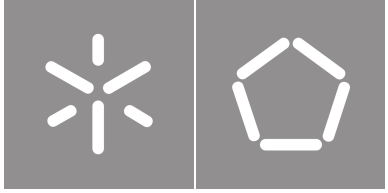
Universidade do Minho
Escola de Engenharia

Jorge Gabriel Alves Cerqueira

Automatic Repair of Behavioural Specifications

October, 2022



Universidade do Minho
Escola de Engenharia

Jorge Gabriel Alves Cerqueira

Automatic Repair of Behavioural Specifications

Master Thesis
Master in Integrated Master's in Informatics Engineering

Work developed under the supervision of:
Manuel Alcino Pereira Cunha
Nuno Filipe Moreira Macedo

October, 2022

Acknowledgements

I would like to thank both of my advisors, Alcino Cunha and Nuno Macedo. Through the entire process of writing this thesis, they have made themselves available and shown interest in its success, constantly providing feedback, suggesting and discussing ideas, and providing advice. Their support was invaluable for the development of this thesis.

I would also like to thank my parents, for their support and their trust not only this year but throughout my entire academic journey. Also thanks to my girlfriend, for the support and for helping me keep motivated to finish.

This work 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.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

_____, _____

(Place) (Date)

_____

(Jorge Gabriel Alves Cerqueira)

"Simplicity does not precede complexity, but follows it."

- Alan Perlis

Resumo

Reparação Automática de Especificações Comportamentais

De forma preocupante, o software está a tornar-se cada vez mais complexo com a passagem do tempo. A sociedade está completamente dependente dele. No entanto, mesmo nos dias que decorrem, não há ensino nem ferramentas suficientes que permitam aos engenheiros de software verificar a correção das suas soluções. Para além disto, soluções construídas rapidamente e com negligência relativamente à qualidade são incentivadas em contraste a uma abordagem mais rigorosa.

Todo o tipo de software terá inevitavelmente defeitos. No entanto, as linguagens de especificação formal permitem modelar sistemas complexos através da especificação das entidades relevantes, como as mesmas interagem, e testes das garantias esperadas. Consequentemente isto auxilia profissionais a compreender mais aprofundadamente os sistemas em que trabalham. Esta abordagem tem algumas desvantagens, como ser custosa temporalmente, sendo que adiciona mais um passo no processo de desenvolvimento, para além de ser difícil de aprender. A causa disto vem da natureza abstrata de especificações quando comparadas a programação, em conjunto da necessidade de estar confortável a trabalhar com conceitos de lógica formal.

Alloy é uma linguagem de especificação formal, capaz de análise estrutural e também temporal. É uma framework popular para validar e verificar requisitos, em grande parte por ser bastante expressiva e flexível. Isto torna-a uma excelente candidata para desenvolver e testar novas técnicas de reparação automática. Estas podem ser capazes ajudar estudantes a aprender mais rapidamente, tal como acelerar o desenvolvimento para utilizadores experientes. Com esta finalidade, algum trabalho já foi feito em relação à reparação de modelos estruturais em Alloy; no entanto, nada se encontra feito quando se põe em consideração os aspetos temporais.

Sendo assim, este trabalho apresenta um resumo da linguagem Alloy, em conjunto com as técnicas de reparação automática já propostas; propõe a primeira técnica baseada em mutações para reparação automática de especificações de lógica de primeira ordem em Alloy 6; para alem disso, descreve a integração de um sistema de geração automática de dicas para a plataforma Alloy4Fun.

Palavras-chave: Métodos Formais, Especificações Temporais, Reparação Automatica de Especificações, Alloy

Abstract

Automatic Repair of Behavioural Specifications

Somewhat worryingly, software is becoming increasingly complex with the passing of time. Even though society has become completely dependent on it, there's still not enough quality teaching and tooling to help software engineers verify the correctness of their solutions. Furthermore, quickly put together solutions are often incentivized over a more rigorous approach.

Software is always bound to have bugs. However, formal specification languages allow the modeling of complex systems by specifying the relevant entities, how they interact, and testing the expected guarantees. Hence, helping developers gain valuable understanding of the systems they work with. This approach has the drawbacks of not only being time costly, adding another step in the development process that requires deep understanding of the problem, but also being difficult to learn. The cause is due to the more abstract nature of specification compared to programming, paired with the need to be comfortable working with formal logic concepts.

Alloy is a formal specification language capable of structural and behavioral analysis. It is a popular framework for validating and verifying requirements, in part due to its expressiveness and flexibility. This makes it a prime candidate to develop and experiment new automatic repair techniques. They can help experienced developers speed up the process of writing specifications and new developers to learn quicker. With this in mind, some work has been done on repairing flawed structural Alloy models, but none considering behavioral aspects.

Thus, this thesis presents an overview of the Alloy language, along with previously proposed automatic repair techniques; it proposes the first mutation-based technique for the automatic repair of first-order temporal logic specifications using Alloy6; also, it describes the integration of an automatic hint generation system for Alloy4Fun, an online platform for teaching Alloy.

Keywords: Formal Methods, Behavioural Specifications, Automatic Specification Repair, Alloy

Contents

List of Figures xii

List of Tables xiv

1 Introduction 1

1.1 Contextualization . . . 1
1.2 Motivation . . . 2
1.3 Main Contributions . . . 2
1.4 Document Structure . . . 3

2 Formal Specification with Alloy 4

2.1 Structural Specification . . . 4
2.2 Behavioral Specification . . . 11
2.3 Syntax and Semantics . . . 17

3 State of the Art 22

3.1 Overview . . . 22
3.2 AlloyFL . . . 23
3.2.1 AUnit . . . 23
3.2.2 AlloyFL$_{co}$ . . . 25
3.2.3 AlloyFL$_{un}$ . . . 27
3.2.4 AlloyFL$_{su}$ . . . 27
3.2.5 AlloyFL$_{mu}$ . . . 27
3.2.6 AlloyFL$_{hy}$ . . . 28
3.3 ARepair . . . 28
3.4 FLACK . . . 30
3.5 BeAFix . . . 34

4 Temporal Alloy Repair **37**
4.1 Overview 37
4.2 Implementation 40
4.2.1 Mutators and Candidate Generation 40
4.2.2 Mutation-based repair with counterexample-based pruning 44
4.3 Evaluation 45

5 Hint Generation in Alloy4Fun **52**
5.1 Alloy4Fun Usage Overview 52
5.2 Implementation 54
5.3 Automatic Hints 54

6 Conclusion **59**

Bibliography **61**

List of Figures

1 Example of possible instances for `Track` and `next`. 5
2 Instaces with Signal atoms. 6
3 Instance following layout fact. 9
4 Counterexample for `JunctionSafety`. 10
5 Check passed for the assertion. 10
6 Variable relations freely changing. (States 1 and 2). 12
7 First two states of an instance with a moving train. 15
8 First three states for a counterexample of the `NoCollision` assertion. 16
9 Instance with the active tracks changing. 21

10 Previously defined tests passing. 24
11 Altered test failing. 24
12 Coverage results. 25
13 Coverage results. 25
14 Illustration for AlloyFL$_{un}$ and AlloyFL$_{su}$ presented by the AlloyFL researchers [21]. 27
15 Diagram illustrating the various components in ARepair [34]. 29
16 Overview of the components used in FLACK [45]. 30
17 Comparison between FLACK and AlloyFL [45]. 34
18 Comparison between ARepair and BeAFix for ARepair benchmarks [8]. 36
19 Comparison between ARepair and BeAFix for Alloy4Fun models [8]. 36

20 Counterexample to a faulty predicate shown by Alloy. 39
21 Diagram of the result of applying the candidate {`File` $\rightsquigarrow$ `Trash`} to the expression `some File + Trash`. 41
22 Percentage of specification challenges repaired by the proposed approach under a certain time threshold, for different search depth levels and with and without pruning. 49
23 Percentage of submissions to static challenges correctly repaired by the proposed approach under a certain time threshold, for different search depth levels and with and without pruning. 50

24 Classification of submissions to static challenges according to which tool was able to effectively repair under 1 second. 50

25 Root path of Alloy4Fun. 52
26 Sharing a model in Alloy4Fun. 53
27 Example of a shared model in Alloy4Fun, with secret paragraphs. 53
28 Architecture of Alloy4Fun. 54
29 Errors in the current Alloy4Fun version. 57
30 Errors in the updated Alloy4Fun version. 58
31 Hints in the extended Alloy4Fun version. 58

List of Tables

1 Available suspiciousness ranking formulas for AlloyFL$_{co}$, AlloyFL$_{mu}$ and AlloyFL$_{hy}$ [21]. . 26
2 Mutation operators used in AlloyFL$_{mu}$ [36]. 28

3 List of mutators for Boolean formulas. 43
4 List of mutators for relational expressions. 43
5 Performance of the 3 techniques under a 1-minute threshold and maximum depth 3. . . 48

6 Hints for each of the mutator classes. 56

1 Introduction

With time, software systems are getting more and more complex. Instead of simple sequential programs we now have gigantic interconnected distributed systems. The former will often present bugs and failures, in the latter these problems are even more frequent and severe.

The question then becomes, how to understand and manage the complexity of these systems, and ensure that they are resilient and secure. In an idealized world, mathematical proofs would be given to prove the correctness of these systems. Unfortunately, such proofs are in many situations too costly or not possible.

Lightweight formal methods are a middle ground. They keep the rigor of a mathematical approach, however, they still allow enough flexibility to guarantee that assumptions about a system are correct up to a level of certainty.

1.1 Contextualization

As previously mentioned, software systems are getting increasingly complex. The reliability of these systems is currently a necessity with so many people depending on them.

Fortunately, the correctness concern has been recognized and techniques such as property-based testing is being integrated into software engineering workflow. This is a good improvement, however, using formal specifications in addition to these techniques would be a massive help, both for finding bugs and also for wrong assumptions made for certain systems. Unfortunately, due to the abstract and complex nature of formal specification languages, professionals still often struggle with writing, understanding and validating them [22]. This increases the costs of training and actively maintaining specifications. It ultimately undermines the quality and reliability of the software development process.

The topic of automatic program repair is very active in software engineering [28]. It has many applications such as bug fixing, detection of vulnerabilities, and support for autonomous learning. Automatic specification repair is not as developed. Some of the challenges in automatic repair of programs are not present for specifications, however, new ones exist. The further development of these techniques might open a lot of possibilities in easing the usage of specification languages.

1.2 Motivation

Alloy [17] is a formal specification language. It was previously tailored for structural analysis, but now, it is also capable of handling behavioural analysis in the latest version 6 [4], with the merge of the Electrum extension [9]. It is a popular framework for validating and verifying requirements, due to being a high-level language with good expressiveness and flexibility. The same reasons also make it an excellent candidate to introduce students to formal specification languages. Due to this, it is already being used in various universities [11]. Alloy has been successfully used in various domains. Examples of this are the validation and discovery of flaws in distributed protocols [43], detection of security flaws in mobile and IoT devices [6, 3, 7], analysis of filesystems [19], automation of software testing [20], validation of rail traffic management systems [13] and many other domains.

Alloy4Fun [25] is an online platform, aiming to help students be able to study specification languages (specifically Alloy) autonomously. Currently, tutors can write and share exercises for students to solve. It already gives some feedback, checking if an answer is correct and giving counterexamples otherwise, but the students often have difficulty interpreting these counterexamples. Repair hints have shown to be helpful in autonomous learning platforms [30].

All these reasons make Alloy a good candidate to experiment and develop new techniques of automatic repair in the context of specification languages. These would help improve the learning experience for new users. Specifically, by opening the possibility of implementing hint generation systems in the Alloy4Fun learning platform, since current feedback is difficult for students to interpret. Work has been done on repairing Alloy structural specifications, however, none of it takes into consideration the recently added support for behavioural specifications.

1.3 Main Contributions

This thesis focuses on the exploration of the feasibility of using automatic repair techniques to repair and generate hints of rich specifications in Alloy.

The main contribution has been the development of a new repair technique. It is capable of repairing issues in incorrect specifications and able to report the result to the user. Current repair techniques were insufficient:

- They did not provide support for Alloy 6 specifications, which is the most recent version, supported in the Alloy4Fun platform;
- They did not focus on performance, specifically in providing results in short time.

Thus, the technique developed focuses on these points: by building on top of Alloy 6, and taking into account the introduced temporal operators; and by basing itself on previously implemented techniques, while providing additional ways to speed up the repair process.

The technique was also expanded to provide a way to report possible fixes as hints. This was then integrated into the Alloy4Fun platform, allowing for students to ask for hints when their submissions are incorrect.

The contributions presented in this thesis also originated a research paper on the International Conference on Software Engineering and Formal Methods (SEFM'22) [10].

1.4 Document Structure

This thesis has the following structure.

Chapter 2 goes over the Alloy language, both the structural and behavioural aspects. It presents both the syntax and semantics of the language. This starts with a practical explanation, guided by an example, and ends with a formal overview of the language.

Chapter 3 explores the state of the art in automatic repair of specifications. It starts with a high-level overview of automatic repair and the relevant concepts. Then, it proceeds to present the current techniques developed along with a succinct explanation of how they work.

Chapter 4 describes the technique developed for automatic repair. It starts with an explanation of the context leading to some of the decisions made. This is followed by a description of the implementation. In the end, a comparison is made between the similar tools available.

Chapter 5 talks about the extensions and changes made to the Alloy4Fun platform. It begins with an overview of how the platform is implemented and what its regular use looks like. It ends by explaining the changes made to it in the implementation of the hint system.

Chapter 6 concludes the thesis, reflecting on the contributions made in it and also thoughts about future works.

2 Formal Specification with Alloy

Alloy [17] is both a tool and language that allows the analysis of structural specifications using relational logic. Electrum [9] is an extension of the Alloy language that introduces first-order temporal expressions and extends the backend to support temporal analysis, allowing behavioral specifications. It has recently been merged into the latest version Alloy 6.

In this chapter, the Alloy 6 language will be introduced along with a practical example of a specification. Starting with a structural specification and moving into a behavioral specification. The example presented will be a basic railway system having connected tracks with signals and trains moving through them. Note that this example is only meant as an illustrative exercise and not a real railway system, although some have been implemented [12, 33]. After that, the syntax and semantics of the Alloy language will be presented formally.

2.1 Structural Specification

Firstly, it is necessary to define the entities within the system. For this, Alloy has the concept of signatures. A signature is used to express a new type within the system. In an instance of the specification, it will be represented by a set (a unary relation) of elements, called atoms. To declare a signature the `sig` keyword is used. The set of tracks in our system can be declared the following way.

```
sig Track {}
```

A signature has now been defined, and Alloy will show instances where tracks are present. These are, however, not very useful, since they have no connections. Alloy allows declaring fields for signatures. These define relations between types. Fields are declared the following way.

```
sig Track {
    next: set Track
}
```

This creates a relation named `next`, the `set` keyword indicates that each track can be linked to any number of tracks.

After declaring this signature, the command `run` can be used to generate instances for the model. In Alloy, the analysis is always made within a finite universe, so instances can be exhaustively generated. The size of this universe is defined by the scope. The scope limits the number of atoms that can be contained in a signature. The `for` keyword can be added as a modifier to a command to define the scope. Formulas can also be added to the body of the command to filter instances where these formulas evaluate to `true`. A command to show all the instances possible with a maximum of three tracks can be expressed as follows.

```
run {} for 3
```

Figure 1 shows two instances that can be obtained with the previous `run` command. Both figures present instances with two atoms, `Track0` and `Track1`.

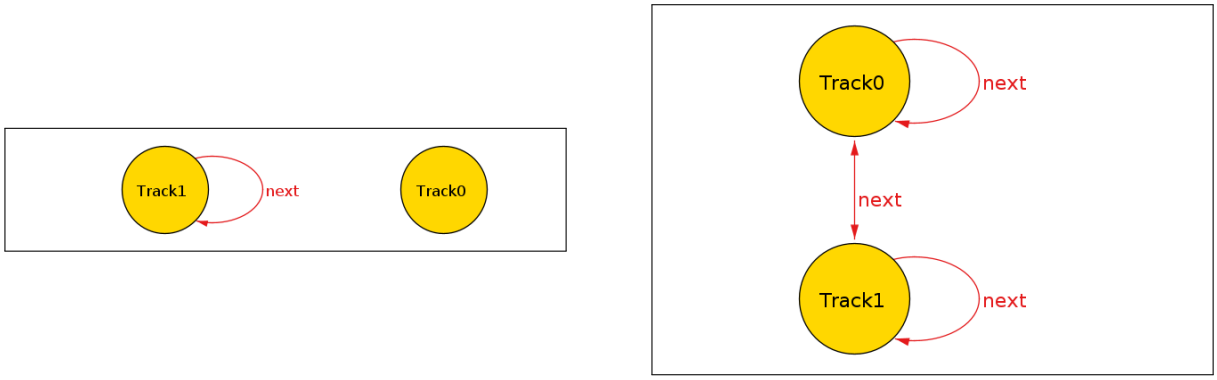


Figure 1: Example of possible instances for `Track` and `next`.

With tracks now declared, light signals can be taken into account. The types of lights can be enumerated, in this case, they'll be either red or green and never both at the same time. For cases like this, the keywords `extends` and `abstract` are useful. The former allows declaring a signature that is a subset of another signature. The latter ensures that all the members of the parent abstract signature belong to an extension. Signatures that extend or abstract the signature will also be disjunct. The light signals can be declared in the following way.

```
abstract sig Color {}
one sig Red, Green extends Color {}
sig Signal {
        color: one Color
}
```

If the declarations are identical for multiple signatures, they can be declared within the same line, as is demonstrated with the red and green colors. The `abstract` declaration above guarantees that atoms

contained in `Color` are contained in one of the extending signatures (either red or green). The `extends` guarantees that the sets of red and green colors are disjoint, this disallows a color from being red and green at the same time.

The keyword `one` has also been used; in the case of the `Red` and `Green` signatures, it guarantees the number of atoms in the sets `Red` and `Green` is one, in other words, there aren't multiple reds or greens. In the case of `color: one Color`, the keyword `one` ensures that, in the relation `color`, each signal is always linked to exactly one color.

The pattern described above is the color signature is so common that Alloy provides a way to declare it more concisely with the keyword `enum` the following way.

```
enum Color { Red, Green }
sig Signal {
        color: one Color
}
```

An alternative approach would have been making `Green` and `Red` as subsets of `Signal`, instead of creating an attribute.

Finally, the signals can be added as a field in the `Track` signature, with the keyword `lone` assuring that a `Track` can't have more than one signal.

```
sig Track {
        next: set Track,
        signal: lone Signal
}
```

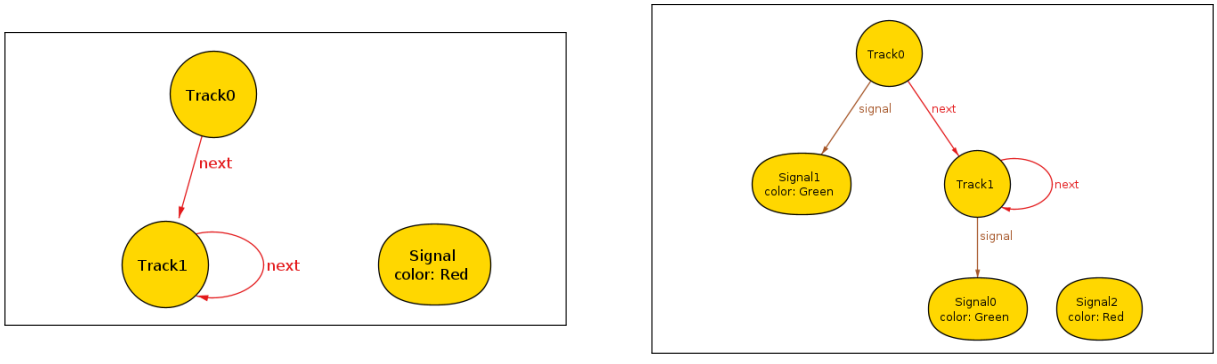


Figure 2: Instaces with Signal atoms.

