Alloy repair hint generation based on historical data

Ana Barros[1,3], Henrique Neto[2,3][0009-0001-2701-7318], Alcino Cunha[2,3][0000-0002-2714-8027], Nuno Macedo[1,3] (✉)[0000-0002-4817-948X], and Ana C. R. Paiva[1,3][0000-0003-3431-8060]

[1] Universidade do Porto, Porto, Portugal {nmacedo,apaiva}@fe.up.pt
[2] Universidade do Minho, Braga, Portugal {alcino}@di.uminho.pt
[3] INESC TEC, Porto, Portugal

Abstract. Platforms to support novices learning to program are often accompanied by automated next-step hints that guide them towards correct solutions. Many of those approaches are data-driven, building on historical data to generate higher quality hints. Formal specifications are increasingly relevant in software engineering activities, but very little support exists to help novices while learning. Alloy is a formal specification language often used in courses on formal software development methods, and a platform—Alloy4Fun—has been proposed to support autonomous learning. While non-data-driven specification repair techniques have been proposed for Alloy that could be leveraged to generate next-step hints, no data-driven hint generation approach has been proposed so far. This paper presents the first data-driven hint generation technique for Alloy and its implementation as an extension to Alloy4Fun, being based on the data collected by that platform. This historical data is processed into graphs that capture past students' progress while solving specification challenges. Hint generation can be customized with policies that take into consideration diverse factors, such as the popularity of paths in those graphs successfully traversed by previous students. Our evaluation shows that the performance of this new technique is competitive with non-data-driven repair techniques. To assess the quality of the hints, and help select the most appropriate hint generation policy, we conducted a survey with experienced Alloy instructors.

Keywords: formal specification · intelligent tutoring system · automated hints · Alloy.

1 Introduction

Formal specification languages are based on mathematical formalisms and are used to describe the expected behaviour of a software component. Formal specifications are increasingly embraced by software engineering professionals, in *lightweight* formal development techniques such as automated synthesis, testing or monitoring. Moreover, they will possibly become even more relevant as advances in large language models push programming activities into higher levels of abstraction [29].

Alloy [12, 13] is a formal specification language that allows the automatic analysis of software design models with rich structure and behaviour. Due to its high-level of abstraction, flexibility and simplicity, Alloy is often used in introductory formal methods courses[4]. Yet, studies show that novices, and even experienced professionals, struggle with understanding and writing Alloy specifications [17]. The Alloy4Fun [16] web platform was developed in this educational context to ease the sharing of specification challenges with auto-grading, supporting instructors in classes and allowing students to study autonomously. Intelligent tutoring systems (ITS) for programming have long relied on automated feedback to support students in large classes and outside the classroom. Alloy4Fun, like regular Alloy, is solver-based and provides feedback for incorrect specifications as graphical counter-examples. This is a popular feature among Alloy practitioners and could, in principle, act as hints to help students progress towards solving a challenge when learning autonomously. However, studies find visual counter-examples have mixed results with novices [7, 8]. In fact, a recent user study [6] with different kinds of manually encoded hints concluded that only *next-step hints*, which highlight faults in incorrect specifications and provide tips on how to fix them, improved the immediate performance of the participants without jeopardizing learning retention.

Next-step hints are one of the most common feedback approaches in ITSs for programming [21]. A possible approach to generate these hints is through automated repair techniques. After repairing a faulty program to obtain a correct one, a next-step hint can be obtained by comparing both. One such technique has been proposed for Alloy [4], but it is only effective when students are already close to a correct specification, and the quality of the generated hints is not clear. An alternative approach is to rely on historical student submission data for the generation of hints, in order to guide the student towards paths that led to successful submissions. The expectation is that more understandable hints can be generated by mimicking successful peer behaviour.

This work proposes the first history-based hint-generation technique for Alloy, and presents its implementation as an extension to Alloy4Fun. Alloy4Fun was also designed to support research on formal methods education, and thus every interaction with the tool is anonymously recorded and made available to the instructors [16]. Based on this collected data, the proposed extension creates a directed graph encoding all attempts by previous students. Then, upon a hint request, it finds a path between the student submission and a solution using a customizable policy, and generates a next-step hint based on this path. The developers of Alloy4Fun maintain a publicly available dataset [15] of student attempts collected from their classes over the years. We relied on this dataset to evaluate our technique both for performance (effectiveness and efficiency) and for the quality of the hints (based on the opinions of experts on teaching Alloy). It achieved better results than the state-of-the-art tools. Furthermore, it can gen-

[4] http://alloytools.org/citations/courses.html

```
sig User {
   follows : set User,
   posts   : set Photo
}
sig Influencer extends User {}
sig Photo {
   date    : one Day
}
sig Ad extends Photo {}
sig Day {}

/* Every photo is posted by one user. */
pred spec1 {
}
//SECRET
pred oracle1 { all p: Photo | one posts.p }
//SECRET
check spec1 { oracle1 iff spec1 } for 3

/* Influencers are followed by everyone else. */
pred spec2 {
}
//SECRET
pred oracle2 { all u:User | Influencer - u in u.follows }
//SECRET
check spec2 { oracle2 iff spec2 } for 3
```

