problem	#Sig	#Rel	#Exe	#AST
classroom_fol	5	3	15	10.00
classroom_rl	5	3	15	10.00
cv_v1	5	4	4	19.75
cv_v2	5	4	4	21.75
lts	3	1	7	19.71
production	5	3	4	14.25
train	6	3	18	23.44
trash_rl	3	1	10	4.80

significant number for the several uncovered infinite categories. This could cause ambiguity in some edge cases. Still, since Alloy is used to model structures within a bounded scope for each category of sets, there will unlikely be any long enumerations over 10 or 15 items, so we use $p = 17$ for the categories without a precise, finite maximum children number. While completeness is not guaranteed, we aim to solve completeness for the majority of cases. While it is possible to break the enumerations by putting the list of items into a linked structure, in most cases, the tradeoff of completeness in the edge cases delivers more benefits in the intuitiveness and compactness of the representation.

Inside each category, the nodes are differentiated by their semantic contents, which are still finitely enumerable since we have access to the signatures defined within a model file, and there are almost no numerical constants. Moreover, the numerical constant will be bound by the scope. Therefore, it is safe to assume that constants are limited to booleans and small integers. In our approaches, we divided the AST nodes under a paragraph of an expression or formula into 19 categories, plus the zero-signature node capturing the local root. Devolved from the syntactic categories are semantic subcategories, and the number of subcategories within a syntactic category ranges from 1 for the simple, invariant nodes to 32, which is the number of possible binary expressions.

Finally, we need to assign the angular signatures for each node. In our practice, we used a simple encoding by dividing the interval of a circle $(-\pi, \pi)$ into equal-length intervals with $D = \pi/13$ as the length of each interval. For each interval, we assign subticks that divide it into 128 or 65536 sectors, depending on the syntactic category. By the scheme above, we have obtained each AST node's syntactic-semantic pair (Ψ, ψ) using the results generated from the parser. Then, we map them on the unit circle as their signatures: each syntactic category goes with the beginning of a $\pi/13$-length interval, and each subtick above the interval starting point corresponds to a semantic subcategory. Note that this is just a demonstration of encoding and is only defined for convenience but not learned for optimality, and the angular correspondence could be any bijective mappings.

Figure 4 gives an example of the encoding from an Alloy predicate AST to its corresponding CSBASG using a polynomial-based static scheme. The syntactic-semantic pairs in the table give the properties of each node in the AST and combine the AST nodes with identical semantic and syntactic properties. The angular signature assigned to each node is $\Psi D + \psi\delta$, where $\delta = D/128$ for expression or formula nodes and $\delta = D/65536$ for the relational declarations that would appear below a quantifying expression or formula, since by Alloy syntax multiple variables could be declared under the same relational declaration node. In our example, there are six JOIN binary operators (.), which are represented as six distinct nodes in the original AST; in the ASG approach, the nodes are combined as a single node with the label BinaryExpr.JOIN.

The complex values could be used to compare two ASGs with shared parts, as discussed in Section 5.3.

5 EXPERIMENT

We address the following research questions:

- **RQ1:** What degree of compactness does the CSBASG representation of an Alloy model achieve?
- **RQ2:** Does our CSBASG representation of an Alloy model enable comparison between different predicates?

5.1 Dataset

Our dataset encompasses 6,307 models from Alloy4Fun[24], sourced from 8 different problem sets, filtered from a larger dataset to ensure each is compliable, runnable, nonempty, and not completely identical to the ground truth. For each problem set, Alloy4Fun has a base model that outlines all the signatures and relations, as well as a collection of empty predicates with English descriptions. Users can then attempt to fill in the predicate to match the English description, and their submission is checked against a backend oracle for correctness. Table 1 gives an overview of the complexity of each problem set: Column **#Sig** is the number of signatures, **#Rel** is the number of relations, **#Exe** is the number of different predicates uses can try to complete on Alloy4Fun, and **#AST** is the average number of AST nodes in the oracle solutions.