Figure 2 shows the new instances with Signals added, but the instances don't always make sense, for example, tracks can link to themselves and signals can sometimes not be connected to a track. To fix this, it makes sense to start writing properties for the system.

Alloy has support for functions. They can evaluate an expression and return a value and can also take arguments. The keyword `fun` is used to define functions. Functions to calculate the set of tracks that are forks and junctions can be defined the following way.

```
fun Fork : set Track {
        { track : Track | not lone track.next }
}

fun Junction : set Track {
        { track : Track | not lone next.track }
}
```

The `Fork` function filters the atoms in the set Track that pass the assertion. The dot join operator, '.', can be applied between two relations or a relation and a set. Let `x` be a set of type `X` and `rel` be a relation of type `X`×`Y`. The resulting type of `x.rel` will be of type `Y` and contain all the atoms that are related to an atom in `x` based on the relation `rel`. Keep in mind that every signature and variable in Alloy evaluates to a relation, which allows for this kind of expression. In this specific example, `track` is a singleton set of type `Track` and `next` a relation with type `Track`×`Track`. Therefore, `track.next` will evaluate to the set of tracks `track` is linked to based on `next`.

The expression `not lone` expresses the negation of `lone`, where `lone` checks that the cardinality of a set is zero or one. Thus, `not lone` will check that the cardinality of a set is greater than one.

The expression for `Fork` can therefore be read as the set of tracks that have more than one successor. Similarly, the expression for `Junction` can be read as the set of tracks with more than one predecessor.

Alloy also supports predicates. These are similar to functions but can only be true or false. The keyword `pred` is used to define predicates. A predicate to check if a signal is always connected to exactly one track can be defined the following way.

```
pred signal_is_connected[s : Signal] {
        one signal.s
}
```

The predicate filters from the relation `signal` the tracks that have `s` as its signal and checks if the number of elements in this set is one.

Similarly, predicates can be defined to check that no tracks are left unconnected and no tracks are connecting to themselves.

```
pred track_is_connected[t : Track] {
        some t.next or some next.t
}
```

```
pred track_no_loop[t : Track] {
        t not in t.^next
}
pred track_no_multiple_signal[t : Track] {
        lone t.signal
}
```

The transitive closure operator, '^', can be applied to a binary relation between equal types. It returns all the reachable atoms from a given atom (i.e. `^next = next + next.next + next.next.next + ...`).

These predicates can now be used within facts, facts are formulas that are assumed to be valid. Alloy will only check instances where facts evaluate to true. The keyword `fact` is used to define facts. Since facts cannot be called in other places naming them is optional. The previously shown predicates can be used within a fact in the following way.

```
fact Layout {
        all s : Signal | signal_is_connected[s]
        all t : Track |
                track_is_connected[t] and
                track_no_loop[t] and
                track_no_multiple_signal[t]
        all s : Signal | s.color = Red
}
```

The expression quantifier `all` allows checking if an expression is true for every element in a set. On the first line, `s : Signal` binds an atom in the `Signal` set to the variable `s`. Then, `signal_is_connected[s]` checks that the signal is connected, and `all` ensures that the expression is true for every possible value of `s`. Other expression quantifiers are `some`, `one`, `lone` and `no`.

Note that expressions within the same scope separated by a line are connected by logical conjunction.

Figure 3 shows one of the instances obtained by using the `run` after adding the `Layout` fact. The instances from figures 1 and 2 will not appear now since the `Layout` fact invalidates them.

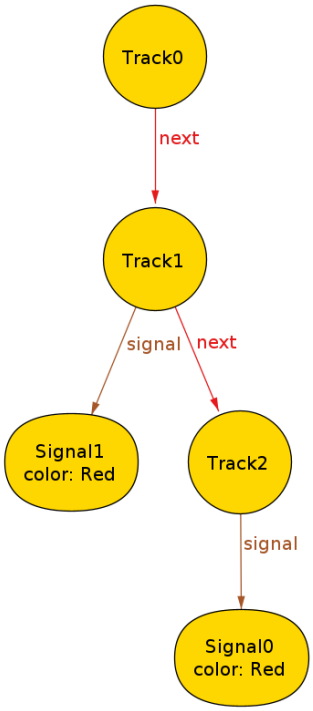


Figure 3: Instance following layout fact.

Finally, expressions can be checked for the specification written. For these, Alloy uses the keywords `assert` and `check`. The `assert` keyword is used to define the expression to be checked. The `check` keyword is used to define the boundaries of the search, for example checking instances with less than 5 tracks.

The safety property of enforcing tracks leading into a junction to have signaling, and not having two green lights leading into the same junction track can be expressed and checked the following way.

```
fun RedSignal : set Signal {
        { s : Signal | s.color = Red }
}
assert JunctionSafety {
        all j : Junction |
                all disj t, t2: next.j |
                        some t.signal and
                        some t2.signal and
                        (t.signal in RedSignal or t2.signal in RedSignal)
}

check JunctionSafety for 10
```

The property checks all tracks leading to the same junction. It verifies that each track has signals,

and, within each disjunct pair, at least one of them has a red signal (or equivalently that signals aren't green in both tracks). The `for` used in the `check` keyword indicates that the program will perform the check for instances with up to 10 elements for each type.

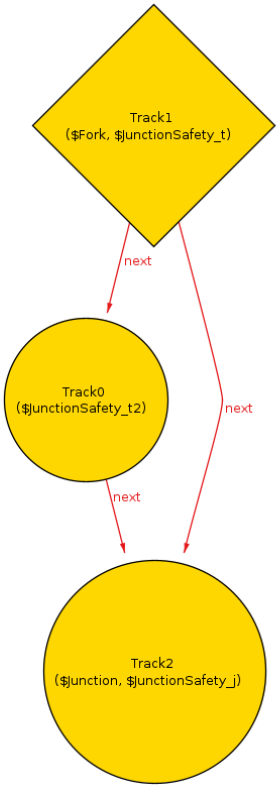


Figure 4: Counterexample for `JunctionSafety`.

This check will fail, as shown in Figure 4. This is because in the `Layout` fact no expression has been written to guarantee that tracks leading to junctions have a signalling light. To fix this the following expression can be added, and the assertion no longer produces counterexamples, as shown in Figure 5.

```
fact Layout {
        ...
        all t : next.Junction | one t.signal
}
```

```
Executing "Check JunctionSafety for 10"
   Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20 Mode=batch
   8042 vars. 270 primary vars. 13515 clauses. 107ms.
   No counterexample found. Assertion may be valid. 29ms.
```

Figure 5: Check passed for the assertion.

2.2 Behavioral Specification

In the previous section a basic structure of the railway was modeled, however instances found are static. It would be more useful to be able to specify the possibility of changes in signal color and also add moving trains. To achieve this with Alloy 6, it is first needed to express the relations or signatures that can change over time, which can be done with the `var` keyword. A new relation representing the currently active next track (it dictates which of the tracks in `next` a train is authorized to move to), along with the indication that the signal colors can change can be added the following way.

```
sig Track {
        var active: lone Track,
        ...
}
sig Signal {
        var color: one Color
}
fact Layout {
        all t : Track | t.active in t.next
        ...
}
```

This enables the relations `color` and `active` to change freely. This raises a problem: the value of these relations will be arbitrary over time, for obvious reasons leading to invalid instances. This can be seen in Figure 6. From the first state to the second, various invalid changes are made, for example `Track1`×`Track1` being added to the `active` relation. Notice that the instance in Figure 6 has colors, these were achieved with the theming feature in Alloy. Themes are used to improve the clarity and readability of instances. It is possible to select different colors and shapes for each atom, depending on the signatures they belong to, or if they belong to the result of a function. In this case, functions like `RedTrack` and `GreenTrack` were added, which filter tracks with red and green signals, respectively. These functions were then used to set the color of the atoms in `Track`, and the atoms in `Signal` have been hidden for clarity.

To solve the previously mentioned problem there's the need to establish the possible events within the system. In this case, one of the events that make sense to allow is a switch in the color of a signal linked to a track, this event can be expressed as follows.

```
pred switch_color[t: Track] {
        one t.signal and (t in GreenTrack or active.(t.active) in RedTrack)

        t.signal.color' != t.signal.color
```

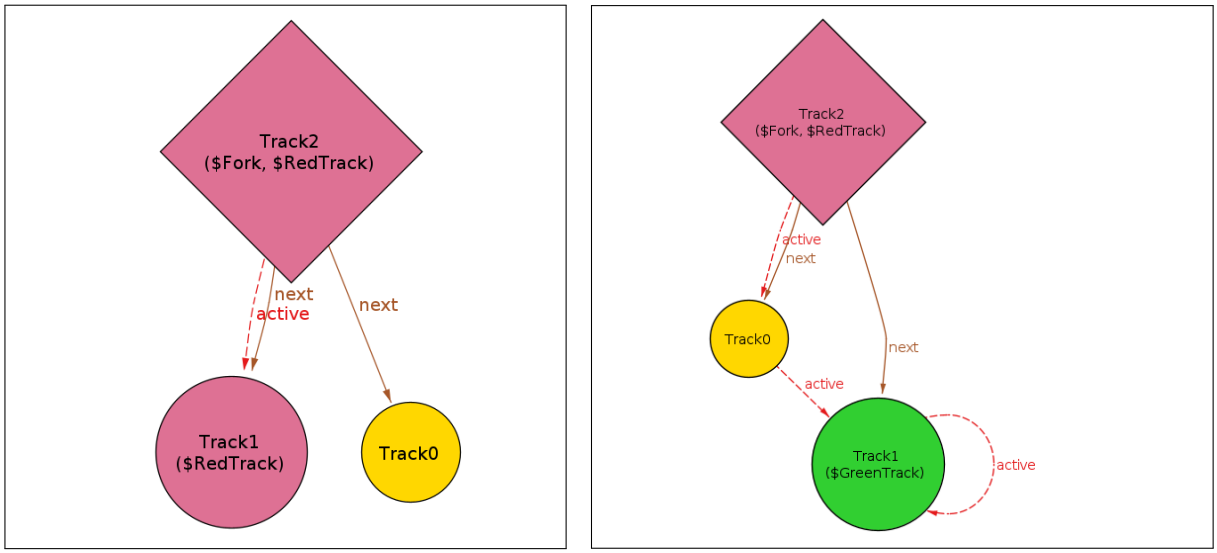


Figure 6: Variable relations freely changing. (States 1 and 2).

```
        active' = active
        all s: Signal - t.signal | s.color' = s.color
}
```

The prime symbol returns the value of the sets in the next state. For example, `t.signal.color' != t.signal.color` enforces that, in the next state, the signal color for `t` will be different from the current one.

Transitions can often be separated into three sections, highlighted in the code with an empty line separating them:

- the preconditions, where the properties necessary for the transition are expressed;
- the effect, where the changes that are going to occur from a state to the next are specified;
- the frame conditions, expressions to prevent relations unrelated to the transaction to change.

In this example, the precondition is the first line. There, it is verified that the track has a signal, and that the light is green or all the tracks leading to the currently active track has a red signal. The effect is expressed by the change of the signal color in the next state. Then, it is stated that for this transition to occur, active tracks must stay the same, and all signals except the one being changed must also stay the same.

A predicate to express the transition of switching active tracks can be expressed the following way.

```
pred switch_active[t: Track] {
        t.signal.color in Red and some t.next
```

```
        t.active' != t.active
        t.active' in t.next

        all track: Track - t | track.active' = track.active
        color' = color
}
```

In the predicate for switching the next active track, firstly it is checked that if the track signal exists it is red (if it is green it might change into a track with a green signal already leading to it). Also, it is checked that there's a next track to change to in the first place. Secondly, it is enforced that in the next state the active track for `t` will be different from the current one. Not only that, this new active track must be, for obvious reasons, in the set of tracks `t` is connected to. Finally, it is enforced that other tracks don't change their active connections and signal colors don't change.

Generally, it is also useful to define a transition, usually given the name `nop`. It simply keeps all variables and relations the same in the next state. The reason for this is that the traces (sequences of states in time) Alloy generates are infinite, so there's always a state after the current one. The way to support this within a bounded analysis is by considering only the sequences that loop at some point. This is the main use for `nop`, at any point, the system can freeze with an infinite loop of `nop` events. This allows the analysis of traces where a loop would not exist otherwise. One definition for `nop` for the current model is the following.

```
pred nop {
        active' = active
        color' = color
}
```

Finally, a `Transitions` fact can be added with the allowed events.

```
fact Transitions {
        always nop or (some t: Track | switch_active[t] or switch_color[t])
}
```

The `always` keyword has been added in Alloy 6, it evaluates to true if, and only if, the expression below it evaluates to true at every state for an instance. So the expression will simply force that at every change of state either: `nop` is taken; some tracks change their next active track; some tracks change their signal color. The specified system has become quite complex. With the possibility of signals and active track connections changing, it makes sense to add the final signature relevant to the model, trains. Trains shall have a variable position corresponding to the track they are currently on, and they can be expressed in the following way.

```
sig Train {
        var pos: lone Track
}
```

With trains added, there's a need to rewrite the previous transition predicates to ensure positions don't change when they occur (called frame conditions, as explained previously). Not only that, the previously defined `Layout` fact is not enough since it doesn't prevent the initial state from having multiple trains in the same position. It is sensible to turn `Layout` into a predicate and create a new fact encompassing all the desired initial conditions of the system.

```
pred nop {
        ...
        pos' = pos
}
pred switch_active[t: Track] {
        ...
        pos' = pos
}

pred switch_color[t: Track] {
        ...
        pos' = pos
}
fact Init {
        layout
        all t: Track | lone pos.t // A track can't start with multiple trains
}
```

Now there's also a need to add a transition to move the train. It shouldn't be able to move when the light is red at its position. In addition, its position on the next state should be the actively connected track from its current position.

```
pred move_train[t: Train] {
        t.pos.signal.color not in Red
        some t.pos.active or no t.pos.next

        t.pos' = t.pos.active

        active' = active
```

```
        color' = color
        all train: Train - t | train.pos' = train.pos
}
fact Transitions {
        always {
                ... or
                (some t: Train | move_train[t])
        }
}
```

Figure 7 shows an instance where the `move_train` transition is taken, and the train moves to the track linked to its current one.

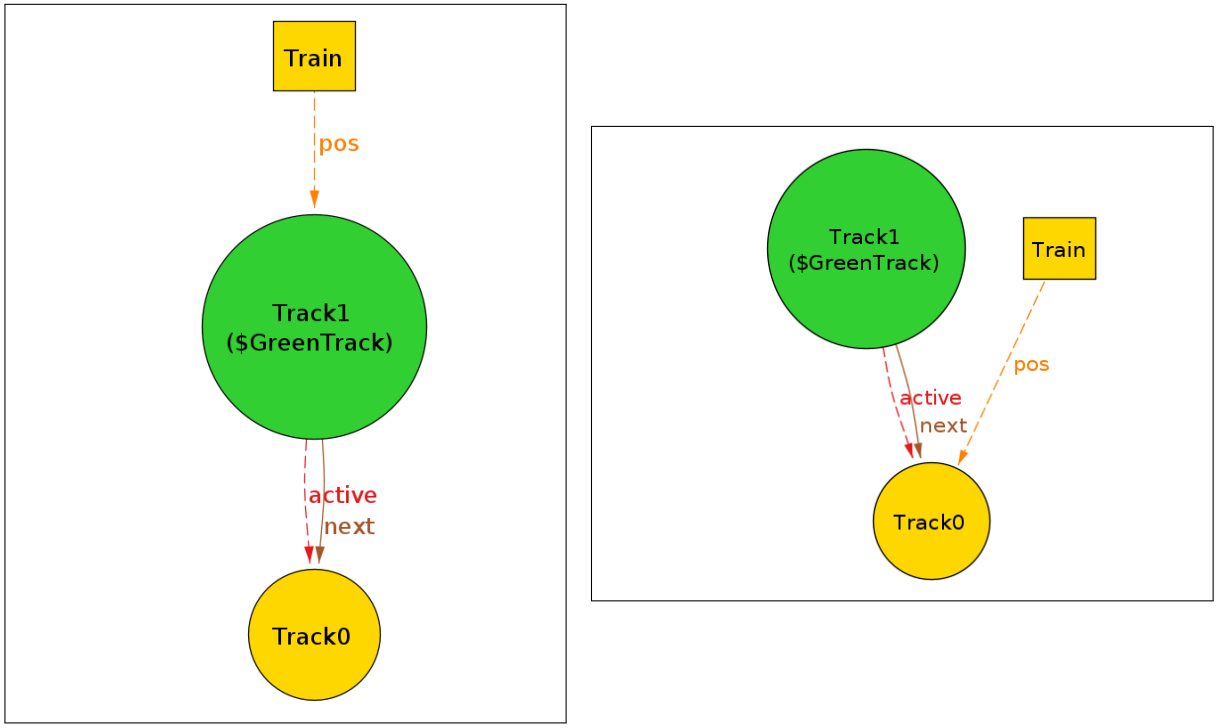


Figure 7: First two states of an instance with a moving train.

