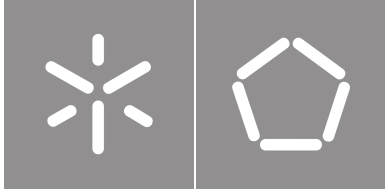
Universidade do Minho
School of Engineering

Henrique Gabriel dos Santos Neto

Mining hints for fixing formal specifications

October, 2023



Universidade do Minho
School of Engineering

Henrique Gabriel dos Santos Neto

Mining hints for fixing formal specifications

Master Thesis
Master in Informatics Engineering

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

October, 2023

Acknowledgements

First and foremost, I would like to extend my heartfelt thanks to my advisors, Professor Manuel Alcino Cunha and Professor Nuno Moreira Macedo. Their expertise, patience, and unwavering support have been instrumental in shaping this work.

Special thanks are also due to INESC TEC - Institute for Systems and Computer Engineering, Technology and Science, for granting me the scholarship that enabled me to pursue my studies. The financial support has been crucial in ensuring the successful completion of this thesis.

Lastly, I would like to extend my sincere thanks to my family, friends and colleagues who have provided moral support and shared their valuable insights.

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.

Braga, January 4, 2024

(Place) (Date)

Assinado por: **HENRIQUE GABRIEL DOS SANTOS NETO**
Num. de Identificação: 30302655
Data: 2024.01.04 18:11:46+00'00'

(Henrique Gabriel dos Santos Neto)

“Simplicity is prerequisite for reliability.”
(Edsger Dijkstra)

Resumo

Mineração de sugestões para corrigir especificações formais

O crescimento da complexidade de aplicações informáticas tornou falhas e erros de software uma inevitabilidade. Para ajudar a garantir que uma aplicação funciona como previsto, profissionais recorrem a modelos de software para detetar e corrigir problemas nas fases iniciais de desenvolvimento. Especificações formais são modelos de software que permitem a desenvolvedores especificar rigorosamente estruturas e comportamentos de software. Infelizmente, a sua complexidade inerente também pode impor problemas nos principiantes que as tentam aprender. Uma maneira possível de abordar este problema seria o emprego de práticas de reparação de especificações e geração automática de sugestões para ajudar os alunos a corrigir tentativas erradas.

Alloy4Fun é uma plataforma online para a aprendizagem de Alloy, uma linguagem de especificação formal com capacidades de analise automática. Alloy4Fun permite a instrutores criar e partilhar desafios de especificação formal com avaliação automática. Recentemente, uma técnica de geração automática de sugestões for desenvolvida para esta plataforma, mas provou ser insatisfatória devido ao seu fraco desempenho. O objetivo desta tese foi explorar outras técnicas para geração de sugestões, nomeadamente técnicas de geração de sugestões baseadas em dados, que poderiam usar o conjunto de dados publico de submissões históricas de estudantes do Aloy4Fun para fornecer dicas de forma mais eficiente.

O principal resultado desta tese, SpecAssistant, é um novo sistema de geração de dicas baseado em dados para Alloy. Este extrai informação do conjunto de dados do Alloy4Fun para construir grafos de submissões, dos quais são extraídas sugestões a partir de regras personalizadas pelos desenvolvedores de cada desafio. Para avaliar o SpecAssistant, realizamos uma série de experiências quantitativas, com o objetivo de avaliar a disponibilidade e o desempenho do nosso sistema. As nossas descobertas demostram que o SpecAssistant consegue fornecer dicas para uma porção significativa de submissões, apresentado um desempenho que supera o sistema de sugestões precedente.

Palavras-chave: Alloy, Geração de Dicas Automáticas, Métodos Formais, Mineração de Dados

Abstract

Mining hints for fixing formal specifications

The increasing complexity of software applications has made software bugs and errors an inevitability. To help ensure that software functions as intended, professionals rely on software models to detect and correct faults early in the development process. Formal specifications are software models that allow developers to precisely specify software structures and behaviors. Unfortunately, their inherent complexity can also pose problems to newcomers while learning them. One possible way to address this issue could be to employ automated hint and specification repair techniques to help students fix incorrect attempts.

Alloy4Fun is an online platform for learning Alloy, a formal specification language with automated analysis features. Alloy4Fun allows educators to create and share specification challenges with automated assessment. Recently, a hint generation technique based on automated repair has been developed for this platform, but proved unsatisfactory due to its poor performance. The goal of this thesis was to explore other techniques for hint generation, namely data-driven hint generation techniques which could leverage the Alloy4Fun public data-set of past student submissions to provide hints more efficiently.

The main outcome of this thesis, SpecAssistant, is a new data-driven hint generation system for Alloy. It mines the Alloy4Fun data-set to build submission graphs, from which hints are extracted using policy rules customized by the challenge developers. To evaluate SpecAssistant we performed a series of quantitative experiments, with the goal of assessing our system's availability and performance. Our findings show that SpecAssistant can provide hints for a significant portion of invalid submissions with a performance that greatly surpasses that of the previous hint system.

Keywords: Alloy, Automated Hint Generation, Data Mining, Formal Methods

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Acronyms

APTED All Path Tree Edit Distance *(pp. 60, 62)*
AST Abstract Syntax Tree *(pp. xi, 37, 45–48, 53, 54, 58, 60–62, 76)*

BFS Breadth-First Search *(p. 62)*
BSON Binary Javascript Object Notation *(p. 66)*

CRUD Create, Read, Update, Delete *(pp. 57, 65)*

DTO Data Transfer Object *(pp. 56, 57)*

HINTS Hint Iteration by Narrow-down and Transformation Steps *(pp. 34–36)*
HTTP Hypertext Transfer Protocol *(pp. xiii, 29, 30, 55, 56)*

ITAP Intelligent Teaching Assistant for Programming *(pp. xi, 40, 41)*

JSON JavaScript Object Notation *(pp. 29, 31, 56, 63, 64)*

MDP Markov Decision Process *(p. 39)*

REST Representational State Transfer *(pp. 29, 54–57)*
RNN Recurrent Neural Network *(p. 42)*

SAT Boolean Satisfiability Problem *(pp. 46, 72)*

TAR Temporal Alloy Repair *(pp. xiii, 36, 37, 51–53, 69, 73–75)*
TED Tree Edit Distance *(pp. 49, 51, 62, 63)*

1 Introduction

In the past decades there has been a rush in software growth as the world becomes more and more digitized. Applications are also becoming increasingly more complex. As complexity increases, bugs and errors become an inevitable problem. If not properly identified and solved, these errors can cause failures in the applications, which can be problematic and result in damages that vary according to the system in question. An error on a web-page will cause an inconvenience to a user attempting to use it. An error in a hospital's equipment could cost someone's life. Assuring that an application works as expected is essential in software development and has become a very important subject in computer science.

1.1 Contextualization

The core of software development is the design of abstractions [17]. An abstraction is a model of the product itself, only a simple and pure representation of the structure and the core ideas, which are conceived before production. The quality of such abstractions is one of the major factors in the quality of the final product. A good abstraction will normally cause a natural flow in the software's development, from its design to its production, resulting in simple and optimal components that can easily be modified or expanded without the need for extensive reorganization or changes. On the other hand, bad software abstractions tend to cause major problems in the production phase, where components will become clumsy and burdensome as they are forced to accommodate for unexpected problems, and even the simplest changes become problematic resulting in enormous development costs. In this case, the final product is usually composed of extremely complex components, that require developers to master a mass of superfluous details, while either developing workarounds or accepting the frequent and inexplicable failures.

Abstractions are also important to the developers themselves, since they want simple and easy to understand models that don not compromise on factors such as robustness and scalability. With this in mind it is expected that with careful consideration of the problem and its future endeavors one could easily develop a fine product. Unfortunately, this is rarely the case due to a problem commonly referred as *wishful thinking* [30]. Developers come up with simple and robust abstractions which turn out to be incoherent and possibly inconsistent in production, resulting in the sort of nasty problems previously

mentioned. These flaws are not directly derived from the developed abstractions, but from the environment imposed by programming, where compilers and tools admit no vagueness whatsoever and tests instantly revel errors and faults.

There are several ways of combating this issue, the most interesting and effective ones rely on *formal specifications*. These abstractions are built using unambiguous notations based on mathematics, which enable automated verification and validation. Over the last few decades there have been several developments is this regard, resulting in the appearance of several advanced tools for developing formal specifications. One of these tools is Alloy [17], which is the main focus of this work.

1.2 Problem

Even with their overwhelmingly advantages in software development, formal specifications are still not used by a significant portion of developers. This is because newcomers tend to have several difficulties in understanding and adopting the mathematical techniques and the inherent level of precision and rigor. Even with the most recent state-of-the-art tools, where these concepts keep getting simpler an simpler, this problem still endures.

Alloy is a clear example of this issue. It relies on first-order logic extended with a few relational operators, and has a straightforward and minimal syntax and semantics. However a significant number of college students still found themselves confounded when using it [23]. A big part of this issue was the poor feedback of the Alloy tools, as the provided counter-examples were difficult to understand by the users. A possible and feasible way of mitigating this would be the introduction of automatic repair and hint generation techniques in these tools, to help newcomers learn the language.

1.3 Objective

Automatic program repair and hint generation are very prominent and active topics in software engineering [44, 27]. Currently, Alloy4Fun [23] is one of the few automated assessment platforms for learning formal specifications, and thanks to its support for specification challenges with autograding it has been used as learning platform for Alloy in graduate courses for a few years. Recently, this application has been the target of the development of a hint system based on automatic program repair techniques [10]. However, the developed system presents some shortcomings: besides its performance issues, it usually fails to provide hints on submissions which deviate a lot from a valid solution.

Based on the success of recent data-driven hint generation systems on educational programming platforms [36, 38], it became apparent that the existing data-set of submissions of Alloy4Fun [22] could be used to significantly upgrade its current hint generation mechanism, allowing not only to speed up the whole generation process but also to increase its availability.

The main objective of this dissertation is to explore several of the most recent and useful hint generation techniques for learning how to program [26], and develop a new technique for Alloy specifications. To guarantee the quality of experience of the user, the new technique should have some minimum performance requirements, i.e. the system should present the hints nearly instantly after reading the user's current submission, otherwise it will most likely fall into disuse. The new technique should also be data-driven, i.e. a technique which will mine existent student submissions to isolate patterns and solution tactics. The reasoning behind this selection is that, by using the historical data from a diverse group of students, we expect the resulting technique to be able to provide hints for most of the erroneous submissions of new students, since they usually fall into the same erroneous patterns. Additionally, we expect an improved performance, since upon being prompted by a student for help, the system would only need to consult the stored information to produce a hint. The information required to start and maintain this system will originate from the historical data of Alloy4Fun [22] and also the new submissions that the system to be developed will receive overtime.

1.4 Alloy Specification Assistant

The technique developed in this dissertation resulted in a tool named SpecAssistant, a hint generation system for Alloy which can be integrated within Alloy4Fun. It achieves its purpose by mining a vast repository of submissions obtained from multiple attempts at solving a few Alloy4Fun exercises. This data is organized into a submission graph, akin to a directed graph, from which hints can be computed.

After creating the submission graph, our system allows the developer of a new challenge to customize the policy used to compute the hints. This customization is accomplished by defining value ranking rules based on pre-computed attributes of the submission graph.

When SpecAssistant is presented with a user's submission it attempts to match it to the historical submission graph. If a match is successful, it retrieves a submission that would make the user get closer to a solution, based on a customizable policy. Finally, using a simple text generation and code highlighting technique, the gathered information is transformed into a hint.

We performed several benchmarks to evaluate our system, and we were able to confirm that the developed hint generation technique is capable of instantly providing hints for a significant amount of invalid submissions. As a result, our system may help students learn Alloy without the efficiency issues of the previously implemented hint system of Alloy4Fun [10]. Furthermore, there is an expectation that our system will enhance its performance over time, as the Alloy4Fun data-set of historical submissions grows.

1.5 Document Structure

This thesis has the following structure.

Chapter 2 describes version 6 of the Alloy modeling language. The chapter presents an illustrative model which is used to describe every essential concept of the language, such as its static and dynamic modeling features.

Chapter 3 presents Alloy4Fun, the target application of this project. This chapter mainly explores this application's features and architecture, alongside the structure of its database.

Chapter 4 explores the state-of-the-art in automated hint and repair systems. It begins by presenting an abstract framework which details the common concepts and processes of most hint generation systems. Afterwards, it explores and details several different hint generation techniques, discussing their features, advantages and disadvantages, and how they could be used in our system.

Chapter 5 describes the developed hint generation system, named SpecAssistant. The chapter starts by describing the fundamental principles and techniques within our system. Subsequently, we describe the implementation of these techniques in Alloy4Fun.

Chapter 6 presents the evaluation preformed on SpecAssistant. A series of data-mining validation techniques were used to answer a series of research questions, with the main purpose of quantifying our system's availability and performance.

Chapter 7 presents some conclusions and perspectives on future work.

2 Alloy

Alloy [17] is a popular formal specification language complemented by an analyzer capable of automated instance generation and validation. Just like any formal specification tool, it helps developers find issues and holes in their systems. Its biggest advantage over its competitors is the analyzer. Unlike most tools which require conventional analysis based on theorem proving (i.e. mathematical proofs), Alloy provides fully automatic analysis on its models which give instant feedback to the developers. This comes at a cost of not having a "complete" analysis, since it only examines a finite number of cases. However, thanks to its optimizations, the space of cases examined for each model can be immense, which makes this less of an issue.

This chapter offers a presentation of Alloy's essential and unique features, as well a basic understanding of model specification.

2.1 Static Modeling

2.1.1 A Basic Model

Structural (or static) models form the basis for any formal specification. These define the structure as well as the rules which the respective instances should satisfy. To demonstrate Alloy's structural modeling capabilities we have built a simple but descriptive example of a basic traffic system, composed of both a road network and the vehicles which traverse it. To begin, we will first model the roads of our network using two basic constructs: signatures and binary relations. The respective Alloy declarations are presented in Listing 2.1.

```
sig Road {
    adjacent : set Road
}
```

Listing 2.1: First model

This model declares the signature `Road` using the keyword **`sig`**, and the `adjacent` binary relation, which associates each road to a set of its adjacent neighbors. It is not a correct model though, since it allows several invalid configurations. The most problematic cases include roads that are adjacent to themselves, roads that contain unidirectional adjacencies, and finally traffic systems that do not represent a connected graph, consisting therefore of multiple scattered road groups.

2.1.2 Instance Representation

Before diving into the constraint language it is important to clarify how Alloy represents its instances. Signatures, relations and generally any type of data is represented as a set of tuples of *atoms* in Alloy. Atoms are primitive entities which have a few basic properties. First, as implied by their names, their are indivisible and cannot be broken into parts. Furthermore they are immutable, meaning their properties remain constant over time and, finally, they are uninterpreted and thus do not contain any meaning associated to them unlike other forms of data, for example, numbers.

Since they have no inhered meaning, the user must declare relations to attach to them the desire meaning. A relation is a set of tuples of the same arity and, as implied by the name, relates the atoms. The different arities of the sets describe the "shape" of value they represent. Signatures, like the `Road` we previously described, are in fact sets of atoms similar to unary relations. Binary and multi-relations describe predicates. For the case of our `adjacent` relation, a tuple would indicate that the road in the second element is adjacent to the road in first element.

It is important to comprehend this notion since most constrains revolve around manipulating these sets, for example intersecting them and composing them. For example, let us consider a signature a and two binary relations named R and S whose values are $a = \{(A), (C)\}$, $R = \{(A, X), (B, Y), (C, Z)\}$ and $S = \{(C, Z)\}$. Composing `a` with `R` (denoted in Alloy as `a.R`), would result in a unary set with the second elements of R's tuples in which the first element is present in a, i.e. the set $\{X, Z\}$. Additionally, we can see that S is contained within R (denoted in Alloy as `S in R`) since the only tuple of S (C, Z) is present in relation R. Their set intersection would be S and their set union would be R.

2.1.3 Model Constraints

To prevent erroneous configurations in our model we must restrict it with the use of constraints. Alloy's language provides a type system for constraint definition, which shares several concepts with other modeling languages but also defines new Alloy specific constructs. The language includes constructs such as quantified variables, set and predicate operators, and so forth. Most of these common constructs will not be explained in detail, with the exceptions of some of the Alloy specific constructs.

In addition, the variety of syntax options allows for a constraint to be written in many different forms. For example, specifying that a road cannot be adjacent to itself can be done with any of the expressions present on Listing 2.2.

```
all r : Road | (r->r) not in adjacent
all r : Road | r not in r.adjacent
no adjacent & iden
```

Listing 2.2: The not self adjacent constraint specified in different styles

All of these constraints are semantically equivalent and represent three possible styles of specification allowed by in Alloy's logic. The first constraint is represented with the *predicate calculus* style where the formulas consist of checking the membership of tuples in relations. The formula follows this style, by universally quantifying (`all`) a variable `r` of type `Road` and checking that the tuple `(r->r)` is not a member of the relation `adjacent`. This first-order logic style usually results in overly verbose constraints. The second constraint is specified with the *navigational expression* style, which somehow relates sets formed by *navigating* relations from specific quantified variables. This is visible in our example, where we claim that for every variable `r` of type road, `r` is not contained within the set of `r`'s adjacent roads represented by `navigating` (i.e. composing) `r` with the `adjacent` relation. The final constraint uses the *relational calculus* style, where each rule is described solely with relational expressions without quantifiers. As such, these tend to be very compact and syntactically simple, although this style can also result in extremely enigmatic constraints. Our example states that the intersection (`&`) of the relations `adjacent` and `iden` (where `iden` is the predefined identity relation which maps every atom to itself) must be empty (`no` checks for emptiness).

The remaining constraints can be specified as follows. To specify that a road must be adjacent to each of its adjacent roads we can simply write the constraint in Listing 2.3.

```
adjacent in ~adjacent
```

Listing 2.3: Bidirectional adjacent constraint

This expression dictates that the relation `adjacent` is contained within `~adjacent`, its converse relation. As such, this forces the relation to be symmetric and so, if a road is adjacent to another, then the reverse will also have to be true.

Finally, we must specify that all the roads are connected. Thanks to our previous constraint, we can assure this by saying that every road allows for a path to any location on the map.

```
all r : Road | Road in r + r.^adjacent
```

Listing 2.4: Connected roads constraint

In Listing 2.4 we specify that for every Road `r`, the entire Road set is contained within the union (`+`) of the road itself and every road reachable from it. To specify the reachable roads, we use the transitive closure (`^`) on the relation `adjacent`. The set `r.^adjacent` represents the union of the sets `r.(adjacent)`, `r.(adjacent.adjacent)`, `r.(adjacent.adjacent.adjacent)`, and so on.

With every constraint specified, to enforce them within our model we simply must add them inside a `fact`. Facts declare our constraints as assumptions and so Alloy will only generate instances that respect all constraints within them. Additionally, to make it easier to identify in the future, we will name our fact `RoadConstraints`.

```
fact RoadConstraints {
  no adjacent & iden
  adjacent in ~adjacent
  all r : Road | Road in r + r.^adjacent
}
```

Listing 2.5: Road constraints

2.1.4 Subtyping

Our model now characterizes features common to all roads. However, in real life not all roads are equal. Depending on their expected traffic flow and their accessibility, these can be subject to different rules and designations based on their needs and purposes. For this example we will take into consideration three types of roads commonly used by urban planners. The first type will be labeled as `Local` roads, which are roads with high accessibility and low traffic flow to buildings and leisure areas. These roads also present large active fronts for public transport. The second type will be labeled `Collector` roads which are responsible for gathering and distributing traffic of multiple Local roads, and thus are able to provide higher traffic flows at the cost of lower accessibility to buildings and other civilian infrastructures. Finally, the third type will be labeled `Freeways`. These have have limited access points in order to be able to provide a high traffic flow between their collectors and other types of medium flow roads. Roads also have other characteristics regarding, for example, construction characteristics and naming schemes. However, since these factors are not relevant to our model we will not consider them for better clarity.

To specify that a road can be categorized within one of these types, we simply must indicate that these represent subsets of the original signature. This is done by declaring these road types as sub-signatures, by either adding `extends Road` or `in Road` to their declaration, which will cause different behaviors in the instance generation. Although both the `extends` and `in` keywords will specify these entities as sub-signatures of `Road`, the `extends` keyword will force these subsets to be disjoint, unlike the `in` keyword. By having the subsets disjoint, a road will only be able to have one designation at a time and so there will not be cases where, for example, a road is both a collector and a freeway. As such, in our example the sub-signatures will be declared with `extends`.

Finally, we must force the signature `Road` to be the union of its extensions. This is achieved by qualifying the original signature with the keyword `abstract`. Since the signature itself has not changed, every constraint previously defined will remain valid. These changes can be viewed in Listing 2.6.

```
abstract sig Road {
```

```
    adjacent : set Road
}
sig Local, Collector, Freeway extends Road {}
```

Listing 2.6: Road subtypes

With the road types specified, the only thing remaining is to describe their constraints. Previously we implied that, with the exception of local roads, all road types connect multiple roads. As such, roads with only one access point, often designated as "dead end streets", must be local roads. To specify this, we can simply specify that if a road has only one adjacency then this road is part of the set of local roads.