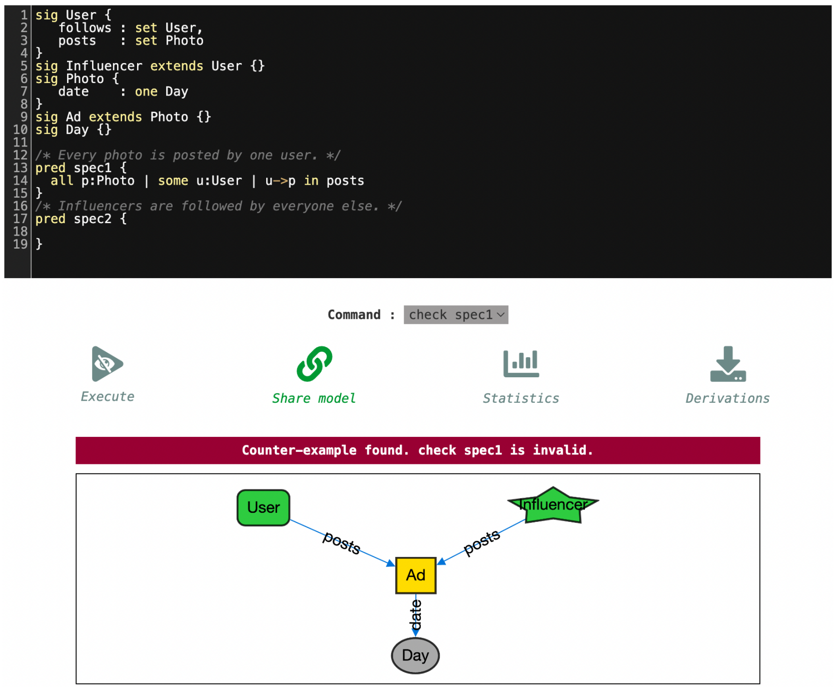
Fig. 1. Social network model with specification challenges

erate timely feedback, which is especially important in the educational context since students might easily feel frustrated if hints take too long to generate.

The remainder of the paper is structured as follows. Section 2 provides a short introduction to Alloy education, and Section 3 describes techniques for hint generation and Alloy repair. Section 4 presents our solution and its implementation, which is evaluated in Section 5. Section 6 presents conclusions and future work.

2 Teaching Alloy with Alloy4Fun

The Alloy language is based on temporal relational logic, but for simplicity, we'll restrict this presentation to the static subset of the language. Structure in an Alloy model is introduced through the declaration of *signatures* and *fields*. These can be restricted by multiplicity constraints and be hierarchically organized. The upper part of Fig. 1 depicts the structure of a social network system, a simplified version of an exercise in the Alloy4Fun dataset [15]. A signature `User` models users, with binary fields `follows` and `posts` relating each user with a **`set`** of users being followed, and a **`set`** of posted photos, respectively. Signature `Influencer`



```
sig User {
   follows : set User,
   posts   : set Photo
}
sig Influencer extends User {}
sig Photo {
   date    : one Day
}
sig Ad extends Photo {}
sig Day {}

/* Every photo is posted by one user. */
pred spec1 {
  all p:Photo | some u:User | u->p in posts
}
/* Influencers are followed by everyone else. */
pred spec2 {

}
```

Command : check spec1

Counter-example found. check spec1 is invalid.

Fig. 2. Incorrect submission to `spec1` in Alloy4Fun

extends users, denoting a subset of `User`. Signature `Photo` has a field `date` that relates each photo to exactly **`one`** day when it was posted; advertisements are a particular kind of photo, introduced by sub-signature `Ad`.

When validating a system design, one would impose additional restrictions over this model using temporal relational logic through *facts*. To promote maintainability, reusable formulas and expressions can be introduced through *predicates* and *functions*, respectively. Then *run* and *check* commands would be defined to animate the model or verify desirable properties, respectively. Commands are automatically executed by the Alloy Analyzer within a given bound for the universe. When teaching Alloy, a typical kind of challenge presented to students is to encode some of these logical constraints.

With this in mind, Alloy4Fun introduced the concept of model *secret*, allowing such challenges to be auto-graded [16]. Instructors write an oracle as a secret predicate and then use the Analyzer to check whether a student submission is equivalent to it. Two examples are shown in the bottom of Fig. 1. The student is asked to write in predicate `spec1` the constraint *"every photo is posted by one user"*. Hidden from the student through annotation *//SECRET*, predicate `oracle1` specifies a possible solution: for every photo `p`, there is exactly one user related with it through `posts`. Command `spec1` simply checks whether the student specification and the oracle are equivalent (with at most 3 atoms in each signature). Being a semantic test, the correct submission can be syntactically different from the oracle. A single Alloy4Fun model (which we call an *exercise*) can contain multiple *challenges*; the one in Fig. 1 has 2.

If a check command is invalid, the Analyzer (and Alloy4Fun) returns a graph-shaped counter-example where the equivalence does not hold. The user can navigate through alternative counter-examples and customize the visualization for better comprehension. As an example, Fig. 2 shows the student view of the exercise from Fig. 1 (i.e., secrets are hidden), where the student submitted an incorrect attempt to the `spec1` challenge and a counter-example was returned. In principle, counter-examples are helpful when debugging specifications, but studies show they are not the most adequate feedback for novice users [6].

Alloy4Fun collects anonymous data from all user interactions. So, whenever a student runs a command, it stores information such as the full model, the selected command and its outcome, and the identifier of the model it derived from. The resulting derivation tree allows the reconstruction of student paths, by identifying sequential attempts to the same challenge. The already mentioned dataset [15] collects this data for various editions of formal methods courses in the Universities of Minho and Porto, Portugal, between the Fall of 2019 and the Spring of 2023, totalling about 100 000 models.

3 Automatic Hint Generation

Next-step hints Although next-step hints are a popular kind of feedback in ITSs, there are some concerns that such hints may be counter-productive, namely due to hint abuse and avoidance [1], or the fact that they indicate students 'how' to fix rather than 'why' [18]. Nonetheless, studies [10, 14, 26, 25] suggest that next-step hints have no impact on long-term learning retention but often improve immediate performance, enabling students to learn more efficiently. A recent study on Alloy reached similar conclusions [6]. Moreover, there's an indication that accompanied by prompts for self-explanation, such hints may improve learning retention [20], although the results could not be replicated [19].