Having the system behavior specified, some assertions should be added to verify if the desirable safety properties hold. An assertion to verify that for all the tracks going into a certain track, only one of them can be green. In addition, an assertion to check that a track cannot have multiple trains in it at the same time. These can be expressed in the following way.

```
assert SignalSafety {
        always all track: Track | all disj t, t2: active.track {
                t in RedTrack or t2 in RedTrack
        }
```

```
}
assert NoCollision {
        always all t: Track | lone pos.t
}
check SignalSafety for 5
check NoCollision for 5
```

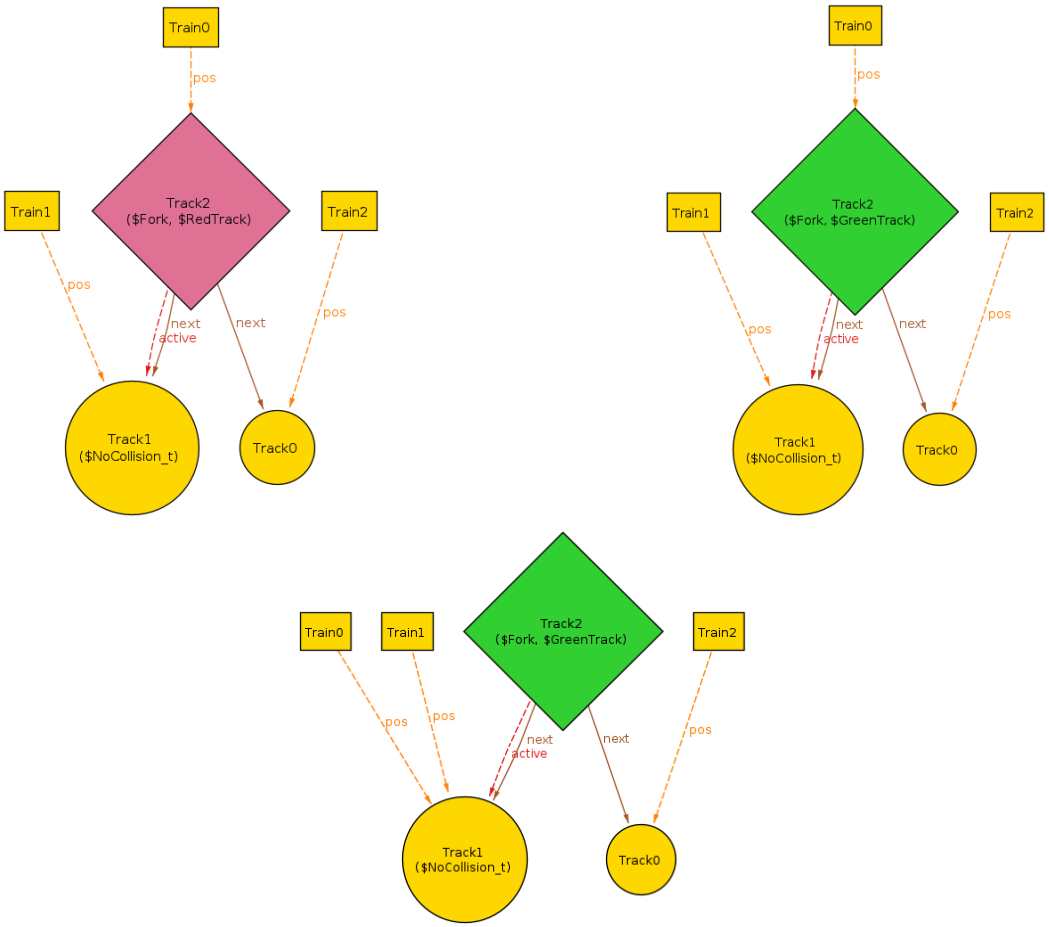


Figure 8: First three states for a counterexample of the `NoCollision` assertion.

The first check `SignalSafety` does not yield any counterexamples. This means that it is valid for any instance within the scope, but not necessarily for instances bigger than it. An infinite number of instances can not be tested, but the scope can be increased until there is enough confidence that the check is valid.

Unfortunately, the check for `NoCollision` fails, as shown by the counterexample in Figure 8. Analyzing the counterexample, the problem is evident, the transition to change signal color disregards the presence of trains in the actively connected tracks. In the first state of the counterexample, `Track1` has a train in it, and therefore, the signal in `Track2` cannot change to green, since this later leads to the collision between `Train0` and `Train1`.

The fix starts with calculating the reachable tracks in the case lights don't change, and checking if there's currently a train on any of those tracks. The following change can be added to the pre-condition of the transition to take this into account.

```
fun reachable_tracks[t: Track] : Track {
        t.^(active - RedTrack -> Track + t -> t.active)
}
pred switch_color[t: Track] {
        t in GreenTrack or {
                active.(t.active) in RedTrack
                all track : reachable_tracks[t] | no pos.track
        }
        ...
}
```

2.3 Syntax and Semantics

In previous sections, Alloy has been presented with a focus on the practical usage of the language. This section will attempt to formalize the concepts used previously.

Below the syntactic rules are presented both in the Alloy syntax and in the more traditional mathematical notation. It is based on previously presented definitions [17, 16, 26] and doesn't attempt to encompass the entire syntax for Alloy. The presented operators are expressive enough to capture all the language, since missing operators can be derived from the presented ones.

The identifiers used are the following:

- c for constants;
- v for variables;
- Φ for n-ary relational expressions;
- ϕ for formulas.

The formation rules for expressions are the shown below. On the left, the alloy representation is presented, and, on the right, the corresponding mathematical representation.

Φ ::= e	Φ
| c	c
| v	v

`e1 + e2`	$\Phi_1 \cup \Phi_2$
`e1 & e2`	$\Phi_1 \cap \Phi_2$
`e1 - e2`	$\Phi_1 \backslash \Phi_2$
`e1 -> e2`	$\Phi_1 \times \Phi_2$
`e1 . e2`	$\Phi_1 . \Phi_2$
`~e`	Φ_1°
`^e`	Φ_1^+
`*e`	Φ_1^*
`{v : e1 | f1}`	$\{v \in \Phi_1 \mid \phi_1\}$

The formation rules for formulas are shown below. Like previously, in the left is presented the Alloy representation, and, in the right, the corresponding mathematical representation. For the temporal operators the corresponding standard operator from linear temporal logic is presented.

ϕ ::= `e1 in e2`	$\Phi_1 \subseteq \Phi_2$
`e1 = e2`	$\Phi_1 = \Phi_2$
`not f1`	$\neg\phi_1$
`f1 or f2`	$\phi_1 \vee \phi_2$
`f1 and f2`	$\phi_1 \wedge \phi_2$
`f1 implies f2`	$\phi_1 \rightarrow \phi_2$
`all v : A | f1`	$\forall v \cdot v \nsubseteq A \vee \phi_1$
`some v : A | f1`	$\exists v \cdot v \subseteq A \wedge \phi_1$
`no e1`	$\lvert\phi_1\rvert = 0$
`lone e1`	$\lvert\phi_1\rvert \leq 1$
`one e1`	$\lvert\phi_1\rvert = 1$
`some e1`	$\lvert\phi_1\rvert \geq 1$
`always f1`	$G\phi_1$
`eventually f1`	$F\phi_1$
`after f1`	$X\phi_1$
`f1 until f2`	$\phi_1 U \phi_2$
`f1 releases f2`	$\phi_1 R \phi_2$
`historically f1`	$H\phi_1$
`once f1`	$O\phi_1$

`before f1`	$Y\phi_1$
`f1 since f2`	$\phi_1 S \phi_2$
`f1 triggered f2`	$\phi_1 T \phi_2$

Below are presented the semantics for the evaluation of both expressions and formulas. The double brackets represent the evaluation, and it takes two arguments:

- π is an infinite sequence of interpretations for constants, predicates and variables;
- n is a natural number, denoting which interpretation is being considered.

The semantics for evaluation of the value of an expression are the following.

$$
\begin{aligned}
[\![v]\!]^i_\pi &= \pi(i)(v) \\
[\![c]\!]^i_\pi &= c \\
[\![\Phi_1 \texttt{ + } \Phi_2]\!]^i_\pi &= [\![\Phi_1]\!]^i_\pi \cup [\![\Phi_2]\!]^i_\pi \\
[\![\Phi_1 \texttt{ \& } \Phi_2]\!]^i_\pi &= [\![\Phi_1]\!]^i_\pi \cap [\![\Phi_2]\!]^i_\pi \\
[\![\Phi_1 \texttt{ - } \Phi_2]\!]^i_\pi &= [\![\Phi_1]\!]^i_\pi \setminus [\![\Phi_2]\!]^i_\pi \\
[\![\Phi_1 \to \Phi_2]\!]^i_\pi &= [\![\Phi_1]\!]^i_\pi \times \in [\![\Phi_2]\!]^i_\pi \\
[\![\Phi_1 \texttt{ . } \Phi_2]\!]^i_\pi &= \{(a_1, ..., a_{n-1}, b_1, ..., b_{m-1}) | (a_1, ..., a_n) \in [\![\Phi_1]\!]^i_\pi \\
&\qquad \wedge (b_1, ..., b_n) \in [\![\Phi_2]\!]^i_\pi \wedge a_n = b_n)\} \\
[\![\texttt{\~{}} \Phi_1]\!]^i_\pi &= \{(b, a) | (a, b) \in [\![\Phi_1]\!]^i_\pi\} \\
[\![\texttt{\^{}} \Phi_1]\!]^i_\pi &= \{(a, b) | (a, b) \in [\![\Phi_1]\!]^i_\pi \vee \exists c_1, ..., c_n \cdot ((a, c_1) \in [\![\Phi_1]\!]^i_\pi \\
&\qquad \wedge (c_1, c_2) \in [\![\Phi_1]\!]^i_\pi \wedge ... \wedge (c_n, b) \in [\![\Phi_1]\!]^i_\pi)\} \\
[\![\texttt{*} \Phi_1]\!]^i_\pi &= \{(a, a) | a \in A\} \cup [\![\texttt{\^{}} \Phi_1]\!]^i_\pi \\
[\![\{x_1 : \Phi_1, ..., x_n : \Phi_n | \phi_1\}]\!]^i_\pi &= \{(a_1, ..., a_n) | a_1 \in [\![\Phi_1]\!]^i_\pi \wedge ... \wedge a_n \in [\![\Phi_n]\!]^i_\pi \\
&\qquad \wedge \pi, i \models \phi_1[a_1/x_1, ..., a_n/x_n]\} \\
[\![\Phi']\!]^i_\pi &= [\![\Phi]\!]^{i+1}_\pi
\end{aligned}
$$

The semantics for the evaluation of the value of a formula are the following.

$$
\begin{aligned}
\pi, i \models \Phi_1 \textbf{ in } \Phi_2 &\equiv [\![\Phi_1]\!]^i_\pi \subseteq [\![\Phi_2]\!]^i_\pi \\
\pi, i \models \Phi_1 \texttt{= } \Phi_2 &\equiv [\![\Phi_1]\!]^i_\pi = [\![\Phi_2]\!]^i_\pi \\
\pi, i \models \textbf{not } \phi_1 &\equiv \pi, i \not\models \phi_1 \\
\pi, i \models \phi_1 \textbf{ or } \phi_2 &\equiv \pi, i \models \phi_1 \vee \pi, i \models \phi_2 \\
\pi, i \models \phi_1 \textbf{ and } \phi_2 &\equiv \pi, i \models \phi_1 \wedge \pi, i \models \phi_2 \\
\pi, i \models \phi_1 \textbf{ implies } \phi_2 &\equiv \pi, i \not\models \phi_1 \vee \pi, i \models \phi_2
\end{aligned}
$$

$$\pi, i \models \textbf{all}\ v : A \mid \phi_1 \quad \equiv \pi, i \models \bigwedge_{c \in [\![A]\!]^i_\pi} \phi_1[c/v]$$

$$\pi, i \models \textbf{some}\ v : A \mid \phi_1 \equiv \pi, i \models \bigvee_{c \in [\![A]\!]^i_\pi} \phi_1[c/v]$$

$$\pi, i \models \textbf{no}\ \Phi_1 \quad \equiv [\![\Phi_1]\!]^i_\pi = 0$$

$$\pi, i \models \textbf{lone}\ \Phi_1 \quad \equiv [\![\Phi_1]\!]^i_\pi \leq 1$$

$$\pi, i \models \textbf{one}\ \Phi_1 \quad \equiv [\![\Phi_1]\!]^i_\pi = 1$$

$$\pi, i \models \textbf{some}\ \Phi_1 \quad \equiv [\![\Phi_1]\!]^i_\pi \geq 1$$

$$\pi, i \models \textbf{always}\ \phi_1 \quad \equiv \forall i \leq j \cdot \pi, j \models \phi_1$$

$$\pi, i \models \textbf{eventually}\ \phi_1 \quad \equiv \exists i \leq j \cdot \pi, j \models \phi_1$$

$$\pi, i \models \textbf{after}\ \phi_1 \quad \equiv \pi, i+1 \models \phi_1$$

$$\pi, i \models \phi_1\ \textbf{until}\ \phi_2 \quad \equiv \exists i \leq k \cdot (\pi, k \models \phi_2 \wedge \forall i \leq j < k \cdot \pi, j \models \phi_1)$$

$$\pi, i \models \phi_1\ \textbf{releases}\ \phi_2 \equiv \forall i \leq k \cdot (\pi, k \models \phi_2 \vee \exists i \leq j < k \cdot \pi, i \models \phi_1)$$

$$\pi, i \models \textbf{historically}\ \phi_1 \equiv \forall j \leq i \cdot \pi, j \models \phi_1$$

$$\pi, i \models \textbf{once}\ \phi_1 \quad \equiv \exists j \leq i \cdot \pi, j \models \phi_1$$

$$\pi, i \models \textbf{before}\ \phi_1 \quad \equiv 0 < i \wedge \pi, i-1 \models \phi_1$$

$$\pi, i \models \phi_1\ \textbf{since}\ \phi_2 \quad \equiv \exists k \leq i \cdot (\pi, k \models \phi_2 \wedge \forall k < j \leq i \cdot \pi, j \models \phi_1)$$

$$\pi, i \models \phi_1\ \textbf{triggered}\ \phi_2 \equiv \forall k \leq i \cdot (\pi, k \models \phi_2 \vee \exists k < j \leq i \cdot \pi, j \models \phi_1)$$

Concrete examples of evaluation are shown below, relating to Figure 9.

- $[\![Track]\!]^n_\pi = \pi(n)(Track) = \{Track_0, Track_1\}, n \in \mathbb{N}$;
- $[\![Track_0]\!]^n_\pi = \pi(n)(Track_0) = \{Track_0\}, n \in \mathbb{N}$;
- $[\![active]\!]^0_\pi = \pi(0)(active) = \{Track_1 \times Track_0\}$;
- $[\![\sim active]\!]^0_\pi = \{Track_0 \times Track_1\}$
- $[\![Track]\!]^1_\pi = \pi(1)(active) = \{\}$;
- $[\![next - active]\!]^0_\pi = [\![next]\!]^0_\pi \backslash [\![active]\!]^0_\pi = \{Track_1 \times Track_0\} \backslash \{Track_1 \times Track_0\} = \{\}$;
- $[\![next - active]\!]^1_\pi = [\![next]\!]^1_\pi \backslash [\![active]\!]^1_\pi = \{Track_1 \times Track_0\} \backslash \{\} = \{Track_1 \times Track_0\}$;

The previous examples show the evaluation of a formula for a specific state. The evaluation of a formula for an entire trace is made by checking if it is valid for every state of that trace, $\pi \models \Phi \equiv \forall i \in \mathbb{N}_0 \cdot \pi, i \models \Phi$. For a formula to be generally valid, it has to be valid for every possible trace, $\models \Phi \equiv \forall \pi \cdot \pi \models \Phi$.

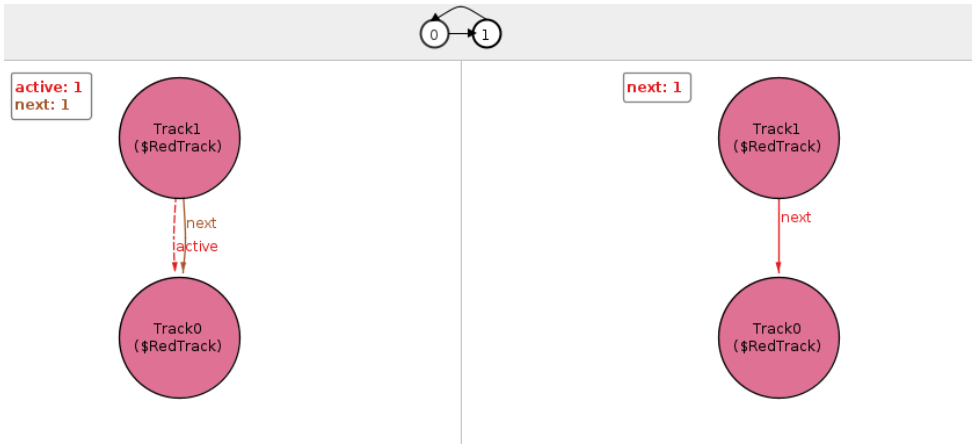


Figure 9: Instance with the active tracks changing.

3

State of the Art

This chapter will provide an overview of automatic repair. It will first introduce some concepts relating to the area and the various stages of an automatic repair technique. It will then analyze the previous work on Alloy automatic specification repair that works for static models, namely, the fault localization extension `AlloyFL`, the automatic repair tool `ARepair` [35, 34], the fault localization technique `FLACK` and the automatic repair technique `BeAFix` [8].

3.1 Overview

Automatic repair revolves around bugs, namely finding, correcting or improving them. The term 'bug' encompasses multiple terms [28]: *failures* refer to unacceptable behaviour; *errors* refer to an incorrect state prior to a failure; *faults* refer to the root cause of the error. A bug can then be seen as a difference in the expected versus the actual behavior of a system.

In the definition of a bug, there's an implicit notion of an observer, the observer is the one who dictates what the expected output is. This observer is very often a human simply looking at the behavior and saying it is not correct. In automatic program repair, however, the machine needs to be able to differentiate faulty outputs from correct ones. For this, the concept of an *oracle* is often introduced; it determines whether the behavior of the system is correct. It can do this using, for example, unit tests that map inputs to their expected outputs. In the case of Alloy, assertions can be added, allowing the use of the logic system and, therefore, a more precise way to differentiate correct specifications from incorrect ones.

Fault localization refers to the process of finding the exact localization or possible localizations that can be causing a fault [41]. Finally, program repair refers to the process of finding patches for the current code that the *oracle* deems as correct.

Unlike automatic program repair, there is little work for the automatic repair of specifications, even outside of Alloy. In the sections below, the current state of the art for automatic repair of Alloy models will be presented. Two techniques for fault localization and two techniques for model repairing, as well as some other techniques that complement these.

3.2 AlloyFL

AlloyFL is a fault localization tool for Alloy [21, 5]. It also denotes the first set of fault localization techniques for faulty Alloy models and also the first set of fault localization techniques at the AST node granularity [38]. AlloyFL relies on AUnit, a test automation tool for Alloy [31], that is responsible for checking the expected validity of certain instances for the model. AlloyFL does not rank all AST nodes due to the lack of control flow leading to many equally suspicious nodes. Furthermore, to reduce the number of returned nodes, the root node of the AST subtree with the highest number of equally suspicious AST nodes is the one returned. After presenting AUnit five variants of AlloyFL are presented, AlloyFL$_{co}$, AlloyFL$_{un}$, AlloyFL$_{su}$, AlloyFL$_{mu}$ and AlloyFL$_{hy}$.

3.2.1 AUnit

AUnit is inspired by the unit testing technique that is mostly used for imperative languages. In AUnit, it is possible to define a test case and the expected outcome for that test case. It also can generate coverage results, allowing the user to verify the parts of the code being tested by the test cases.

To quickly illustrate the capabilities of AUnit a simple model of a list will be presented along with a function and predicate. A list can either be empty or have a beginning node, which can then be connected to 0 or 1 other nodes.

```
sig List { begin: lone Node }
sig Node { tail: lone Node }
pred empty[l: List] {
    no l.begin
}
```

Valuations can now be specified with the `val` keyword. Valuations can be used in tests, to create a new test the `@Test` keyword is used. A `@cmd:...` keyword is also introduced to allow embedding formulas to the AUnit test. This is particularly useful to test predicates with parameters, whose variables wouldn't be possible to bind outside the valuation scope. For example in `Test1` shown below, writing `run testEmpty and empty[List0]` wouldn't make sense since `List0` isn't bound on that scope.

```
val testUnitList {
    some List0: List, Node0: Node {
        List = List0
        begin = List0->Node0
        Node = Node0
    }
}
val testEmpty {
```

```
    some List0: List, Node0: Node {
        List = List0
        begin = List0->Node0
        Node = Node0
        @cmd: { empty[List0] }
    }
}
@Test Test0: run testUnitList for 3 expect 1
@Test Test1: run testEmpty for 3 expect 0
```

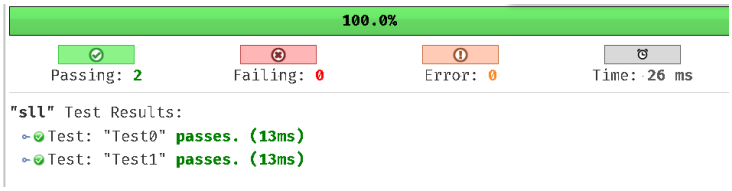


Figure 10: Previously defined tests passing.