Additionally, road access between each type of road is restricted. Local roads can only connect to other local roads or collectors, while freeways can only connect to other freeways or collectors. As such, to enforce these properties we will specify a constraint for each type of road, dictating that the set of it is possible adjacent roads must be contained in the union of the roads that they are allowed to connect. For the specific case of the collectors this is redundant since these are allowed to connect to any kind of road. Finally we can present these constraints in a new `fact`, presented in Listing 2.7, which we will name `RoadTypeConstraints`.

```
fact RoadTypeConstraints{
  all r : Road | one r.adjacent implies r in Local
  Local.adjacent   in Local + Collector
  Freeway.adjacent in Freeway + Collector
}
```

Listing 2.7: Road type constraints

2.1.5 Other Declarations

To complete the structure of our model the only thing missing are the vehicles which drive through the traffic network. Similarly to how our road network is defined, all the modeled vehicles will be an extension of an abstract signature named `Vehicle`, with the location of each vehicle being captured by a simple relation between the vehicles and the roads. The specified vehicles will be both cars and buses. Cars will have no additional trait specified. Buses, on the other hand, continuously follow a circuit within specified regional areas, which only includes local roads or collectors. The resulting declarations are presented in Listing 2.8.

```
abstract sig Vehicle {
  location : one Road
}
sig Car extends Vehicle {}
sig Bus extends Vehicle {
```

```
    area : set Local+Collector,
    circuit : area -> one area
}
```

Listing 2.8: Vehicle specification

Here we can see two new types of relations which possess interesting properties. Starting with the `area` relation, we can see that it pairs each Bus to a **`set`** containing either the collectors or the local roads that it will navigate, specified by using the union operator in the declaration. Later, some constraints may require the restriction of this relation to a particular type of road. This can be specified with the operator `:>` followed by the restricting set. For example, a restriction of `area` to the local roads would be specified as `area :> Local`.

Range restrictions have a counterpart named domain restrictions which are specified with the `<:` in a similar way. As implied, these act in the same way but on a relation domain instead of its range. These are not particularly useful regarding the `area` relation, although the same cannot be said for other relations, such as `location` for example. The `location` relation maps each vehicle to the road it is located. These vehicles can be either buses or cars, and as such if the user finds itself in a situation were it must differentiate a specific relation based on one of these signatures, for example cars, it would specify the relation with `Car <: location`, where the location's domain has been restricted to the set `Car`.

Continuing to the `circuit` relation, its purpose is to specify the circuit in which each bus will move. The circuit itself is defined as a relation, which pairs each road belonging to the area of the bus with the following one according to its itinerary. As such the entire relation couples two "inputs" (the bus and the road in question) to a single "output" (the next road following the circuit), thus being a ternary relation. As presented, declaring such relation is as simple as preprending the additional inputs (in this case the `area` representing the road in question) to the relations range, splitting them with the operator `->`. This is not restricted to one additional input: if the user finds itself in need of specifying additional inputs, it can add them in the exact same way, and thus declare more complex relations of increased arity.

With the new relations defined, we still need to ensure that the model respects the required constraints, specially on the circuits. First, we must ensure that the circuit is in fact a periodic chain. To start, the easiest way is to declare the relation as being simple (or functional), i.e., for every bus, each road of the area has exactly one path forward. This can be done with a constraint. However, due to how common these types of relations are, Alloy provides keywords which can enforce this in the relation's declaration. In our case, this keyword is the **`one`** that is present before the relation's range. Next, we must ensure the circuits properties themselves, this being that each is a periodic chain of all the roads present in the given bus' area. Finally, we must ensure that the circuit respects the actual road system. The constraints are presented in Listing 2.9.

```
fact Circuit {
  all v : Bus {
    v.circuit in adjacent
    all r : v.area | v.area in r.^(v.circuit)
  }
}
```

Listing 2.9: Circuit constraints

As we can see, in this fact we specify two constrains for all possible Buses. The first constraint dictates that the circuit of any bus is a subset of the road network (defined through the `adjacent` relation) and thus forces it to be respected. The second constraint specifies that the area of the bus in question is reachable via the circuit from any of its roads. This intricate constraint imposes three restrictions when in conjunction with the simple relation constraint previously imposed in the declaration of circuit. The first one is periodicity: since every quantified road is contained within the buses area and each road can only specify one other road as its successor, the relation is forced to be a cyclic ring. The second and third behavior are related, by specifying that the entire area of the bus is contained within the previously mentioned cycle, we guarantee that any road of the bus' area has both a successor and predecessor in the circuit, thus forcing the circuit to model the entire area of the bus.

2.2 Structural Model Analysis

2.2.1 Analysis Commands

To start analyzing our model we first need to specify the desired analysis commands. There are two types of analysis a user can enact, these being model validation and model verification. Model validation commands, which are described by **`run`** blocks, allow the user to browse several automatically generated instances of the specified model. This is extremely useful, since it allows the user to quickly detect possible errors that had yet to be recognized. Model verification on the other hand, described by **`check`** blocks, are designed for property checking. When presented with properties from the user, these will check their validity against a wide range of instances in search for counter-examples. If the properties fail for an instance, then Alloy will present it to the user, otherwise the analyzer will notify the user that the properties may be valid. This uncertainty on the validity occurs due to the check not being complete, since the analyzer exhaustively searches every possible case within a finite universe. However, the scope of this universe can cover a very large number of possible cases and as such provide enough assurance for most real world scenarios.

For our static model we will mostly focus on model validation, since by being immutable there are yet not many relevant properties for us to check. The most typical way to start model validation is to declare

a simple empty `run` block. These blocks have a similar syntax to the `fact` block, allowing the user to optionally name them and also specify additional constraints for it to consider during instance generation. These constraints are not part of the model like facts, but simply allow for a more fine search within the generated instances. Our first run block, named `Empty`, is presented in Listing 2.10.

```
run Empty {}
```

Listing 2.10: Empty run

2.2.2 Visualization Types

To visualize the generated instances we simply have to execute the specified command. This can be done through the button **Execute** on the editor or though the program's execution context menu. If all the constraints are consistent, Alloy will present an instance which when accessed will launch a new window within the analyzer with its depiction. The first instance for our model is presented in Figure 1.

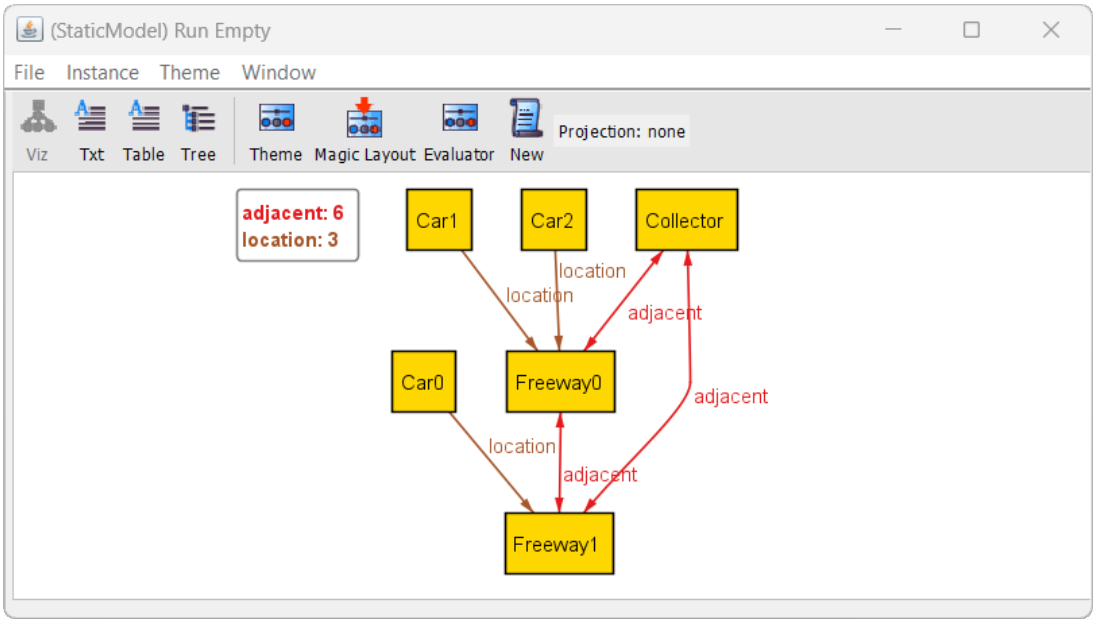
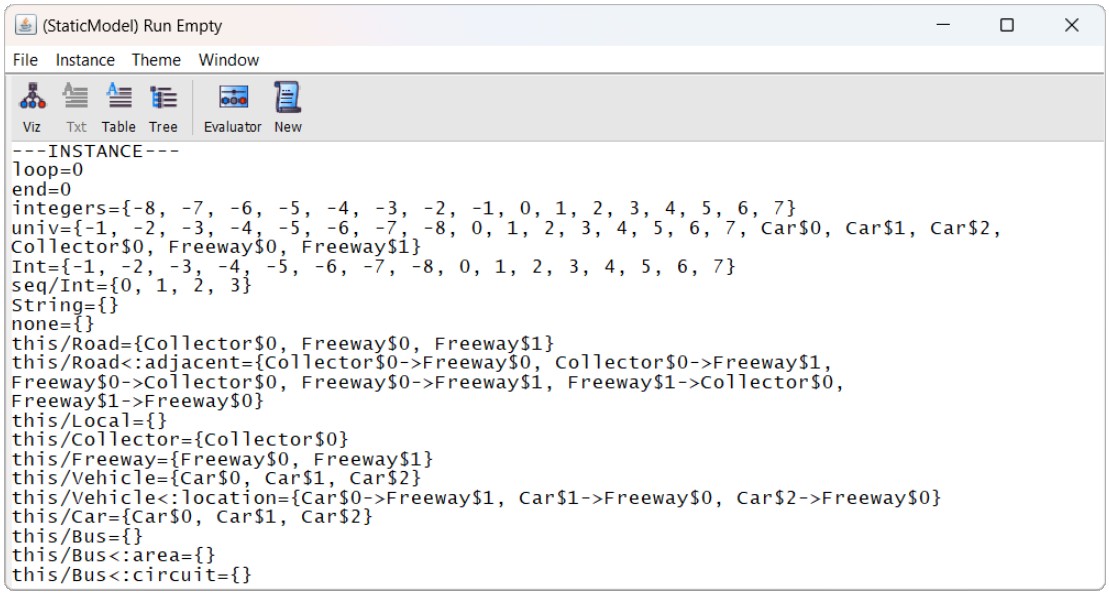
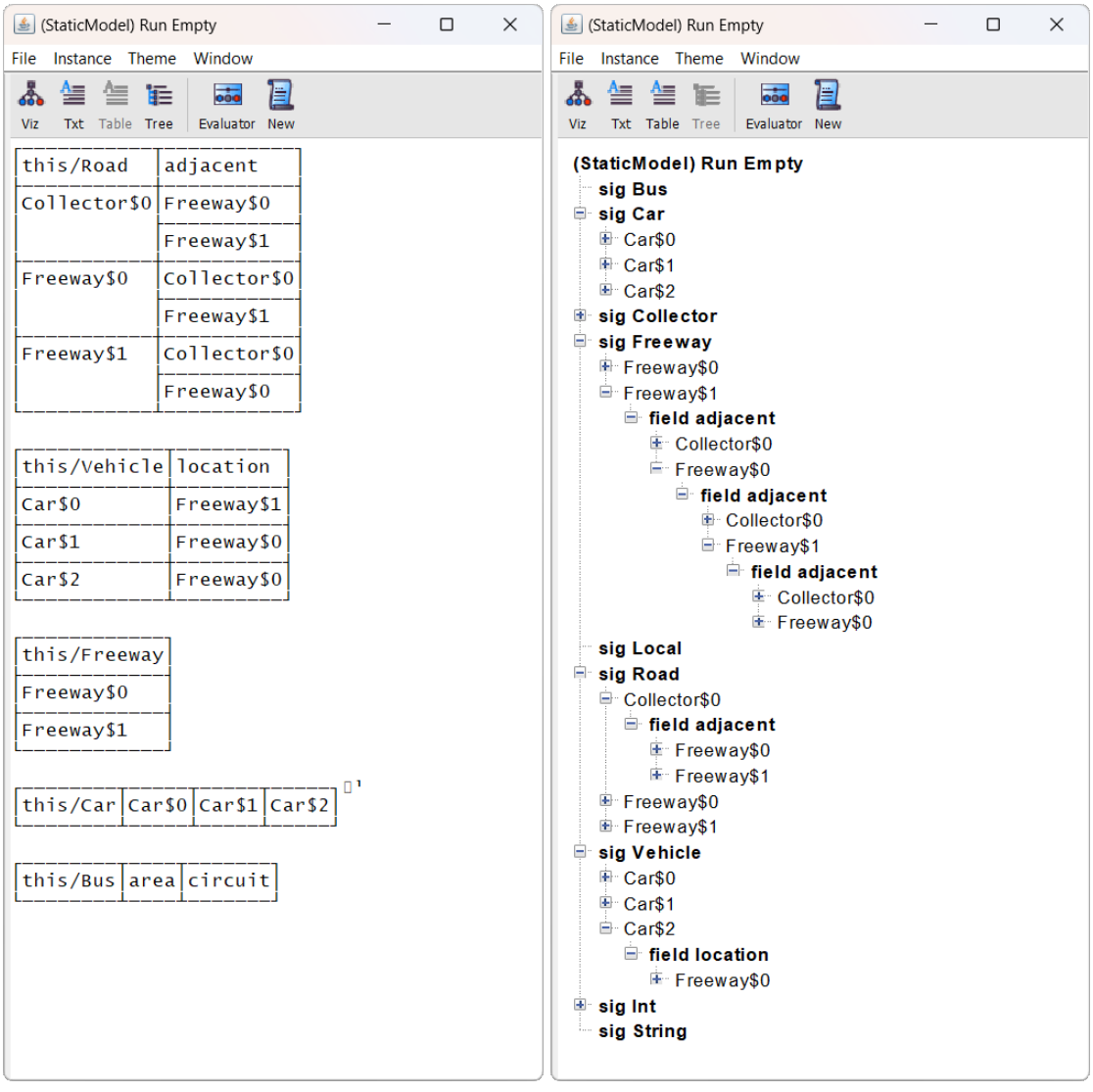


Figure 1: Default visualization of an instance of the static model

As we can see, by default the analyzer will display the instance as a graph while providing simple variations on the relations colors. This is not however the only type of instance visualization. The first four buttons off the window offer different styles of presenting the model. **Viz** corresponds to the default graph representation already presented, **Txt** lists each set and their atoms in a plain textual format, **Table** pretty prints each set as a table, and finally **Tree** displays an instance as an horizontal hierarchy based fork tree. These three options of visualization are presented in Figure 2. The most interesting view and the one that we will be using from now on is the graphical view.



(a) Text View



(b) Table View

(c) Tree View

Figure 2: Other visualization options

2.2.3 Themes

Themes are an essential part of the graphical view. They allow for the user to specify not only the shapes and color of atoms and relations, but most importantly the form in which these are represented, i.e. whether these should be present as atom attributes, as nodes of the graph, or if they are not relevant to be displayed at all. The default theme, presented in Figure 1 displays all available information, where every atom is presented as a yellow rectangle and every relation as a different colored arc between its atoms. Some sub-signatures may be also displayed as attributes depending on their context. This is usually sufficient in small instances, however as these grow this style can become extremely difficult to understand. Besides the default, regarding Alloy generated themes there is also the *magic layout*. This functionally that can be called with the button labeled **Magic Layout**, and proceeds to assign different shapes and colors to the atoms, which can mitigate a few of the problems of the default layout. However, the resulting visualization can still be very difficult to understand. To exemplify this, we will execute the block `run OneBus {one Bus}` which will force the instance to contain a single bus. The resulting instance after applying the Magic Layout is presented in Figure 3.

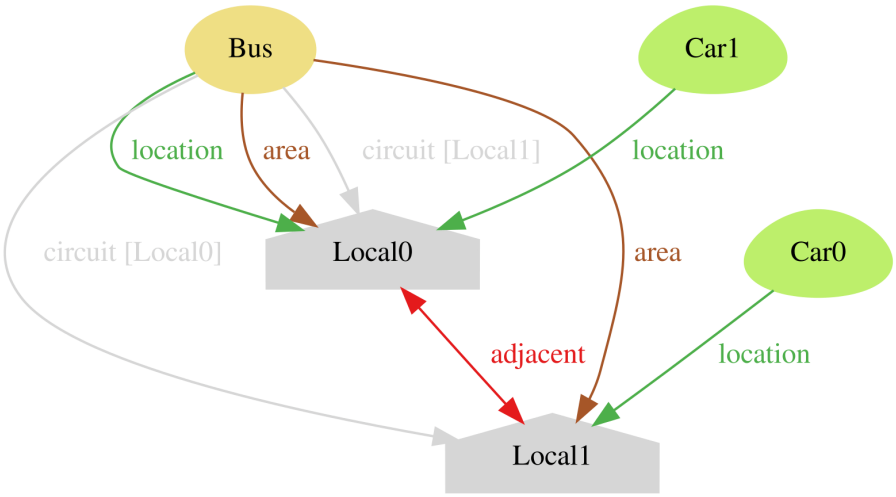


Figure 3: Magic layout example

From this example we can observe that Alloy's automatic themes do not provide much help in easing the instance's comprehension, more precisely regarding the `circuit` relation, which can barely be distinguished from the `area` relation and requires us to search for the scattered labels in order to piece it together. If we were to add a second bus it would become even worse, as in addition to finding the labels we would also need to match them to the correct vehicle. The best way to visualize complex instances is to define a personalized theme. To do this, we must use the **Theme** button which will in return display a menu allowing us to customize the visualization. On this menu, our signatures and types will be displayed in a hierarchical form which corresponds to how they where defined in our model. As a result, in addition to the precise tuning of our signatures, changes regarding multiple extensions of the same signature can be defined solely within their parent signature. For example, if we choose to see all atoms corresponding

to roads as hexagons, we simply need to change our `Road` signature's shape, which will then cause all road types to inherit said change. Beyond both types and sets, the most relevant customization options concern relations, and the user can specify these to be either hidden or represented with arcs or atom attributes. An example of the theme customizer is presented in Figure 4.

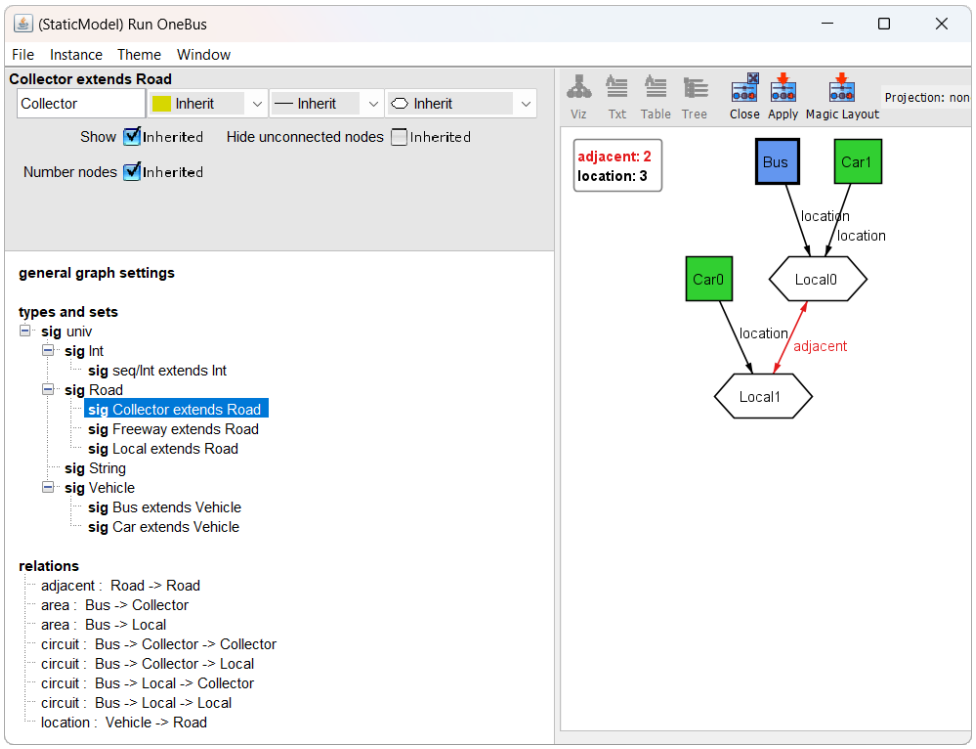


Figure 4: Theme customizer

Since themes are burdensome to setup, Alloy allows for them to be saved as `.thm` files, which can later be loaded to visualize an instance. This not only prevents the user from wasting time on redefining themes, but also provides it with the ability to hot-swap between them and thus providing multiple perspectives on the same instance.

Our personalized theme will display vehicles as different colored boxes and roads as different colored Hexagons. Additionally, both `area` and `circuit` will be hidden since their comprehension will be difficult with either arcs or attributes. Instead, we will rely on the evaluator to analyze these relations.

2.2.4 Evaluator

The evaluator is a console like interpreter which allows the user to evaluate arbitrary expressions regarding the model. Although it is very useful for a precise analysis, it does not provide full access to the language, for example it is not possible to define new facts. Its utilities include listing the values of sets and evaluating predicate expressions. For our instances, we will mostly use it to consult the values for the buses areas and circuits. Exemplifying, asking for the circuit of the bus in the instance of Figure 3 and Figure 4 can be done with the expression `Bus$0.circuit` , as seen in Figure 5.

```
> Bus$0.circuit
```

Local$0	Local$1
Local$1	Local$0

Figure 5: Prompt for the circuit of a Bus

Since the expression already specifies the bus in the circuit's relation, each of the tables line will simply pair a road in the first column with its respective successor in the second one. As such, we can visualize that the circuit of this bus follows the roads `Local0 -> Local1 -> Local0 -> ...` periodically.

2.2.5 Model Validation

As previously mentioned, model validation involves the inspection of several automatically generated instances in order to detect potential model problems. In most model specifications, these examinations usually are not enough to completely validate the model. However, thanks to their automatic and arbitrary nature, these instances provide instant feedback to the user, empowering it with extremely useful knowledge of its model and the possible unwanted scenarios it allows. As a result, this forms a crucial step in any efficient model validation.