There are several techniques to automatically generate a next-step hint from an incorrect submission [21]: searching for steps that take the student closer to a reference solution, using previous successful submissions by peers, identifying known patterns in the incorrect submission, or trying to repair a solution to pass an oracle. Repair-based approaches have been proposed for Alloy, which we discuss below. However, these are often affected by scalability issues, and it's unclear how to select high-quality hints from alternative repair suggestions. In contrast, data-driven approaches do not suffer from performance issues and may generate more intuitive hints since they are based on historical submissions. The tradeoff is that they may be ineffective in large solution spaces or assignments with small historical logs. We are not aware of such techniques for specification ITSs, so we discuss them in the context of programming ITSs next.

Data-driven hint generation The first data-driven hint generation approach was proposed in the context of a logic-proof tutoring system [2]. It has since been adapted to platforms for programming [11, 23, 28], although not for specifications, as far as we are aware. The main idea behind these approaches is to use historical student submissions to build a graph of all traversed solution paths. Each node in the graph is the AST of a submitted attempt in a student path, and the transitions register the sequence of edit actions that lead from one submission to the other. To build the hint graph, all student paths are combined into a single graph by matching identical submissions, keeping the popularity of each state and/or transition, and marking correct submissions as goal states. When a student asks for a hint, if the current state is present in the hint graph, it calculates the optimal path towards a correct solution and generates a hint. In [2] Markov Decision Processes (MDP) were used to calculate the optimal path, but various other policies have since been proposed [22, 24]. Studies have used expert input to evaluate the quality of the hints resulting from different polices [22, 24].

The main challenge for this kind of approach is the size of the solution space. Besides being an obvious issue for assignments with little historical data, the solution space for expressive programming languages is so large that getting hits in the graph may be unlikely even with substantial historical data. Several approaches have been explored to address this, such as creating intermediate states [28], using program outputs rather than the actual AST as graph states [11], or employing canonicalization techniques to group semantically equivalent ASTs in the same graph state [27].

Automated Alloy Repair Automated program repair techniques generate fixes for programs that fail to pass a certain oracle. In education, this oracle can be written by the instructor, either a reference solution or a suite of tests, and then used to generate hints to fix student submissions. Some automated repair techniques have been proposed for Alloy specifications [30, 3, 4, 31].

ARepair [30] was the first repair technique for Alloy, using test cases as oracle. This makes it prone to overfitting, generating fixes that pass the tests but still break the expected properties. Moreover, Alloy models are typically not accompanied by test cases. In contrast, BeAFix [3] uses as oracles check commands. This is more natural in Alloy (and Alloy4Fun challenges) since models are typically accompanied by commands defining expected properties. Unfortunately, the pruning techniques proposed to improve performance rely on multiple commands and suspicious locations, and are not effective for simple Alloy4Fun specification challenges. TAR [4] was developed for the educational context and integrated into Alloy4Fun. It is focused on producing timely feedback to avoid student frustration (and to support the temporal aspects of Alloy 6). Its pruning technique evaluates previously seen counter-examples to avoid costly calls to the solver. It was shown to considerably outperform ARepair and BeAFix within a 1-minute timeout, but it is unfeasible for specifications far from a correct solution. ATR [31] is another technique to repair Alloy 4 specifications with commands as oracles. Although developed independently from TAR, it also uses counter-examples (and the closest valid instances) to avoid calls to the Analyzer. ATR

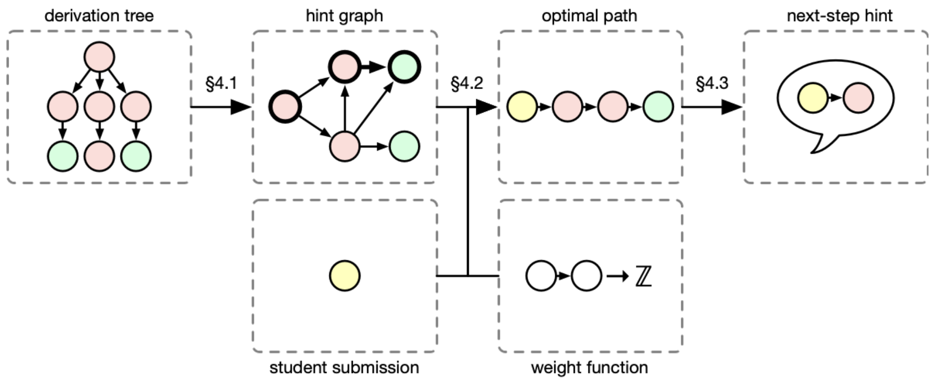


Fig. 3. Overview of the approach when submissions are present in historical data

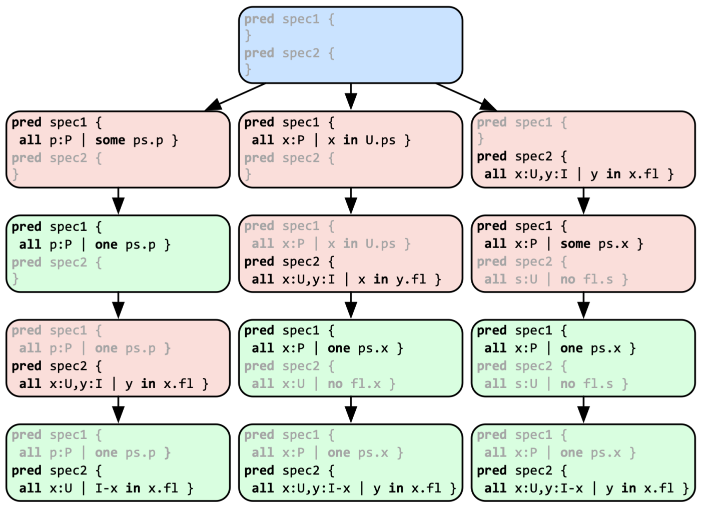
was shown to outperform the repair rate of ARepair and BeAFix, and to be more efficient than BeAFix.