The tests pass, as can be seen on Figure 10. Altering the `@cmd` to `not empty[List0]` will make the test fail, yielding the result seen in Figure 11.

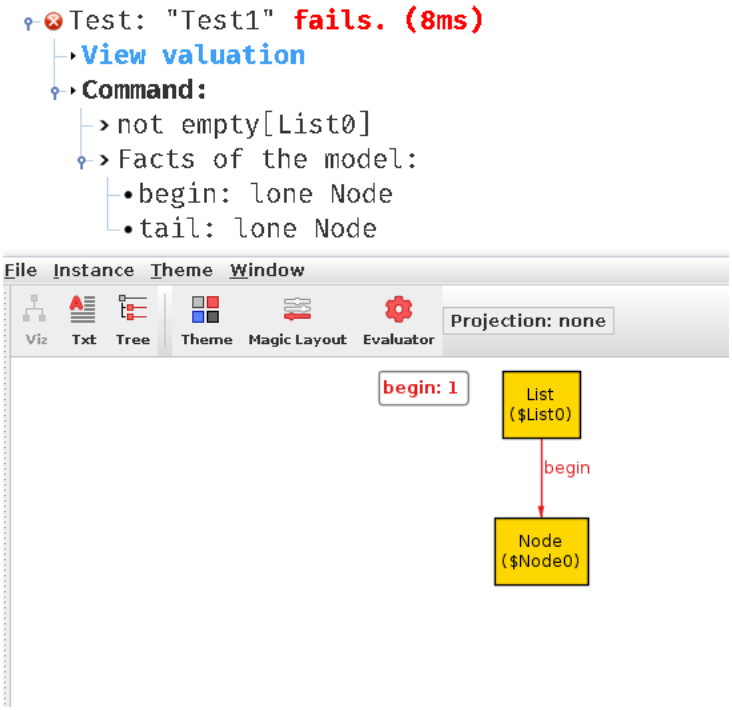


Figure 11: Altered test failing.

The generated coverage results are also handy to understand the cases missing in the test cases. For example, from Figure 12, it can easily be seen that there are no tests for instances with no lists, or more than one list.

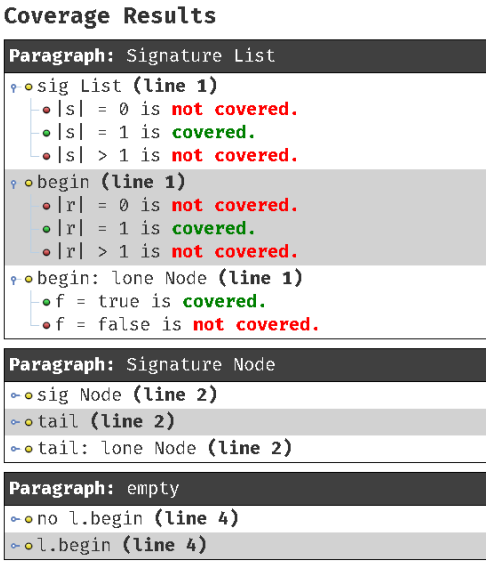


Figure 12: Coverage results.

The coverage is calculated by checking each test. If a signature is set with a cardinality of 0, 1 or above 1, then `|s| = 0`, `|s| = 1`, `|s| > 1` is marked as covered, respectively. Similarly, for the case of boolean expressions, if their result is false or true in a test, that result is marked as covered. Adding a simple test case for no list is reflected in the coverage, as seen in Figure 13.

```
val noList {
    no List
}
Test Test2: run noList for 3 expect 1
```

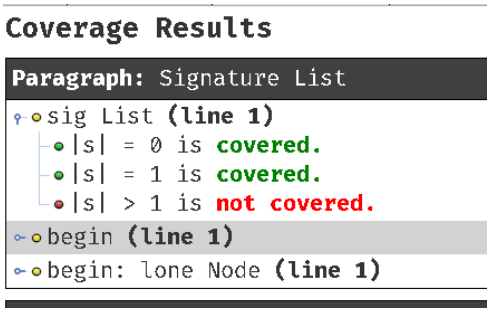


Figure 13: Coverage results.

3.2.2 AlloyFL$_{co}$

AlloyFL$_{co}$ implements the *spectrum-based FL* (SBFL) technique. It consists of running the test suite and checking the blocks used in each test. In imperative languages, these blocks can be, for example, single

instructions. Since Alloy is a declarative language, without control flow or execution traces, the expressions/formulas within a paragraph (i.e. signature, predicate, function, fact and assertion) are either all or none executed by a given test. This means nodes within the same paragraph will share the same suspiciousness scores.

By default, all facts are implicitly used for each test. So, this technique will only yield different suspiciousness scores for predicates called in the test suite but not in facts. A static analyzer is used to find all the paragraphs transitively used in each test. It then ranks them based on the number of passing/failing tests using a suspiciousness formula. This technique is expected to be inaccurate due to the limitations presented above.

AlloyFL$_{co}$ allows selecting one out of five suspiciousness ranking formulas. These are presented in Table 1, where *passed(e)* and *failed(e)* denote the amount of passed and failed tests with the expression being examined; and *totalpassed totalfailed* represent the total of passed and failed tests in the test suite.

Name	**Formula**
Tarantula [18]	$\frac{\frac{failed(e)}{totalfailed}}{\frac{failed(e)}{totalfailed}+\frac{passed(e)}{totalpassed}}$
Ochiai [2]	$\frac{failed(e)}{\sqrt{totalfailed \times (failed(e)+passed(e))}}$
Op2 [29]	$failed(e) - \frac{passed(e)}{totalpassed+1}$
Barinel [1]	$1 - \frac{passed(e)}{passed(e)+failed(e)}$
DStar [42]	$\frac{failed(e)}{passed(e)+(totalfailed-failed(e))}$

Table 1: Available suspiciousness ranking formulas for AlloyFL$_{co}$, AlloyFL$_{mu}$ and AlloyFL$_{hy}$ [21].

3.2.3 AlloyFL$_{un}$

AlloyFL$_{un}$ modifies the standard Alloy toolset to be able to return the potentially faulty AST nodes using unsat core (a set of formulas for which no satisfying instance exists). This is only available for the MiniSat solver since it is the only solver in Alloy that returns this information. Note that this technique only takes into account unsatisfiable failing tests, AlloyFL$_{su}$ expands this to satisfiable failing tests. It constructs a hit map for the entire AST with each node having an initial count of 0, and whenever a node is returned by the unsat core, the count for the node and its descendants increases by one. After checking every test it collects nodes whose counts are greater than their parents (note that a node count can only be greater or equal to its parent). The collected nodes are ranked in descending order of the corresponding count, in case of a tie in the count, nodes with fewer descendants are prioritized.

The researchers present the Figure 14 to illustrate this behavior. Firstly, all nodes have the count set to 0. There's then a hit on the node with a square, it and its descendants have their counts increased. After that occurs a second hit and the same process applies. In the end AlloyFL$_{un}$ would return the node hit first and second, with the first one being ranked higher in suspiciousness.

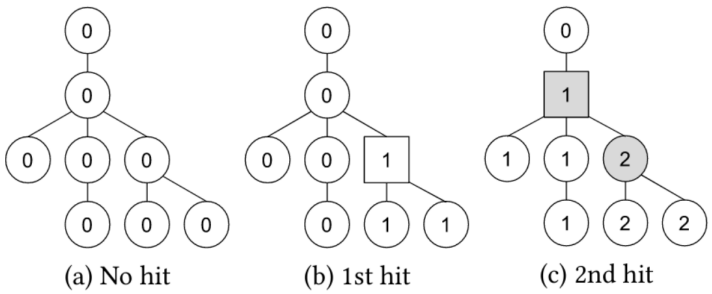


Figure 14: Illustration for AlloyFL$_{un}$ and AlloyFL$_{su}$ presented by the AlloyFL researchers [21].

3.2.4 AlloyFL$_{su}$

AlloyFL$_{su}$ is similar to AlloyFL$_{un}$, however, it also takes into account satisfiable failing tests. The nodes reported from the unsatisfiable tests are the same as the ones from AlloyFL$_{un}$. For satisfiable tests, it uses the same static analyzer from AlloyFL$_{co}$ and increases by one of the nodes returned by it.

3.2.5 AlloyFL$_{mu}$

AlloyFL$_{mu}$ implements a *mutation-based FL* (MBFL) technique. It mutates AST nodes to non-equivalent mutants (i.e. `a&&b` to `a||b`) and compares the results of the mutated model to the ones from the original model.

The algorithm can be stated simply as follows: collect all the nodes covered by failing tests using the static analyzer; for each of these nodes, attempt to apply every mutation (one at a time); if the mutation is not applicable, leads to compilation error or is equivalent to the original model, skip it; otherwise, rerun the

tests for the mutated model and compute a suspiciousness score for each node using one of the formulas presented in Table 1; the final score for each node is the maximum suspiciousness score it got through the mutated models.

Table 2 shows the list of mutation operators presented by the researchers, along with examples.

Operator	Description	Alloy
MOR	Multiplicity Operator Replacement	"one sig"⟶"lone sig"
QOR	Quantifer Operator Replacement	"some"⟶"all"
UOR	Unary Operator Replacement	"a.^b"⟶"a.*b"
BOR	Binary Operator Replacement	"a<=>b"⟶"a=>b"
LOR	Formula List Operator Replacement	"a&&b"⟶"a||b"
UOI	Unary Operator Insertion	"a"⟶" a"
UOD	Unary Operator Deletion	" a"⟶"a"
LOD	Logical Operand Deletion	"a&&b"⟶"b"
PBD	Paragraph Body Deletion	"fact{ e }"⟶"fact{}"
BOE	Binary Operand Exchange	"a-b"⟶"b-a"
IEOE	Imply-Else Operand Exchange	"a=>b else c"⟶"a=>c else b"

Table 2: Mutation operators used in AlloyFL$_{mu}$ [36].

3.2.6 AlloyFL$_{hy}$

AlloyFL$_{hy}$ implements a hybrid between the previously mentioned SBFL and MBFL techniques. It takes the average suspiciousness score obtained from AlloyFL$_{co}$ and AlloyFL$_{mu}$ and assigns a score to each AST node. In the case an AST node is not mutable (and therefore doesn't have a suspiciousness score from Alloy$_{mu}$) only the score from AlloyFL$_{co}$ is used. If multiple nodes have the same score, the ones with fewer descendants are prioritized.

In the benchmarks performed by the researchers, AlloyFL$_{hy}$ was able to perform better than the other variants under various metrics. Intuitively, it is designed to combine the strengths of the two other techniques, since AlloyFL$_{mu}$ tends to perform badly for omission errors (since it can't add extra information), where AlloyFLco performs relatively well.

3.3 ARepair

ARepair is a command-line tool, and it is the first automatic repair technique for Alloy [35, 34]. ARepair depends on multiple Alloy extensionsl, namely: the previously presented AlloyFL; AUnit, for the notion

of unit testing; MuAlloy, [32, 36] to define mutation operators for Alloy grammar and perform mutation testing; RexGen, [40] to generate non-equivalent relational expressions; and ASketch, [37, 39] to complete missing fragments of models in a way that all AUnit tests pass.

ARepair follows the *generate-and-validate* approach [15]. This means it receives an Alloy model and an AUnit test suite where the model triggers failing tests. ARepair then searches through various candidates and tries to fix the model in such a way that all tests pass. ARepair may run for multiple iterations to fix a model. This is because at each iteration it only applies a change in such a way that some of the previously failing tests pass, while the previously passing tests keep passing. This greedy approach means that ARepair may not always be able to find a patch to make all tests pass, and therefore it can fail to fix a model.

Since ARepair relies on many previously defined tools, the technique can be explained rather simply. The model and unit tests are fed into AlloyFL, which will output the AST nodes ranked by their suspiciousness score. ARepair attempts to apply mutations on the most suspicious node. The mutations are generated using MuAlloy. If a mutation making some failing tests pass while preserving passing tests, it is applied. In the case no such mutation is found, for each suspicious node, ARepair creates holes at each level of the AST in a bottom-up fashion, and tries to synthesize code for them. The holes are made using ASketch, and the synthesizations are made using RexGen. Assuming a suspicious AST with depth *D*, ARepair will replace all nodes at depth *D* with holes and try to synthesize code for these nodes such that some failing tests pass while keeping passing tests. In case no valid patch is found, ARepair will instead replace all the nodes at depth *D-1* with holes and repeat the process; if no patch is found at depth 0, ARepair moves on to the next suspicious AST node. Figure 15, presented by the ARepair researchers, shows how the various components link together as explained above.

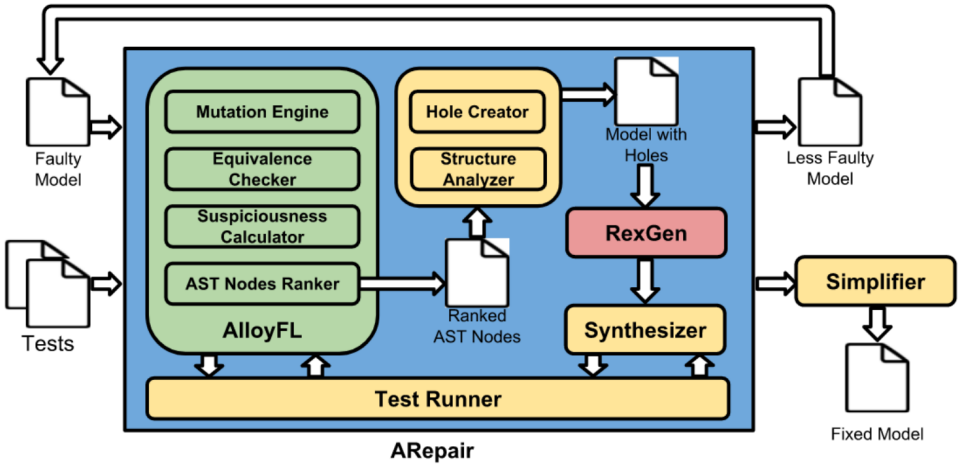


Figure 15: Diagram illustrating the various components in ARepair [34].

3.4 FLACK

FLACK is a tool suite for fault localization in Alloy models using counterexamples [45]. Unlike AlloyFL, it doesn't rely on unit tests to localize faults in models. Instead, it relies on Alloy assertions, more specifically, on the counter-examples presented by the Alloy Analyzer. These are already commonly used by Alloy users, so their usage is more natural and makes for a better user experience.

FLACK relies on various components, namely:

- the Alloy Analyzer, capable of generating counterexamples having as input an Alloy model and a violated assertion. A counterexample is an instance I of the model M that is invalid for the asserted predicate p, in logic notation: $I \models M \wedge \neg p$;
- A PMAX-SAT (*Partial-Max SAT*) solver, more specifically, the Pardinus solver [14]. Given an unsatisfiable normal form formula, a PMAX-SAT solver can calculate the maximum amount of clauses that can be made true. For FLACK, this is used to find the satisfiable instance most close to a counterexample;
- A Comparator, capable of identifying differences in instances (counterexamples generated by the Alloy Analyzer and the satisfiable model generated by the PMAX-SAT solver). These differences involve atoms, tuples and relations. Intuitively these differences are the ones causing faults so relations dependent on them are more suspicious;
- A Diff Analyzer that given the model, pairs of counterexamples and their satisfiable instances, along with their differences, can generate a ranked list of suspicious expressions in the model.

Figure 16 presents an overview of these components and how they interact, as shown by its researchers.

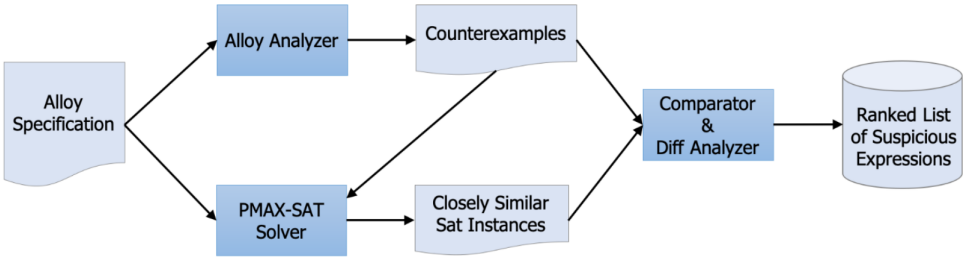


Figure 16: Overview of the components used in FLACK [45].

Algorithm 1 shows the base FLACK algorithm. It receives as input the model and a property not satisfied by the model (a violated assertion). FLACK uses the Alloy Analyzer to get the counterexample instances that violate the property. It then uses the Pardinus PMAX-SAT to generate pairs of counterexamples and their most similar satisfiable instances. After this, FLACK computes the differences between the pairs of instances and runs the diff analyzer to locate the error. If FLACK cannot generate a similar satisfiable instance, it instead uses the *unsat core* returned by the Alloy Analyzer to locate the error.

Algorithm 1: FLACK fault localization process [45].

Input: Alloy model M, property p not satisfied by M

Output: Ranked list of suspicious expressions in M

$AlloySolver \leftarrow AlloyAnalyzer(M, p)$
$pairs \leftarrow \emptyset$
while $|pairs| < max\_instance\_pairs$ **do**
    $c \leftarrow AlloySolver.gencex()$
    $AlloySolver.blockcex(c)$
    $s \leftarrow PMaxSolver(M, c)$
    if $s = nil$ **then**
        $U \leftarrow AlloySolver.get\_unsatcore()$
        return $unsat\_analyzer(M, U, c)$
    $pairs \leftarrow pair \cup (c, s)$
end
$diffs \leftarrow comparator(pairs)$
return $diffs\_analyzer(diffs)$

Algorithm 2 shows the algorithm FLACK uses to calculate the suspiciousness score of expressions based on the differences between counterexamples and their satisfiable instances. It takes the model, pairs of counterexamples, their satisfiable instances and the differences between them and returns a ranked list of suspicious expressions. Firstly, the diff analyzer tries to compute the suspicious expressions based on the diff. For example, if two atoms are part of a relation in the counterexample but not in the satisfiable instance, expressions dependent on these atoms or this relation will be flagged as suspicious. After these suspicious expressions are calculated, a score is computed for each of them, the expressions are given as an AST tree and computed recursively.

- If the expression is a leaf, then FLACK instantiates it with the atoms from the instance differences, and then it evaluates it for each pair of counterexamples and satisfiable instances. If the evaluated result for an instantiated expression contains all atoms involved in the differences, it increases the score for the expression. If no evaluated result contains all atoms involved in the differences, the returned score will be 0.

- In case the expression evaluates to a Boolean (i.e. 'foo in bar'), then it is instantiated with atoms from the differences and evaluated for each pair. If the result is different for the counterexample and valid instance, the score is incremented. This score is then added to the sum of the scores of the children expressions.