As mentioned before, these instances are generated with `run` blocks. These blocks can optionally be named and are accompanied by a constraint were the user can optionally restrict the analyzer's instance generator. When executing the run block, Alloy will generate a random instance in a new window. For the case of our empty run (Listing 2.10), the resulting instance was presented in Figure 1.

What we have not explained yet is the fact that a run command is not restrained to this instance. By pressing the button **New**, the analyzer will attempt and possibly generate a new instance that also satisfies the specified constraint and model. This process can be repeated innumerous times so long as there are enough valid instances that satisfy the specified restrictions. In Figure 6 two additional valid instances of the empty run are displayed.

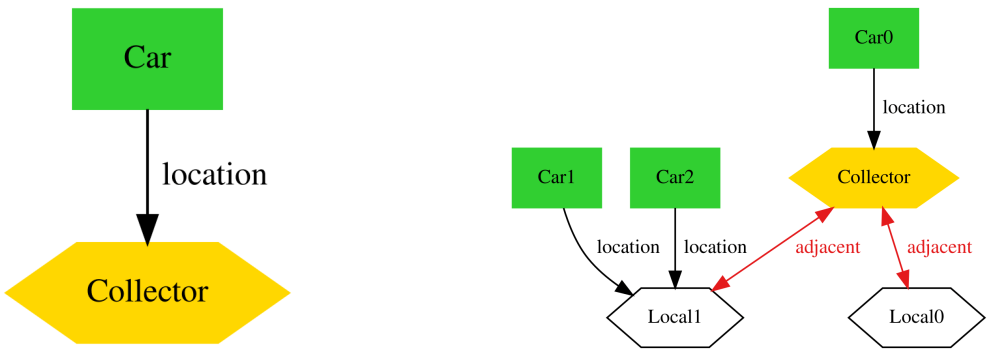


Figure 6: Additional instances of the empty run

Additionally, `run` blocks also provide keywords to specify the size of the domain that will be analyzed. This can be done by using the keyword `for` after the block declaration, to specify the desired scope for executing the commands. For example, the command `run {} for 6` will generate instances with at most six atoms in each top-level signature (`Road` and `Vehicle` in our case). If we want to specify an exact scope we can add the keyword `exactly` resulting in `run {} for exactly 6`. It is also possible to precisely control the scope for each signature, for example, limiting the analysis for up to three roads and up to two vehicles can be made with the command `run {} for 3 Road, 2 Vehicle`.

2.3 Dynamic Modeling

2.3.1 Behavior Specification

Dynamic models describe how systems can behave with the advancement of time. Typically, dynamic models are some sort of state transition systems. Alloy 6 introduced the ability to define said specifications, a feature which before was supported by the *Electrum* extension [21, 11]. To demonstrate these dynamic modeling features we will expand our previous example.

In particular, we will explore a traffic system with moving vehicles, which gradually traverse the road network. Every dynamic model should specify the events that can happen to any entity that belongs to the universe in question. Nevertheless, only some of these are actually useful to study in each model. First, the signatures and relations whose value will change over time should be declared as being variable with the keyword `var`. In our example, we plan only to vary each of the vehicle's location and thus this relation must be declared as a variable relation. The new definition is presented in Listing 2.11.

```
abstract sig Vehicle {
  var location : one Road
}
```

Listing 2.11: Declaration of the variable location

After declaring the mutable structures that characterize the state, the first step to describe a dynamic model is to specify its initial states. This involves declaring constraints regarding our variable components using the logic constructs already used in static modeling. For our traffic system we will specify that initially the cars are located in any local road, while buses will be located in one of the roads of their itinerary. These constraints are declared in a fact named `init` as presented in Listing 2.12.

```
fact init {
  all c : Car | c.location in Local
  all v : Bus | v.location in v.area
}
```

Listing 2.12: Initial state fact

After defining the initial states, the next step it to specify the model's behavior and evolution using Alloy's temporal logic. These can be solely implemented within facts, however the best common practice is to describe each possible event with a predicate. Predicates are very similar to facts and specify a list of constraints. The main difference is that they do not inherently declare any assumptions, and are instead evaluated only when called upon in other blocks. Additionally, entities can be shared in said calls, allowing for the abstraction of the constraint block themselves.

The main event that can occur in our model is when a vehicle moves between roads. In the specification of this event, our predicate will have to dictate that the vehicle in question will only move to one of its adjacent roads. In addition, if said vehicle is a bus, the chosen road must correspond to the one specified in its circuit. A possible specification is presented in Listing 2.13.

```
pred move[v: Vehicle] {
  some r : v.location.adjacent {
    location' = location ++ v->r
    v in Bus implies r in v.location.(v.circuit)
}}
```

Listing 2.13: Move event

As we can see, the specified predicate `move` receives the vehicle in question and verifies that for one of its adjacent roads, the new locations (specified by using the prime operator in `location'`) will be equal to the current locations overlapped (++) with the tuple relating the vehicle to its new road. In addition, if the vehicle is a bus, this road must also be specified as the circuit's successor for the current location. This specification will allow for several types of movements between the vehicles and the roads, which, in conjunction with other vehicles and other possible actions, will allow each state to branch out into several valid new states.

The second behavior that we must describe regards the changes caused by any other possible event. As previously mentioned, dynamic models must take into account all possible occurrences regarding their defined universe, otherwise their specifications will not be complete abstractions of their intended systems. As such, even without the ability to precisely identify or describe such events, we must disclose how they affect our model. In most dynamic models, these external events do not affect the declared mutable structures, and can intrinsically be viewed as stuttering steps. Specifying such stuttering is very simple, since we only need to specify that the value of any of our mutable structures remains the same. For our model, the stutter specification will state that the new value of the relation `location` will equal its current value. The resulting predicate is presented in Listing 2.14.

```
pred stutter {
  location' = location
}
```

Listing 2.14: Stutter event

With every possible event specified, the only thing left to specify is how these will be interleaved to form complete instance traces. For this, we must enforce that every state transition follows one of the specified events, i.e. some vehicle moves or the systems stutters. To enforce this in every state we simply need to enclose the resulting expression within the `always` temporal operator. The resulting fact is presented in Listing 2.15.

```
fact Traces {
  always (stutter or some v : Vehicle | move[v])
}
```

Listing 2.15: Traces fact

With the valid traces specified, the only thing remaining in our model is to define the liveness rules it should satisfy.

2.3.2 Temporal Logic

Dynamic model constraints can be specified through the use of temporal operators. These specify how the constrains will be checked throughout an instance trace, and can be distinguished between future or past operators.

Regarding future operators, the most commonly used are the unary operators `always`, `eventually` , and `after`. As previously seen, `always` states that its enclosed expression is true in any possible state. On the other hand, `eventually` declares an expression as an inevitability, while `after` forces its enclosed expression to be valid in the next state. Additionally, these operators can be mixed to form more complex expressions. For example, if we want to specify an intermittent event we can use the combination `always eventually` ϕ. Alloy also supports the binary temporal operators ϕ `until` ψ and ψ `releases` ϕ, the former dictates that ψ will happen and ϕ must be true until then, and the latter dictates that ϕ can only stop being true after ψ.

Past operators function mostly as counterparts for the future ones. The unary operators `once`, `before`, and `historically`, dictate that the enclosed expression was once true, was true in the previous statement, or was always true, respectively. It is also possible to use the binary operators ϕ `since` ψ and ψ `triggered` ϕ, the former stating that ϕ became true after ψ occurred, and the latter that ϕ was always true, or at least true after ψ.

2.3.3 Liveness Constraints

With the previously presented operators we can complete our model and specify the final constraints to correct its behavior. The first regard buses, and how these vehicles must continuously move throughout their circuit in order to provide their public service. The second regards freeways and vehicles, and dictates that no vehicles are allowed to stop in freeways. This is to ensure the drivers safety in these high flow

roads, since stopped cars form obstacles which can easily cause traffic jams at best or traffic accidents at worst. The resulting constraints are presented in Listing 2.16. These are called liveness constraints because they force some "good" behaviors to happen, unlike safety constraints that only forbid some "bad" behaviors from happening.

```
fact Liveness {
  all b : Bus | always eventually move[b]
  all v : Vehicle |
   always (v.location in Freeway implies eventually move[v])
}
```

Listing 2.16: Liveness constraints

As depicted, the first constraint dictates that every bus will always eventually move, and in conjunction with the definition of `move` will thus provide its intended service. The second constraint states that for every state if a vehicle is located in a freeway, then it will eventually move to an adjacent road.

2.4 Dynamic Model Analysis

Similarly to what was described in our static model, our model analysis will be divided into two phases. In the first phase we will begin by validating our model, by inspecting arbitrary instances generated with a `run` block, to have a first glance of our model, and confirm that there are no evident mistakes. In the second phase, we will be resorting to `check` blocks to verify expected properties about our model.

2.4.1 Model Validation

In order to validate our model we can start by executing the command:

```
run Empty {} for 6 but 1 Vehicle
```

When opening the generated instance the result is the trace presented in figure 7.

When compared to the static analyser, presented in Figure 1, we can see that the visualization now depicts two different states of the model as well as a depiction of the trace on top, highlighting which transition is currently being displayed. In addition, the **New** button is gone, and is now replaced by six buttons. The first one named **New Config** works in the same way as the old button, and thus will cause the analyzer to generate a new configuration with new values for the static signatures and relations. As the name suggest the next button, named **New Trace**, will generate a new trace for the same configuration, if possible. Following we have the **New Init** button which generates a new initial state on the current configuration. The **New Fork** button will attempt to branch the current displayed transition generating a different next state. Finally we have two arrow buttons which allow us to move forwards and backwards along the trace.

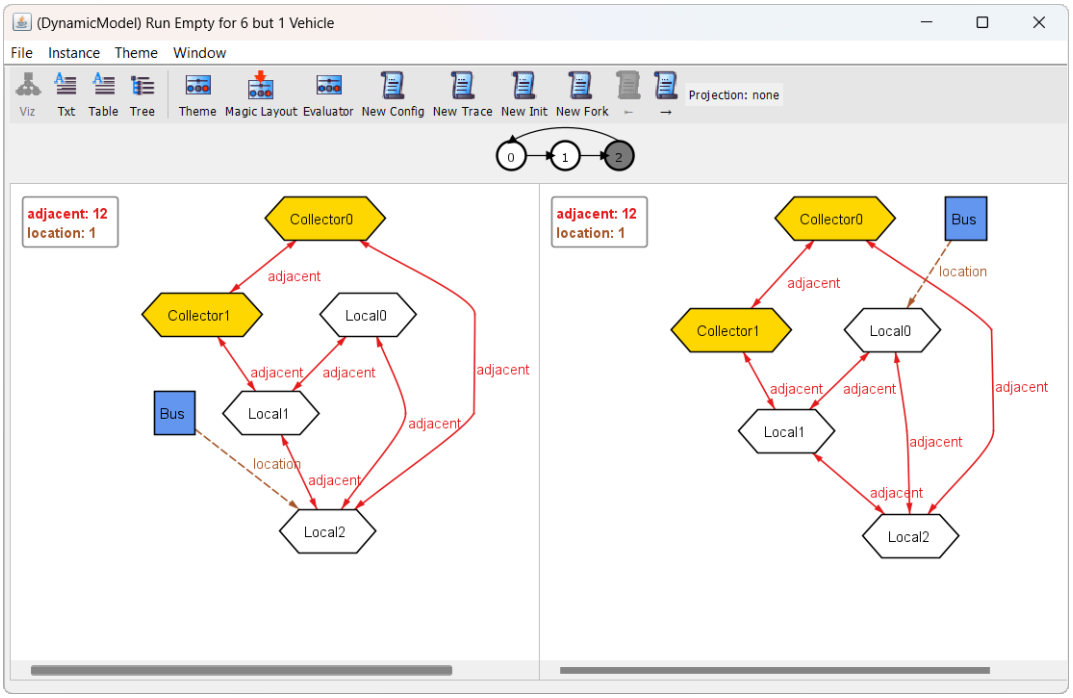


Figure 7: Dynamic model visualizer

2.4.2 Model Verification

To verify our model we will be examining two different types of properties on both buses and cars, namely *Safety* and *Liveness* properties.

2.4.2.1 Safety Properties

We will begin by studying some of the vehicles' *Safety* properties, which specify the absence of unwanted behaviors on our system. These properties tend to be easier to model check, since these can in fact be refuted by exhibiting only a finite number of steps that leads to a "bad" state. These are typically specified using the pattern `always` ϕ, where ϕ specifies an invariant that all reachable states must satisfy.

Starting with buses, we want to specify that a bus will never leave its area of operations. To specify this we can simply state that for every bus we will always have its location contained within its area. The resulting `check` is presented in Listing 2.17.

```
check BusSafety {
  all v : Bus |  always v.location in v.area
} for 6 but 1 Vehicle
```

Listing 2.17: Bus safety

When executed, Alloy will attempt to search for a counter-example within ten steps using a *bounded model checking* verification technique, where it will generate and test several possible examples. If it fails to find such a counter example it reports that the property may be valid for every case. A user has the flexibility to modify the verification behavior either by adjusting the scope of the bounded verification or by eliminating the restriction, thus performing an *unbounded model checking* verification.

Proceeding to the cars, we can specify that, for example, to reach a freeway these must have passed through a collector. Expressing this can be done by saying that for all cars in all states, if they are located in a freeway then they were once located in a collector. The resulting `check` is presented in Listing 2.18.

```
check CarSafety {
  all v : Car |
    always (v.location in Freeway implies
                once v.location in Collector)
} for 6 but 1 Vehicle
```

Listing 2.18: Car safety

Just like the previous property, Alloy will not be able to find any counter-example and thus report that this propriety may be valid.

2.4.2.2 Liveness Properties

To finish our verification we will be checking *Liveness* properties which specify behaviors that the system must have. These are an essential part of the verification, however they are also more difficult to express and to verify, since a counter-example to a liveness property must be an complete infinite trace where the desired behavior never happened. Most liveness properties conform to the pattern `eventually` ϕ, were ϕ dictates an expected state the system should reach.

For the buses, we will verify if these in fact serve every road in the area while moving through the road network. A possible specification could be that, for every bus, its location will periodically pass through every road in its area. As mentioned in Section 2.3.2, an intermittent property can be described with the combination `always eventually`. The resulting `check` is presented in Listing 2.19.

```
check BusLiveness {
    all v : Bus, r : v.area |
        always eventually v.location = r
} for 6 but 1 Vehicle
```

Listing 2.19: Bus Liveness

When executed, Alloy did not find a counter example, implying that this property could be valid.

Finally, for the cars we will attempt to verify that these keep periodically returning to local and collector roads (they cannot remain in a freeway forever). This is very similar to the previous property, being that now instead of buses we have cars, and instead of the roads of the bus area we have our Local and Collector roads. The resulting `check` is presented in Listing 2.20.

```
check CarLiveness {
  all v : Car | always eventually v.location in Local + Collector
} for 5 but 1 Vehicle
```

Listing 2.20: Car Liveness

Upon executing, Alloy will find that this is not true for all possible scenarios. To see why this does not hold we can inspect the counter-example trace. In this case, it consists of an instance with a circular freeway with three road segments, one collector, one local road, and one car. In the initial state (0) the car is located in the local road. Then the car will move from the local road as depicted in Figure 8.

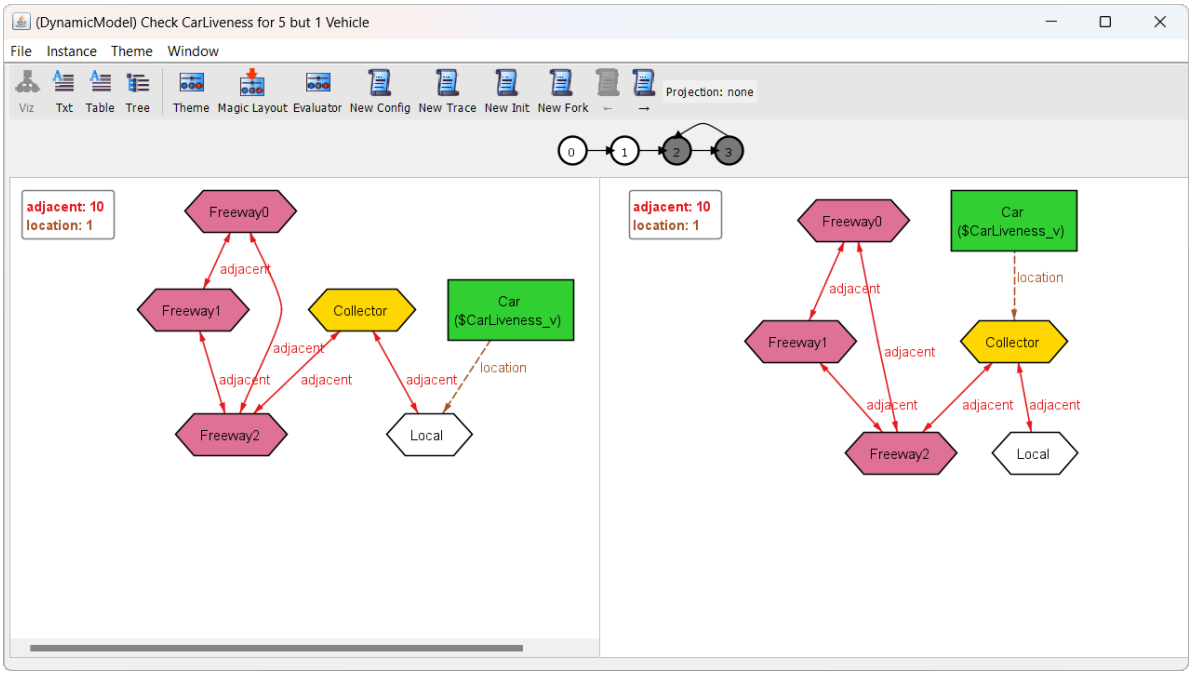


Figure 8: Counter-example's first transition

Proceeding from the collector in state (1), the car will move to `Freeway2`, as depicted by Figure 9.

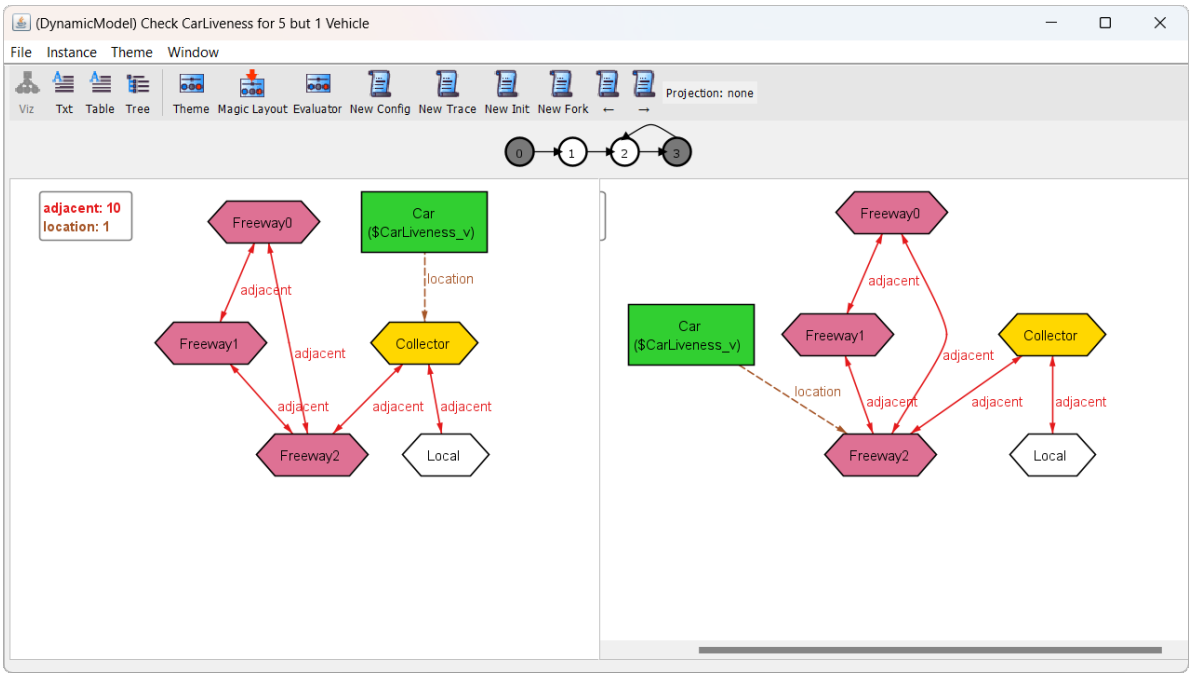


Figure 9: Counter-example's second transition

From this point onward the car will loop through the states (2) and (3) indefinitely. In the 3rd, 5th, 7th, ... transitions the car will move from `Freeway2` to `Freeway1`. In the 4th, 6th, 8th, ... transitions the car will reverse its previous move, going from `Freeway1` to `Freeway2`. This is visible in Figure 10.