Our 6,307 models contain the base signatures from the Alloy4Fun problem set along with two predicates: a student-written predicate named InvX (or PropX, X is an identifying number) and another predicate called InvXC that is oracle solutions from Alloy4Fun. For each model, we run two predicates, *overconstrained* and *underconstrained*, which determine if there are cases that satisfy InvX but not InvXC, or vice versa. If a model has no cases for both *overconstrained* and *underconstrained* predicates, then InvX is formally correct. This gives four categories of the models: correct ($InvX = InvXC$), overconstrained only ($InvX \subsetneq InvXC$), underconstrained ($InvXC \subsetneq InvX$), and both overconstrained and underconstrained ($InvX \not\subset InvXC \land InvXC \not\subset InvX$), covers all possible cases.

5.2 Metric 1: Compactness of Representation

We evaluate the compactness of the representation by comparing the number of nodes in the ASG representation compared to the number of nodes in the AST. Table 2 gives an overview of these results. Column **Problem** displays the problem set from Alloy4Fun [24], and column **#Models** displays the number of unique student submissions. From each problem set, the student-written submissions, which we refer to as mutant predicates, are categorized further into four mutant types given their coverage of the set indicated by the ground truth: CORRECT, OVERconstrained, UNDERconstrained, or BOTH over- and underconstrained (Column **Mut.Type**). The percentages given are the ratio of the reduction

Table 2: Compactness test results of the dataset.

Problem	#Models	Mut.Type	Mut.%	Oracle%
lts	113	OVER	24.65	40.52
lts	138	BOTH	24.76	35.9
lts	66	UNDER	27.14	34.31
lts	61	CORRECT	36.66	39.19
trash_rl	150	OVER	13.42	7.10
trash_rl	267	BOTH	14.85	6.99
trash_rl	62	UNDER	31.67	11.68
trash_rl	252	CORRECT	20.25	6.73
train	170	OVER	23.47	25.23
train	277	BOTH	28.18	28.54
train	174	UNDER	32.70	24.43
train	75	CORRECT	29.44	19.43
classroom_fol	223	OVER	30.48	16.86
classroom_fol	1115	BOTH	36.48	26.08
classroom_fol	166	UNDER	37.19	28.54
classroom_fol	495	CORRECT	32.83	14.67
cv_v2	40	OVER	38.28	43.79
cv_v2	21	BOTH	38.37	43.08
cv_v2	12	UNDER	29.16	36.21
cv_v2	31	CORRECT	30.95	40.47
cv_v1	118	OVER	36.05	41.48
cv_v1	68	BOTH	32.49	39.97
cv_v1	54	UNDER	32.03	37.94
cv_v1	52	CORRECT	31.73	39.83
classroom_rl	223	OVER	30.48	16.86
classroom_rl	1115	BOTH	36.48	26.08
classroom_rl	166	UNDER	37.19	28.54
classroom_rl	495	CORRECT	32.83	14.67
production	25	OVER	18.84	20.73
production	22	BOTH	16.02	20.41
production	36	UNDER	22.89	22.80
production	25	CORRECT	14.75	21.12

Table 3: Comparability quantification results of the dataset.

Problem	#Models	Mut.Type	Mut.%	Oracle%
lts	113	OVER	69.66	31.03
lts	138	BOTH	71.46	36.55
lts	66	UNDER	60.87	42.91
lts	61	CORRECT	51.61	49.31
trash_rl	150	OVER	84.08	61.93
trash_rl	267	BOTH	89.39	62.35
trash_rl	62	UNDER	90.21	75.24
trash_rl	252	CORRECT	90.64	73.49
train	170	OVER	70.06	49.00
train	277	BOTH	75.16	50.97
train	174	UNDER	73.29	60.55
train	75	CORRECT	66.66	63.86
classroom_fol	223	OVER	77.54	63.51
classroom_fol	1115	BOTH	79.21	64.01
classroom_fol	166	UNDER	69.61	61.49
classroom_fol	495	CORRECT	82.07	71.77
cv_v2	40	OVER	65.78	33.73
cv_v2	21	BOTH	67.67	35.04
cv_v2	12	UNDER	70.70	36.82
cv_v2	31	CORRECT	63.93	38.38
cv_v1	118	OVER	58.99	34.84
cv_v1	68	BOTH	66.96	36.03
cv_v1	54	UNDER	69.96	37.52
cv_v1	52	CORRECT	62.88	38.86
classroom_rl	223	OVER	77.54	63.51
classroom_rl	1115	BOTH	79.21	64.01
classroom_rl	166	UNDER	69.61	61.49
classroom_rl	495	CORRECT	82.07	71.77
production	25	OVER	71.80	45.60
production	22	BOTH	70.67	42.82
production	36	UNDER	43.86	45.62
production	25	CORRECT	68.12	49.4