- If the expression is neither a leaf nor evaluates to a Boolean, the score is the sum of the scores of the children expressions.

Algorithm 2: FLACK Diff Analyzer. [45].

Input: Alloy model M, pairs of counterexamples and satisfiable instances *pairs*, differences between the pairs *diffs*

Output: Ranked list of suspicious expressions in M

$exprs \leftarrow get\_susp\_exprs(M, diffs)$
$results \leftarrow \emptyset$
foreach $expr \in exprs$ **do** $compute\_score(expr, results)$
return $sort(results)$

Function $compute\_score(expr, results)$ **is**
    $score \leftarrow 0$
    if $isleaf(expr)$ **then**
        $isexpr \leftarrow instantiate(expr, diff)$
        foreach $(c, s) \in pairs$ **do**
            $cvals \leftarrow eval(c, isexpr)$
            $svals \leftarrow eval(s, isexpr)$
            $instscore \leftarrow 0$
            if $diff \subset cvals$ **then**
                $instscore \leftarrow instscore + \frac{|diff|}{|cvals|}$
            if $diff \subset svals$ **then**
                $instscore \leftarrow instscore + \frac{|diff|}{|svals|}$
            $score \leftarrow score + \frac{instscore}{2}$
        end
        $score \leftarrow \frac{score}{|pairs|}$
    else
        if $isbool(expr)$ **then**
            $isexpr \leftarrow instantiate(expr, diff)$
            foreach $(c, s) \in pairs$ **do**
                if $eval(c, isexpr) \neq eval(s, isexpr)$ **then**
                    $score \leftarrow score + 1$
            end
        foreach $child \in getchildren(expr)$ **do**
            $score \leftarrow score + compute\_score(child, results)$
        end
    end
    $results \leftarrow results \cup (expr, score)$
end

There might be cases where a counterexample is generated, but similar satisfiable instances can't be

found. This indicates that the model has a logic conflict with the property we want to check (a contradiction). In these cases FLACK inspects the *unsat core* generated by Alloy. This is a smaller subset of the original constraints containing the conflicting constraints.

Algorithm 3 explains the process of obtaining the faulty expressions based on the unsat core. It receives as input an Alloy model, a counterexample and the unsat core generated by Alloy. FLACK starts by producing a sliced model of the original where the expressions in the unsat core are omitted. Since the conflicting instances are removed, it should now be possible to generate a satisfiable instance with the sliced model to compare with the counterexample. The counterexample is compared to the satisfiable instance and the differences between them are identified. Finally, FLACK attempts to identify which removed expressions are really conflicting with the assertion by evaluating them on obtained differences. Intuitively, if an expression evaluates to true this means adding it back to the model would still allow the sat instance to be generated. Therefore, expressions evaluating to false are the ones conflicting with the assertion, and so they return as suspicious.

Algorithm 3: FLACK UNSAT Analyzer [45].

Input: Alloy model M, unsat core U, counterexample c
Output: a set of expressions in M
$M' \leftarrow slice(M, U)$
$s \leftarrow PMaxSolver(M', c)$
$diffs \leftarrow comparator((c, s))$
$exprs \leftarrow collect\_exprs(U)$
$conflicts \leftarrow \emptyset$
foreach *expr* $\in$*exprs* **do**
  foreach $diff \in diffs$ **do**
    if $eval(expr, diff, M') = false$ **then**
      $conflicts \leftarrow conflicts \cup expr$
  end
end
return $conflicts$

The FLACK researchers present various benchmarks for the technique, notably, where some models were compared to AlloyFL. The results of this comparison are shown in Figure 17. As can be seen, for the 152 benchmark models, FLACK was able to better rank the faulty expressions (a rank *n* means the faulty expression was ranked as the *n-th* most suspicious position). Perhaps more importantly, FLACK can generate the ranking of suspicious nodes two orders of magnitude faster than AlloyFL. FLACK failed to find the faulty expression more often than AlloyFL.

tool	top 1	5	10	> 10	failed	avg rank	time(s)
FLACK	91	126	136	6	10	2.4	0.2
AlloyFL	76	128	137	8	7	3.1	32.4

Figure 17: Comparison between FLACK and AlloyFL [45].

3.5 BeAFix

BeAFix is a tool for bounded exhaustive search of Alloy specification repairs [8]. This means it explores every possible repair candidate within certain bounds, applying pruning techniques to reduce the search space without losing valid candidates. Similarly to FLACK, this technique goes along the spirit of Alloy which also searches all possible instances within a bounded scope to verify assertions.

For the purposes of repair, BeAFix describes a faulty specification as one where at least one of the analysis commands for it has an outcome contrary to expected. This encompasses things like failing assertions or unsatisfiable predicates. Similarly, a fix is defined as a patch capable of making the outcome for all the commands align with its expected outcome. This technique, therefore, requires the user to specify the assertions, predicates, or tests that the technique will use for its fix acceptance criterion. This specified portion of the code cannot be changed by the generated patch as it would lead to trivial solutions.

The fault detection stage resorts to the underlying mechanism behind Alloy Analyzer. Once a specification is regarded as faulty, there's a need to identify the parts of the specification that are more likely to be causing the fault. For this, BeAFix assumes an external tool able to provide a ranking of suspicious specification locations. The implementation of this technique was developed with FLACK as a backend for fault localization. However, the technique doesn't rely on unit testing. Thus, unlike ARepair, it should be compatible with any fault localization technique, as long as the acceptance criteria is valid for said fault localization technique.

With the suspicious nodes identified, BeAFix can start the process of generating fix candidates. These candidates are mutations of the original model. Examples of mutation operations considered are: logical and relational operator insertion, removal and replacement; quantifier replacement; multiplicity constraint replacement; field or variable swap and replacement. The specification is processed for type information to prevent generating mutations that would lead to type errors. Contradictory formulas are also disregarded. Mutations can be applied to already mutated models, and both the available mutation operations and the maximum number of combined mutations can be configured. These are then exhaustively generated (up until the configured maximum amount of combined mutations) as the space of fix candidates is traversed.

Since there will be a very high amount of possible generated fixes, BeAFix presents pruning techniques capable of eliminating candidates, without the loss of candidates that will possibly be fixes. The search space is organized as a search tree, the root being the original specification with the obtained faulty locations. If specification *s* is in the search tree and *s'* can be obtained by applying a mutation to *s*, then s' will also be in the tree with the same locations marked as faulty. For BeAFix nodes of the tree are visited

in a breadth-first fashion. Other traversals could be adopted without impacting the ability to find a fix (but possibly changing execution time).

BeAFix proposes two pruning techniques. The first pruning technique is referred to as *Partial repair checking*. It consists in identifying one of the suspicious locations for which a current repair candidate fails independently of changes in other suspicious locations. Let $Spec$ be an Alloy specification, $Check_1$, ..., $Check_k$ the checks used as oracles and L_0, L_1 the suspicious locations previously identified. Each $Check_i$ refers only to specific parts of $Spec$, these can be identified by syntactic analysis (recursively checking the formulas, relations, etc. used inside the check). With this analysis, it can be determined for every check which of the locations it involves. Let m_0 and n_0 be the modifications for L_0 and L_1, respectively. If a failing check $Check_i$ refers to only one of the suspicious locations, say L_0 and its current expression m_0, this means that $Check_i$ will still be false for any n_0. Therefore, if m_0 is maintained, modifying L_1 cannot be a fix for $Spec$, and this means all the candidates (m_0, n_i) can be excluded from the search.

It is important to note that the previously presented pruning technique relies on some properties of Alloy's relational logic. Namely, that the validity and satisfiability of a formula only depends on symbols it refers to; and the property that adding more assumptions to a formula cannot reduce the conclusions drawn from it originally (this technique relies on unsatisfiability, where $false$ is concluded).

The second pruning technique is referred to as *Variabilization*. It consists in identifying modifications that won't generate fixes when paired with any modifications in a specific location. Let $Check_i$ be a failing assertion referring to suspicious locations; L_0 and L_1, (m_0, n_0) the current failing fix candidates; and CEX_i a counterexample as result of the assertion violation ($CEX_i \not\models Spec[m_0, n_0] \rightarrow Check_i$). The purpose of variabilization is to check whether the current fix for L_0 may work with some candidate for L_1 (n_0 is already known not to work). Let F_1 be a formula containing L_1, T the most general type for L_1 in context F_1, $Spec_{L1}$ the specification obtained by replacing F_1 in $Spec$ with the expression $\exists l1 : T | F1[l1/L1]$. In $Spec_{L1}$, $L1$ is replaced by the existence of any variable of type T, which will be either the same or more generic type than the type of $L1$. The expression $CEX_i \models Spec_{L1} \rightarrow Check_i$ can be now checked, and will verify if there exists any value of type T for which CEX_i stops being a counterexample. If this check fails, it means no value that is put in L_1 would work while m_0 is in L_0, therefore all candidates (m_0, n_i) can be discarded from the search.

Figures 18 and 19 show the benchmarks presented in the BeAFix paper. The benchmarks are made with FLACK being used as the method for BeAFix to get the suspicious locations. In general, BeAFix seems to be able to find repairs more consistently than ARepair and with better average times. Notably, there is a discrepancy between the success of ARepair for its own benchmarks and the ones from Alloy4Fun. The explanation presented by the BeAFix researchers is that the unit tests for Alloy4Fun tests have been automatically generated using AUnit. Which lead to incorrect fixes that passed the generated tests, while the benchmarks from ARepair were manually designed preventing as much overfitting. This shows that the ARepair has its results very dependent on writing the right set of tests. This would be time-consuming since, as shown, automatically generated tests don't yield the best results.

Model	Total Cases	ARepair				BeAFix			
		Repaired (%)	*Avg. time*	*Correct* (%)	*Incorrect* (%)	*Repaired* (%)	*Avg. time*	*Correct* (%)	*Incorrect* (%)
addr	1	1 *(100%)*	*9010*	1 *(100%)*	0 *(0%)*	1 *(100%)*	*351*	1 *(100%)*	0 *(0%)*
arr	2	2 *(100%)*	*7651*	2 *(100%)*	0 *(0%)*	2 *(100%)*	*2394*	2 *(100%)*	0 *(0%)*
balancedBST	3	2 *(67%)*	*120276*	1 *(33%)*	1 *(33%)*	1 *(33%)*	*358*	1 *(33%)*	0 *(0%)*
bempl	1	0 *(0%)*		0 *(0%)*	0 *(0%)*	0 *(0%)*		0 *(0%)*	0 *(0%)*
cd	2	2 *(100%)*	*3302*	0 *(0%)*	2 *(100%)*	2 *(100%)*	*742*	2 *(100%)*	0 *(0%)*
ctree	1	1 *(100%)*	*6774*	1 *(100%)*	0 *(0%)*	0 *(0%)*		0 *(0%)*	0 *(0%)*
dll	4	3 *(75%)*	*22585*	0 *(0%)*	3 *(75%)*	3 *(75%)*	*2624*	3 *(75%)*	0 *(0%)*
farmer	1	0 *(0%)*		0 *(0%)*	0 *(0%)*	0 *(0%)*		0 *(0%)*	0 *(0%)*
fsm	2	2 *(100%)*	*6068*	2 *(100%)*	0 *(0%)*	1 *(50%)*	*313*	1 *(50%)*	0 *(0%)*
grade	1	1 *(100%)*	*124797*	0 *(0%)*	1 *(100%)*	0 *(0%)*		0 *(0%)*	0 *(0%)*
other	1	0 *(0%)*		0 *(0%)*	0 *(0%)*	1 *(100%)*	*3120*	1 *(100%)*	0 *(0%)*
student	19	12 *(63%)*	*76120*	9 *(47%)*	3 *(16%)*	13 *(68%)*	*71197*	13 *(68%)*	0 *(0%)*
Total:	38	26 *(68%)*	*41843*	16 *(42%)*	10 *(26%)*	24 *(63%)*	*10137*	24 *(63%)*	0 *(0%)*

Figure 18: Comparison between ARepair and BeAFix for ARepair benchmarks [8].

Model	Total Cases	ARepair				BeAFix			
		Repaired (%)	*Avg. time*	*Correct* (%)	*Incorrect* (%)	*Repaired* (%)	*Avg. time*	*Correct* (%)	*Incorrect* (%)
Graphs	305	276 *(90%)*	*2625*	18 *(6%)*	258 *(85%)*	248 *(81%)*	*6734*	248 *(81%)*	0 *(0%)*
LTS	282	165 *(59%)*	*8729*	7 *(2%)*	158 *(56%)*	42 *(15%)*	*5999*	42 *(15%)*	0 *(0%)*
Trash	229	220 *(96%)*	*4077*	68 *(30%)*	152 *(66%)*	199 *(87%)*	*4915*	199 *(87%)*	0 *(0%)*
Production	63	47 *(75%)*	*6232*	8 *(13%)*	39 *(62%)*	56 *(89%)*	*4124*	56 *(89%)*	0 *(0%)*
Classroom	1168	911 *(78%)*	*95717*	92 *(8%)*	819 *(70%)*	418 *(36%)*	*82856*	418 *(36%)*	0 *(0%)*
CV	162	132 *(81%)*	*4966*	4 *(2%)*	128 *(79%)*	82 *(51%)*	*2805*	82 *(51%)*	0 *(0%)*
Total:	2209	1751 *(79%)*	*20391*	197 *(9%)*	1554 *(70%)*	1045 *(47%)*	*17905*	1045 *(47%)*	0 *(0%)*

Figure 19: Comparison between ARepair and BeAFix for Alloy4Fun models [8].

4

Temporal Alloy Repair

This chapter will provide an overview of the technique developed, along with explanation of the implementation, and the decisions made in the development process. It will also provide benchmarks comparing the results of previously developed tools with TAR, the technique presented in this chapter, as well as benchmarks for temporal models and other observations.

The developed tool is capable of repairing Alloy 6 temporal specifications, as expected by the main goal of this thesis. It was, therefore, named `TAR`, standing for *Temporal Alloy Repair*.

4.1 Overview

The benchmarks shown in Figures 18 and 19 seem to indicate that the mutation-based repair technique, presented in `BeAFix`, is able to produce better results than the mainly synthesis-based technique, presented in `ARepair`. For this reason, the technique in `TAR` gets its roots from `BeAFix`. Namely, it is also mutation-based, exhaustively searching the candidates up to a certain depth; and uses pruning methods to skip certain candidates known not to be a valid solution. The main differences between the techniques are: the ability to repair temporal expressions; a different set of mutators; and differences in pruning techniques.

`TAR` has also been developed with the intent of being used in a teaching and learning context, as explained in the motivation. Specifically, to generate hints for students using the Alloy4Fun platform. Alloy4Fun is a platform that allows the instructor to create challenges for students, by permitting the definition of hidden predicates and checks. In typical usage, the instructor will write a base model. Then, a hidden predicate, corresponding to the correct answer. Also, a hidden check, which verifies if the hidden predicate is equal to the answer given by the student. This has two main implications for the repair: the tool does not need to have fault localization implemented because the fault will always be in the predicates written by the student; and second, in typical usage, the student only has to write one predicate per challenge. This means pruning techniques relying on having multiple suspicious locations in different paragraphs are not useful. Both techniques in `BeAFix` are only applied in these cases (although variabilization could be applied more generally). To further illustrate how the platform works, consider the

```
var sig File {
    var link : lone File
}
var sig Trash in File {}
var sig Protected in File {}

//SECRET
pred prop4o { eventually some Trash }

//SECRET
check prop4ok { prop4 <=> prop4o }

// some file will eventually be sent to the trash
pred prop4 { }
```

Listing 1: Portion of the code from a challenge in Alloy4Fun.

Alloy snippet below, which is part of the code behind an exercise in Alloy4Fun.

The exercise presents the model of a simple filesystem. It contains three signatures: `File`, to represent the files in the system; `Trash` which is a subset of `File` and represents the files that have been sent to the trash; and `Protected`, also a subset of `File`, to represent protected files. It also contains the `link` relation, indicating if a file is a regular file or a link to a different file. All these signatures and relations are marked as variable, indicating that files can appear, disappear, and change status throughout temporal states.

The fourth challenge asks the student to write a predicate that evaluates to true if, and only if, a file is at any time sent to the trash. The solution proposed by the tutor can be seen in the `prop4o` predicate. It is marked with the `//SECRET` so the platform knows this paragraph should be hidden from the student. The student is expected to fill the body of the `prop4` predicate with the solution to the challenge. To verify the answer, the `prop4ok` check is used to test if the solution proposed is semantically equivalent to the one written by the student. Here the previous points are clear, the student can only change one predicate, `prop4`, and the fault, if it exists, is always present in that predicate since `prop4o` and `prop4ok` are correct.

A common faulty solution submitted by students is the following.

```
pred prop4 { some f : File | eventually f in Trash }
```

At first glance, this solution might seem correct. However, without temporal operators, expressions are only evaluated for the first state. This means that the property actually only checks if some file present in the initial state is eventually in the trash. Knowing this, it's easy to figure out what counterexample Alloy could give, an instance where a file is added to the system after the first state, and then also sent to the

trash, Figure 20 shows this counterexample. This counterexample would then be shown to the students, which they are expected to interpret to find the fault in their reasoning.

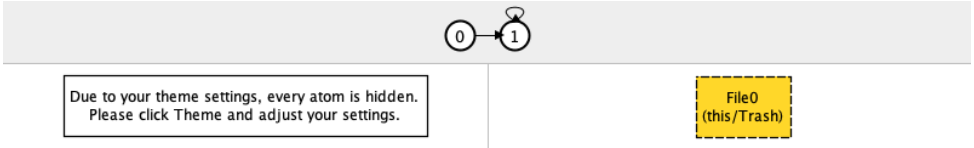


Figure 20: Counterexample to a faulty predicate shown by Alloy.

In mutation-based approaches to repair, candidates are generally obtained by mutating an AST node into a new one based on certain mutation rules. The depth of a candidate is defined by the number of mutators that compose it. For instance, the incorrect submission above could have the following candidates at depth 1, meaning these candidates are composed of only one mutator:

- `some f : Trash | eventually f in Trash`

  In this candidate, the `File` set is replaced by `Trash`;

- `some f : File | eventually f in File`

  In this candidate, the `Trash` set is replaced by `File`;

- `some f : File | always f in Trash`

  In this candidate, the `eventually` operator is replaced by `always`;

- `some f : File | after f in Trash`

  In this candidate, the `eventually` operator is replaced by `after`;

- `some f : File | eventually not (f in Trash)`

  In this candidate, a `not` unary operator is added, negating the `f in Trash` expression;

- `one f : File | eventually f not in Trash`

  In this candidate, the `some` quantifier is replaced by `one`;

- `eventually some f : File | eventually f in Trash`

  In this candidate, the `eventually` operator is added behind the entire expression.

Keep in mind that this is not an exhaustive list of candidates that might be generated at depth 1. Of these mutants, only the last one, `eventually some f : File | eventually f in Trash`, is equivalent to `prop4o` and therefore a correct solution. Note that this solution is not syntactically equivalent to the oracle, it is however semantically equivalent, and closer to the shape of the predicate submitted by the student. This means that for every instance within the scope of the validation command, the oracle expression and the candidate evaluate to the same value. Having to evaluate all the instances within the

scope implies that making sure a candidate is a correct solution amounts to calling the solver in Alloy to run the check command. Calls to the solver can be expensive, especially if a simple counterexample cannot be found. Since a lot of candidates will be generated, the process might generally take a long time. Thus, the necessity for pruning methods, without them, the repair times will be too high and, therefore, the educational application will be infeasible.