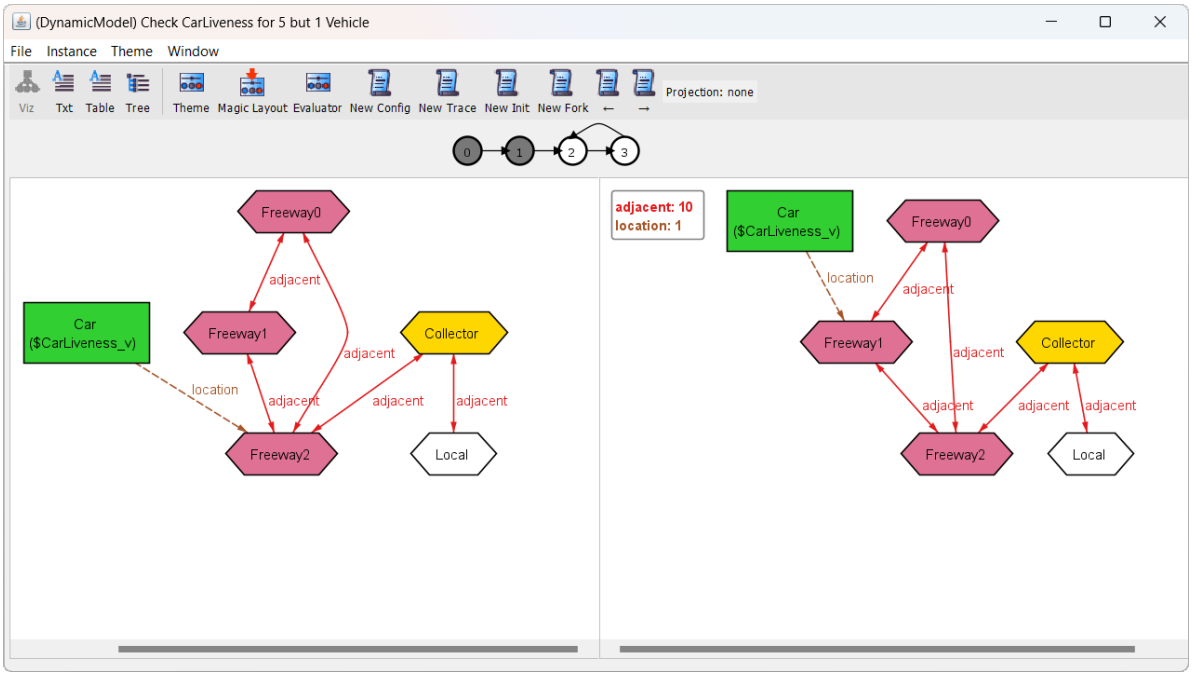


Figure 10: Counter-example's third transition

As a consequence of the car looping forever in the freeway, it will never return to any local or collector road, thus making the property invalid.

3

Alloy4Fun

As presented previously, the Alloy Analyzer provides several ways to specify and analyze systems. Just like most modern programming and modeling languages, these models are saved in files on local storage, which allows the individual users to freely manage them as they see fit. This approach although very versatile, can prove very impractical when taken into a collective scenario, most notably in an educational context where students must constantly send models back to their teachers in other to get feedback on their results. In this case, besides the dilatory and error-prone process involved in manually sharing the model files, the teachers and tutors also need to dedicate a lot of time on repetitious tasks for validating, organizing, and transmitting feedback to their students. Alloy4Fun was conceived precisely to solve these issues.

Alloy4Fun is a web application that allows online editing, verification, and sharing of Alloy models and instances [23]. It provides a fully working online Alloy platform, where users can specify models and run commands similar to the Alloy Analyzer. Additionally, it allows users to create specification exercises, which can then be shared with other users. A user which receives an exercise can attempt to solve it and will be instantly provided with feedback. Additionally, any user can save their models and instances in the platform at any time, upon which they will be provided with a *permalink* which can be used to later retrieve the saved model. This *link* is anonymous, as it does not store any information about its creator or access history. The user has full control over the livelihood of the link and can use it as they see fit, whether it is simply storing it or sharing it with other users.

3.1 Interface

Alloy4Fun replicates the main features of the Alloy analyzer in a web application format. Upon accessing `alloy4fun.inesctec.pt` (the current deployment's URL) the user will be presented with an empty online editor, where it can specify a model. In Figure 11 we have a screen capture which displays a model of a File System's trashcan.

Figure 11: Alloy4Fun overview

In addition to the editor, we can see three buttons on the right, where the first one, **Execute**, as implied, executes a command from the model, which can be chosen from the drop-down presented directly above the button. When pressed, the web page will expand depending on the output of the execution. Valid *check* commands (or invalid *run* commands) will merely present a success (or insuccess) message, while invalid *check* commands (or valid *run* commads) will display their respective counter-examples (or instances). This behavior is presented in Figure 12, where an erroneous check was run.

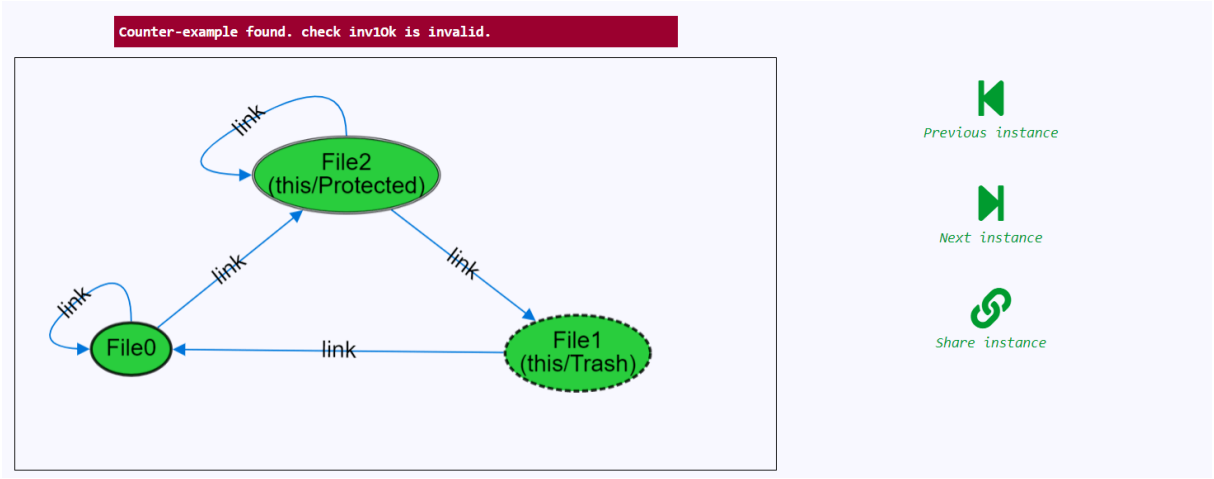


Figure 12: Counter-example depiction

The second button, labeled **Share model**, allows for the user to save the model within the platform. Upon this, the platform will return to the user a *permalink* of the saved model. This link is anonymous,

and it is the responsibility of the user to keep track of it, as the platform does not provide any means of recovering said links. This is by design, since it keeps the interaction with the platform as simple as possible, and avoids the hassle of storing credentials and ensuring the mandatory data protection regulations that are associated with it. The third and final button allows for the user to download the derivations of a previously shared model. This is useful for gaining insights about the users attempts and learning processes, which could prove useful for future research and developments.

3.2 Usage

To use the platform a developer simply needs to write an Alloy model and the commands it wishes to verify. The user can then execute the commands and visualize their results within the analyzer and also share/save the model. The example built in Chapter 2 can be found in `http://alloy4fun.inesctec.pt/56DQPgL2jSnHAMKbi`.

3.2.1 Exercises

Additionally, the developer can also create exercises with this platform. These are defined solely with the Alloy language and an additional functionality to hide code from the model. To exemplify this process we will be building an exercise based on the previously developed static model (Section 2.1) where the objective will be for the user to specify every modeled constraint.

A kind of exercises that can be easily encoded in Alloy4Fun are specification challenges. These are composed of an empty predicate block where a user would write their solution, alongside an hidden oracle, i.e. a construct that allows to automatically test the submissions. To create a challenge from our model the simplest way would be to have a single empty predicate and to use all the current facts as an oracle, and simply have the student test its implementation with a single check. However, this would not be optimal as there is no way to uniquely isolate faulty constraints. A student would be required to write and validate the entire model at once, which will quickly overwhelm it.

The best way is to split the current assumptions into several predicates and only compare the relevant ones with the student's solution. For example, for any of the constraints in Listing 2.2 we would specify the challenge as presented in Listing 3.1. We will firstly define an empty predicate `inv1` which corresponds to the block the student must complete. Following, we define the predicate `inv1o` which will contain the solution to the problem. Finally, to allow automatic feedback to the user, we simply define a **`check`** command which will verify if the student's solution is semantically equivalent to the specified oracle.

After specifying the constraint challenge we will want to hide some of its blocks, since we do not want to present the solution to the users or allow them to cheat by changing the validation process. Alloy4Fun allows this by adding the comment *`//SECRET`* before a code block. When saving and sharing, the platform will now provide two different *permalinks* for the exercise, a public one which hides the secrets and a private one which will display the complete model.

```
pred inv1 {
  // A road cannot be adjacent to itself
}
//SECRET
pred inv1o {
  no adjacent & iden
}
//SECRET
check inv1{
  inv1 iff inv1o
}
```

Listing 3.1: Constraint challenge 1

This approach allows the challenge developer to precisely control which constraints to verify in each challenge. However it also decouples all of the assumptions from the model, which can cause errors. For example, the connected constraint in Listing 2.4 is correct thanks to the previous assumption in Listing 2.3. To ensure the same correct behavior on our challenge we must declare our assumptions in the verification command. This is commonly done with an implication as seen in Listing 3.2, were we have two assumptions. Additionally, this tactic can also be used to filter irrelevant invalid states to better depict counter-examples to the users.

```
//SECRET
check inv3{
  inv1o and inv2o implies
  (inv3 iff inv3o)
}
```

Listing 3.2: Command of `inv3`

When the developer finishes specifying an exercise containing one or more challenges it can then share it. Upon this, Alloy4Fun will generate two *permalinks* which correspond to the public and private views previously mentioned. The public link can then be distributed among competitors, while the private link allows the retrieval of the complete model and also the submissions which derived from it. The permalinks of our exercise can be accessed through Table 1.

PUBLIC	http://alloy4fun.inesctec.pt/BFrGk6AiedYr6Q46b
PRIVATE	http://alloy4fun.inesctec.pt/dJF3ab5vwCxTS4tAh

Table 1: Permalinks to an exercise

3.3 Architecture

The Alloy4Fun application is composed of three components. The core component is the web-service itself which provides the interfaces required to access the main web-page and thus the application's services. Next, we have a component responsible for dealing with the Alloy language, labeled API. The final component is a NoSQL database implemented with MongoDB. A visualization of these components and their interactions is presented in Figure 13.

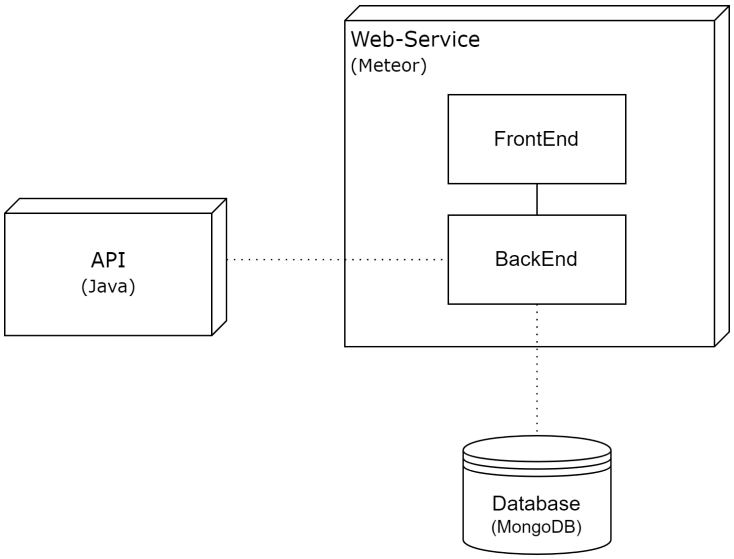


Figure 13: Alloy4Fun architecture

Web-Service The web-service is developed in JavaScript with the use of *Meteor*[1], a full-stack framework for web-based applications. Being a full stack application it can essentially be described as two modules, the front-end and the back-end. The front-end is responsible for defining and providing the user interface (i.e. the web-page) and maintaining the editors. The back-end is responsible for processing the user actions, which range from simple database reads and writes to more complex methods involving the external API component.

API This component provides some of the Alloy Analyzer features as a simple Representational State Transfer (REST) service which formats its responses in JavaScript Object Notation (JSON). Just like Alloy, it is written in Java to ensure full compatibility. Its main function is to analyze models provided by the framework in order to provide results, namely counter-examples. One key fact about these requests is that they are session based, since it is desirable to keep an analyzer instance alive as long as the user has the web-service opened on a given model. A brief description of its HTTP interface is presented in Table 2.

[1] https://www.meteor.com/

Method	Parameters	Description
POST "/getInstances"	• The session's unique identifier • The parent session's identifier • The Alloy model • The command to execute • The number of instances required	This method parses and computes the provided Alloy model and generates its results in an internal temporary state (i.e. a session) recognizable by the provided unique identifier. Subsequently it will generate the requested number of Alloy instances from the session and serialize them within the method's answer, along with any errors or warnings that came up. Future requests with the same identifier will skip the parsing and solving and provide the next set of instances. Sessions are deleted when they have been idle for 10 minutes or when they are identified as the parent of a new session.
POST "/getProjection"	• The session's unique identifier • The target instance index • The required projection types	This method provides Alloy's projection feature, i.e. a method for changing and applying theme features on instances generated by the previous method. It requires the session's identifier and the instance's index in order to retrieve the instance, which will then be projected based on the types of atoms specified, serializing each projection in the method's answer.
POST "/validate "	• The Alloy model	This method simply validates (syntax and type-checking) the provided Alloy model, returning either a success code or the serialized error thrown by the compiler.

Table 2: API's HTTP interface

This state-based solution offers several performance benefits, since by maintaining the models in memory it is able to skip parsing and instance computations on several requests. However, it also introduces a set of major challenges on the deployment. One of the primary drawbacks is the inability to scale the application to a large amount of users and also to ensure its fault tolerance. This is essentially due to the difficulty of replicating state based applications, since to maintain the integrity of the sessions each user would have to be mapped to the exact same replica in every request. As a result, a replication of this component would require a sophisticated solution to maintain an efficient load balance of the replicas which would be made even more complex when accounting for failure resilience.

Database The application's database, implemented over MongoDB[2], is mainly responsible for storing the data handled by the web-service, namely the saved models and their respective instances and counter-examples. It is comprised of the collections **Link**, **Instance**, **Navigation**, and **Model**, which are described in the dictionaries presented in Tables 3, 4, 5, and 6, respectively.

Link				
Field	**Format**	**Optional**	**Example**	**Description**
_id	String	NO	`3cy7jaB4ESqdb2txK`	Unique identifier of the object
private	Boolean	NO	`false`	Indicates whether the link is private or public
model_id	String	NO	`2Htt7FYT7dY9ETGRX`	Reference to the *Model* object.

Table 3: Dictionary of the collection *Link*

[2] https://www.mongodb.com/

Instance				
Field	**Format**	**Optional**	**Example**	**Description**
_id	String	NO	`3cy7jaB4ESqdb2txK`	Unique identifier of the object
model_id	String	NO	`2Htt7FYT7dY9ETGRX`	Reference to the *Model* object
cmd_i	Int32	NO	`4`	Index of the executed command
graph	JSON Object	NO	`{ "types":[...], "sets":[...], "rels":[...] }`	The serialized Alloy instance
time	String	NO	`2020-12-13 23:28:11`	Serialized date of the instance

Table 4: Dictionary of the collection *Instance*

Navigation				
Field	**Format**	**Optional**	**Example**	**Description**
_id	String	NO	`zKAYz8BCDmHKgNoSx`	Unique identifier of the object
time	String	NO	`2021-03-19 17:28:11`	Serialized date of the saved instance
operation	Int32	NO	`0`	Enumeration for the navigation action: 0 for forward, 1 for backward
instIndex	Int32	NO	`6`	The index of the navigated instance
model_id	String	NO	`2Htt7FYT7dY9ETGRX`	Reference to the *Model* object

Table 5: Dictionary of the collection *Navigation*

Model				
Field	**Format**	**Optional**	**Example**	**Description**
_id	String	NO	`zKAYz8BCDmHKgNoSx`	Unique identifier of the object
time	String	NO	`2020-12-13 23:28:11`	Serialized date of the model
code	String	NO	`var sig File{ var link : lone File ...`	The complete code of the saved model
derivationOf	String	YES	`dvhCng5AdxC8MqjFy`	Reference to the parent model from which this one is a derivation
original	String	YES	`9jPK8KBWzjFmBx4Hb`	Reference to the root ancestor model
sat	Int32	YES	`1`	Status code which classifies the executed command as Satisfiable (1), Unsatisfiable (0) or Erroneous (−1)
cmd_i	Int32	YES	`4`	Index of the executed command
cmd_n	String	YES	`inv5`	Name of the executed command
cmd_c	Boolean	YES	`true`	Specifies if the command was a check (`true`) or a run (`false`)
msg	String	YES	The name "Traash" cannot be found.	The message presented to the user for the erroneous execution
theme	JSON Object	YES	`{"generalSettings":{...}, "relationSettings":[...], ...}`	The customized visualization theme, if applicable

Table 6: Dictionary of the collection *Model*

Although this database schema supports all of Alloy4Fun's functionalities, it has two noteworthy flaws regarding some field formats. The first pertains to the choice of representing object identifiers as strings, as opposed to utilizing *MongoDB*'s `ObjectId` types. As a result, the schema places the responsibility of generating and maintaining unique keys on the application instead of using the database's automatic generation feature. This process can prove cumbersome and error-prone, especially in scenarios where multiple applications interact with the database.

The second and more substantial flaw concerns the representation of the time fields as strings instead of *MongoDB*'s *Date*, resulting in two different issues. Firstly, by forcing each program to work with strings it requires specialized string handlers which are error prone thanks to the second issue, which is the possible inconsistencies in date serialization. By using strings, to ensure consistency within the database, we are trusting that each database writer uses the same date serializing technique. Unfortunately, this is not necessarily the case for Alloy4Fun's *meteor web-service*, as it uses the date formats of its host operating system, which can change between deployments. Currently Alloy4Fun's public dataset [22] contains two date formats as result of this flaw, a "month/day/year" pattern with a 12 hour clock (example: `2/12/2023, 8:23:47 PM`) and a "year-month-day" pattern with a 24 hour clock (example: `2020-12-13 23:28:11`), which compromise the data consistency.

The essential piece of the database is the *Model* collection. Each saved model (e.g. a submission to an exercise) pairs its source code with several context data such as the theme used, the command executed, the resulting state (i.e. whether the submission's execution was valid, invalid or erroneous), a reference to the root ancestor model and, most importantly, a reference to the model it was derived from.

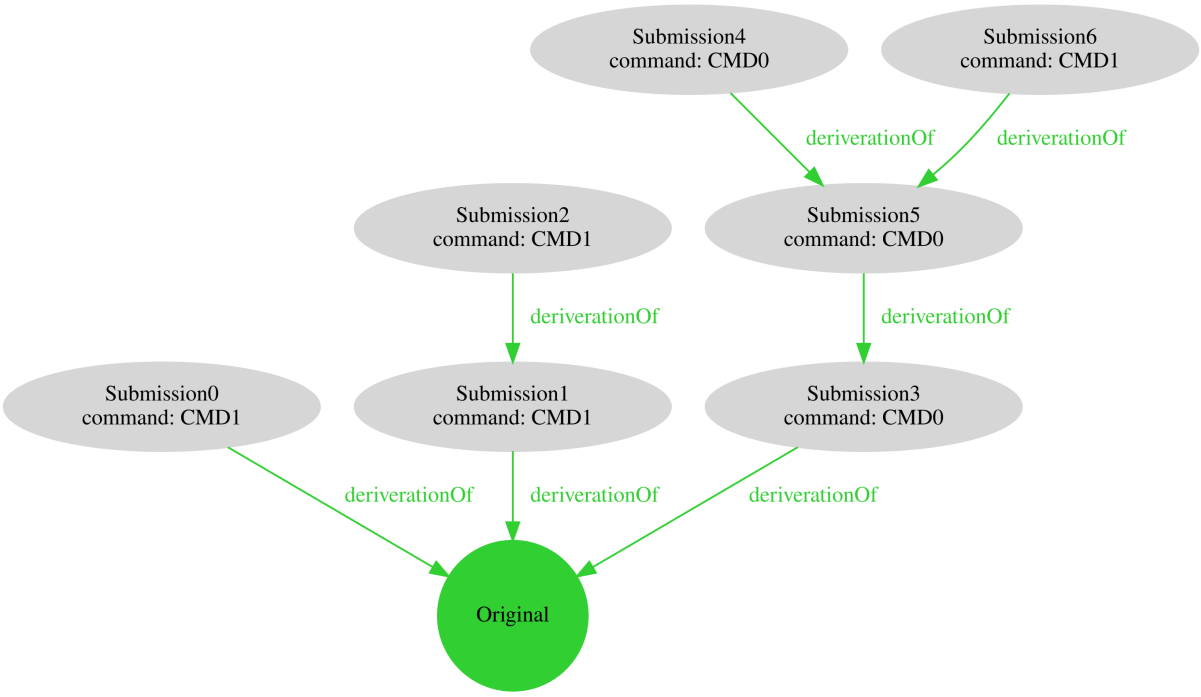


Figure 14: Visualization of a derivation tree

It should be noted that every time secrets are introduced in the public view of a model already containing secrets, a fresh derivation tree is created. This means that in each branch of a derivation tree there is at most one model with secrets. The root ancestor model is this model with secrets, if it exists, or the root of the tree if there are no models with secrets. With these references, the database effectively stores a derivation tree for each exercise, where each possible path from the root ancestor model to a leaf mostly consists of a student's attempt in solving the problem. A simple visualization of one of these trees is presented in Figure 14, were we have three different branchs of submissions. The first two branches, composed of submissions 0 to 2 are likely to have been created from two different attempts at solving the original exercise. The last branch, composed of submissions 3 to 6 entails that a *permalink* was created on submission 5, which was accessed and modified at least one time, resulting in the observable branch. Due to the anonymous nature of the links, it is impossible to determine whether the branched submissions were created by the author of the *permalink* or if it was the result its distribution among other individuals.