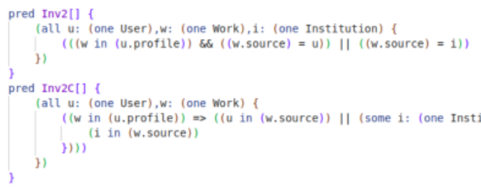
of nodes in the ASG representation compared to the raw AST for the student submission (Column **Mut.%**) and the oracle solution (Column **Oracle%**), e.g., there are 32.83% fewer nodes in the ASG of correct classroom_fol student submissions than the AST.

Overall, we achieved an impressive 27.25% reduction of nodes over the dataset with no loss of information. Interestingly, a smaller oracle does not guarantee less reduction. While the model with the smallest oracles by number of AST nodes, trash_rl, does in fact see the smallest reduction, the train model has the largest oracle but three models have larger reductions in nodes. All told the reduction the CSBASG provides over the AST is tied to the likelihood that the formula to be expressed has redundancy within its structure. Note that since the dataset is mostly single predicates designed as after-class practices for college students, the probability of semantic identical nodes in the AST is relatively low, and for a sub-AST with a large scale, the performance would increase accordingly in the single-metric of node reduction. Moreover, the size of the adjacency matrix scales up with $O(N^2)$ with the number of nodes so that a denser matrix could be achieved with the ASG scheme.

However, this metric should not be confused with saving spaces. Since in an AST the entries in the adjacency matrix are either 0 (indicating no links) or 1 (indicating a direct link), which only occupy 1-bit per entry in an optimized language or computational method, but *Lemma 2* indicates that there is no complete encoding without information loss that takes a number smaller than $O(2^{pT})$ which occupies pT-bits at least, so space saving is neither guaranteed nor indicated by the inferences and experiment above.

5.3 Metric 2: Comparability of Representation

Comparing a pair of code snippets to train a model for automated correction could be tedious, especially for syntactically correct but faulty predicates. Therefore, the CSBASG representation's comparability could enable graph mutations to annotate the pairs and create training data. We will call an atomic mutation either adding or removing an edge in the ASG. A relationship change between two nodes could also be represented as removing the old edge and adding the new one. To explore this idea, we have implemented a



```
pred Inv2[] {
    (all u: (one User),w: (one Work),i: (one Institution) {
        (((w in (u.profile)) && ((w.source) = u)) || ((w.source) = i))
    })
}
pred Inv2C[] {
    (all u: (one User),w: (one Work) {
        ((w in (u.profile)) => ((u in (w.source)) || (some i: (one Institution) {
            (i in (w.source))
        })))
    })
}
```