Note, however, that Figure 20 can also serve as a counterexample for all the candidates presented above (except the correct one), since it is accepted by the oracle but not the candidates. This is the main idea behind the pruning method in TAR. Counterexamples can be shared between candidates, this means that to disqualify a candidate it might only be necessary to call the Alloy *evaluator*, which is less expensive than calling the solver. Also, by storing the counterexamples obtained, these can then be used to attempt to skip future candidates.

4.2 Implementation

This section explains the concepts and implementation of the tool. It starts by explaining the concept of mutators. Then, it goes over how these mutators are combined to create candidates. Finally, how counterexamples are used to prune candidates.

It is also important to introduce the concept of a location in TAR, a location is simply an AST node. In the implementation, it also carries some extra metadata relating to the AST node, for example, the variables in scope. To generate the suspicious locations, the user has to specify a predicate named `__repair` and put the suspicious predicates there. This predicate name has no special meaning for Alloy. It was chosen with the intent to make its purpose clear, while not colliding with predicates with names more relevant to the model. TAR will then go to each of the predicates referenced and add every expression and subexpression of them as a suspicious location.

4.2.1 Mutators and Candidate Generation

A mutator can be seen as a pair of a location (reference to the original expression), and the mutated expression. Mutators also carry a list of *blacklisted* locations, meaning these locations will disappear or are in some way not allowed to change when the mutator is applied. For example, a mutator can take an originally binary expression and keep only one of its sides. In this case, the other side will be blacklisted, since its subexpressions will cease to exist after the mutator is applied. Mutators are also able to generate children, these children are new mutators that apply to locations created by the current mutator. The application of a mutator is simple, take the example of Figure 21. The original expression is `some File + Trash`, and the mutator is `File` $\rightsquigarrow$ `Trash`. The numbers represent the reference to the AST node object. Notice that locations are compared by reference, not syntactically. If the latter was the case, there could be ambiguity on which node a mutator is applied to, however, the reference is unique. Using the nodes reference for comparison has favorable properties. For example Figure 21, after the mutator is

applied, there will be two nodes syntactically equivalent (`0x3` and `0x4`) and there would be ambiguity if a mutator was to be applied to one of them.

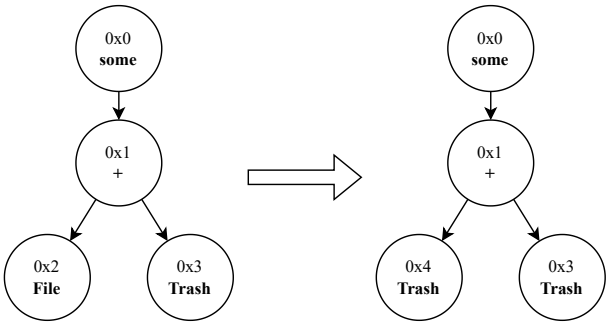


Figure 21: Diagram of the result of applying the candidate {`File` ⇝ `Trash`} to the expression `some File + Trash`.

A candidate is obtained from the application of multiple mutators. It is useful to be able to treat a candidate as a set of mutators instead of a list, because this means candidates $[m_0, m_1]$ and $[m_1, m_0]$ are actually equivalent. The issue arises when m_0 and m_1 are applied to the same location. In this case, the order of application does matter. To prevent this, two mutators belonging to a candidate cannot be applied to the same location. Instead, the mutator can generate a new location and more fine-tuned children mutators for that location. This allows skipping a lot of candidates, for example, if $[m_1, m_0]$ is skipped this way, then $[m_1, m_0, m_n]$ can also be skipped, since an equivalent candidate will be covered by $[m_0, m_1, m_n]$.

Tables 3 and 4 show the available mutators in TAR. These are not a complete representation of the implementation as some of the mutators were merged into the same line due to having similar behavior. The tables have three columns: one with the name attributed to the mutator; one showing the general format of the mutator; and one giving an example for a possible instance of the mutator. The tables are split into two categories, which are based on the type of expression they target. An expression can either be Boolean or relation, and the mutations in TAR always keep Boolean expressions as Boolean and relational expressions as relations.

Each mutator might only be implemented for certain expressions, for example, `RemoveBinary` is (for obvious reasons) only applicable when the expression is binary. Most mutators also have generation rules to reduce the number of candidates that will cause type errors. The following explains these rules:

- `RemoveBinary` is only implemented for binary expressions. The only rule is that the side of the expression being taken has the same type as the original. For example, `in` operator will never be removable since it takes two relations and returns a Boolean. However, we can always generate two mutators for the operator `and`;

- `ReplaceBinary` is only implemented for binary expressions. It defines three groups of operators, $Bool \times Bool \rightarrow Bool$, $Relation \times Relation \rightarrow Relation$ and $Relation \times Relation \rightarrow Bool$, and

only replaces operations within these same groups. For example, the operator + can be replaced with −, since both are in the $Relation \times Relation \rightarrow Relation$ group, but it cannot be replaced with the operator `or`, which is in the $Bool \times Bool \rightarrow Bool$ group;

- `RemoveUnary` is only implemented for unary expressions. The only rule is that the subexpression of the original is of the same type. For example, the `not` operator can always be removed, but multiplicities, like `one` and `some`, cannot;

- `ReplaceUnary` is only implemented for unary expressions. It defines three groups of operators, $Bool \rightarrow Bool$, $Relation \rightarrow Bool$ and $Relation \rightarrow Relation$, and only replaces operations within the same group. For example, `always` is replaceable by `eventually` since both belong to $Bool \rightarrow Bool$, but it is not replaceable by multiplicities, which belong to $Relation \rightarrow Bool$;

- `BinaryToUnary` is only implemented for binary expressions. The original operator has to be of type $Relation \times Relation \rightarrow Bool$. This allows us to replace the original operator with one of the group $Relation \times Relation \rightarrow Relation$ and apply to the result an operator from the group $Relation \rightarrow Bool$, matching the type of the original expression. For example, the expression `A in B` takes the relations `A` and `B` and returns a Boolean, this can be changed to `no (A + B)` which takes the same relations and also returns a Boolean;

- `InsertUnary` is implemented for any expression. The only rule is that types are kept. For example, the `not` operator can always be added to a Boolean expression;

- `QuantifierToUnary` is implemented only for quantifiers. It takes the relation being bounded to one of the variables and applies an operator of the group $Relation \rightarrow Bool$ to it. For example, `some x: X | foo[x]` would result in the mutator `some X`;

- `ReplaceQuantifier` is implemented only for quantifiers. It simply replaces quantifiers, it will always type check since it does not consider `sum` and `comprehension`;

- `ReplaceRelation` is implemented for any expression that is a relation. It guarantees that the arity of the relation is maintained, but the type can be changed;

- `InsertBinary` is split into multiple implementations. It tries to maintain the arity of the relation and also checks that the resulting type does not result in an empty set. For example, if `A` is a subset of `B`, then the mutator `A - B` would not be generated from the expression `A`, as this would result in an empty set. Similarly, if `A` has an arity of 1 and `B` has an arity of 3, the mutator `A . B` would not be generated, even if the expression type checks, since this would result in an expression of arity 2.

Name	Mutation	Example
RemoveBinary	A [bop] B ⇝ A A [bop] B ⇝ B	A and B ⇝ A
ReplaceBinary	A [bop] B ⇝ A [bop'] B	A and B ⇝ A or B
RemoveUnary	[uop] A ⇝ A	always no A ⇝ no A
ReplaceUnary	[uop] A ⇝ [uop'] A	no A ⇝ some A
InsertUnary	A ⇝ [uop] A	no A ⇝ always no A
BinaryToUnary	A [bop] B ⇝ [uop] (A [bop'] B)	A in B ⇝ no (A + B)
QuantifierToUnary	[qtop] a:A | B ⇝ [uop] A	no a:A | foo[a] ⇝ no A
ReplaceQuantifier	[qtop] a:A | B ⇝ [qtop'] a:A | B	no a:A | foo[a] ⇝ some a:A | foo[a]

Table 3: List of mutators for Boolean formulas.

Name	Mutation	Example
RemoveBinary	A [bop] B ⇝ A A [bop] B ⇝ B	A + B ⇝ A
ReplaceBinary	A [bop] B ⇝ A [bop'] B	A + B ⇝ A - B
RemoveUnary	[uop] A ⇝ A	~A ⇝ A
ReplaceUnary	[uop] A ⇝ [uop'] A	^A ⇝ *A
InsertBinary	A ⇝ A [bop] B	A ⇝ A + B
InsertUnary	A ⇝ [uop] A	A ⇝ ~A
ReplaceRelation	A ⇝ B	A ⇝ B

Table 4: List of mutators for relational expressions.

Algorithm 4 illustrates the candidate generation process in TAR. The algorithm starts by initializing the various variables necessary for the generation. The list of candidates, $cands$ is initialized with a single value, a candidate with no mutators, which will return the original model when applied; a variable $curr\_candidates$ keeps track of the index of the candidate that will be returned next call to the iterator; another variable, $curr\_generation$ keeps track of the index of the candidate that will be used to generate more candidates when the ones currently in the list have all been returned; finally, the variable $mutators$ stores the mutators for the initial repair locations (the ones not generated by other mutators), the generation is made one location at a time, using the rules presented previously. The function `next` is seen here as an iterator, which returns the next candidate to be used. If some of the candidates inside $cands$ have not already been returned, meaning the variable $curr\_candidate$ is smaller than the length of $cands$, then the candidate is simply extracted using the index $curr\_candidate$ and the variable is incremented. Otherwise, if all the candidates inside $cands$ have been returned, there is the need to generate more. To achieve this, the candidate in index $curr\_generation$ is retrieved and its children candidates are generated. These children candidates are generated by appending a mutator to the candidate, under the rules that this mutator is not incompatible with one of the mutators inside the candidate. After these new candidates are generated they are traversed and added to the list of candidates if there is no

previously equivalent one. In this case, an equivalent candidate means one that has the same mutators inside it, even if in a different order. After this process, the value of $curr\_generation$ is incremented. The generation might yield no new candidates, in which case this process is repeated with a new index in $curr\_generation$. If $curr\_generation$ reaches a value equal to the length of $cands$, this means no new candidates can be generated, and thus, the function returns $\perp$.

Algorithm 4: Candidate generation process in TAR.

```
Input: A set of repair locations.
Output: An iterator over the candidates.
cands ← [[]] // list of candidates generated, initialized with just an empty candidate
curr_candidate ← 0 // index of the candidate that will be returned in the next call
curr_generation ← 0 // index of the next candidate new children candidates will be generated
 from

mutators ← {} // set of mutators generated
// generate and store the mutators for each location
foreach location ∈ locations do
  | mutators.add(generate_mutators(location))
end

Function next() is
  | // try to generate more candidates while all the currently generated ones have already been
  |    returned
  | while curr_candidate ≥ cands.length() do
  |   | if curr_generation ≥ cands.length() then return ⊥
  |   | curr_generation_cand ← cands.get(curr_generation)
  |   | generated_cands ← curr_generation_cand
  |   | foreach cand ∈ generated_cands do
  |   |   | if ¬cands.any_prev_equivalent(curr_generation) then
  |   |   |   | cands.add(cand)
  |   | end
  |   | curr_generation ← curr_generation + 1
  | end
  | ret ← cands.get(curr_candidate)
  | curr_candidate ← curr_candidate + 1
  | return ret
end

return next
```

4.2.2 Mutation-based repair with counterexample-based pruning

In Alloy, a check command with a formula ϕ over a specification defined by formula ψ (in Alloy, defined through `fact` constraints) is converted into a model finding problem for a single formula, $\neg\phi \wedge \psi$. If

for any instance, this formula evaluates to true, then that instance is a counterexample, it also means that both $\neg\phi$ and ψ are true. If we mutate the model and either ψ or $\neg\phi$ evaluate to false, then the instance is not a counterexample for the new model. This is the basis for why the counterexamples of previous candidates can be used to prune future candidates. If the instance of a previous counterexample is stored, it can be evaluated for the new model and check formula. If it keeps being a counterexample, the candidate can be pruned.

Algorithm 5 gives a good idea of how the repair is done in TAR. It starts by getting the iterator over the candidates, which is shown in Algorithm 4, and, by initiating an empty priority queue for the counterexamples. Then, while more candidates can be generated, first it is checked if the candidate generated is over the max depth. If so, no solution can be found up to the max depth (note that the depth of a candidate is always equal or superior to the previous one); after that, the candidate is applied to the original model to obtain the mutated model; next, the counterexamples are traversed, starting with the one with highest priority (the one that has been able to prune the most candidates). If at any point a previous counterexample is also a counterexample for the current candidate, then the candidate is skipped and the priority for the counterexample is incremented; if the candidate has not been pruned, a final solve command is run. This will either pass, meaning the candidate is a correct solution, or return a counterexample. In the former case the candidate is simply returned, in the latter case the counterexample is added to the queue with a priority of one.

To improve efficiency, the implementation has some slight modifications to Algorithms 4 and 5. The last counterexample used to prune is tracked and tested first. The reasoning is that candidates coming after one another will likely mutate the same locations, due to the way candidate generation is implemented. Thus, they are also more likely to be pruned by the same counterexamples.

The technique has been implemented as an extension to Alloy 6. However, it does not make any changes to the original core source code files. Instead, all the functionality is added in new packages, which use the public methods from the original source code. This is in contrast to BeAFix, which does change the original Alloy source code. Furthermore, since it was based on a previous Alloy release it was more viable to write a new mutation-based technique from scratch then upgrading BeAFix source code to the Alloy 6 version. Since TAR does not modify the original source code, it should be easier to update it to newer Alloy versions, and, also easier to understand and modify the implementation. The source code is public and can be found on GitHub [1]. The docker image used to obtain the results in the next section can also be obtained on DockerHub [2].

4.3 Evaluation

The main goals in TAR are being able to repair temporal models and obtaining a good enough performance for students to get timely feedback. Improving the number of models able to be solved is also a secondary

[1] https://github.com/Kaixi26/TAR
[2] https://hub.docker.com/r/kaixi26/tar

Algorithm 5: Repair process in TAR.

Input: A model M, a formula ϕ representing an invalid check and a list of suspicious locations $locs$.
Output: A candidate that turns the check valid or $\perp$.

```
cexs ← []
cands ← candidates(locs)
while cands.hasNext() do
    cand ← cands.next()
    if cand.length() > MaxDepth then return ⊥
    M' ← M.apply(cand)
    valid ← true
    cexs' ← cexs.clone()
    while valid ∧ cexs' ≠ ∅ do
        cex ← cexs'.pullHighest()
        valid ← cex.evaluate(M', φ)
        if ≠ valid then cexs.incrementPriority(cex)
    end
    if valid then
        cex ← solve(M', phi)
        if cex ≠ ⊥ then
            return cand
        else
            cexs.pushPriority(cex, 1)
        end
end
return ⊥
```

goal. However, the mutators implemented in previous techniques seem to already cover most of the mutators that would be useful. Adding mutators that would only produce a fix on a few occasions can conflict with the main goal of performance. In this section, the performance of the proposed technique is evaluated, with the intent to answer the following research questions.

RQ1 What is the performance of mutation-based repair with counterexample-based pruning for temporal Alloy 6 specifications?

RQ2 How does its performance compare with that of existing automatic repair techniques for static Alloy specifications?

RQ3 What is the actual impact of counterexample-based pruning?

One teaching team has been using alloy4fun in classes and shares the students submissions after anonymization [25, 24]. The data from these challenges is the source for these benchmarks. Thus, to answer RQ1, TAR was executed for all erroneous submissions to challenges with temporal aspects in their models. This amounts to 2 models (`TrashLTL` and `Trains`), composed of 38 challenges and a total of

3671 submissions. To answer RQ2, TAR was executed, along with BeAFix and ARepair, to a subset of the submissions, the same used by the researchers BeAFix in their benchmarks. This amounts to 6 models (`TrashRL`, `ClassroomRL`, `CV`, `Graphs`, `LTS` and `Production`), comprised of 48 challenges and a total of 1935 submissions. Note that the total cases tested for these benchmarks do not match Figure 18, which shows 2209 submissions. The reason for this is that the missing submissions were marked with two faulty locations. This does not make a lot of sense in the context of Alloy4Fun challenges, which are tested one at a time. Moreover, although TAR can be used with multiple suspicious paragraphs, this was not a focus in the development, since it does not apply to the use case. To answer RQ3, all the executions of TAR were also run with counterexample-based pruning disabled.

The challenges follow the same shape seen in Listing 1. To adapt these challenges for usage in benchmarking TAR, a script was developed to extract the student predicate and put it in a new separate predicate. The file for repair will look similar to the one below. This is the file for the submission with id `Sf4uc8gXRxwtv6pZ2`, `main` refers to the full source code of the challenge. All the files for benchmarking within the same challenge share the same main predicate, `__repair` indicates which predicate is suspicious and able to be mutated, the `__repair` check is the oracle. When it passes, TAR knows it has a correct candidate.

```
open main
pred idSf4uc8gXRxwtv6pZ2_prop4 {

  always some f:File | eventually f in Trash
}
pred __repair { idSf4uc8gXRxwtv6pZ2_prop4 }
check __repair { idSf4uc8gXRxwtv6pZ2_prop4 <=> prop4o }
```

The benchmarks in Figure 19, performed by the BeAFix team, shows a huge discrepancy between the results of ARepair and BeAFix. Since Figure 18 did not show as big of an improvement, it seemed that the results were in big part because of the lack of quality in the unit tests. The unit tests from the benchmarks with ARepair models were manually selected, while the ones from Alloy4Fun were generated automatically using AUnit. Since there is a huge amount of challenges in the Alloy4Fun benchmark, it is not feasible to write tests manually for all of them. Not only that, these tests could end up being biased.

However, in an attempt to provide a fairer comparison, since TAR stores the counterexamples, and since the expected solution is known, the expected value of the instance can also easily be found by evaluating the oracle. With this insight, new unit tests were generated by running TAR for all the submissions of a challenge, and then merging and adding the occurrences of all the counterexamples for that challenge. Note that, to prevent models testing a lot of candidates from weighting too much, the individual occurrences of a counterexample do not matter. It only matters if the counterexample appeared or not during the repair process.

		ARepair (25 Tests)			BeAFix			TAR		
Exercise	Cases	Fixed (%)	TO	Failed	Fixed (%)	TO	Failed	Fixed (%)	TO	Failed
`Classroom`	999	102 (10%)	246	651	311 (31%)	578	110	408 (41%)	52	539
`CV`	137	26 (19%)	6	105	77 (56%)	44	16	85 (62%)	1	51
`Graphs`	283	181 (64%)	0	102	220 (78%)	28	35	240 (85%)	4	39
`LTS`	249	20 (8%)	5	224	35 (14%)	144	70	33 (13%)	5	211
`Production`	61	18 (30%)	1	42	47 (77%)	10	4	50 (82%)	0	11
`Trash`	206	89 (43%)	6	111	182 (88%)	14	10	193 (94%)	0	13
total (static)	1935	436 (22%)	264	1235	872 (45%)	818	245	1009 (52%)	62	864
`TrashLTL`	2890	-	-	-	-	-	-	1832 (63%)	116	942
`Trains`	781	-	-	-	-	-	-	213 (27%)	47	521
total (temporal)	3671	-	-	-	-	-	-	2045 (56%)	163	1463

Table 5: Performance of the 3 techniques under a 1-minute threshold and maximum depth 3.