The system stores the *permalinks* for accessing each shared model in the **Link** database collection described in Table 3. These links are associated with privacy settings, distinguishing them as either private or public. This distinction determines whether the *backend* displays or conceals the secrets of the accessed model.

4

State of the Art

Throughout this chapter we will present a variety of program hint and repair systems and their corresponding methodologies. We will begin by introducing an abstract hint generation framework which broadly describes the data flow across most recent automatic feedback systems. Afterwards, we will present several hint generation systems in the context of this framework, grouping them in four types of feedback methodologies: Fault Localization, Synthesis-Based, Curated, and Data-Driven. Each methodology is explained in relative detail with an extra focus on its strengths and compromises. We conclude by exploring the application of text generation in the context of hint generation systems.

4.1 The HINTS Framework

In recent years, there has been a proliferation of diverse approaches for generating hints in programming languages [26]. These approaches encompass a wide spectrum of unique ideas and techniques, spanning various research domains, including behavioral and state repair [27], machine learning, data mining [2], among others. Consequently, the hints generated by these approaches exhibit several degrees of diversity, manifesting as error highlights, recommended actions, explanatory comments, and various other forms of support. Despite this, there is a consistent data flow pattern that underlies most methodologies. We always begin with a step where we analyze user submissions, which provide information that can be regarded as intermediate unprocessed hints. The program sequentially processes hint intermediaries through various components, which either reduce or transform the crude suggestions into useful hints for the user. This data flow commonality does not occur by chance; rather, it constitutes the fundamental underpinnings of any hint generation system.

The Hint Iteration by Narrow-down and Transformation Steps (HINTS) Framework [26] is an architecture which abstractly depicts all of the common steps among many automated hint generation systems. A visualization of this framework is presented in Figure 15.

The process starts with a pool of data as an input, which is commonly referred as hint data since it is either composed of or used to generate hints. These can include data-sets produced by peers, property based tests for the models, direct user submissions, or any other static feature that could be useful to the

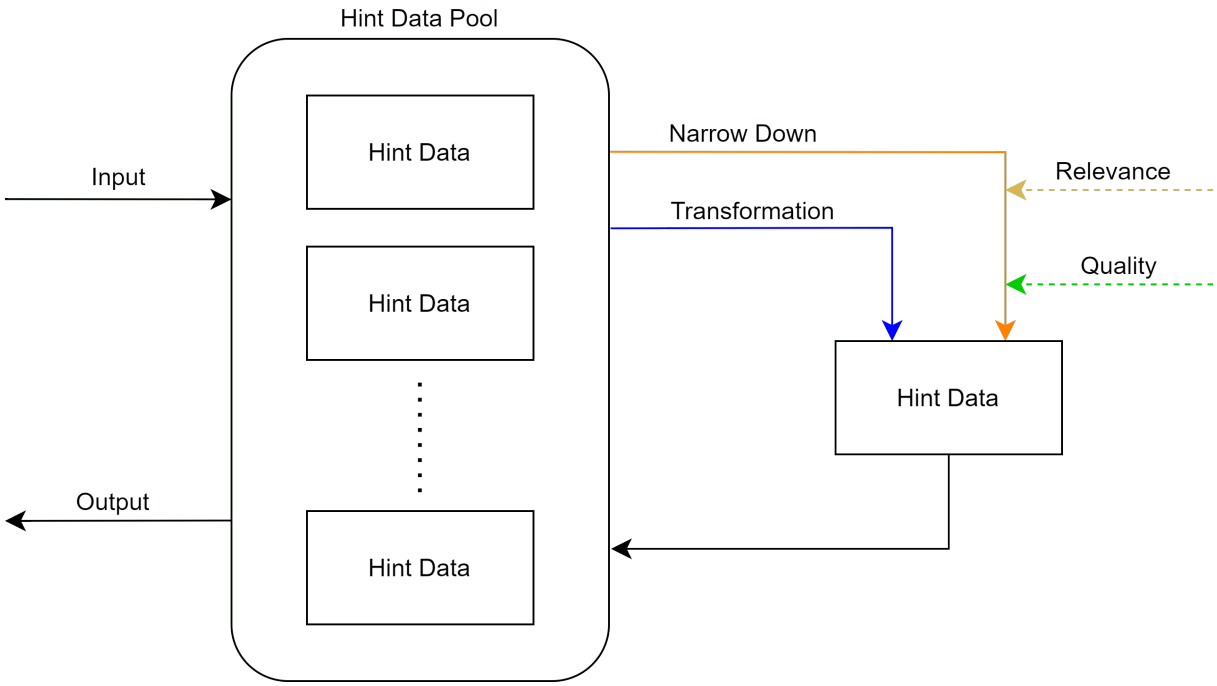


Figure 15: The HINTS framework

system. Once a data pool has been established, a hint system will then perform one or several processing steps to generate new hint data, which can be classified within two types of actions.

- A *transformation* action as implied involves changing the way existing hints or data are represented. Some examples of this are partitioning hint data, restructuring an existing entry, standardization, or discarding duplicates. For example, an Alloy model can be broken down into its signatures, predicates and functions, over which some could be discarded while other could be modified to allow for custom evaluations.

- A *narrow down* action involves selecting subsets of hint data from the original pool. These selections take into account two different factors: the first one is the *relevance* of the hints when applied to a student's program; the second one is their overall *quality* for the situation, i.e. their capability to help the student produce a correct solution.

Each time new hint data is produced it is added to the original pool. Finally a hint system delivers the student one or more subsets of hint data which represent the resulting hint's information.

The HINTS framework does not imply any order or restriction in which these steps must be taken. However most approaches need to make compromises in this regard, for example a system can become limited to its initial input data if it has a requirement to assure the integrity of their hints during their lifespan. Other examples include approaches that do not require any *transformation* to be made upon its data, being thus able to supply hints directly from the programs.

Over the following sections we will be discussing several examples of well developed hint generation systems. As previously implied, these systems can be abstracted and described within the HINTS framework and thus several of these concepts will be recurrent through out our presentation.

4.2 Fault Localization Hint Generation

Fault localization systems stand on the boundary of what can be defined as a program hint or repair system. Their main focus is to identify defects in a program's source code, classifying the detected faults and present the user with suggested actions, i.e. hints. The purpose of these suggestions is to present the user with options in achieving a state free of faults. This can be valuable in a debugging context but it does not necessarily aid in comprehending or resolving the underlying engineering issues [45]. Overall, their general processes are very similar: faulty models are put through a series of validating artifacts which isolate suspected faults within the original model; the suspicions faults are then presented directly to the user, along with predefined suggestions for solving them.

The main distinction between Fault Localization systems is their fault identifying techniques. One of the oldest techniques is program slicing, a technique that abstracts a program into a reduced form by statically or dynamically deleting irrelevant parts from it. Numerous studies have explored this technique [48]. A few examples are eXVantage [46], a coverage-based testing system focused on minimizing performance overhead, and JSDiagnosis [47], an AI-based dynamic slicing system. Another common technique is spectrum-based fault localization, a technique which identifies faults by comparing the execution traces of failing and passing program executions, classifying each identified difference with a fault suspicion metric. A key challenge in these systems is ensuring their metrics achieve an effective balance between precision and coverage of their fault detection. Similar to the previous techniques, there are also numerous works on this topic [39], some examples include LAURA [1], a system that applies program transformations to validate the executions, and PROUST [18], which contains a set of predefined *programming plans* which are executed on the user input in order to generate a diagnostic of the bugs found in the code.

4.3 Synthesis-Based Hint Generation

Synthesis-based tutoring systems rely on generating new programs or models from a user's submission in order to provide their hints. These generation processes vary depending on the implementation, some might use unit or property-based tests, while others will exhaustively mutate the prompted models into new ones. In the specific case of Alloy there exists already several of these tools, such as ARepair [42], BeAFix [8, 7], and TAR [10]. The techniques used by these tools are similar. Upon receiving a student's model the system will then begin synthesizing new submissions from the wrong ones. The used tactic depicts the main difference between the mentioned implementations.

ARepair employs the fault localization tool AlloyFL [43] to identify a set of suspicious declarations. Following this, the declarations undergo systematic modifications through a predefined set of rules referred to as *mutators*, creating an array of mutated declarations. This procedure repeats until either the computed declarations passes the tool's fault-localization tests or when it meets the limit of its search criteria. One issue with ARepair is that its fault localization tests may not always be enough to properly validate the synthesized model. Consequently, there is a risk of incorrect fixes being generated under the assumption that they are valid. BeAFix is similar to ARepair as it uses FLACK [50], a fault localization tool which benefits from Alloy's counter-examples, to extract and mutate a set of suspicious declarations from an incorrect model. In contrast to ARepair, it leverages the oracles of the individual models to validate each synthesized model, which eliminates the potential for falsely validated models.

Unfortunately, both ARepair and BeAFix only support the non-temporal version of Alloy. This issue was the main motivation to the development of the Temporal Alloy Repair (TAR) tool, which was the first technique for fixing Alloy 6 first-order temporal logic specifications. The technique employed is designed to exhaustively preform syntactical mutations until the tool either finds a solution or it reachs a limit imposed by the developer. When a new model is synthesized it is stored as an Abstract Syntax Tree (AST), a structure representing the core model as seen when parsed by Alloy's compiler. After computing the repaired model, the resulting AST, along with the actions that produced it, can be employed to either rectify the original model or provide hints to the user. A visualization of the typical architecture of these systems is presented in Figure 16.

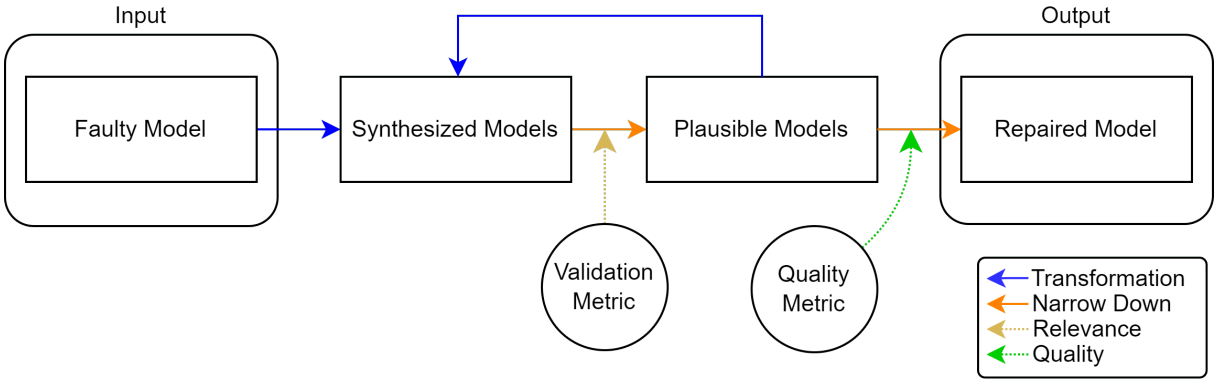


Figure 16: Synthesis-based hint generation

As we can see, there are two types of artifacts which influence the main processing steps. The validation metrics are mostly specified by the teacher, which generally consist of an oracle or unit tests that can be used to check the correctness of the synthesized models. Additionally, these may also include optimizations, such as TAR's counter-example pruning techniques. Some quality metric is then used on the process of selecting a final hint in the form of a tree edit. This can be as trivial as selecting the first correct model found or more robust ones such as scoring each synthesized model based on the number of edits required to obtain it from the original incorrect one.

Overall these methods can be reliable at not only generating hints but also at ensuring their quality, i.e. their ability to help the student solve their problem [10]. However their main drawback tends to be their performance. Most implementations can often take minutes to generate a hint, which is very unpractical in a tutoring environment where the student wants to receive instant error/success feedback.

4.4 Curated Hint Generation

This technique produces hints by converting sets of model solutions written by the teachers in an oracle to generate a set of steps leading to a solution, labeled *program strategies* [13]. These strategies are then used to examine the students submissions in order to find the ones that can help them with the next steps of the development. To achieve this, the student's submission must be first normalized in order to be more likely to match with any of the stored strategies. If the match is successful, its corresponding steps are retrieved, as well as the teacher annotations which are directly related. A few examples of these systems include a Haskell helper named *Ask-Elle* [12], *MistakeBrowser* [15], and *CoderAssist* [19]. A nice advantage of these system is that their hints can be extremely robust and rich. Since the annotations are defined by the teachers they can include anything, from simple messages, to documentation references, and possibly URLs or other examples. A visualization of the typical architecture of these systems is presented in Figure 17.

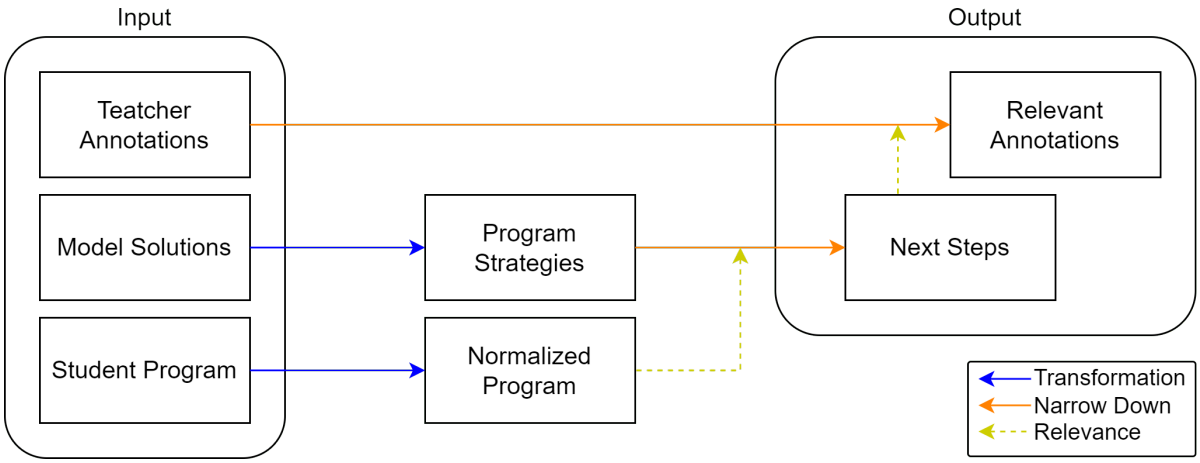


Figure 17: Curated hint generation

Some variations of this technique exist, such as AutoTeach [4], where the students program is not even used to compute the hints. In this case the system will merely provide hints to a problem sequentially as they are requested by the user, strictly following the program strategies chain of steps.

The main disadvantage of this kind of systems is their inability to deal with unexpected mistakes. Even when the system finds a match to an erroneous submission, and subsequently suggests program strategies and annotations, in the general case there are never guarantees that the student will follow them. This can result in situations were the student goes "out of bounds" and can no longer be helped by

the hint system. This problem can be solved with edit-based approaches, where the teachers add models and annotations to new unmapped strategies, however such a system would need to be supported by teachers throughout its entire lifespan in order to maintain its usefulness.

An interesting note regarding this disadvantage is if a students' solution partially matches a known strategy, the precision, and consequently the quality, of the final hints tends to degrade. For these cases, several hints are filtered based on the relevance for the provided solution. This filter dictates the trade-off between quality and availability, since stricter filters will provide less hints with better quality and vice-versa. This trade-off is essential in the overall system's success, since although an automated system main purpose is to be able to provide hints to the end user, poor quality hints have been proven to discourage them from seeking help from the tool [37].

4.5 Data-Driven Hint Generation

A hint generation system is considered to be Data-Driven when it relies on historical data originated from the users to generate and provide hints. This data is composed of any object capable of classifying a user's performance, such as their submissions, their program outputs, or their overall progression within a problem.

4.5.1 Hint Factory

One of the first data-driven tutoring systems was the "Hint Factory" [40, 5]. This system uses historical student submissions and a Markov Decision Process (MDP) for hint generation. For every programming exercise, it starts by abstracting and aggregating the historical submissions into objects know as states. These states are then converted to nodes of a connected graph, where the edges correspond to steps that the students took when solving the exercise. Between all the parsed states, there can be many that correspond to different solutions for the exercise. After the graph is built a teacher can provide hints associated with each edge. When a student submits a solution the system will abstract the data and attempt to match it to an existing state. If no match is found the generation process is canceled and no hint is given, otherwise the system will then calculate the best possible path between the incoming state and a solution with the help of a MDP. The resulting path is then used to generate and provide an optimal hint to the student. A visualization of the the architecture of this framework is presented in Figure 18.

Hint Factory uses a MDP for selecting the best quality path, however these systems can also rely on many other options for scoring and choosing the best path available. Overall, there is not any definitive or standardized method for this step, and generally most algorithms tend to preform similarly in helping the students progress [35]. This is mainly due to the human factors peculiar to each student, such as, for example, their experience with the language. These factors are extremely hard to predict, and thus they are always a source of deviation between the expected student's progression and the actual performance with the provided hints.

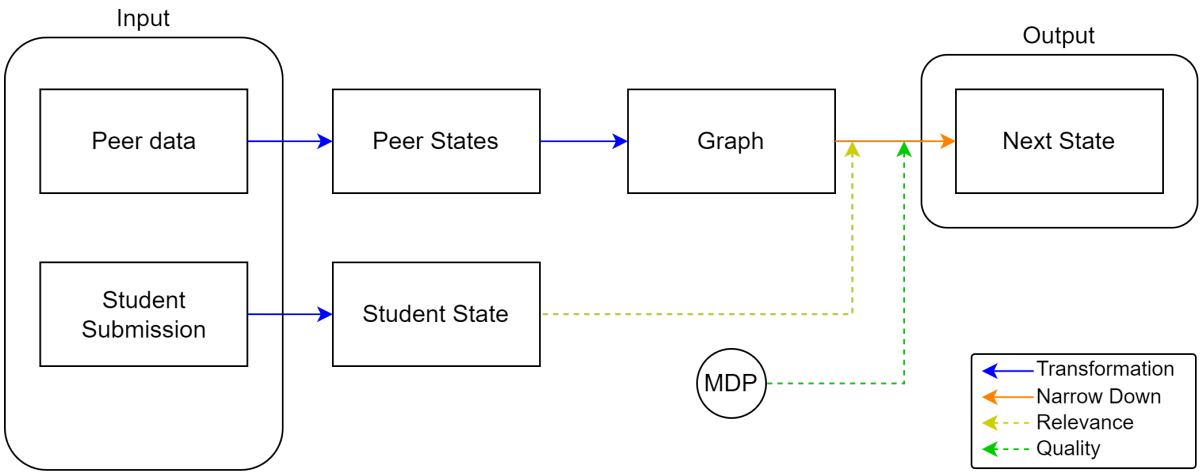


Figure 18: Hint Factory architecture

A key decision in a Hint Factory system is how to represent the states, and therefore the nodes of the graph. These should allow for the abstraction of many fundamentally similar submissions while not compromising on their information, and consequently the produced hints. A state that cannot abstract many submissions would become useless since it will rarely be matched with anything, while a state that abstracts too many submissions loses the ability to provide precise and useful hint data to the user. As a result, several approaches based on Hint Factory [31, 24] tend to have some trade-off between these two factors, dictated by their established abstraction techniques.

4.5.2 Intelligent Teaching Assistant for Programming

The Intelligent Teaching Assistant for Programming (ITAP) [38] is a Python tutoring system which expands the previously presented Hint Factory by introducing the ability for the graph to evolve over time. As such, instead of an initial set of historical student data, the only thing the algorithm needs to start is a reference solution or a way to identify if a solution is correct. A visualization of its architecture is presented in Figure 19.

After starting, similarly to the Hint Factory, each submission will be abstracted and converted into a state. Since ITAP is developed for Python, these states contain simply what the language's compiler will act upon (i.e. its grammatical tokens). With the resulting state it will then attempt to find an existing match in the current graph. However, unlike the Hint Factory, if no match is found, the system will incorporate the new discovery into the current graph instead of just failing. This process involves a synthesis procedure similar to the one introduced in Section 4.3.

When a state is not matched, the system will start synthesizing new states to find solutions to the problem by using a set of mutating rules. An advantage that this process has over a conventional synthesis-based hint generator is the ability to skip the computation of several states when a mutation that happens to be already registered in the graph is found. After calculating the paths to the solutions, it will then score them based on two factors: the amount of mutations they introduce on the submission and the frequency

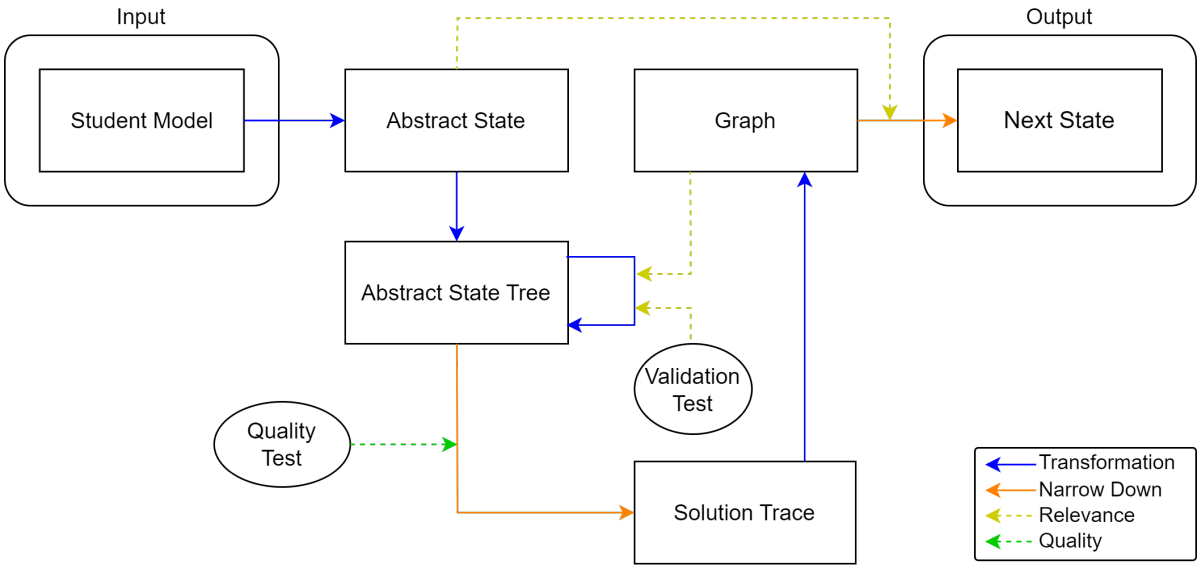


Figure 19: ITAP architecture

in which the path states were queried during the process. The next state with the highest score is then incorporated into the graph, and thus any future request involving the same submission will only need to directly consult the graph, skipping the whole synthesis operation.