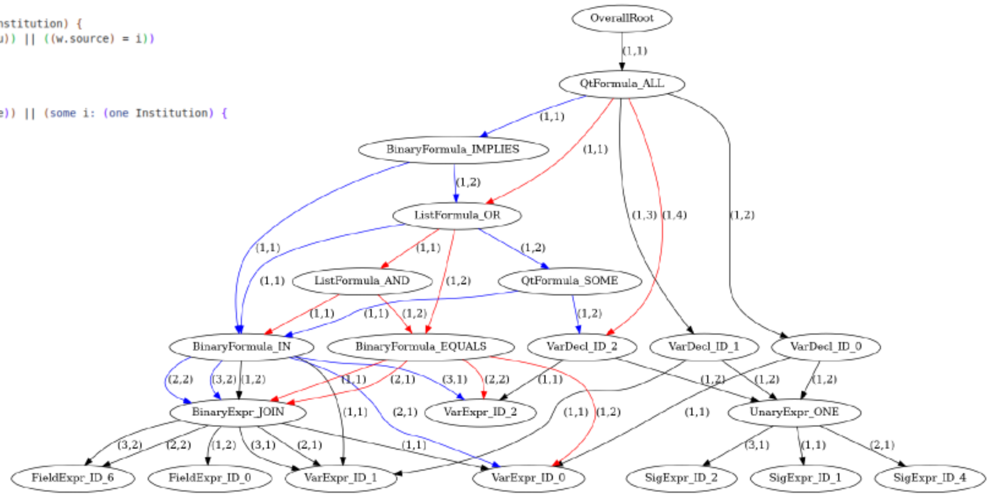


Figure 5: An example of a pair of student-written mutant predicate (Inv2) and the ground truth (Inv2C). The ASG edges in form (x, y) indicate the x-th visit of the sourcing node, and the positional relationship between the nodes is y. Black edges are present in both predicates, red edges are present in the mutant only, and blue edges are present in the ground truth only.

simple formula to decode the multiedges created when constructing the CSBASG using the polynomial method.

To get a holistic view of the comparability after decoding the CSBASG, we count the percentage of the shared edges among the predicate pairs. Table 3 gives an illustration of the per-category performance with regard to this metric. Again we present the problem set (Column **Dataset**), the number of submissions (Column **#Models**), and the mutant type of the submission (Column **Mut.Type**). Then, the next two columns present the percentage of shared edges. Column **Mut.%** is the average percentage of edges the mutant student submissions share with the oracle, and Column **Oracle%** is the average percentage of edges the oracle solution shares with the mutant student submission. The results are promising and show significant evidence of comparability: on average, 60.74% of edges in the oracle solution also appear in the student-written counterpart, and on average, 77.37% of edges in the student-written code also appear in the oracle solution.

To take a closer look at comparability, Figure 5 gives an example of the decoded CSBASG construction for comparing a pair of predicates in the same declarative environment, which, in our case, is the faulty and correct solution to the same problem. The common parts, depicted as black edges, show the correct declaration of the first two variables by the student given the right part of the graph, and the trivial correctness of the fields associated with the variables as elements of the pre-declared sets are shown in the bottom-left corner. However, most of the graph shows significant deviation of logic, as expected since the student predicate is *both* overconstrained and underconstrained. Straightforwardly, each of the red or blue edges in the graph is an atomic mutation between the pair of predicates; the red edge is a link to be broken, and the blue edge is a link to be established in a hypothetical automatic fixing application. In the example above, we could see that out of 41 edges shown in Figure 5, there are 10 common edges, 20 edges are added, and 11 edges are removed from the faulty model, which outlines the atomic mutations.

6 FUTURE WORK

In this section, we present several future improvements to the CSBASG representation and planned applications.

6.1 Laplacian of CSBASG and Mutant as a Control Operation

At the beginning of this research, we attempted to use a Laplacian matrix to represent the matrix. However, since some self-edges exist in the scope of the Alloy ASG, we could not as there is not currently a Laplacian construction for a directed graph containing self-edges. Nonetheless, technically, the CSBASG still holds its structural balance, and the ability to use mutants as control operations in a discrete-time system still holds. There are existing works [1, 30] that could potentially lead to constructions that could apply to various kinds of graphs with self-loops, which are potentially fitting constructions to apply towards a repair process that uses a pair of code segments as a control system in order to explore additional methods of a modification on a software system containing multiple functional code segments under consideration.