4 Hints from Historical Alloy Data

The proposed technique adapts existing data-driven hint generation techniques for programming. Using Alloy4Fun historical data, it creates a graph that captures students' progress when solving a challenge, which is then used to generate hints for future students. This section describes the technique and its implementation, whose overview is presented in Fig. 3.

4.1 Hint Graph Construction

To generate hints, our approach relies on a graph of student submissions for each specification challenge, created from an Alloy4Fun dataset. These graphs are created offline and can be rebuilt from time to time as new data is collected. Each node in the graph is a normalized formula previously submitted by a student, labelled as correct or incorrect, and each edge represents a transition between two submissions. Each formula is unique in the graph, so similar submissions are merged, and the frequency of nodes and transitions are registered to be used in the pathfinding step. Formula comparison is performed at the AST level, so syntactically incorrect entries in the dataset are disregarded. As seen in Section 2, an Alloy4Fun exercise may contain multiple challenges, so the derivation tree must be split per challenge. The Alloy command called by each entry identifies the corresponding target challenge. To exactly identify the student submission and avoid considering the oracle as part of the graph state, we assume that each challenge command calls an empty predicate to be filled by the student, as exemplified in Fig. 1; the formula for each node is extracted from the content of



```
pred spec1 {
}
pred spec2 {
}
```

```
pred spec1 {
 all p:P | some ps.p }
pred spec2 {
}
```

```
pred spec1 {
 all x:P | x in U.ps }
pred spec2 {
}
```

```
pred spec1 {
}
pred spec2 {
 all x:U,y:I | y in x.fl }
```

```
pred spec1 {
 all p:P | one ps.p }
pred spec2 {
}
```

```
pred spec1 {
 all x:P | x in U.ps }
pred spec2 {
 all x:U,y:I | x in y.fl }
```

```
pred spec1 {
 all x:P | some ps.x }
pred spec2 {
 all s:U | no fl.s }
```

```
pred spec1 {
 all p:P | one ps.p }
pred spec2 {
 all x:U,y:I | y in x.fl }
```

```
pred spec1 {
 all x:P | one ps.x }
pred spec2 {
 all x:U | no fl.x }
```

```
pred spec1 {
 all x:P | one ps.x }
pred spec2 {
 all s:U | no fl.s }
```

```
pred spec1 {
 all p:P | one ps.p }
pred spec2 {
 all x:U | I-x in x.fl }
```

```
pred spec1 {
 all x:P | one ps.x }
pred spec2 {
 all x:U,y:I-x | y in x.fl }
```

```
pred spec1 {
 all x:P | one ps.x }
pred spec2 {
 all x:U,y:I-x | y in x.fl }
```

Fig. 4. A sample derivation tree with 3 paths for the exercise in Fig. 1

that predicate[5]. When extracting submissions to a certain challenge and removing syntactically invalid formulas, the pointer to the parent submissions must be updated accordingly to preserve the student paths.

For improved efficacy (i.e., the probability of a submission having a match in the graph), we apply a few canonicalizations specified in [27] that were sensible in the Alloy context, such as *sorting commutative operations* and *normalizing the direction of comparisons*. Additionally, since quantified variables in Alloy cannot be inlined, we apply *variable anonymization*. The same transformation is applied to submissions whenever a hint is requested. Note that we do not want to abuse canonicalization and end up with hints for a formula that differs too much from the concrete student submission. So, for example, we do not propagate **`not`** operators using De Morgan's laws.

To illustrate this process, consider the derivation tree in Fig. 4, that could be collected from the exercise in Fig. 1 (signature and field names abbreviated). It contains 3 paths, with incorrect and correct interleaved attempts to both challenges (`spec1` and `spec2`). The target challenge in each state is the one not greyed-out, green and red nodes represent correct and incorrect submissions, respectively, and the blue node is the root model shared by the instructor[6]. This will result in the two graphs in Fig. 5, with node and transition frequency identified by line thickness. Notice the normalization before merging, here just the name of the quantified variables. Notice also that there may be more than one semantically equivalent valid solution per challenge.

[5] This strategy may not hold for other kinds of Alloy4Fun challenges, in which case additional annotations could be used to identify the submission predicate.

[6] Technically, paths can branch if a student backtracks to a previous model. This phenomenon was negligible in the dataset, and does not affect the general procedure.

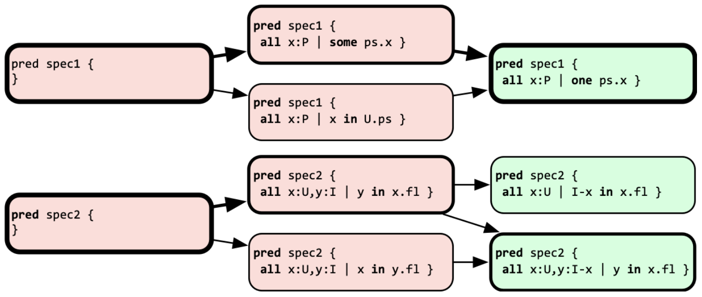


Fig. 5. Hint graphs resulting from the derivations in Fig. 4

4.2 Finding the Optimal Next State

The hint generation algorithm runs on demand when a student requests a hint. After locating the student's submission in the hint graph of the target challenge, the *current state*, the algorithm searches for the optimal path—according to the defined criterion—from it to any correct formula, the *goal state*. The first edge of this path indicates the transition the student should make to progress toward the goal, the *next state* that will be used to create the hint.