For all the benchmarks that will be shown below, the timeout used was one minute. The reason for this is that we care most about repairs in lower time thresholds and the granularity for the timeout in BeAFix is by the minute. All the raw data, as well as the scripts to generate the graphs shown, can be checked in the same URL as the tool itself. [3] All tests were run on a Linux-5.15 machine with docker version 20.10, an Intel Core i5 4460 processor and 8 gigabytes of RAM.

RQ1 The data in Table 5 shows the results for the temporal benchmarks, more specifically, lines `trashLTL`, `Trains` and `total (temporal)`. Figure 22 shows the same results for the temporal benchmarks, but highlighting the number of models fixed within a certain time frame and with different parameters. The data shows that TAR is viable for Alloy 6 repair. It can repair about 35% of the specifications within 2 seconds, and by 1 minute it can repair 56%, results that even surpass those for non-temporal Alloy repair, as we will shortly see. Increasing the depth to 3 does not seem to increase significantly the performance of the approach, and with depth 2 the results stagnated at 45% by 10 seconds. As shown in Table 5, of the 46% challenges that failed to be fixed under 1 minute, 10% were due to time-out while the remainder failed due to exhausting the search space.

[3] https://github.com/Kaixi26/TAR

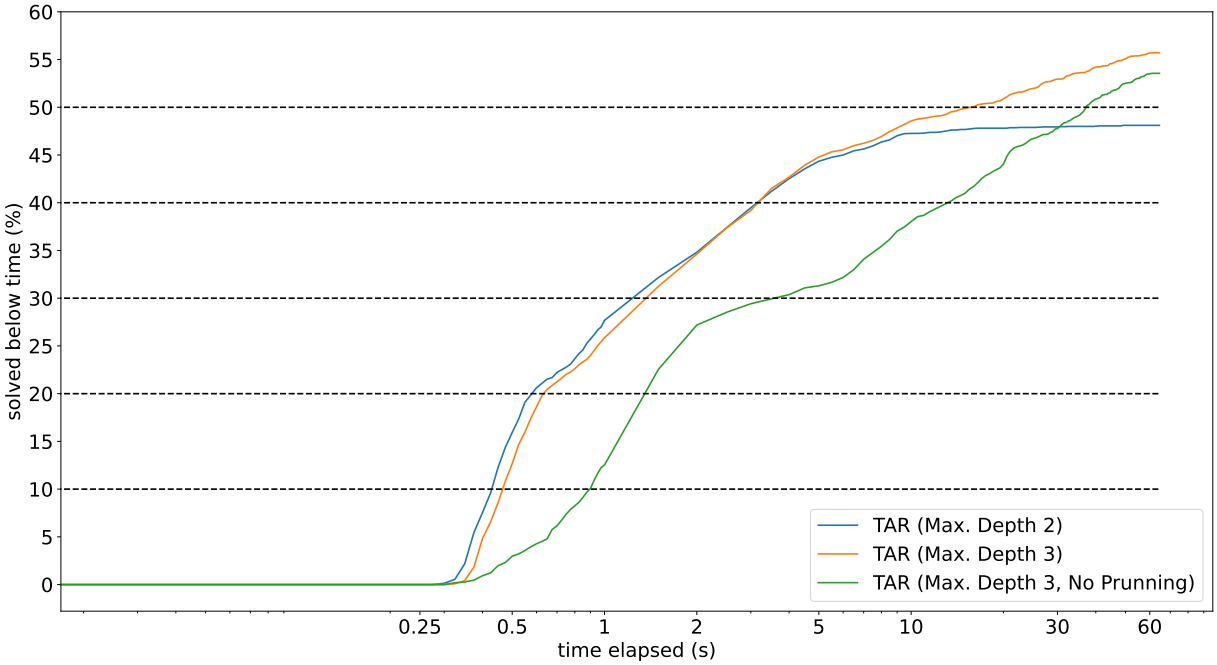


Figure 22: Percentage of specification challenges repaired by the proposed approach under a certain time threshold, for different search depth levels and with and without pruning.

RQ2 Table 5 shows the results of the static benchmarks. Figure 23 shows the same results for the static benchmarks, but highlights the number of models fixed within a certain time frame and with different parameters. The data shows that TAR consistently outperforms the other techniques, particularly in the lower time frames. At 1 second, TAR was able to fix around 35% of the models, while BeAFix was only able to fix around 20%. By the 1 minute threshold, the difference is reduced, with TAR able to fix around 52% of the models compared to the 45% models fixed by BeAFix. However, 1 minute is considered to be already too long for a student to wait for automatic feedback. Figure 24 also further demonstrates this point. It shows the overlap in solved cases at the 1-second threshold, where TAR has fixed 306 cases that BeAFix was not able to. BeAFix only has 9 cases that it has solved but TAR has not, 6 of these are just cases where BeAFix was able to repair faster than TAR; and the 3 remaining are due to differences in mutators, namely, the introduction of a join operation changing the arity of the expression, which is not always supported in TAR. A case could be made that BeAFix still has a lot more timed-out cases, so it might have been able to get more fixed cases given enough time. However, in the benchmarks made by the authors of BeAFix, with a timeout of 1 hour, the number of fixed cases was 50.7%, still below the results obtained by TAR. Similarly to the temporal benchmarks, depth 3 does not seem to increase significantly the performance of the approach compared to depth 2. The unit tests obtained for ARepair (the best 25 counterexamples were selected for the benchmark) also seem to show better results. It was able to more than double the results obtained by the authors of BeAFix, however, it seems even in a near best case scenario, the technique from ARepair cannot compete with a mutation-based approach.

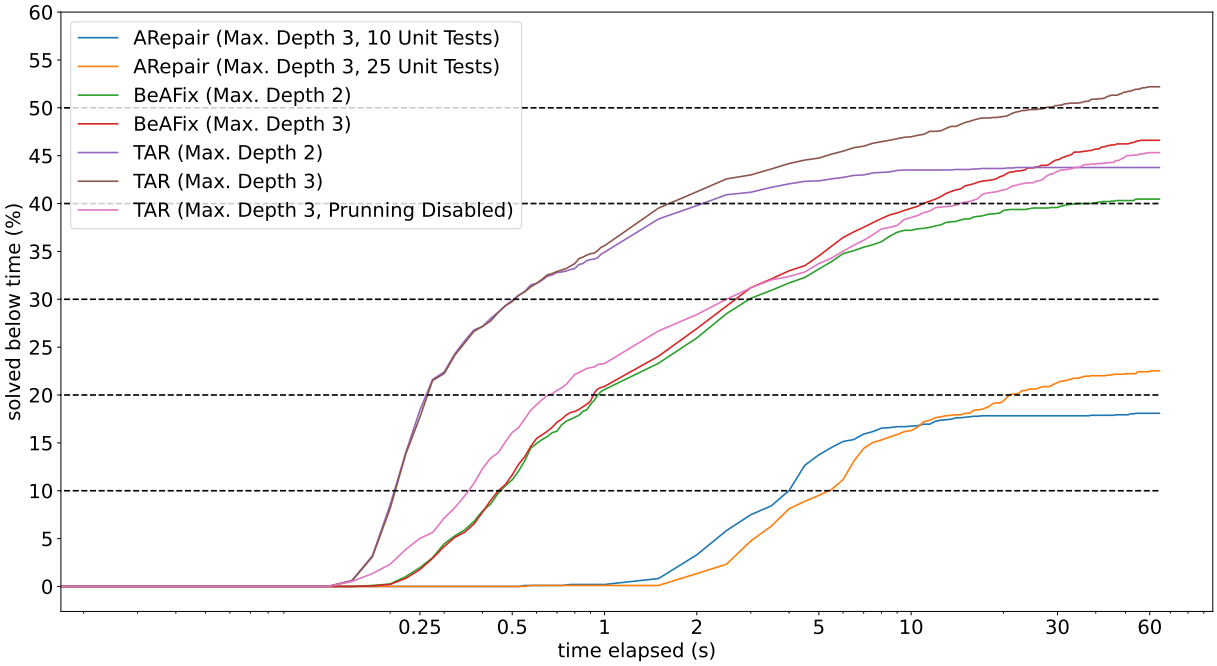


Figure 23: Percentage of submissions to static challenges correctly repaired by the proposed approach under a certain time threshold, for different search depth levels and with and without pruning.

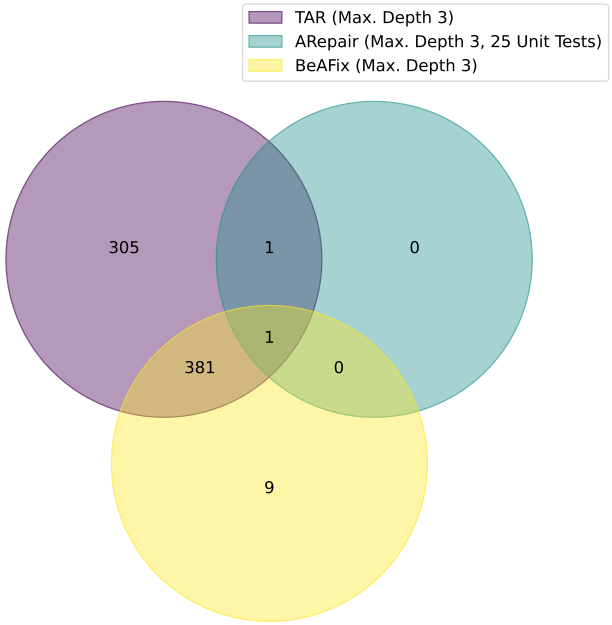


Figure 24: Classification of submissions to static challenges according to which tool was able to effectively repair under 1 second.

RQ3 Looking at the performance of TAR with pruning disabled shown in Figures 23 and 22, it is clear counterexample-based pruning has a very positive impact on performance. This seems especially true for lower time frames, for the higher time frames the differences seem to be lower. The reason for this might be because of the nature of the candidate generation. Generally, the bigger the depth, the bigger the number of candidates generated for that depth, and the difficulty of finding a solution at that depth. The benchmarks presented previously show clearly the efficacy of using counterexamples to prune candidates. Additionally, the average amount of generated counterexamples is low. Recall the number of counterexamples correlates with the number of calls to the solver. For static benchmarks an average of 5.4 counterexamples are generated, this value is around 6.4 for the temporal benchmarks. This small amount of counterexamples ends up being able to prune an impressive amount of candidates. This number is, on average, around 100000 for the static benchmarks, 76% of which are pruned by the same counterexample; and 160000 for the temporal benchmarks, 67% of which are pruned by the same counterexample.

5

Hint Generation in Alloy4Fun

This chapter will provide an overview of the online platform Alloy4Fun, which has been used by tutors at the universities of Minho and Porto for several years. First, there will be an overview of how the platform is usually used. This is followed by an explanation of the implementation. Then, a discussion on some of the possible ways to integrate automatic repair to help students with learning; the way this ended up being implemented; the improvements to the platform; and how the repairs were transformed into hints.

5.1 Alloy4Fun Usage Overview

When a user enters the website in the root path, they are greeted with an empty editor where Alloy models can be written, as seen in Figure 25.

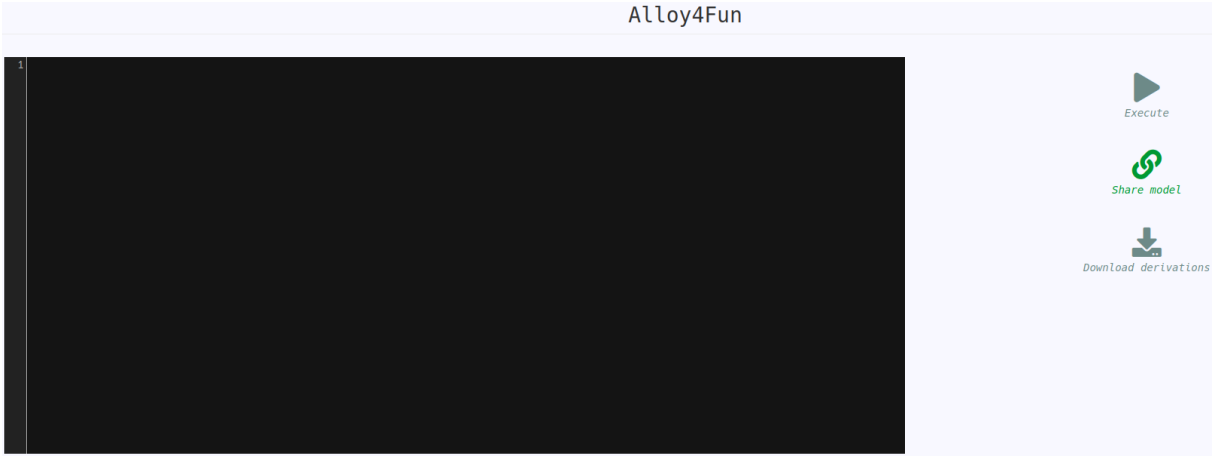


Figure 25: Root path of Alloy4Fun.

After the user writes either a `run` or a `check` command, the `Execute` button becomes available, as can be seen in Figure 26. This button allows the user to execute the analysis. This will show the witness instances in the case of a `run` (or show a warning if no instances are found); or, will show the counterexamples in the case of a `check` (or show a message telling the user the predicate might be consistent).

The `Share Model` button is the feature that makes Alloy4Fun so useful in the educational context. After the user writes the model, they can share it, this will create a fixed URL that can be accessed later, and the editor will start with the shared model instead of being empty. This feature also can share models with some of the runs, predicates, functions or checks hidden. The way it works is by writing a tag `//SECRET` before the block to be hidden. When secret tags are present in the model, the share button instead creates two URLs: one for the public model, with all the secret tags hidden; and one private model, with the original model. This can be seen in Figure 26. After the model is shared Alloy4Fun provides two links, and Figure 27 shows what a user opening the public link would be greeted with. Notice also that the execute icon is different, to indicate that the command being run is hidden. This feature makes it so the checks and oracles can be written by the instructor. It can then be hidden from the students, leaving only, for example, an explanation of the exercise and an empty predicate.

Figure 26: Sharing a model in Alloy4Fun.

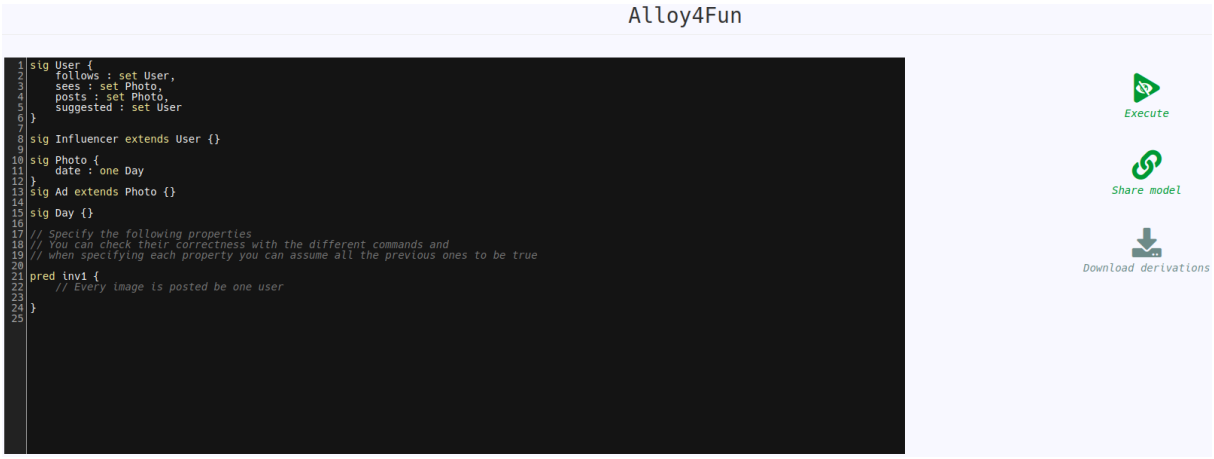


Figure 27: Example of a shared model in Alloy4Fun, with secret paragraphs.

5.2 Implementation

Alloy4fun is a web application composed of three main services: a main server, taking care of the frontend and backend of the application; an API server taking care of tasks related to Alloy; and a mongodb server to store data. Figure 28 shows how the user interacts with the application and how the servers interact with each other.

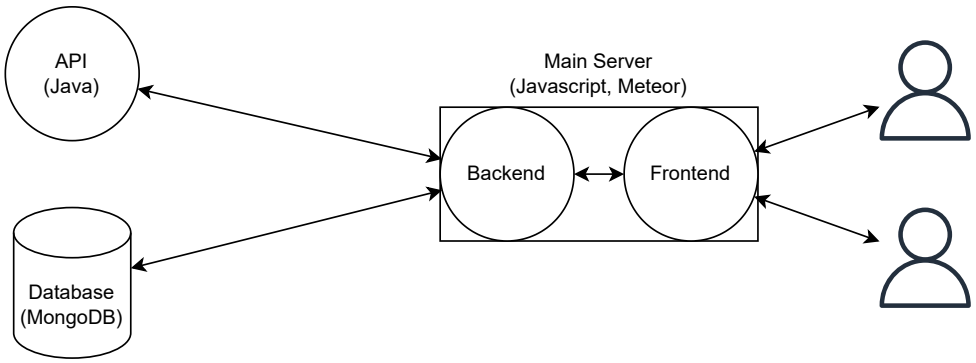


Figure 28: Architecture of Alloy4Fun.

The main server is written in javascript, with meteor, a full-stack framework for web applications. Requests from the user are always sent to this server, which then may query one of the other servers to obtain the result.

The API server is written in java, since this makes it simple to use the Alloy code as a library. The main purpose of this server is to receive Alloy code, run the commands wanted, and return the results from these commands. When the server receives such a request, it creates a new session and stores information about the code parsed and command ran. This makes it possible to later ask for more instances or counterexamples.

Alloy4Fun stores in its database some of the information relating to interactions with the tool. These include models, which are created when the share or execute features are used. They contain the source code of the model, and may also contain extra information like: the model it was derived from; the command executed; and its result. The database also stores links, which simply reference a model and have a flag, indicating if the link is a private one or not (so it can hide the secrets in case it is). Navigation information is also stored, this indicates when a user used the buttons to show the next or previous instances. Finally, themes are also stored, so they can be recovered for shared instances.

5.3 Automatic Hints

The final objective of this thesis was to use the developed automatic repair tool and integrate it into Alloy4Fun in a cohesive way, such that it can help students in the learning process. Several behaviors were discussed to reach the final one. The main objective is having the hints be helpful enough, but, at

the same time, not giving the solution, so the students still have to reach it themselves. The source code for the development can be seen on github [1].

One of the ideas was to use the counterexamples obtained in the repair process. However, due to a lack of evidence that these counterexamples would indeed be more helpful to students; and, since it is likely that the first counterexample shown on a check will likely also be at the top of counterexamples returned from the repair, this idea was ultimately discarded. Another idea was to simply indicate how distant a student is from a correct solution. This was discarded since it doesn't give information about how to get closer to a correct solution, and thus might lead to a frustrating experience.