6.2 Mitigation of Exponential Growth of Matrix Entry

Lemma 2 mentions a strict exponential bound $O(2^{p^T})$to represent code segments within a repeat-free graph representation. Unfortunately, both the value of repeating nodes in the AST T roughly increases linearly with the increase in code segment size, and the maximum number of nodes under a category of nodes p could also end up a high constant that can vary between different code segments, all of which can prevent a universal, complete representation. Furthermore, even in our representation schema for an Alloy predicate body, as mentioned in Section 4, a rough number of $p = 17$ was assigned for any technically infinitely extensible structures such as a ListExpr. While some intuitive solutions exist, like a linked or nested approach to break an infinitely extensible node into multiple nodes linked with each other, they are also costly for time and space. Therefore, the exponential growth of the adjacency matrix remains a challenge for future improvements.

6.3 Automated Repair Using Machine Learning

Our research was initially motivated by a desire to create an input format for machine learning models that would preserve the logical structure of the underlying language, as current machine learning repair techniques treat code as natural language, which limits their likelihood of generating valid Alloy formulas. Since we already have a mechanism that could output the common edges, edges to be removed, and edges to be established, we could begin with a predictive model for the probability of the existence of each edge, as suggested by some state-of-the-art graph neural networks, by adapting those methods on the directed multigraph with a polynomial encoding.

6.4 Learning of Encoding and Code Generation

In this paper, we only attempted to ensure zero-information-loss encoding for Alloy code segments, but we still relied on a map of nodes since we have a static, same-distance pre-defined encoding. Nevertheless, in theory, we can stop this reliance by ensuring no different pair of edges have the same angular signature difference.

Consider a vector $\zeta = [1\angle\theta_1, 1\angle\theta_2, ..., 1\angle\theta_n]^T$ where n is the number of total nodes, such that, by *Lemma 1*, the eigenvector of the CSBASG. We could train this vector as an embedding of the nodes in the CSBASG, giving their angular signatures. In a future scenario, such a vector could be trained with some objective function in applications such as Alloy code generation. A common approach could be an dual-annealing optimization problem like

$$\min \sum_{(v_i,v_j,k)\in\mathcal{E}} |\theta_i - \theta_j|$$

$$\max \sum_{\forall k:(v_i,v_j,k)\notin\mathcal{E}} |\theta_i - \theta_j|$$

given $\theta_i \neq \theta_j$ for any $i \neq j$ and (v_i, v_j, k) is the k-th edge in the multigraph form between v_i and v_j, minimizing the signature differences between any pairs of nodes while maximizing the signature differences for the nodes that are not directly connected. For a sub-ASG containing solely expressions or formulae with such a vector ζ, we could estimate that it could be a high probability for an incomplete ASG to have edges connecting to the most possible nodes that have a lower angular signature distance with the parent node, helping us to fill out the blanks in an incomplete code segment or generate examples within a predefined declarative environment.

6.5 Possible Application on Other Languages

By construction, every programming language is built on a Context-Free Grammar, which we can utilize to create a complete construction and encoding for a CSBASG that can be used in place of an AST. However, our current completeness representation is built on the assertion that each node has limited extensibility. For real-world applications, there could be plenty of structures that either allow infinite extensions or, in a practical sense, have a value of the maximum links down from a category of node sufficiently high enough to hinder the CSBASG's scalability severely. One way to mitigate this is to focus on building CSBASGs of small-scale, simple, and reusable code segments, such as a function body, and then use this representation to help determine its proximity to any known faulty code segments.

7 RELATED WORK

In this section, we give an overview of the related work relevant to our planned future applications of a CSBASG for Alloy.

Automatic Repair of Alloy Models. Automatically repairing faulty Alloy models is a growing research field [7, 15, 16, 37, 49]. ARepair, ICEBAR, and BeAFix are generating valid repair techniques that involve bounded exhaustive searches [15, 16, 37]. TAR is a mutation-oriented repair technique designed specifically for Ally4Fun models [7]. ATR tries to find patches based on a preset number of templates [49]. While these techniques try to strike different balances in establishing a domain of patches to search through, the current state-of-the-art ATR can only correct 66% of the Alloy4Fun benchmark models. In addition to opening up the door to explore contextually aware machine learning repairs, our CSBASG could be incorporated into the backend of these existing techniques to try and improve their scalability, many of which define their search space using ASTs.