As discussed in Section 3, several criteria have been proposed to define the optimal path. Our goal was to keep the path finding process as general as possible, so we allow the instructor to define the desirable policy. This is done through the definition of a weight function on the edges of the graph from a set of available attributes. These attributes may be data-driven—namely the (relative) popularity of the edge in the source state, and the popularity of the source and target states—but also syntactic—namely the complexity of the edge transformation and the source and target formulas. The complexity of the states is given by the size of the respective AST. For the complexity of the edge, recall that a transition between states may encompass several actions between two successive submissions from the student. We measure the complexity of the edge as the tree edit distance (TED) between the two states, calculated using the state-of-the-art algorithm APTED[7].

Given the weight function on edges, the optimal path is calculated through a simple shortest path algorithm for weighted graphs.

4.3 Hint Message Generation

The next-step hint is generated from the optimal path. We consider two aspects to create the hint message: how far the student is from the optimal solution,

[7] https://github.com/DatabaseGroup/apted

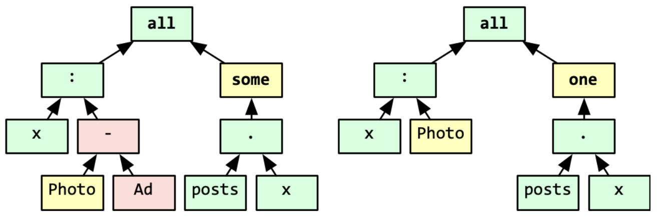


Fig. 6. Example of AST edit operations.

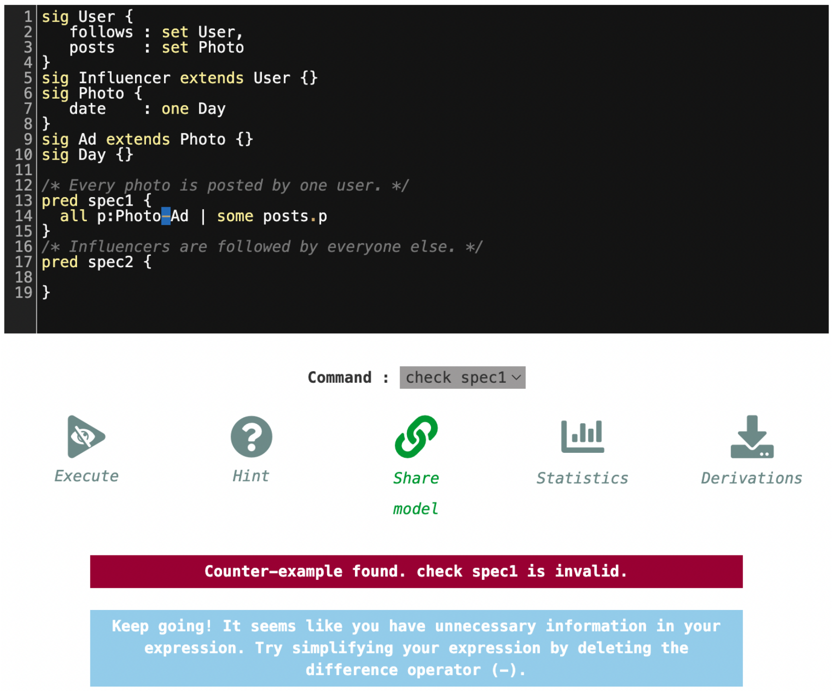
based on the TED between the current and the goal states; and the sequence of edit operations between the current and the next states. To calculate this sequence, we use an implementation[8] of GumTree [9], which calculates a mapping between AST nodes and uses the Chawathe et al. [5] algorithm for computing the edit sequence. The result is a sequence of *inserting*, *deleting*, or *moving* nodes, or *updating* a node's label. Since there may be dependencies between these edit operations, currently we select the first operation of the sequence for the hint. To translate an edit operation to a hint we use a message template for each operation type. The messages try to simulate what a teacher would say to a struggling student, and contain placeholders for operator-specific information that can be tailored for the Alloy language.

Consider, e.g., transforming `all p:Photo-Ad | some posts.p`, incorrect for `spec1`, into the correct `all p:Photo | one posts.p`, shown in Fig. 6. This requires 4 operations: move node `Photo` up, delete nodes `-` and `Ad`, and update node `some` to `one`, resulting in a TED of 4. The resulting hint message looks like this: *"Keep going! It seems like you have unnecessary information in your expression. Try simplifying your expression by deleting the difference operator (-)."*.

4.4 Handling Missing Hits

A pure data-driven approach fails for formulas absent from the historical data. To improve efficacy, one can construct paths from a previously unseen state until one already in the graph. To this purpose, we enhance our data-driven approach with a mutation-based component. Whenever a request does not exist in the graph, we generate variants according to a set of mutators. If a variant happens to already exist in the graph, a temporary edge from the current state to that variant is added with popularity 0, thus connecting the previously unseen formula to the graph and enabling the pathfinding procedure. These mutators—which are comprised by multiple edit actions—represent typical high-level transformations applied to a formula. In particular, we rely on the mutators proposed by TAR [4], which were specifically designed for the Alloy language. Currently, this process is restricted to a single mutation to avoid reaching a path too distinct from the student submission.

[8] https://github.com/GumTreeDiff/gumtree



```
sig User {
   follows : set User,
   posts   : set Photo
}
sig Influencer extends User {}
sig Photo {
   date    : one Day
}
sig Ad extends Photo {}
sig Day {}

/* Every photo is posted by one user. */
pred spec1 {
  all p:Photo-Ad | some posts.p
}
/* Influencers are followed by everyone else. */
pred spec2 {

}
```

Counter-example found. check spec1 is invalid.

Keep going! It seems like you have unnecessary information in your expression. Try simplifying your expression by deleting the difference operator (-).