The idea that ended up being implemented is the generation of hints, telling the student where the code needs to be modified and a general tip about the modification that needs to be made. This still forces students to think about the correct answer, just guiding their attention, and, if no hint can be found, tells them that the attempt might not be close to a correct answer. To implement the hint generation, the following features had to be taken into consideration:

- Conversion of candidates into hints;
- Addition of an endpoint in the API server for hint requests;
- Mechanism to be able to identify which predicate is suspicious;
- Implement the interface for requesting and showing hints;
- Storing metadata about hint requests in the database.

The first step in the implementation is to find a way to generate hints from the candidates. The most obvious way to do this is to implement it directly into TAR. This is because it already knows the type of all the mutators in a repair candidate. It also knows the positions of the original expressions for these mutators. Because of this, a hint method was added to the mutator class and a simple text was given to every mutator. Table 6 shows the hints for each of the mutators classes present.

The next step is to integrate TAR and hint generation into the API server. Since it is written in java, the classes from the compiled file of TAR can simply be imported and used. A new endpoint was created (`/getHint`), it expects to receive JSON in the request with 3 arguments: the model to be repaired; the index of the command to act as the oracle; the suspicious paragraphs for the repair. With this data, it creates a temporary file, dumps the model into it and parses it. With the parsed model, it finds the oracle command and the suspicious predicates. After this, it attempts the repair. The timeout and maximum depth are defined as global variables. Both are set at low values, to lower the time spent waiting by the students and the resources used. However, these values can be changed very simply. Finally, the reply can be built, which will contain metadata about the repair process, namely: the time elapsed in the repair process; if the model was repaired or not; the hints, and the locations they refer to; the resulting fixed expression.

[1] https://github.com/Kaixi26/Alloy4Fun

Mutator Class(es)	Hint
BinaryToUnary	Transform into a unary expression.
RemoveBinary ReplaceBinary	Binary operator has to be changed or removed.
InsertUnary InsertPrime	Insert an operator.
QtToUnary ReplaceQt	Quantifier has to be changed.
ExtendOrReduce ReplaceRelation	A different relation is required.
InsertJoinMutator RelationToBinary RelationToUnary	Add a binary or unary operator.
RemoveUnary ReplaceUnary	Unary operator has to be changed or removed.

Table 6: Hints for each of the mutator classes.

The third step is to have a way to know the predicate being modified by the student, so it can be marked as a suspicious location. This could have been done automatically, for example, by checking if the body of a check is in the form `A[] <=> B[]` and the scope of either `A` or `B` is empty, being very likely this is the suspicious predicate. However, this would have a lot of edge cases (for example, checks are sometimes in the form `C => (A[] <=> B[])`), likely resulting in bugs, and would lock the platform into this way of doing exercises. Our approach to doing this was to continue with the already established feature of using tags. Similarly to the `//SECRET` tag, a `//REPAIR` tag has been added to mark the suspicious predicates for the following check. This is tedious but also simpler and more powerful. However, current exercises have to either be recreated or the shared models need to be altered directly in the database. The implementation is simple. The tag is found using a regular expression, then, the name of the suspicious predicate, which comes after the tag is parsed. After this, the code searches for the first check command appearing after the tag and saves the information in a map, connecting commands to their corresponding suspicious expressions. The repair tag works well with the secret tag, when the repair tag is put between a secret tag and a paragraph it is also hidden, however, the repair tag can also be used by itself.

For the request of hints, the main server was altered. A new button was added to the interface, which is only available after execution is performed. The idea of limiting the usage of hints was discussed, for example making it available only after solving certain smaller challenges, or after a certain number of attempts were made, but these implementations seem needlessly complex and bad for user experience. When clicked, it simply calls the backend of the main server which then performs a request to the API server.

Alloy4Fun already has a system to handle messages, like errors or warnings, to be shown to the user, as can be seen in Figure 29. Since the characteristics of the hints are very similar to errors and warnings, it makes sense that all these are handled the same. However, the problem with the current system is

that it is very intrusive. Since messages are presented in a popup they block a lot of the editor, which makes it harder to verify the code. To remove the popup, it is necessary to click either the OK button or outside the popup window. This leads to cases where the user removes the popup so they can have better visibility of the code, and then losing the context of the message, having to execute the request again. Furthermore, after execution, the button is locked until the model is changed. So, if the user wants to check the error again, they will have to, for example, add and remove a character from the editor so they can perform execution again. To improve the user experience, the proposal is to make it more similar to the behavior of the Alloy editor. In the Alloy editor, the position of errors or warnings is highlighted until the code is modified, and the error or warning message is displayed continuously. To achieve this change, modifications were made to the main server. The editor marker was already supported by the editor library and the markers are only cleared on change events. The code for log messages, which was mostly used to notify the user if a check was likely correct (no counterexamples found) or not, was extended to allow not only a value to be passed but also an array. The result of the changes can be seen in Figure 30, which shows the new error messages; and Figure 31, which shows how the hints are presented.

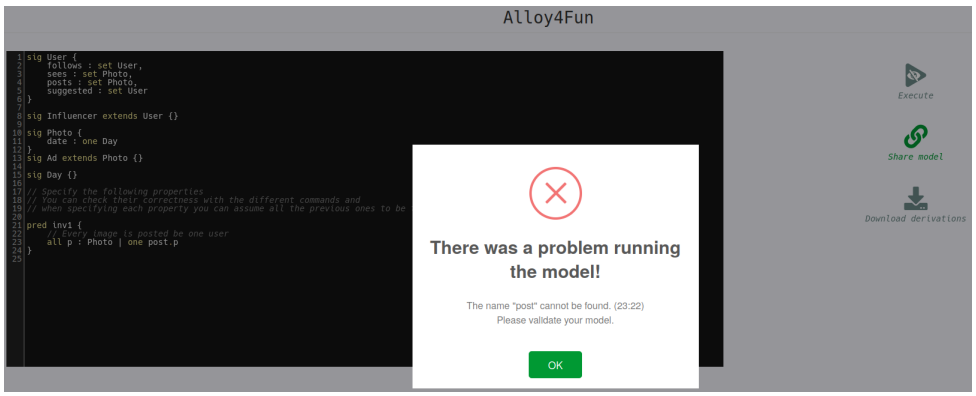


Figure 29: Errors in the current Alloy4Fun version.

A new collection was also added to the database. It contains information about the execution of hint commands, which can then be analyzed to test, for example, the effectiveness of the hints, by measuring how long the student takes to reach the correct answer and also to find possible bugs. The information stored is: the time elapsed; if the model was repaired or not; the hints given; the obtained repair; the model used for the repair; and the timestamp for when the information was stored.

Figure 30: Errors in the updated Alloy4Fun version.

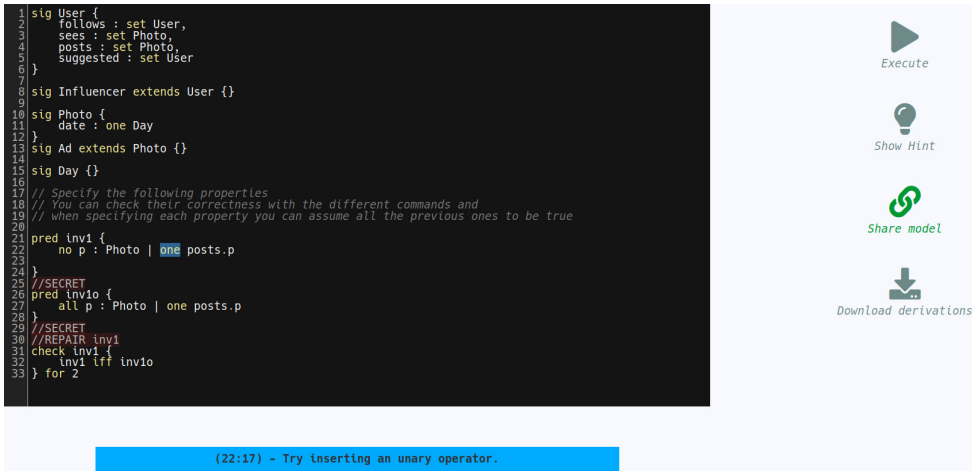


Figure 31: Hints in the extended Alloy4Fun version.

6

Conclusion

The main goals of this thesis were the development of a tool and technique capable of repairing Alloy 6 temporal specifications and its integration into the Alloy4Fun platform. We believe these goals were achieved successfully. This thesis presented a mutation-based technique for the automatic repair of Alloy 6 specifications, being the first to consider its full temporal first-order logic. The tool developed has been able to outperform existing Alloy repair tools. Both in time and also in the total amount of correct repairs. It also presents the hint generation system implemented to guide students to correct solutions, without revealing them right away.

Towards the end of the development of this thesis, a new technique was proposed named ATR [44] (Alloy Template-Based Repair). The technique presented in this work was developed independently from it. ATR takes inspiration from both ARepair and BeAFix, employing the use of both mutations and synthesizations. It uses FLACK as the means to find the suspicious locations, and checks as the oracle for correctness. The technique starts the repair process by trying a few simple mutators. This will either generate a fix or some counterexamples, and ATR also uses previous counterexamples to try to skip some mutators, similar to TAR. After this process, for each counterexample, ATR tries to obtain a satisfying instance as close to it as possible. To achieve this, it uses a process similar as the one described in FLACK, handling the problem with a PMAXSAT solver. ATR then goes through a synthesis process. It defines repair templates, which are rules for how expressions can be synthesized, and the generated expressions are then pruned by checking the differences in pairs of counterexamples and satisfying instances. The benchmarks made for ATR by its researchers show impressive results, being able to repair around 66% of the incorrect submissions, for the same dataset shown in Table 5. However, this value was obtained for a timeout of 1 hour. We replicated the tests with a timeout of 1 minute, and the value obtained was around 17% of the submissions being repaired. This means ATR is not suitable for the use case proposed in this thesis, and as such, TAR is still a very valuable contribution. ATR is also based on Alloy 4, so it does not provide support for temporal models.

Automatic repair research, especially in the case of specification languages, is still in its infancy. There are likely many improvements to come moving forward. After the development of this thesis, the capabilities of mutation-based repair seem to be near its limits. The most relevant mutators have already

been taken into consideration, so adding more mutators would likely not improve the benchmarks by a considerable value. However, mutation-based repair has shown to be incredibly useful within the lower time bounds. More specifically, for lower depths of mutators, and this might still be improved with new pruning techniques. Due to this, future techniques will still likely benefit from attempting mutation-based repair within a constrained depth of 1 or 2. Currently, the 4 techniques are very isolated, this is of course expected since all have very different implementations. However, it would be interesting to have the various techniques consolidated in a single codebase. This would allow the isolation and evaluation of each component. For example, knowing if the counterexample-based pruning of TAR could have any significance in improving the repair process of ATR.

The hint generation system has not yet been proven effective. It is important to find ways to verify if it is, in fact, a valuable addition to Alloy4Fun or if it doesn't help students with learning. This can be done in several ways: testing students that have not yet been introduced to Alloy, and comparing how fast they can solve and comprehend the exercises; testing students that have been introduced to Alloy, and verifying if they can correctly understand the hint messages and diagnose the issue; analyzing the data stored, for example, checking if a student asking for a hint to a challenge can reach solutions to the following challenges faster.

In this thesis, the proposed way to generate hints was through a process of automatic repair. This is a very versatile technique since it can generate hints for various types of exercises, only requiring the model and the oracle. However, there are other ways to obtain hints, specifically in the case of Alloy4Fun. Not only a possible final solution is already known (the oracle), but it is also likely that other solutions and candidates match those already stored at its database, obtained from previous submissions from students. In addition to this, the dataset also contains information about the paths (incorrect submissions) traversed by a student to reach a solution. This might help to divide hints into small steps. With this information, it is possible to calculate a path that is likely to lead the student to a more correct solution. A technique has already been explored before for hint generation with code [27].

Bibliography

[1] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. "Spectrum-based multiple fault localization". In: *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2009, pp. 88–99 (cit. on p. 26).

[2] R. Abreu et al. "A practical evaluation of spectrum-based fault localization". In: *Journal of Systems and Software* 82.11 (2009), pp. 1780–1792 (cit. on p. 26).

[3] M. Alhanahnah, C. Stevens, and H. Bagheri. "Scalable analysis of interaction threats in IoT systems". In: *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 2020, pp. 272–285 (cit. on p. 2).

[4] *Alloy 6 major version*. `http://web.archive.org/web/20211101221007/https://alloytools.org/alloy6.html` (cit. on p. 2).

[5] *AlloyFL*. `https://web.archive.org/web/20211204171243/https://alloyfl.github.io/` (cit. on p. 23).

[6] H. Bagheri et al. "A formal approach for detection of security flaws in the android permission system". In: *Formal Aspects of Computing* 30.5 (2018), pp. 525–544 (cit. on p. 2).

[7] H. Bagheri et al. "Practical, formal synthesis and automatic enforcement of security policies for android". In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2016, pp. 514–525 (cit. on p. 2).

[8] S. G. Brida et al. "Bounded exhaustive search of alloy specification repairs". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 1135–1147 (cit. on pp. 22, 34, 36).

[9] J. Brunel et al. "The electrum analyzer: model checking relational first-order temporal specifications". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 884–887 (cit. on pp. 2, 4).

[10] J. Cerqueira, A. Cunha, and N. Macedo. “Timely Specification Repair for Alloy 6”. In: *International Conference on Software Engineering and Formal Methods*. Springer. 2022, pp. 288–303 (cit. on p. 3).

[11] *Courses on Alloy*. http://alloytools.org/citations/courses.html (cit. on p. 2).

[12] A. Cunha and N. Macedo. “Validating the hybrid ERTMS/ETCS level 3 concept with Electrum”. In: *International Journal on Software Tools for Technology Transfer* 22.3 (2020), pp. 281–296 (cit. on p. 4).

[13] A. Cunha and N. Macedo. “Validating the hybrid ERTMS/ETCS level 3 concept with electrum”. In: *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer. 2018, pp. 307–321 (cit. on p. 2).

[14] A. Cunha, N. Macedo, and T. Guimaraes. “Target oriented relational model finding”. In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2014, pp. 17–31 (cit. on p. 30).

[15] J. Hua et al. “Towards practical program repair with on-demand candidate generation”. In: *Proceedings of the 40th international conference on software engineering*. 2018, pp. 12–23 (cit. on p. 29).

[16] D. Jackson. “Automating first-order relational logic”. In: *ACM SIGSOFT Software Engineering Notes vol. 25 iss. 6* 25 (6 2000). doi: 10.1145/357474.355063 (cit. on p. 17).

[17] D. Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012 (cit. on pp. 2, 4, 17).

[18] J. A. Jones and M. J. Harrold. “Empirical evaluation of the tarantula automatic fault-localization technique”. In: *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 2005, pp. 273–282 (cit. on p. 26).

[19] E. Kang and D. Jackson. “Formal modeling and analysis of a flash filesystem in Alloy”. In: *International Conference on Abstract State Machines, B and Z*. Springer. 2008, pp. 294–308 (cit. on p. 2).

[20] S. A. Khalek et al. “Testera: A tool for testing java programs using alloy specifications”. In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE. 2011, pp. 608–611 (cit. on p. 2).

[21] T. A. Khan, A. Sullivan, and K. Wang. “AlloyFL: a fault localization framework for Alloy”. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021, pp. 1535–1539 (cit. on pp. 23, 26, 27).

[22] S. Krishnamurthi and T. Nelson. “The human in formal methods”. In: *International Symposium on Formal Methods*. Springer. 2019, pp. 3–10 (cit. on p. 1).

[23] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. url: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf.

[24] N. Macedo, A. Cunha, and A. C. R. Paiva. *Alloy4Fun Dataset for 2020/21*. 2021. doi: 10.5281/zenodo.4676413 (cit. on p. 46).

[25] N. Macedo et al. "Experiences on teaching alloy with an automated assessment platform". In: *Science of Computer Programming* (2021), p. 102690 (cit. on pp. 2, 46).

[26] N. Macedo et al. "Lightweight specification and analysis of dynamic systems with rich configurations". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016, pp. 373–383 (cit. on p. 17).

[27] J. McBroom, I. Koprinska, and K. Yacef. "A survey of automated programming hint generation: The hints framework". In: *ACM Computing Surveys (CSUR)* 54.8 (2021), pp. 1–27 (cit. on p. 60).

[28] M. Monperrus. "Automatic software repair: a bibliography". In: *ACM Computing Surveys (CSUR)* 51.1 (2018), pp. 1–24 (cit. on pp. 1, 22).

[29] L. Naish, H. J. Lee, and K. Ramamohanarao. "A model for spectra-based software diagnosis". In: *ACM Transactions on software engineering and methodology (TOSEM)* 20.3 (2011), pp. 1–32 (cit. on p. 26).

[30] P. M. Phothilimthana and S. Sridhara. "High-coverage hint generation for massive courses: Do automated hints help CS1 students?" In: *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. 2017, pp. 182–187 (cit. on p. 2).

[31] A. Sullivan, K. Wang, and S. Khurshid. "Aunit: A test automation tool for alloy". In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2018, pp. 398–403 (cit. on p. 23).

[32] A. Sullivan et al. "Automated test generation and mutation testing for Alloy". In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2017, pp. 264–275 (cit. on p. 29).

[33] A. Svendsen et al. "Formalizing train control language: automating analysis of train stations". In: *12th Int Conf on Computers in Railways (COMPRAIL 2010)*. 2010, pp. 245–256 (cit. on p. 4).

[34] K. Wang, A. Sullivan, and S. Khurshid. "Arepair: a repair framework for alloy". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE. 2019, pp. 103–106 (cit. on pp. 22, 28, 29).

[35] K. Wang, A. Sullivan, and S. Khurshid. "Automated model repair for alloy". In: *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2018, pp. 577–588 (cit. on pp. 22, 28).

[36] K. Wang, A. Sullivan, and S. Khurshid. “MuAlloy: a mutation testing framework for alloy”. In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE. 2018, pp. 29–32 (cit. on pp. 28, 29).

[37] K. Wang et al. “Asketch: A sketching framework for alloy”. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018, pp. 916–919 (cit. on p. 29).

[38] K. Wang et al. “Fault localization for declarative models in Alloy”. In: *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2020, pp. 391–402 (cit. on p. 23).

[39] K. Wang et al. “Solver-based sketching of alloy models using test valuations”. In: *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer. 2018, pp. 121–136 (cit. on p. 29).

[40] K. Wang et al. “Systematic generation of non-equivalent expressions for relational algebra”. In: *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer. 2018, pp. 105–120 (cit. on p. 29).

[41] W. E. Wong and V. Debroy. “A survey of software fault localization”. In: *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45* 9 (2009) (cit. on p. 22).

[42] W. E. Wong et al. “The DStar method for effective software fault localization”. In: *IEEE Transactions on Reliability* 63.1 (2013), pp. 290–308 (cit. on p. 26).

[43] P. Zave. “Reasoning about identifier spaces: How to make chord correct”. In: *IEEE Transactions on Software Engineering* 43.12 (2017), pp. 1144–1156 (cit. on p. 2).

[44] G. Zheng et al. “ATR: template-based repair for Alloy specifications”. In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2022, pp. 666–677 (cit. on p. 59).

[45] G. Zheng et al. “FLACK: Counterexample-guided fault localization for alloy models”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 637–648 (cit. on pp. 30–34).