Code Generation for Alloy Models. HiGenA is a hint generator for Alloy4Fun exercises that uses historical edits to suggest a series changes a user can make to get to a correct solution [9]. HiGenA's user study highlights that users only find the hints helpful when their solution is already close to the correct answer, indicating room for improvements for hint generation. Overall, while restricted to Alloy4Fun exercise, HiGenA does establish a baseline for us to compare our code generation techniques against. Similar to hint generation, there is an existing body of work that completes partial Alloy models. ASketch takes as input a partial Alloy model with holes and a test suite that outlines the expected behavior [39]. ASketch then generates substitutions for the holes [38] and tries to find a completed model that passes all tests. However, despite several advances, ASketch still times out on the largest benchmark model [20, 33]. While ASketch does not currently utilize ASTs, other synthesis strategies could be built utilizing our CSBASG representation and compared to ASketch.

8 CONCLUSION

This paper introduces a novel graph representation of a programming language as a complex-valued multigraph with a polynomial-based adjacency matrix encoding. We evaluate the effectiveness of this representation by investigating the compactness improvements the CSBASG achieves compared to an AST for Alloy models, and the comparability CSBASG enables of Alloy predicates. In addition, we point out several future research directions that utilize this representation, including repair, code generation and improvements to the representation itself.

REFERENCES

[1] Behcet Acikmese. 2015. Spectrum of Laplacians for Graphs with Self-Loops. arXiv:1505.08133 [math.OC]

[2] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell, and Dawn Song. 2010. Towards a Formal Foundation of Web Security. In *2010 23rd IEEE Computer Security Foundations Symposium*. 290–304.

[3] Abdulaziz Alhefdhi, Khanh Hoa Dam, Xuan-Bach Dinh Le, and Aditya K. Ghose. 2020. Adversarial Patch Generation for Automatic Program Repair. *ArXiv* abs/2012.11060 (2020). https://api.semanticscholar.org/CorpusID:260510563

[4] Uri Alon, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *International Conference on Learning Representations*. https://openreview.net/forum?id=H1gKYo09tX

[5] Hamid Bagheri, Eunsuk Kang, Sam Malek, and Daniel Jackson. 2018. A formal approach for detection of security flaws in the Android permission system. *Formal Asp. Comput.* (2018).

[6] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. 368–377. https://doi.org/10.1109/ICSM.1998.738528

[7] Jorge Cerqueira, Alcino Cunha, and Nuno Macedo. 2022. Timely Specification Repair for Alloy 6. In *Software Engineering and Formal Methods: 20th International Conference, SEFM 2022, Berlin, Germany, September 26–30, 2022, Proceedings* (Berlin, Germany). Springer-Verlag, Berlin, Heidelberg, 288–303. https://doi.org/10.1007/978-3-031-17108-6_18

[8] Nathan Chong, Tyler Sorensen, and John Wickerson. 2018. The Semantics of Transactions and Weak Memory in x86, Power, ARM, and C++. *SIGPLAN Not.* 53, 4 (2018), 211–225.

[9] Ana Inês Oliveira de Barros. 2023. Data-Driven Hint Generation for Alloy using Historial Student Submissions. (2023).

[10] Edward B. Duffy and Brian A. Malloy. 2012. Design and Implementation of a Language-Complete C++ Semantic Graph. In *Proceedings of the 50th Annual Southeast Regional Conference* (Tuscaloosa, Alabama) *(ACM-SE '12)*. Association for Computing Machinery, New York, NY, USA, 170–175. https://doi.org/10.1145/2184512.2184552

[11] J. P. Galeotti, N. Rosner, C. G. López Pombo, and M. F. Frias. 2013. TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds. *TSE* (2013).