Finally, with the solution procedure obtained from either the data-driven model or the synthesis process, the system will finally narrow down the meaningful hints to be delivered to the student. Unlike the Hint Factory, this system is capable of a fully automatic hint generation, based on predefined template which is capable of generating phrases which depict the code location and the required actions that the student should follow.

4.5.3 Hint Generation with Deep Learning

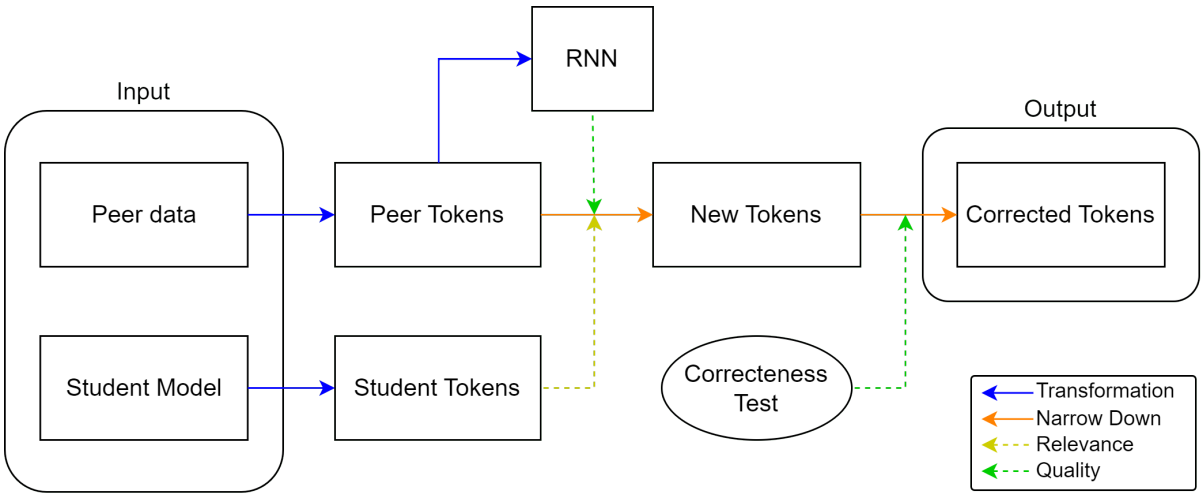


Figure 20: Hint generation with Deep Learning

Deep Learning based systems, such as SYNFIX [6], DeepFix [14] or BIFI [49], train Recurrent Neural Networks (RNNs), usually with reinforcement learning, for the purpose of correcting segments of a student's submission. To achieve this, the system must process a data-set of existing submissions into smaller objects know as tokens, which represent the relevant pieces of code from the original submission. These tokens are then used to train a RNN. When a student requests help with a submission, the system will tokenize the given code and present it to the RNN that processes the information into new or alternative tokens. These replacement tokens are then tested against the original submission and delivered to the user, provided they solve the original problem. A visualization of the system is presented in Figure 20.

These systems present two differences when compared to the previously described data-driven systems. The first one is the output object, as the previous systems produce solution procedures and this technique produces correction tokens. As a result, when targeted for hint generation, an additional technique for extracting the solution procedures from each token must be employed. The second distinction would be the evaluation method, the previous systems use graph-like structures with submissions to generate hints while this method resorts mostly to deep learning models. Besides these factors, the whole process is essentially the same: we start with peer data that gets processed into submission data (states/-tokens), that are then used to setup a hint generation technique (graph/RNN). Upon being prompted by a student, the system will transform the submission data onto the established data structure, and then generate a set of possible solutions/hints which are validated and provided back to the student.

4.6 Text Generation in Hint Systems

Text generation plays a pivotal role in most hint generation systems, since it is one of the most dynamic and versatile ways of conveying information to the users. Using text we can generate a variety of suggestions such as step-by-step guides and informative/analytical prompts. The process of generating text for hints however is challenging due to the intricacies of natural languages. Generating coherent and relevant text hints requires sophisticated grammar, semantics, and syntax systems, which are not infallible due to the general ambiguity and subjective nature of most idioms. Generating textual hints typically follows two distinct methodologies: Heuristic Techniques and Machine Learning Techniques.

4.6.1 Heuristic Techniques

These techniques rely on an explicit logic in order to generate the messages, which consequently makes them more understandable and easier to implement and validate. However, they scale badly, since they tend struggle when dealing with large amounts of data or highly complex problems, as manually crafting rules for every situation becomes cumbersome. Some examples of these methodologies include:

Rule-Based Generation [25] involves having a human specify a set of predefined conditions (known as rules) and the content associated with it. When attempting to present a hint, the program will sequentially test the chosen action on known rules, displaying the content of the successful matches to the user. Nonetheless, this method can become unusable when dealing with complex or diverse data, since the sheer volume of situations can make creating rules for every possible scenario impractical.

Template Filling [51] is an expansion of the previous approach where, instead of providing fixed content, the human now specifies templates which are composed of predefined placeholders that are dynamically replaced with tokens derived from the matched action. This implementation improves on its predecessor by allowing for a greater customization of the hint. However, it is more difficult to implement and it still has the previously mentioned problem.

4.6.2 Machine Learning Techniques

These techniques leverage preexisting data to automatically learn how to generate the required text. They tend to be harder to understand and validate since their generation logic varies according to the training data, making them extremely difficult to interpret and explain. Nonetheless, these techniques have several benefits, namely their adaptive capabilities allow them to effortlessly scale to multiple text variations, increasing the overall quality of the hints. Some popular examples of these techniques include:

Markov Chains [28] are formal methods that use probabilistic models to predict the next word of a phrase based on the current one. These models are built by statistically analyzing preexisting text and while being relatively simple they can produce surprisingly realistic text, but tend to lack long-term context and coherence in their results.

Generative Pre-trained Transformers (GPTs) [9] are deep learning models based on the transformer architecture [41]. They are pre-trained on a large corpus of text data and generate text by predicting the next word in a sequence using its preceding text. As a result they generate flexible and adaptive text with the downside of occasionally producing convincing but factually incorrect content.

Generative Adversarial Networks (GANs) [29] consist of two neural networks that are trained together, a text generator responsible for creating new content, and a text discriminator that evaluates how close the generated content is to real data. By making both networks compete with each other we can obtain exceptional text generating models, however this isn't an easy process to setup since their duality nature also makes them extremely difficult to train.

5

Alloy Specification Assistant

In this chapter we will present a new hint system for Alloy4Fun, which we named *SpecAssistant*, that relies on existing data from the users in order to generate its output, being thus classified as data-driven. The overall process is similar to the Hint Factory [40] approach, as it starts by processing historical data into unique states, which are then assembled into a Submission Graph, upon which a traversal policy will be used to select the next state, and the respective hint. Upon receiving an invalid submission to a challenge, it will process the user's formula and use it to match a state of the submission graph. The next state is then determined by the predefined policy rules which assess the quality of the candidate states. Finally, the state's information is processed into a hint, composed of text highlights and custom messages. A graphical view of this architecture is presented in Figure 21.

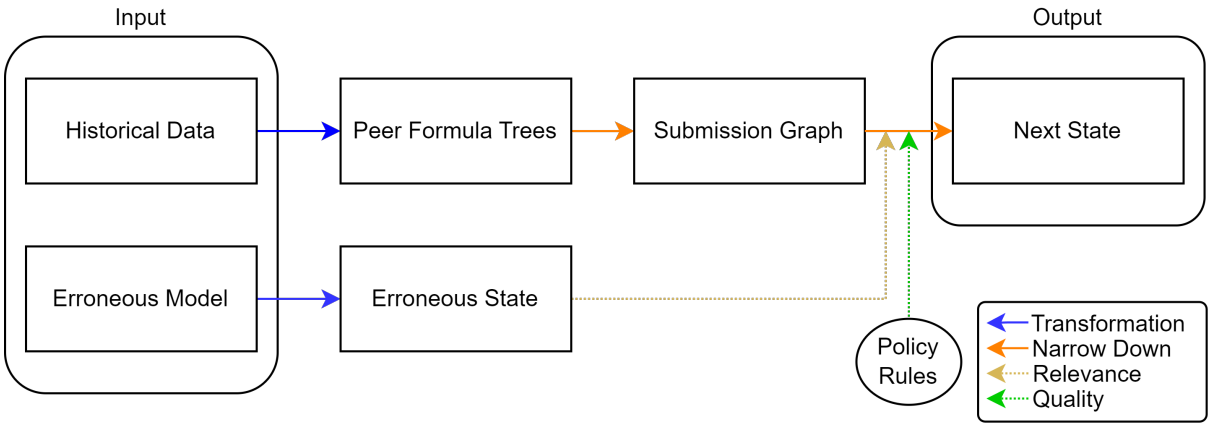


Figure 21: SpecAssistant architecture

In this chapter, we delve into the intricacies of the developed hint generation system. We will begin by describing the fundamental concepts and processes used by the system, and how these can allow us to generate hints. We will then describe the implementation of these techniques on Alloy4Fun, elaborating on aspects such as the framework and libraries used, the data formats, and the user interface.

5.1 Core techniques

This section aims to describe the core techniques of our system. We begin with a description of our formula comparison procedures and their impact in the system. Afterwards, we proceed to characterize how the peer data is stored in the system's data model. Finally, we describe how the data model is processed into the final hints which are displayed to the user.

5.1.1 Formula Comparison

Before presenting the core techniques we must first describe the data that these were designed to handle. Every process in our system revolves around Alloy formulas which were specified by the users in their attempts to solve challenges in a exercise. These formulas can be parsed into Abstract Syntax Trees (ASTs), which detail the hierarchical structure of the syntactical elements within the formulas. In Figure 22, we depict the AST of the formula `all s: Student | s in Human` , which asserts that every student is a human. It is composed of a quantifier that is described by a declaration and a body. The body is defined by applying the `in` operation to the variable `s` and the set `Human`.

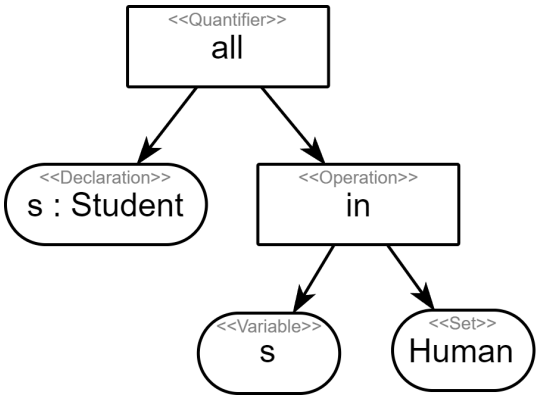


Figure 22: Example of an AST

When handling formulas, we can compare them in three distinct ways: through their lexical representation, their syntactical representation, or their semantic meaning. Each approach comes with its set of benefits and compromises, usually involving a trade-off between the final hint's availability and quality. These different approaches are described as follows.

Lexical Comparison A lexical approach views and compares each formula directly as written by the user. As such it allows the system to precisely tailor each hint to each submission, for example the system can directly mention the user's custom variable names or highlight portions of the text in order to draw the user's attention to them. Unfortunately, this greatly decreases the availability of the hints, since providing the user with a hint requires the submission to exist in the database exactly as written by the user, which is very unlikely thanks to all the personal choices the users can instill on their submissions.

Syntactical Comparison A syntactical approach discards most of the user's personal decisions, such as parameter names and order, by focusing only on the meaning conveyed by the arrangement and structure of keywords in the formula. Intrinsically, this abstracts multiple submissions under the same AST, which increases the chances of providing the user with a hint. This comes at a slight cost in the quality, as it results in the loss of certain contextual details within the formula elements. These details may include custom names, macros and other "syntactic sugar" constructs, file locations, among others which are valuable for providing more specific and customized hints. However, it is possible to build translations between lexical and syntactical formats, which allow for approaches with the best of both worlds. An example of two syntactically equivalent formulas can be found in Listing 5.1.

```
some r1,r2 : Road | r1=r2   ≡   some x,y : Road | y = x
```

Listing 5.1: Syntactical equivalence example

Semantic Comparison A semantic approach focuses solely on the meaning of the formula itself. This allows for an large degree of abstraction, which should increase the systems ability to generate hints. Regrettably, this semantic approach removes almost all of information that is traceable to the original submission, and thus it cannot be used to generate contextual hints, which severely degrades the system's quality. Additionally, this approach also comes with some performance issues since semantic evaluations would require the use of Boolean Satisfiability Problem (SAT) solvers to check the equivalence, which are notoriously slow. An example of semantically equivalent formulas can be found in Listing 5.2.

```
adjacent = ~adjacent   ≡   all r : Road | r in r.adjacent
```

Listing 5.2: Semantic equivalence example

Our system mixes the lexical and the syntactical approach, by having each formula inspected and compared syntactically using lexical formats. This is achieved by normalizing the original formula to a predefined lexical standard, so that when compared directly only syntactical differences will be visible, allowing the system to take advantage of simple lexical comparisons without compromising the abstraction power of the syntactic approach. The used normalization technique can vary across implementations and generally dictates a trade-off between quality and availability. Our normalization technique implements three rules:

Commutativity The sub-expressions of a commutative operation are arranged following some order. For example, in the expression `B and A`, if `A` precedes `B` then its normalized version would be `A and B`.

This rule is applied in formulas using the following operators: intersection (&), union (+), equal (=), different (≠), and (∧), or (∨), equivalent (`iff`), and disjoint (`disj`). When dealing with these operators,

our normalization process organizes the arguments of each operation based on their lexicographic order. We have opted for the lexicographic order as it effortlessly ensures a consistent ordering across all formulas.

Equivalence If two different operators are equivalent then only one should be used in the normalized version. For example, if we consider that `>` will be the inequality operator of choice, then any expression containing $\leq$ would be rewritten with it. For example, `A`$\leq$`B` would be normalized as `not B<A`.

This rule is primarily focused on the transformation of inequality operators. When confronted with the operators $\geq$ and $\leq$, our implementation replaces them with the operators `<` and `>`, respectively, while simultaneously switching the order of their arguments.

Identifiability Identifiers (such as function parameters or quantified variables) must be normalized under the same naming scheme. For example, the syntactical equivalent formulas `all x: A|x in B` and `all a : A |a in B` should have the variables `x` and `a` rewritten to the same name. The assigned names can be defined as wished by the developer, so long as it is an alphanumeric identifier which assures that syntactically equivalent formulas use the exact same names on their identifiable nodes.

For our application, we only need to update identifiers specified by users. Specifications from default Alloy modules are equal across all environments, which as a result ensures their consistent naming scheme. Alloy formulas include four constructs that introduce identifiers: functions, predicates, `let` blocks, and quantifiers. Functions, predicates and `let` constructs introduce identifiers to denote one or several formulas. The simplest way to handle these cases is to expand these identifiers to their complete definition, effectively removing them from the AST.

For quantifiers we rename each variable identifier with the concatenation of the prefix `ref` and the index of its declaration. Before this renaming, we also reorder variables according to the lexicographical order of the respective types, and refactor disjoint (`disj`) clauses into the quantified formula. This reordering process also involves merging sequential quantifiers that have the same operator. A few examples of this normalization process are depicted in Listing 5.3.

```
all x: X | all n: x.next| n ≠ x
    ⇝  all ref0:X, ref1: ref0.next | ref0 ≠ ref1
all u1: U | some u2: U | u in u.next
    ⇝  all ref0: U | some ref1: U | ref1 in ref0.next
all disj n1,n2 : N | n1 in adj.n2
    ⇝  all ref0:N, ref1:N | disj[ref0,ref1] ⟹ (ref0 in adj.ref1)
```

Listing 5.3: Examples of normalized quantifier formulas

An additional issue that is addressed by the normalization is the treatment of constant `true` formulas. Alloy lacks a dedicated syntax for expressing these constants directly, but they are equivalent to empty code blocks. When custom empty functions or predicates are invoked within formulas, expanding their definition can introduce these empty code blocks into an AST, which is syntactically incorrect. So we have implemented some rules to remove them. For example, in `and` operations we simply drop the empty block, and an expression `if condition then x else y` will be rewritten to its `then` branch when presented with an empty block as the condition, among others.

5.1.2 Submission Graph

The data model is the main component of a data-driven system. In our case it is the direct byproduct of the peer data and it plays the pivotal role in the system's output. Our system will process this historical data of each challenge into a model named **Submission Graph**, a structure which shares properties with Control Flow [3] and Decision Graphs [32]. This graph monitors the decision-making of multiple agents in their pursuit of solving a specified challenge, by aggregating the control flow that illustrates how their actions and choices helped them progress through their difficulties. An abstract depiction of this model is presented in Figure 23, which will be analyzed and explained throughout the following paragraphs.

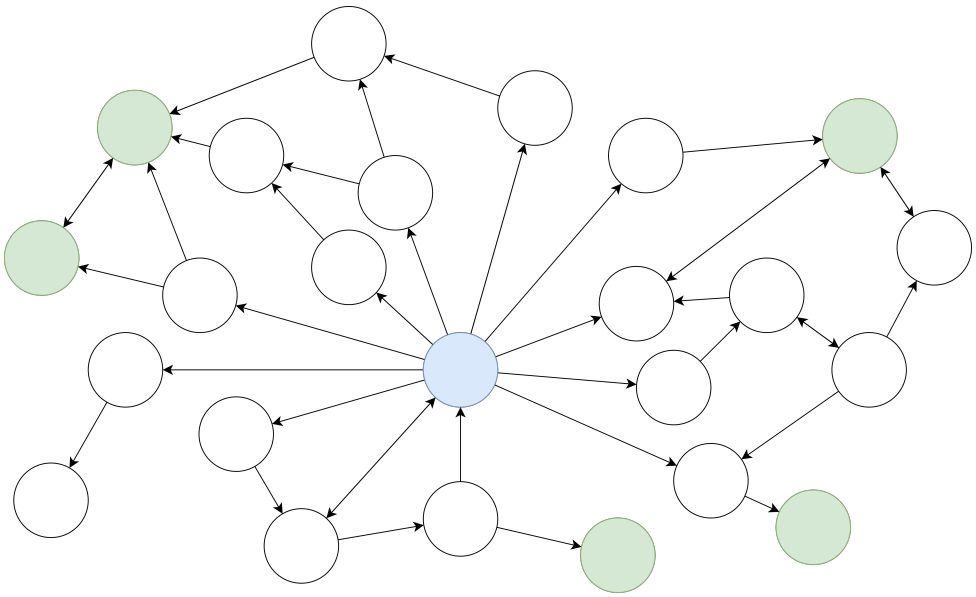


Figure 23: Example of a Submission Graph

Similar to generic graph structures, the representation under consideration is depicted using nodes and edges. These correspond to the two essential components of our model, **States** and **Transitions**, which are described as follows:

States S – States correspond to the Alloy formulas (ASTs) submitted by users while attempting to solve a challenge and processed by our previously described normalization technique. In our system, we also register a few key attributes of each state: the formula's validity (i.e. indicating whether

it solved the challenge), its anticipated truth table size (latter referred to as its complexity), the frequency the formula was submitted, and the frequency the formula was modified (i.e. the sum of the out-edge's frequencies).

Transitions T – Transitions capture the sequence of states users followed while solving challenges. Similarly to states, for each transition we register: the frequency, the minimum Tree Edit Distance (TED) between its states, the proportion of times that a transition was chosen over its competitors, i.e. the peer transitions that share the same source state.

Our system creates a different submission graph for every challenge of every exercise. These graphs tend to have a similar shape to the graph of Figure 23, which steams from the format of the derivation tree structures that originate them. Typically, there is a central node, here colored blue, corresponding to the challenge's initial formula, usually an empty block, which branches into several neighbor formulas. This node typically has a path to every node of the graph, making the submission graph weakly connected. As we progress outwards from the central node we traverse several paths of invalid nodes, here colored white. Occasionally, these paths can branch to or join other paths or return back to the central node. Each path can be a dead end, when a student interrupted a session without solving the challenge, or reaches a valid submission, the nodes colored green. Solution nodes tend to have a higher-than-average in-degree, because many student sessions converge into the same valid submission.