Fig. 7. Hint provided for incorrect submission to `spec1` in extended Alloy4Fun

4.5 Deployment in Alloy4Fun

The proposed approach was implemented as a REST service, and we implemented an extension to the Alloy4Fun platform that uses the service to automatically provide hints to challenge attempts[9]. A new button was added to the interface that allows users to request a hint when an incorrect specification is submitted to a challenge. If the tool is able to generate a hint, it highlights a location in the editor and provides an explanatory message. This is shown in Fig. 7 for the example used in Section 4.3.

The service was implemented in Java—to take advantage of the Alloy Analyzer parser and AST—using the Quarkus framework. The hint graphs are stored in a new collection for the MongoDB database of Alloy4Fun. The weight function that determines the policy is provided through a JSON file that defines an arithmetic expression over the complexity and frequency attributes presented in Section 4.2.

Although optimal paths could be calculated live from the graph whenever a hint is requested, in practice, to make hint generation as fast as possible, we pre-compute the optimal next state for every state of the graph offline. When a hint is requested, it is just a matter of fetching the next state from the graph.

[9] https://github.com/anaines14/Alloy4Fun

Table 1. Statistics for the considered exercises

Exercise	Id	Challs.	Specs.	Syntatic	Training	Testing
Social Network	SN	8	22690	14943	10428	2793
Courses	Co	15	22516	14911	10431	2418
Train Station	TS	10	8158	6388	4394	1331
Production Line	PL	10	8078	6058	4156	1102
Trash LTL	TL	19	5279	4352	2788	890
Classroom FOL	CF	14	5893	4376	2702	663
Classroom RL	CR	14	6341	4248	2474	687
Trash RL	TR	9	4361	3059	1530	347
Trash FOL	TF	9	4092	2719	1425	194
Graphs	Gr	7	3211	2481	1281	370
Labelled Transition System	TS	6	3382	2076	995	393
Curriculum Vitae	CV	4	1199	854	596	218

Table 2. Quantitative evaluation results, all times in seconds

Id	Construction			Data-driven		Data+Mutations		TAR		
	States	T_G	T_P	Hits	T_H	Hits	T_H	Hits	T_H	Cmn.
SN	3605	165.5	7.9	1265 (45%)	0.01	1740 (62%)	0.7	539 (19%)	44.6	20
Co	4104	215.3	13.2	812 (34%)	0.01	1298 (54%)	0.7	502 (21%)	38.6	30
TS	1874	69.8	9.5	529 (40%)	0.02	751 (56%)	0.8	320 (24%)	38.2	9
PL	1862	113.6	5.4	373 (34%)	0.01	525 (48%)	0.7	297 (27%)	38.5	15
TL	1219	52.2	7.7	357 (40%)	0.01	569 (64%)	0.4	771 (87%)	9.5	23
CF	903	45.2	6.0	331 (50%)	0.01	463 (70%)	0.5	182 (27%)	26.7	13
CR	985	46.7	5.5	239 (35%)	0.02	330 (48%)	0.3	159 (23%)	35.9	7
TR	446	31.5	2.4	195 (56%)	0.02	260 (75%)	0.2	237 (68%)	0.9	3
TF	343	28.6	1.9	100 (52%)	0.02	154 (79%)	0.3	167 (86%)	0.9	5
Gr	599	27.7	3.7	162 (44%)	0.02	251 (68%)	0.2	175 (47%)	21.0	0
TS	489	22.8	3.7	74 (19%)	0.02	141 (36%)	0.2	37 (9%)	8.2	0
CV	324	11.7	1.6	44 (20%)	0.01	48 (22%)	0.7	61 (28%)	48.2	2

Table 3. Incorrect specifications selected for the questionnaire

Spec	Id	Incorrect
spec1	I1a	`all p:Photo	some u:User	u->p in posts`
	I1b	`all p:Photo	p in User.posts`
	I1c	`all p:Photo,u:User	p in u.posts`
spec2	I2a	`all i:Influencer,u:User	i in u.follows`
	I2b	`all u:User	Influencer in u.follows`
	I2c	`all u:User	u.follows in Influencer`

5 Evaluation

We evaluate the proposed hint generation technique quantitatively—addressing its effectiveness and efficiency—and qualitatively—comparing the generated hints with those suggested by experts. Specifically, we aim to answer the following research questions:

Table 4. Most popular answers by expert Alloy tutors

Id	Most popular location	#	Most popular hint	#
I1a	`some u:User | …`	8	update @ `some u:User | …`	8
I1b	`all p:Photo | …`	3	update @ `… in …`	2
			update @ `all p:Photo | …`	2
I1c	`all p:Photo,u:User | …`	7	update @ `all p:Photo,u:User | …`	5
I2a	`all i:Influencer,u:User | …`	4	delete @ `u:User`	2
			update @ `all i:Influencer,u:User | …`	2
			insert @ `all i:Influencer,u:User | …`	2
I2b	`Influencer`	5	insert @ `Influencer`	4
I2c	`… in …`	6	update @ `… in …`	5

RQ1 How effective is the tool when a hint is requested, i.e., how often can it generate a hint?

RQ2 How efficient is it in the various steps of the process, i.e., how long does it take to construct the graph and to generate a hint?

RQ3 How does it compare with repair-based approaches?

RQ4 What is the quality of the generated hints, and what is the impact of the specified policy?