[12] Hongyang Gao, Zhengyang Wang, and Shuiwang Ji. 2018. Large-Scale Learnable Graph Convolutional Networks. *CoRR* abs/1808.03965 (2018). arXiv:1808.03965 http://arxiv.org/abs/1808.03965

[13] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. 2011. Specification-Based Program Repair Using SAT. In *TACAS*. 173–188.

[14] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (San Francisco, California, USA) *(AAAI'17)*. AAAI Press, 1345–1351.

[15] Simón Gutiérrez Brida, Germán Regis, Guolong Zheng, Hamid Bagheri, Thanhvu Nguyen, Nazareno Aguirre, and Marcelo Frias. 2023. ICEBAR: Feedback-Driven Iterative Repair of Alloy Specifications. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (<conf-loc>, <city>Rochester</city>, <state>MI</state>, <country>USA</country>, </conf-loc>) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 55, 13 pages. https://doi.org/10.1145/3551349.3556944

[16] Simón Gutiérrez Brida, Germán Regis, Guolong Zheng, Hamid Bagheri, ThanhVu Nguyen, Nazareno Aguirre, and Marcelo Frias. 2021. Bounded Exhaustive Search of Alloy Specification Repairs. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1135–1147. https://doi.org/10.1109/ICSE43902.2021.00105

[17] Shan Huang, Xiao Zhou, and Sang Chin. 2021. Application of Seq2Seq Models on Code Correction. *Frontiers in Artificial Intelligence* 4 (2021). https://doi.org/10.3389/frai.2021.590215

[18] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (apr 2002), 256–290. https://doi.org/10.1145/505145.505149

[19] Daniel Jackson and Mandana Vaziri. 2000. Finding Bugs with a Constraint Solver. In *ISSTA*.

[20] Ana Jovanovic and Allison Sullivan. 2022. Towards Automated Input Generation for Sketching Alloy Models. In *10th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2022, Pittsburgh, PA, USA, May 22-23, 2022*, Arnd Hartmanns, Ina Schaefer, Stefania Gnesi, and Nico Plat (Eds.). ACM, 58–68. https://doi.org/10.1145/3524482.3527651

[21] Ruoyu Li, Sheng Wang, Feiyun Zhu, and Junzhou Huang. 2018. Adaptive Graph Convolutional Neural Networks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence* (New Orleans, Louisiana, USA) *(AAAI'18/IAAI'18/EAAI'18)*. AAAI Press, Article 434, 8 pages.

[22] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 727–739. https://doi.org/10.1145/3106237.3106253

[23] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[24] Nuno Macedo, Alcino Cunha, José Pereira, Renato Carvalho, Ricardo Silva, Ana C. R. Paiva, Miguel S. Ramalho, and Daniel Silva. 2019. Sharing and Learning Alloy on the Web. arXiv:1907.02275 [cs.CY]

[25] Darko Marinov and Sarfraz Khurshid. 2001. TestEra: A Novel Framework for Automated Testing of Java Programs. In *ASE*.

[26] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2010. The Margrave Tool for Firewall Analysis. In *LISA*.

[27] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58 (March 2015), 66–73.

[28] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. 2016. Learning Convolutional Neural Networks for Graphs. *CoRR* abs/1605.05273 (2016). arXiv:1605.05273 http://arxiv.org/abs/1605.05273

[29] Benjamin Paassen, Jessica McBroom, Bryn Jeffries, Irena Koprinska, and Kalina Yacef. 2021. Mapping Python Programs to Vectors using Recursive Neural Encodings. *Journal of Educational Data Mining* 13, 3 (Oct. 2021). https://doi.org/10.5281/zenodo.5634224

[30] Ugasini Preetha P, M. Suresh, and Ebenezer Bonyah. 2023. On the spectrum, energy and Laplacian energy of graphs with self-loops. *Heliyon* 9, 7 (2023), e17001. https://doi.org/10.1016/j.heliyon.2023.e17001

[31] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2017. Modeling Relational Data with Graph Convolutional Networks. arXiv:1703.06103 [stat.ML]

[32] D. Sobania, M. Briesch, C. Hanna, and J. Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE Computer Society, Los Alamitos, CA, USA, 23–30. https://doi.org/10.1109/APR59189.2023.00012

[33] Allison Sullivan. 2017. *Automated Testing and Sketching of Alloy Models*. Ph. D. Dissertation. University of Texas at Austin.