5.1.3 Hint Model

The submission graph encompasses all the potential decisions made by students, rendering it a non-deterministic graph. In order to use it to generate hints for users, it is essential to select a specific path

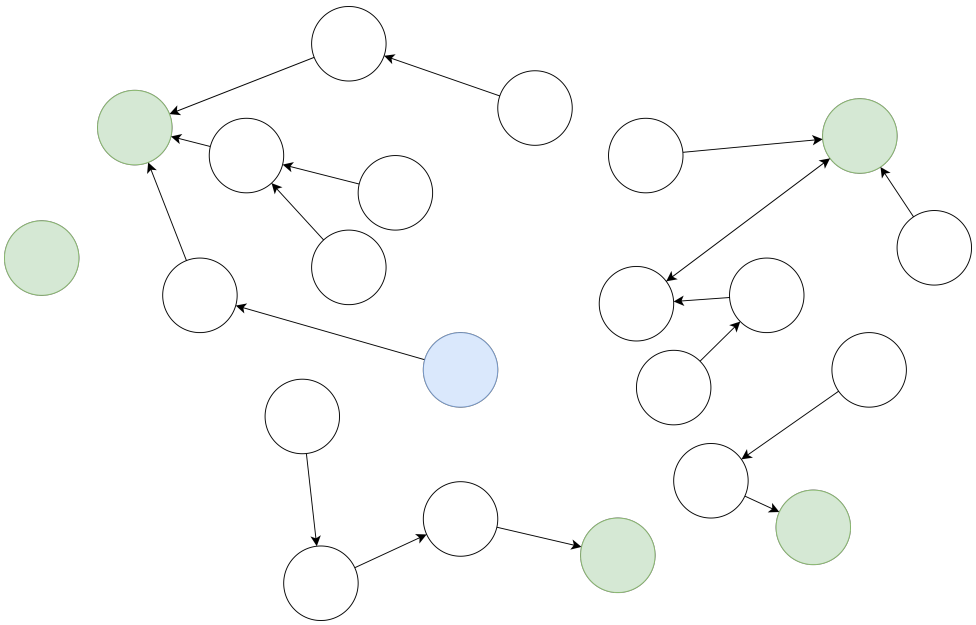


Figure 24: Example of a Hint Model

for each state, essentially narrowing the submission graph into a deterministic one. This procedure is referred to as applying a traversal **policy** on the submission graph. It reduces the submission graph into a forest of state trees, each rooted at a solution, which we have labeled as the **hint model**. Additionally it discards the states that cannot reach a solution. An example of a *hint model* from our previous example is displayed in Figure 24.

To create an effective **hint model**, we need to develop a policy algorithm capable of extracting it from the original submission graph. However, there are some caveats to consider. Depending on the nature of the challenge and its designer's intentions, certain policies may be more successful at generating superior **hint models** compared to others. Since we cannot forecast the challenge designers' preferences, our primary focus has been on developing a modular algorithm, designed to provide designers with the flexibility to customize it to their policy needs.

5.1.4 Policy Algorithm

The basis of our policy algorithm is Dijkstra's shortest path algorithm for weighted graphs. The algorithm operates by maintaining a score on a set of tentative states, which starts as the set of all valid states, i.e. the set of states that correctly solve the challenge. It will iteratively select the state with the best score from this set and use it to update the scores of states that transition to it. After each selection, the algorithm

Algorithm 1: Policy Algorithm

Input: (S, T): Submission Graph
Data: O Objective, N: Neutral Value, R: Policy Rule
Output: $model$: The hint model
begin
  Initialize $queue$, $previous$, $score$ and $model$ as empty sets;
  for *every valid state* s **in** S **do**
    $queue \leftarrow queue + s$;
    $score[s] \leftarrow N$;
  end
  while $queue$ *is not empty* **do**
    $s' \leftarrow$ state in the $queue$ with the best $score$ according to O;
    $queue \leftarrow queue - s'$;
    $model[s'] \leftarrow previous[s']$;
    forall *transitions* t *that leads a state* s *into* s' **do**
      if $s \notin model \wedge score[s]$ *is worse then* $R(s, t, s')$ *according to* O **then**
        $score[s] \leftarrow R(s, t, s')$;
        $previous[s] \leftarrow s'$;
        $queue \leftarrow queue + s$;
      end
    end
  end
end

designates the chosen state and the transition from which the highest score was computed as part of the **hint model** and subsequently excludes them from any potential operations in its future iterations. Once there are no more states to be evaluated the algorithm ends. With this framework we allow the designer to customize three parameters. The first parameter is the objective of the algorithm, i.e. whether the goal is to minimize or maximize the node's scores. The second parameter is the neutral score value, which represents the score assigned to the algorithm's initial set of states (the set of valid states). Finally we allow the designer to customize the rule used to compute the new score of a state given a transition. In this rule the user has access to all of the attributes registered in the submission graph for the transition and the two states involved, and can use their normalized values to define a custom score formula. The pseudo-code representation of this process is presented in Algorithm 1.

In Table 7, we provide an overview of several policy options that we have experimented with. The policy rules in this table will refer to some of the attributes registered in states and transitions, and already discussed in Section 5.1.2. These attributes include the Tree Edit Distance (TED), the state's complexity ($COMPX$) and the state's submission frequency ($FREQ$). Furthermore, the policy can mention algorithm specific attributes, such as the current $SCORE$ associated with the source node.

Name	Objective	Neutral Value	Policy Rule
Shortest Tree Edit Distance (TED)	MIN	0	$TED + SCORE$
MinMax Tree Edit Distance (TED)	MIN	0	$max(TED, SCORE)$
Balanced TED and Complexity	MIN	0	$0.5\,TED + 0.5\,COMPX + SCORE$
Proportional Complexity	MIN	0	$(1 - Proportion) \times COMPX + SCORE$
One	MIN	0	$1 + SCORE$
MaxiMin Frequency	MAX	0	$min(FREQ, SCORE)$
Popularity	MAX	1	$Proportion \times SCORE$

Table 7: Examples of policy parameters

5.1.5 Policy Execution

When processing the users current submission, SpecAssistant will normalize it with the same technique used on the peer data. Subsequently, the obtained formula will then be prompted on the hint model which will give the best next state to guide the user. This procedure is restricted to the pre-existing data and consequently, when a hint is solicited with an unknown submission, the system will be incapable of providing a hint. This problem can be mitigated by querying the hint model with variants derived from the submission's formula, from which we could extract relevant hints.

The implemented approach for expanding the overhaul scope of each query consists in querying the hint model with a set of mutated formulas derived from the submission. The variant formulas are built from a combination of procedures named mutators which have the goal of transforming the original formula based on a predefined rule. In our system, we utilize the mutators conceived in TAR [10] which are presented in Table 8. When a submission is not in the submission graph, this method employs an

exhaustive procedure where each mutator is systematically applied to the submission's sub-formulas, generating a variety of mutated formulas of which we retain in a set those that are contained in the hint model. Finally, the mutated formula with the best score according to the selected policy rule is then used to infer the hint. Since the transitions corresponding to the mutators do not exist in the graph, this technique can only be used with policy rules that rely on attributes that are independent of the submission data. For example, it cannot be used with policy rules that rely on the frequency or the proportion.

Mutator	**Rule**	**Mutation Example**
RemoveBinary	`A [bop] B` ⇝ `A` `A [bop] B` ⇝ `B`	`A or B` ⇝ `A` `A - B` ⇝ `B`
ReplaceBinary	`A [bop] B` ⇝ `A [bop'] B`	`A and B` ⇝ `A or B` `A & B` ⇝ `A - B`
ExtendOrReduce InsertJoin	`A` ⇝ `A [bop] B`	`A` ⇝ `A + B` `A` ⇝ `A . B`
RemoveUnary	`[uop] A` ⇝ `A`	`once no A` ⇝ `no A` `~ A` ⇝ `A`
ReplaceUnary	`[uop] A` ⇝ `[uop'] A`	`once no A` ⇝ `always no A` `~ A` ⇝ `A'`

InsertUnary
InsertPrime
RelationToUnary
	`A` ⇝ `[uop] A`	`no A` ⇝ `always no A`
		`A` ⇝ `A'`
		`A` ⇝ `^A`

ReplaceSet	`[uop] A` ⇝ `[uop] B`	`no A` ⇝ `no B`
BinaryToUnary	`A [bop] B` ⇝ `[uop](A [bop'] B)`	`A in B` ⇝ `no (A + B)`
QuantifierToUnary	`[qtop] a: A | B` ⇝ `[uop] A`	`no a: A |inv[a]` ⇝ `no A`
ReplaceQuantifier	`[qtop] a: A | B` ⇝ `[qtop'] a: A | B`	`no a: A | B` ⇝ `some a: A | B`
RelationToBinary	`A` ⇝ `A [bop] B`	`A` ⇝ `A and B`
ReplaceRelation	`A` ⇝ `B`	`A` ⇝ `B`

Table 8: TAR's mutators

Each rule presented in Table 8 depicts the mutator's effect with a set of arbitrary expressions and operators, where the former is represented by capitalized letters and the latter by the keywords: `[uop]` `[bop]` and `[qtop]`, denoting unary, binary and quantifier operators, respectively.

5.1.6 Hint Generation

After successfully computing the best next state, the system needs to present this information to the user in the form of a hint. For automated repair systems whose goal is to perform auto-corrections, this can be achieved by simply presenting to the user the fixed submission. In hint generation however, this is more complicated since the goal is not to solve the problem but to provide an educational scaffold to the user, i.e. provide just enough information to help a learner overcome its obstacles, without completely solving the problem and miss the educational objective. This is made even more difficult by the fact the

interpretation of each hint is very likely to vary between users, so to ensure the desired effect on the user, our system should present concise information which can also be adapted to the different situations.

Taking into account these limitations, we chose to implement our hints as code highlights. This stemmed from the fact that these serve as a visual focal point to the user and thus allowed us to precisely guide the user to the submissions fault. Alongside the highlight, the system will also provide a text message with further help, generated from the original state and action. The absence of any sufficiently large repository of Alloy hints prevented us from using machine learning methodologies for text generation. As a result our current text generation relies on a rudimentary rule-based approach. This approach generates textual prompts based on the transition that led to the selected next state. For actions that were computed based only on historical data (that is, the wrong submission was seen before), our system first employs a tree difference algorithm [34] to categorize the overall transition as list of changes, specifying them as *deletions*, as *insertions*, or as *replacements*. Subsequently, it will retrieve the closest change to the AST's root, matching its category either the phrase "Try adding something to this declaration" for insertions or the phrase "Try to change this declaration" for the remaining categories. The chosen change is also used to determine the faulty expression to highlight. Conversely, when transitions result from mutations (that is, the wrong submission was not seen before), the displayed hint is drawn from the definition within the mutator's class, resulting in the messages found in Table 9.

Mutator	**Hint**
RemoveBinary ReplaceBinary InsertJoin	Binary operator has to be changed or removed.
InsertUnary InsertPrime	Insert an operator.
RemoveUnary ReplaceUnary	Unary operator has to be changed or removed.
BinaryToUnary	Transform into a unary expression.
QuantifierToUnary ReplaceQuantifier	Quantifier has to be changed.
ExtendOrReduce ReplaceRelation	A different relation is required.
InsertJoin RelationToBinary RelationToUnary	Add a binary or unary operator.
RemoveUnary ReplaceUnary	Unary operator has to be changed or removed.
ReplaceSet	Expression under unary operator has to be changed.

Table 9: Hints for TAR's mutators

5.2 Alloy4Fun Implementation

This section will provide an overview of the technique's implementation within the Alloy4Fun application, which involved some minor developments within meteor's front-end, along some major developments across three modules of the API component: the new hint system, the REST service, and an Alloy Toolbox integration module, which can be found within the modules `specassistant`, `alloy4fun`, and `alloyaddons`. This section discusses the modifications made to existing components, alongside the implementation of the new features. The outcome of this implementation resulted in a system that, when subjected to an invalid submission automatically retrieves a hint targeted at the current situation which the user could consult or ignore. This hint is presented by marking the incorrect section of the code with a blue highlight and offering a hint message about the error and the highlighted portion, which is exemplified in Figure 25.

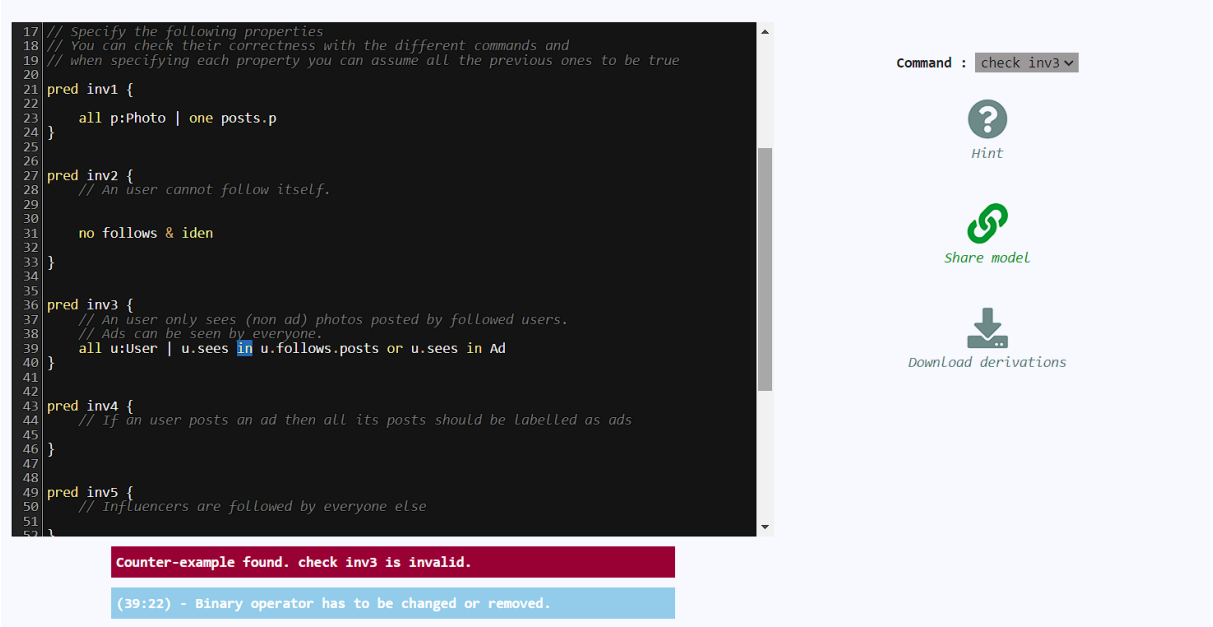


Figure 25: Example of a hint generated by SpecAssistant

5.2.1 Framework Migration

To take full advantage of Alloy's functionalities, namely its parser and AST implementations, our system needed to be developed in Java just like the Analyzer. Additionally, in our case it would be beneficial to have the new functionalities incorporated within the Alloy4Fun API component. However, unlike the current API component that only accesses the submitted Alloy model, the new services would also introduce the need to access the database. To do this we had two possible approaches. The first relied on having each client provide all data that could be required. This approach was quickly abandoned since as a result the clients would need to provide data with sizes ranging from dozens to hundreds of megabytes, which would

quickly overwhelm the service when presented to multiple concurrent clients. The only viable solution was to make the API component itself able to request the data directly from the database.

This solution was not simple however. The *framework* in which the API component is developed (Thorntail) was discontinued in 2020 due to the rise of better alternatives. As a result, continuing the development with this framework would lead to problems due to the lack of support in recent years. One of the biggest problems we would have was the use of deprecated or removed functionalities on the database, which would require several suboptimal workarounds. Additionally, several modern libraries for back-end development would not be available and as such the functionalities needed from them would have to be redeveloped for this system, which would result in a waste of development time.

To avoid these problems, the API was ported to Quarkus[1], a popular and rapidly evolving java *framework* which contains the features required for our development. Additionally it also comes with several welcoming characteristics such as rapid REST response times, optimizations for deployments on virtual containers, and a well-defined long-term support plan. The resulting API, implemented in a module named `alloy4fun`, is composed of different components which form the stack presented in Figure 26. Despite this, the core functionality of the component and its external interfaces remain unchanged and are indistinguishable from the current Alloy4Fun web service.

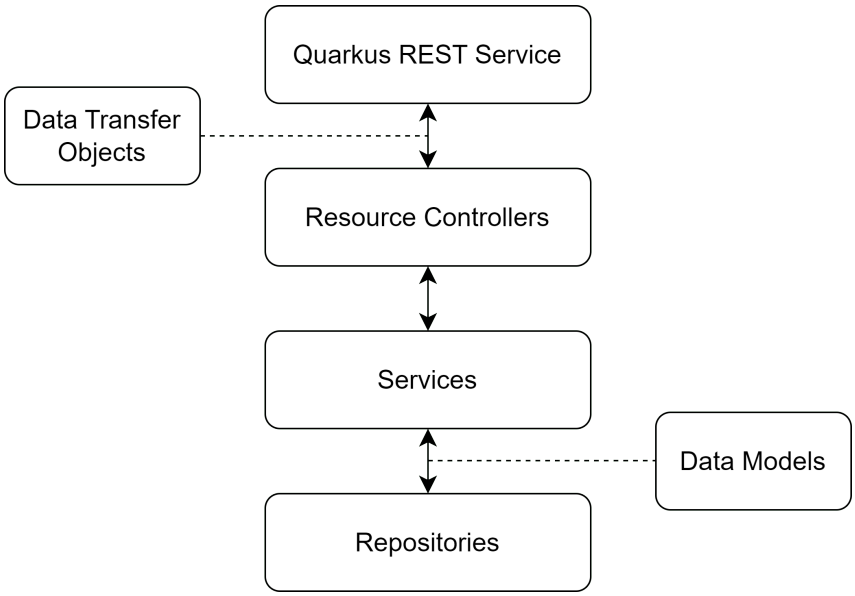


Figure 26: API component stack

At the top of the stack we have the REST service component which is provided by the framework on the package `quarkus-resteasy-reactive-jackson`. The package itself is the combination of two libraries: firstly we have *Quarkus RestEasy Reactive*, the extension that handles asynchronous and non-blocking HTTP endpoints, and secondly we have the Jackson project[2], a data-processing tool that allows

[1] `https://quarkus.io`
[2] `https://github.com/FasterXML/jackson`

for the seamless management of JSON data as Java objects. This component's function is to handle every process of any HTTP communication that is not specific to our developments.

Descending through the stack we will reach the application's core functionality, starting with the resource controllers, declared within the `resource` package. These components define the application's HTTP interface, i.e. the available methods, and how these interact with the system's business logic. Our system declares three functional controllers: `AlloyGetInstances`, `AlloyGetProjection`, and `AlloyValidate`, which are responsible for implementing each of the API's methods. Alongside the controllers we also declare a number of Data Transfer Objects (DTOs) across the packages `data.request` and `data.transfer`. These define the data formats used by our services and are succinctly described as basic Java classes, leveraging from the Jackson library and its corresponding annotations to be automatically parsed into JSON formats. An example of a DTO can be found in Listing 5.4 where we declare the object returned by the HTTP method POST "/getInstances".

```
1  public class InstanceResponse {
2      @JsonInclude(NON_EMPTY) String err;
3      @JsonInclude(NON_NULL) @JsonUnwrapped InstanceMsg warning;
4      String sessionId;
5      Boolean unsat;
6      Boolean check;
7      String cmd_n;
8      Integer cnt;
9      @JsonProperty("static") Boolean is_static;
10     @JsonInclude(NON_NULL) Integer loop;
11     @JsonInclude(NON_EMPTY) List<InstanceTrace> instance;
12 }
```

Listing 5.4: *InstanceResponse* facade

Beneath the Controllers, lies the Service layer which embodies the heart of the application's business logic. It defines the core operations of the application and enforces the business rules, ensuring the integrity and consistency of data interactions. Their data-flow starts from the information provided by the Controllers, which can include any form of data structure, which is subsequently subjected to targeted operations and validations. Within these operations, the system might need to interact with external applications resources by calling the underlying layer, therefore scaling its functions to several other processes and components. Finally, each service replies to the original controller which generally only needs to transform the resulting information into REST responses. Throughout our development we will see each service be declared in the package `service`, however the base application does not declare any since the only business logic of the program is the external Alloy compiler.

Finally, at the base of the stack we find the Repositories, declared within the package `repositories`. This layer is responsible for managing the interaction between the application and the underlying data, such as an external API, the file system, and most commonly the database. The function of a repository is to handle the application's data ensuring its consistency which can include data parsing, persistence,

retrieval, querying, and aggregation. Alongside each repository, within the `data.models` package, we also have the data models that each one is designed to handle. They generally consist of Java classes that comply with one or multiple interfaces required for the functionalities of their endpoints. Exemplifying, for REST services this may simply be the previously described DTOs, for a database this might be an explicit specification of each of the entities keys and indexes alongside each object. The most common kind of implementation focus mainly on declaring Create, Read, Update, Delete (CRUD) repositories, where each developer only needs to worry about defining their models and queries, leaving the possible networking and serialization processes to be defined by the framework. Our base API declares a simple repository used only to read and update the *Session* models stored within the application's temporary memory. Through the implementation of the hint system we have a few CRUD repositories as a result of the new requests issued to the system's database.

5.2.2 Data Model Management

Since each exercise involves not only different challenges but also different checking approaches, it became crucial to define a process to ensure that each submission to a challenge is matched to the correct submission graph. The only viable way to differentiate a challenge is with the combination of its first ancestor with secrets (`original`) and the executed command label (`cmd_n`). However, using them directly to discriminate each challenge will give rise to a few undesirable complications, namely:

- When specifying automatic feedback exercises in Alloy4Fun it is required to split the oracle evaluation from the user's solution with either predicate or functions, in order to allow Alloy4Fun to hide the former. These declarations cannot be accessed systematically, since the lack of any standard for naming or structuring an exercise makes them essentially indistinguishable from other declarations. In order to surpass this problem the system requires the tutor to specify a mapping that relates the command of each problem with their corresponding user-editable predicate where the student will write the submission.

- Alloy, and consequently Alloy4Fun, allows for the specification of multiple commands with the same name. As such, the system has no way of distinguishing duplicate commands defined by the users and thus the possible mismatches can lead to the assimilation of faulty data onto the submission graphs, such as wrong answers being interpreted as valid by the adulterated command. To prevent this we take advantage of the fact that Alloy4Fun appends the exercises hidden commands at the end of the model, which allows us to filter every user defined command simply by knowing the number of challenges in the original exercise.