5.1 Quantitative Evaluation

For the quantitative evaluation, we applied our technique to the Alloy4Fun dataset [15], which contains data for multiple exercises (each with multiple challenges). It contains about 66 000 syntactically correct student submissions to 12 different exercises, collected over 4 years. Table 1 shows the number of challenges per exercise (Challs.) and the aggregated statistics. The dataset was split into a training subset to construct the graphs and a testing subset to evaluate the performance. We split full paths in the dataset randomly 70%/30% (rather than splitting individual submissions, since our approach is based on previously traversed paths). Each entry in the testing subset was then run for a hint request in the purely data-driven technique, in the version that employs mutations for formulas absent in the graph, and also in the existing repair-based approach TAR with a maximum search depth of 2. All tests were performed on a commodity Intel Core i5-13600KF, with 32GB of RAM. Timeout for requests was set to 1min since timely feedback is critical in the educational context. Table 2 summarizes the results.

Regarding **RQ1**, Table 2 shows the hit rate (i.e., the number of specifications for which the tool was able to return a hint) for the purely data-driven and the mutation-enhanced versions. The hit rate of the former ranges from 19% to 56%, with a total average of 39%. Interestingly, the exercises with higher hit rate are not among those with the largest number of specifications in the historical log, which is possibly connected to the complexity of the challenges. Nonetheless, this hit rate will only increase as the exercises collect more submissions. Activating the mutation component for missed requests considerably increases the hit rate to an average of 57%.

For **RQ2**, we start with the graph construction step. Table 2 aggregates the results for each exercise, namely the number of unique formulas resulting in graph states, and the time to construct the graphs (T_G) and to compute the optimal next state (T_P). The selected weight function did not affect the performance significantly (shown values are for minimizing transition complexity). Results show that the whole process takes a few minutes for the exercises with more submissions, which is reasonable since this construction is expected to be performed sporadically offline. Regarding the hint generation step, Table 2 also shows the average time to generate a hint for both approaches (T_H). For the data-driven approach, this time is negligible for all exercises (recall that we precalculate the optimal next state offline). When enhanced with mutations, there is an expected increase on time, although still below 1s in average. This makes the technique feasible in answering live hint requests.

Regarding **RQ3**, Table 2 also shows the hit rate and time to retrieve a hint for TAR. The hit rate seems less predictable, ranging from 9% to 87%, with an average of 30%, well below our approach. Interestingly, the number of formulas for which both our data-driven approach and TAR can generate hints (Cmn.) is very small, suggesting that these approaches are complementary. As expected TAR takes considerably longer to generate a hint, with an average of 27s, since it is search-based and calls the solver to validate potential solutions.

5.2 Qualitative Evaluation

To evaluate the quality of the generated hints (**RQ4**), we asked experienced Alloy instructors how they would suggest a next-step hint for a set of incorrect specifications. For each of the two challenges from Fig. 1, we selected 3 frequently submitted incorrect specifications, shown in Table 3. We created a questionnaire that asked for hints in the shape of a target location and an edit operation (insertion, removal and update). We sent the questionnaire to 12 Alloy instructors unrelated with this work, and received 8 replies. We observed that, except for one case (`I1a`), the experts did not select the same next-step hint, highlighting the difficulty of automatically generating hints. Table 4 shows the most popular answers by the experts, both by location only by the whole hint (i.e., location plus edit operation).

Our approach allows policies to be customized through weight functions. To compare the answers of the experts with the results of our approach, we designed a few simple weight functions, some considering only the complexity of nodes (Cmp_N) and edges (Cmp_E), and others only the frequency of nodes (Frq_N) and edges (Frq_E). We also considered a couple of policies that combined these syntactic and data-driven attributes. For this evaluation, we do not consider the mutation-enhanced version of the technique, as we intend to evaluate the quality of the data-driven approach. For each policy we counted in how many of the 6 incorrect specifications the generated hint: *i*) was selected by *any* expert, and *ii*) was among the most *pop*ular answers by the experts. We consider whether

Table 5. Matches between hints generated by policies and expert hints

Policy	Location		Loc.+Op.	
	Any	Pop.	Any	Pop.
Cmp_N	3	3	3	3
Frq_N	2	2	2	2
Cmp_E	5	3	4	2
Frq_E	4	3	4	2
$Cmp_N \times Frq_E$	6	4	5	3
$Cmp_E \times Frq_E$	6	5	6	3

there was a match only on the identified location or in the whole hint. Table 5 shows the results.

Interestingly, results show that looking uniquely at the complexity of the edges (TED) results in hints closer to the experts than the purely data-driven policies. However, the best results are actually when considering both kinds of attributes simultaneously: with Cmp_E and Frq_E every hint generated was one also suggested by some experts, and often one of the most popular.

6 Conclusion

This paper presented the first data-driven hint generation technique for ITSs for learning formal specifications, namely for the Alloy language, and its implementation in the Alloy4Fun platform. The data-driven technique is complemented with a mutation-based component to handle absences in the historical data. Our evaluation shows that our approach outperforms an existing repair-based technique, and that with the right policy the generated hints can emulate those provided by experts.

Our expert questionnaires included an open question where most experts suggested feedback in shapes other than next-step hints, such as explaining the issue with the incorrect specification. Some studies suggest next-step hints accompanied by self-explanations can improve learning [20], but studies also find hints explaining issues are not well-received by novices [6]. Further studies are needed on how to implement these effectively. On the other hand, the quantitative evaluation showed a small overlap between the cases successfully handled by the data-driven and the repair-based approaches, suggesting that hybrid approaches may be worth exploring.