[34] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2019. Security Verification via Automatic Hardware-Aware Exploit Synthesis: The CheckMate Approach. *IEEE Micro* (2019).

[35] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol.* 28, 4, Article 19 (sep 2019), 29 pages. https://doi.org/10.1145/3340544

[36] Hongwei Wang, Jia Wang, Jialin Wang, Miao Zhao, Weinan Zhang, Fuzheng Zhang, Xing Xie, and Minyi Guo. 2017. GraphGAN: Graph Representation Learning with Generative Adversarial Nets. https://doi.org/10.48550/ARXIV.1711.08267

[37] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. 2018. Automated Model Repair for Alloy. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 577–588. https://doi.org/10.1145/3238147.3238162

[38] Kaiyuan Wang, Allison Sullivan, Manos Koukoutos, Darko Marinov, and Sarfraz Khurshid. 2018. Systematic Generation of Non-equivalent Expressions for Relational Algebra. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10817)*, Michael J. Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl (Eds.). Springer, 105–120. https://doi.org/10.1007/978-3-319-91271-4_8

[39] Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. 2018. Solver-Based Sketching of Alloy Models Using Test Valuations. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10817)*, Michael J. Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl (Eds.). Springer, 121–136. https://doi.org/10.1007/978-3-319-91271-4_9

[40] Kesu Wang, Meng Yan, He Zhang, and Haibo Hu. 2022. Unified Abstract Syntax Tree Representation Learning for Cross-Language Program Classification. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension* (Virtual Event) *(ICPC '22)*. Association for Computing Machinery, New York, NY, USA, 390–400. https://doi.org/10.1145/3524610.3527915

[41] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically Comparing Memory Consistency Models. In *POPL*.

[42] Bingting Wu, Bin Liang, and Xiaofang Zhang. 2022. Turn Tree into Graph: Automatic Code Review via Simplified AST Driven Graph Convolutional Network. *Know.-Based Syst.* 252, C (sep 2022), 15 pages. https://doi.org/10.1016/j.knosys.

2022.109450

[43] Honghui Wu, Ahmet Taha Koru, Guanxuan Wu, Frank L. Lewis, and Hai Lin. 2023. Structural Balance of Complex Weighted Graphs and Multi-Partite Consensus. *IEEE Control Systems Letters* 7 (2023), 3801–3806. https://doi.org/10.1109/LCSYS.2023.3341992

[44] Honghui Wu, Ahmet Taha Koru, Guanxuan Wu, Frank L. Lewis, and Hai Lin. 2023. Structural Balance of Complex Weighted Graphs and Multi-partite Consensus. arXiv:2311.04389 [eess.SY]

[45] Geunseok Yang, Kyeongsic Min, and Byungjeong Lee. 2020. Applying Deep Learning Algorithm to Automatic Bug Localization and Repair. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing* (Brno, Czech Republic) *(SAC '20)*. Association for Computing Machinery, New York, NY, USA, 1634–1641. https://doi.org/10.1145/3341105.3374005

[46] Razieh Nokhbeh Zaeem and Sarfraz Khurshid. 2010. Contract-Based Data Structure Repair Using Alloy. In *ECOOP*. 577–598.

[47] Pamela Zave. 2015. How to Make Chord Correct (Using a Stable Base). *CoRR* abs/1502.06461 (2015).

[48] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 783–794. https://doi.org/10.1109/ICSE.2019.00086

[49] Guolong Zheng, ThanhVu Nguyen, Simón Gutiérrez Brida, Germán Regis, Nazareno Aguirre, Marcelo F. Frias, and Hamid Bagheri. 2022. ATR: Template-Based Repair for Alloy Specifications. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 666–677. https://doi.org/10.1145/3533767.3534369