Acknowledgments. The work by A. Barros and H. Neto is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project EXPL/CCI-COM/1637/2021. The work by N. Macedo is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Aleven, V., Roll, I., McLaren, B.M., Koedinger, K.R.: Help helps, but only so much: Research on help seeking with intelligent tutoring systems. Int. J. Artif. Intell. Educ. **26**(1), 205–223 (2016)
2. Barnes, T., Stamper, J.C.: Toward automatic hint generation for logic proof tutoring using historical student data. In: ITS. LNCS, vol. 5091, pp. 373–382. Springer (2008)
3. Brida, S.G., Regis, G., Zheng, G., Bagheri, H., Nguyen, T., Aguirre, N., Frias, M.F.: Bounded exhaustive search of Alloy specification repairs. In: ICSE. pp. 1135–1147. IEEE (2021)
4. Cerqueira, J., Cunha, A., Macedo, N.: Timely specification repair for Alloy 6. In: SEFM. LNCS, vol. 13550, pp. 288–303. Springer (2022)
5. Chawathe, S.S., Rajaraman, A., Garcia-Molina, H., Widom, J.: Change detection in hierarchically structured information. In: SIGMOD Conference. pp. 493–504. ACM Press (1996)
6. Cunha, A., Macedo, N., Campos, J.C., Margolis, I., Sousa, E.: Assessing the impact of hints in learning formal specification. In: SEET@ICSE. pp. 151–161. ACM (2024)
7. Danas, N., Nelson, T., Harrison, L., Krishnamurthi, S., Dougherty, D.J.: User studies of principled model finder output. In: SEFM. LNCS, vol. 10469, pp. 168–184. Springer (2017)
8. Dyer, T., Nelson, T., Fisler, K., Krishnamurthi, S.: Applying cognitive principles to model-finding output: The positive value of negative information. Proc. ACM Program. Lang. **6**(OOPSLA1), 1–29 (2022)
9. Falleri, J., Morandat, F., Blanc, X., Martinez, M., Monperrus, M.: Fine-grained and accurate source code differencing. In: ASE. pp. 313–324. ACM (2014)
10. Gusukuma, L., Bart, A.C., Kafura, D.G., Ernst, J.: Misconception-driven feedback: Results from an experimental study. In: ICER. pp. 160–168. ACM (2018)
11. Hicks, A., Peddycord III, B.W., Barnes, T.: Building games to learn from their players: Generating hints in a serious game. In: ITS. LNCS, vol. 8474, pp. 312–317. Springer (2014)
12. Jackson, D.: Software abstractions: Logic, language, and analysis. MIT Press, revised edn. (2006)
13. Jackson, D.: Alloy: A language and tool for exploring software designs. Commun. ACM **62**(9), 66–76 (2019)
14. Lazar, T., Sadikov, A., Bratko, I.: Rewrite rules for debugging student programs in programming tutors. IEEE Trans. Learn. Technol. **11**(4), 429–440 (2018)
15. Macedo, N., Cunha, A., Paiva, A.C.R.: Alloy4Fun dataset for 2022/23 (Jul 2023). https://doi.org/10.5281/zenodo.8123547, https://doi.org/10.5281/zenodo.8123547
16. Macedo, N., Cunha, A., Pereira, J., Carvalho, R., Silva, R., Paiva, A.C.R., Ramalho, M.S., Silva, D.C.: Experiences on teaching Alloy with an automated assessment platform. Sci. Comput. Program. **211**, 102690 (2021)
17. Mansoor, N., Bagheri, H., Kang, E., Sharif, B.: An empirical study assessing software modeling in Alloy. In: FormaliSE. pp. 44–54. IEEE (2023)
18. Marwan, S., Lytle, N., Williams, J.J., Price, T.W.: The impact of adding textual explanations to next-step hints in a novice programming environment. In: ITiCSE. pp. 520–526. ACM (2019)
19. Marwan, S., Price, T.W.: iSnap: Evolution and evaluation of a data-driven hint system for block-based programming. IEEE Trans. Learn. Technol. **16**(3), 399–413 (2023)

20. Marwan, S., Williams, J.J., Price, T.W.: An evaluation of the impact of automated programming hints on performance and learning. In: ICER. pp. 61–70. ACM (2019)
21. McBroom, J., Koprinska, I., Yacef, K.: A survey of automated programming hint generation: The HINTS framework. ACM Comput. Surv. **54**(8), 172:1–172:27 (2022)
22. Piech, C., Sahami, M., Huang, J., Guibas, L.J.: Autonomously generating hints by inferring problem solving policies. In: L@S. pp. 195–204. ACM (2015)
23. Price, T.W., Dong, Y., Barnes, T.: Generating data-driven hints for open-ended programming. In: EDM. pp. 191–198. International Educational Data Mining Society (IEDMS) (2016)
24. Price, T.W., Dong, Y., Zhi, R., Paaßen, B., Lytle, N., Catetè, V., Barnes, T.: A comparison of the quality of data-driven programming hint generation algorithms. Int. J. Artif. Intell. Educ. **29**(3), 368–395 (2019)
25. Price, T.W., Marwan, S., Winters, M., Williams, J.J.: An evaluation of data-driven programming hints in a classroom setting. In: AIED (2). LNCS, vol. 12164, pp. 246–251. Springer (2020)
26. Rivers, K.: Automated Data-Driven Hint Generation for Learning Programming. Ph.D. thesis, Carnegie Mellon University, USA (2017)
27. Rivers, K., Koedinger, K.R.: A canonicalizing model for building programming tutors. In: ITS. LNCS, vol. 7315, pp. 591–593. Springer (2012)
28. Rivers, K., Koedinger, K.R.: Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. Int. J. Artif. Intell. Educ. **27**(1), 37–64 (2017)
29. Sarkar, A., Negreanu, C., Zorn, B., Ragavan, S.S., Pölitz, C., Gordon, A.D.: What is it like to program with artificial intelligence? In: PPIG. pp. 127–153. Psychology of Programming Interest Group (2022)
30. Wang, K., Sullivan, A., Khurshid, S.: Automated model repair for Alloy. In: ASE. pp. 577–588. ACM (2018)
31. Zheng, G., Nguyen, T., Brida, S.G., Regis, G., Aguirre, N., Frias, M.F., Bagheri, H.: ATR: Template-based repair for Alloy specifications. In: ISSTA. pp. 666–677. ACM (2022)