- The first ancestor, i.e. the exercise's original model, is not necessarily the only exercise to present the same specification challenge. Any modification to the model's code, for example a new comment or a rewritten command label, is enough to cause the exercise to change even though the

contained challenges themselves remain the same. If such exercises were processed separately, each data model would ignore the states of the peers who contributed to the same challenges, resulting in multiple smaller and less reliable hit models. To address this issue we have decoupled the challenges from the exercises, resulting in the need for the tutor to specify a relation that describes how to access all the exercise variants that propose each challenge.

To address all of these problems the implementation declares two data collections: *Graph* and *Challenge*. The first uniquely identifies each submission graph, pairing it with a bundle of descriptive fields such as a name for the challenge. The second implements all the previously mentioned mappings, relating each challenge to the respective command, and submission graph, predicates and functions where the user will write the submission, and the maximum offset the exercise can have from the end of the command list.

The system requires the user to specify each entity and relation in order to enable hint processing on an exercise. When applied to a single exercise this is generally a simple and straightforward task. However, in the context of our current project, where we have multiple exercises with variable naming schemes, this can be cumbersome, and so we explored a more automated approach. Building an infallible method would be difficult, but thankfully our data follows a few predefined patterns which allows us to streamline the process. Firstly, all exercise variants have equal command names and, secondly, every user modifiable predicate is targeted by one command at most. As a result we can automatically generate all required information simply by indicating which exercises are variants of each other.

5.2.3 Alloy Model Processing

A core element in our development involves the methods responsible for handling the Alloy Analyzer. These processes are used both by the API component and the newly developed hint generation component. As a result, to ensure an ease of integration within both of these contexts, these methods where forked into their own module, named `alloyaddons`.

In order to parse an Alloy model we must call one of the static methods within the class `CompUtil` contained within the package `edu.mit.csail.sdg.parser` of the Analyzer. These methods are the main source of latency within the parsing process, since they require that each model must be written into a file and parsed directly from the file system, causing the application to issue several system calls in really short periods. Unfortunately, solving this problem would require an extensive re-implementation of Alloy's parser and makes processing big data-sets such as the Alloy4Fun data-set really slow. We have mitigated this issue by implementing methods that scale this class to multiple threads, allowing us to parse multiple models in parallel.

When a model is parsed we obtain an object of the class `CompModule` which fully describes the parsed model. By using the mappings defined in the previous section we can navigate this object and find the relevant ASTs. Each AST is an instance of a class named `Expr`. To facilitate coding, this class

implements the *Visitor Design Pattern*, a behavioral design that allows the addition of new methods to each hierarchical class without modifying their structure.

To take advantage of this pattern the developer simply needs to declare a *visitor* class that belongs to the `VisitorReturn` hierarchy where it will define the behavior for each required subclass. One example of this construct is the implementation of the normalization technique, introduced in Section 5.1.1, which implements a traversal using a visitor object described by the facade presented in Listing 5.5.

```
1   public class ExprNormalizer extends VisitReturn<Expr> {
2       Expr visit(ExprBinary exprBinary) throws Err;
3       Expr visit(ExprList exprList) throws Err;
4       Expr visit(ExprCall exprCall) throws Err;
5       Expr visit(ExprConstant exprConstant) throws Err;
6       Expr visit(ExprITE exprITE) throws Err;
7       Expr visit(ExprLet exprLet) throws Err;
8       Expr visit(ExprQt exprQt) throws Err;
9       Expr visit(ExprUnary exprUnary) throws Err;
10      Expr visit(ExprVar exprVar) throws Err;
11      Expr visit(Sig sig) throws Err;
12      Expr visit(Sig.Field field) throws Err;
13  }
```

Listing 5.5: Normalizer facade

Although the class requires a considerable number of definitions, the majority of the methods do not actually change their receiving objects. To implement our normalization algorithm modifications are only required on five different classes: **ExprList**, **ExprBinary**, **ExprCall**, **ExprLet**, and **ExprQt**.

ExprList objects define a series of commutative operations (`and`, `or`, and `disj`) across their list of sub-nodes. To comply with our commutative normalization clause, the list is ordered lexicographically by comparing their string serialization.

ExprBinary objects define every possible binary operations between two sub-nodes, distinguished as the **left** node and the **right** node. These operations include all commutative and equivalent operators targeted by our normalization. Similarly to the previous object, handling the commutative operations involves computing the lexicographic order of its elements using their string serialization value. Afterwards the lexically smaller node is placed on the **left** and the lexically bigger node is placed on the **right**. When it comes to equivalent operations, when using the operators $\geq$ and $\leq$, the implementation swaps the **left** and **right** arguments and rewrites the node using the operators $<$ and $>$, contained within a `not` operator.

ExprCall and ExprLet objects describe function calls and `let` constructs, respectively. As such, they are replaced by their subtrees after each one of their declared variables is replaced by the value provided to them in their arguments.

ExprQt objects describe quantifier declarations. These are composed of a quantifier operator, a list of variable declarations, and the quantified sub-expression. As previously explained in Section 5.1.1, to comply with our identifier normalization clause, we must reorder each of the identifiers in order to rename each using the index of its declaration. Although conceptually simple, the implementation of this renaming poses some challenges. Depending on how the user has written their formula, the identifiers intended for reordering may or may not be separated into multiple `ExprQt` objects, chained by their sub-formula instances. Thus, our process begins by collecting all sequential variable declarations that share the same quantifier operator. Each declaration is an instance of the class `Decl`, which is composed of a list of identifiers, the declared type, and an optional disjoint clause.

After collecting all declarations we then proceed to assign each the minimum $depth$ within the quantifiers AST. This $depth$ field allows us to determine the partial order induced by dependent identifiers, such as, for example, the variable `z` of the formula `all x,y : X, z : x.next` which must always be declared after `x`. In this case x and y would have a minimum depth of 0 while z would have a depth of 1. If a new variable was declared that depends on z it would subsequently have a depth of 2, and so on. Upon computing the $depth$ for each declaration, our normalization will start deconstructing each `Decl` object. Each identifier is ordered and subsequently renamed according to its declaration's $depth$ and the lexical order of the normalized type. Simultaneously, every disjunct clause is normalized by combining it with the quantified sub-formula through the utilization either the operation `and` or `implies`. The final output of the normalization is a new **ExprQt** object, with the original quantifier, the renamed identifiers, and the updated sub-formula.

When normalizing an AST, our system maintains a mapping between the original tree and the normalized one, in other to allow our system to trace both variable names and most importantly the file positions required for marking down a hint's code highlight.

After its normalization an AST can be further processed in two different ways. The first is string serialization, which is used in interactions with the database, and when ordering identifiers. The second is a difference algorithm, were the system is comparing two ASTs in other to determine their changes. To accomplish this, our system uses the All Path Tree Edit Distance (APTED)[3] [33, 34] algorithm, which computes the smallest edit maps and distances between two trees. To use the algorithm we simply must define a match relation between each node and a cost model for its three possible edit operations ($insert$, $delete$, $rename$). Subsequently, the system can compute the minimum edit distance and mapping between two ASTs. Meanwhile, the edit mapping denoting the relation that matches identical nodes between each tree is translated into a sequence of edit operations that are used by the hint generation procedures.

[3] github.com/DatabaseGroup/apted

5.2.4 Hint Model Computation

With the previously defined processes for handling the submissions and the Alloy ASTs we can finally start parsing our historical data into our data models. This process is divided between three steps: firstly, we have the model ingestion were we scan and process each submission tree from the database, and add it to the respective submission graph; secondly, we have a classification step where we pre-compute a series of attributes needed throughout the model's lifetime; and, finally, we have the policy computation step, where we narrow down the submission graph to a hint model.

Model Ingestion

As previously mentioned in Section 3.3, Alloy4Fun's submission data is structured as derivation trees rooted at the exercise's original submission. Within each exercise we have a series of challenges for the user that will have to be processed separately. Additionally, since Alloy4Fun records every execution the students' issued, in each of these derivation trees we can differentiate every attempt at solving a exercise as a path of submissions from the tree's root to its leaves. This data can be overwhelming, since a tree may also include syntactically incorrect submissions, submissions for different exercises in different order, executions of commands not present in the original exercise, and duplicate submission issued by the users on their explorations. Some of these behaviors are visible in Figure 27, were we depict a tree highlighting different commands with different colors. In the example, we have four different paths corresponding to four different attempts at solving three challenges (`inv1`, `inv2`, and `inv3`) in a exercise containing two signatures: `A`, which declares relation `next: set A`, and `B`, which is an extension of signature `A`. In this

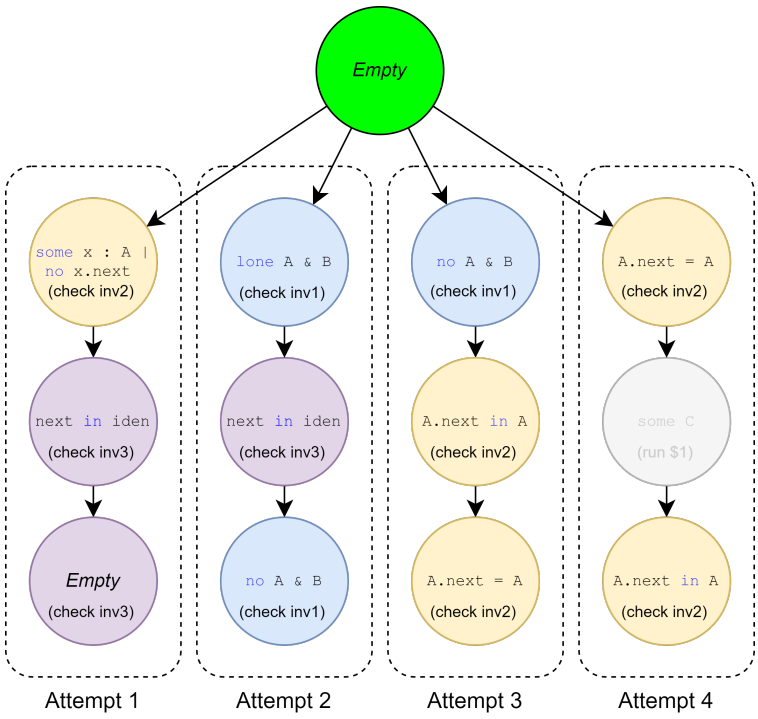


Figure 27: Exercise derivation tree

example the students inconsistently alternated between three challenges and also executed commands not present in the original exercise and not corresponding to any challenge (namely `run $1`).

The goal of this step is to process this data by isolating each challenge from every path of the tree, and aggregating the respective submissions in the challenge submission graph. To achieve this we must simply divide the set of submissions by each targeted challenge, and subsequently relating the submissions of each resulting set by the order defined within the root exercise. Upon finishing this process we simply need to amalgamate the submissions with duplicate formulas and transitions into an initial submission graph where we only record the frequency of each transition, as exemplified in Figure 28.

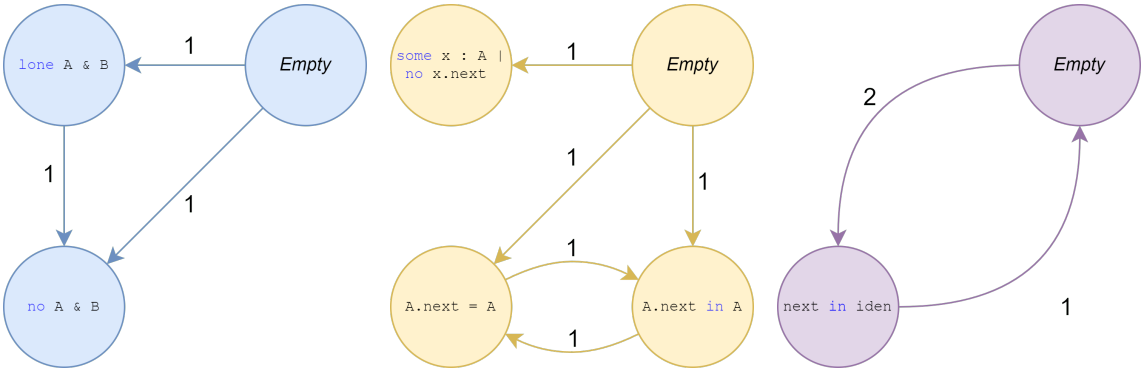


Figure 28: Amalgamated Submission Graphs

This ingestion process is implemented in a specialized Breadth-First Search (BFS) algorithm with the `CompletableFuture` class, a Java class that allows for a simple specification of multi-threaded tasks. The advantage of using a multi-threaded approach is the possible performance boost obtained from parsing multiple models in parallel.

Attribute Computation

After populating each graph, the subsequent step is to compute each attribute needed for the policy rule computation. One of these attributes is the proportion of times each edge was traversed, which is calculated directly by determining the ratio between the number of times the edge was traversed and the number of times a user submitted its source state. Another attribute is the complexity of each formula, which is defined as the expected size of the formulas truth table. To compute this value we must count the number of nodes in the AST m and the number of independent variables n. With these properties we can obtain the truth table's size with the value of n^m.

The final attribute we need to compute is the Tree Edit Distance (TED) between the ASTs at the ends of each transition. For this we use the All Path Tree Edit Distance (APTED) [33, 34] library. When given two distinct trees, the algorithm initiates a process of comparing and pairing identical nodes in an effort to discover the most extensive node matching. Subsequently, after computing a mapping, the algorithm highlights the nodes in each tree that remain unmatched. If there is a discrepancy in the source tree, its token is categorized as deleted. Conversely, if a discrepancy arises in the target tree, it is labeled as an

addition. A visual representation of the resultant mapping is illustrated in Figure 29, with deleted nodes indicated in red and added nodes in green. The TED corresponds to the number of nodes that have been added or deleted between the two trees. In our example, this value is 3.

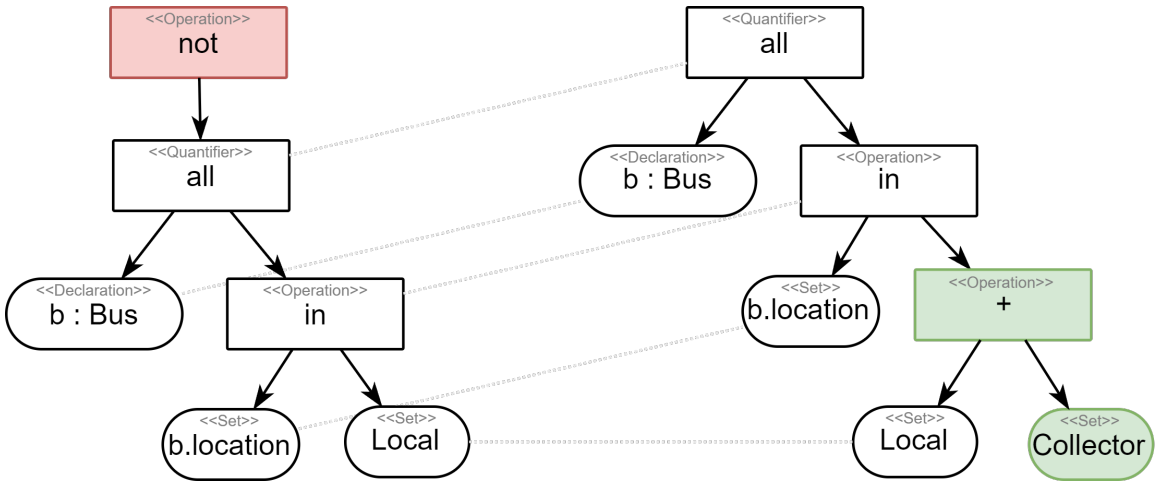


Figure 29: Visualization of a tree difference mapping

Policy Computation

In this step, our program utilizes the configured policy parameters, which are included with the computation request, to calculate the desired hint model for the challenge. The algorithm itself is a translation of the Algorithm 1 into Java, which is designed to process every data point from Java Typed objects. As such, our neutral value and objective parameters are defined using a double and an enumeration, respectively. The policy rule, on the other hand, is defined by a class hierarchy, implementing a Composite Design Pattern. It consists of a super-class named `PolicyRule`, which has three sub-classes: `Constant`, `Var`, and `Binary`. The super-class defines two abstract methods. The first method, `normalizeByGraph`, is designed to standardize the rule's parameters according to a specified graph. The second method, `apply`, plays a crucial role in executing the defined rule on the provided edge and states.

While the `PolicyRule` defines how to handle the specified rule, its actual behavior is implemented by its sub-classes. The `Constant` class represents a constant numerical value that cannot be normalized and simply provides its value upon application. Next, the `Var` class is defined by an attribute reference. When invoked for normalization, this object retrieves the maximum and minimum values of its target. Consequently, when the object is applied to a state or edge, it returns the value of the attribute normalized between 0 and 1 by the considering these limits. Finally, we have the `Binary` object, which can apply one of six types of binary operations (+, −, ×, /, min, max) to its two sub-rules. Its normalization process is defined by the normalization of its sub-formulas. Each one of these classes can be serialized and deserialized as a JSON, with the use of a discriminator field named `"type"` created using the `Jackson` project. Therefore, front-end services like Meteor can effortlessly communicate each rule using

these objects. An example of this object in JSON format is shown in Listing 5.6, which depicts the rule $1 + SCORE$.

```
{
    "type": "binary",
    "operator": "+",
    "left": {"type": "constant", "value": 1.0},
    "right":{"type": "var", "var": "SCORE"}
}
```

Listing 5.6: Policy rule as a JSON object

5.2.5 Updated Database Schema

As result of our development, we have expanded the original MongoDB database schema, resulting in the creation of four new collections that store the data described in the previous sections. In this section, our primary focus is to provide a detailed description of the scheme and data formats of these new collections, described in Tables 10, 11, 12, and 13.

Graph				
Field	**Format**	**Optional**	**Example**	Description
_id	ObjectId	NO	`64b9433022472102a170f762`	Object Identifier
name	String	YES	`Instagram-inv2`	Name of graph, (usually the exercise and challenge names)
policy	Object	YES	`{ "count": 658 "time": 2.4176935 }`	A performance summary of the policy computation, including its final submission count and computation time
parsing	Object	YES	`{ "dkZH6HJNQNLLDX6Aj":{ "count": 659 "time": 45.6081762 } }`	A performance summary of the ingestion process, associating each exercise with its submission count and parsing time

Table 10: Dictionary of the collection *Graph*

Challenge				
Field	Format	Optional	Example	Description
_id	ObjectId	NO	`64b9433022472102a170f76a`	Object Identifier
model_id	String	NO	`dkZH6HJNQNLLDX6Aj`	Reference to the model exercise
graph_id	ObjectId	NO	`64b9433022472102a170f762`	Reference to the submission graph
end_offset	Int32	NO	`8`	Maximum offset the command can have from the end of the model
cmd_n	String	NO	`inv10k`	Name of the challenge's command
targetFunctions	Array	NO	`["inv1"]`	List of functions of the challenge

Table 11: Dictionary of the collection *Challenge*

Node				
Field	**Format**	**Optional**	**Example**	Description
_id	ObjectId	NO	`64b9433022472102a170f762`	Object Identifier
graph_id	ObjectId	NO	`64b9433022472102a170f831`	Reference to the submission graph
formula	Object	NO	`{ "inv1" : "no Trash" }`	Mapping from each managed function to the respective formula
valid	Boolean	NO	`false`	Whether the formula was correct
visits	Int32	NO	52	Number of submissions which specified this formula
leaves	Int32	NO	45	Number of modifications applied to the formula (out-edge count)
hopDistance	Int32	YES	4	Minimum distance to a solution
complexity	Double	YES	120	Complexity of the formula
score	Double	YES	97.568779	Computed policy score of the node
witness	String	YES	`nuWnon2d7N7N7ZFvw`	Reference to a model specifying this node's formula

Table 12: Dictionary of the collection *Node*

Edge				
Field	**Format**	**Optional**	**Example**	**Description**
_id	ObjectId	NO	`64c14e95129a556f2555df88`	Object Identifier
graph_id	ObjectId	NO	`64c14e91129a556f2555dcdf`	Reference to the submission graph
origin	ObjectId	NO	`64c14e95129a556f2555df6b`	Origin node reference
destination	ObjectId	NO	`64c14e95129a556f2555df83`	Destination node reference
count	Int32	NO	3	Number of edge crossings
editDistance	Double	YES	13.0	Edit distance between the origin and destination formulas
policy	Boolean	YES	`true`	Whether the edge is a part of the hint model

Table 13: Dictionary of the collection *Edge*

To communicate with the new data we have defined a CRUD repository for each collection with the use of *Quarkus*'s Panache library for MongoDB. This package simplifies the development process by providing an intuitive querying system without the processes associated with managing the MongoDB client. To take advantage of this tool we must first define our data models as inheritors of the `PanacheMongoEntity` or one of its parent classes. As a result, each model will be presented with a series of methods used to issue queries alongside the required processes used to manage the model's `_id` field. Afterwards the user can declare each query within the model's respective repository, defined by the interface `PanacheMongoRepository`. A simple example of this construct can be found in Listing 5.7, were we present a snippet of the *Challenge* repository containing two queries.

Both of these queries use the `find` method of `Panache` to declare a MongoDB match query, which has the purpose of retrieving Challenge objects based on the conditions provided within its body.

The first query (`findByModelIdAndCmdN`) has the purpose of restricting the retrieved Challenge by their `"model_id"` and `"cmd_n"` fields. It implements its body using a query language implemented

by Panache. It is a simple language to read and process, but it does not support the complete array of features of MongoDB. These unmet features can still be called upon with objects that can format to Binary Javascript Object Notation (BSON) such as the `Document` object. This is the case for the second query (`streamByModelIdIn`), where the provided object declares the body `{"model_id":{$in:[...]}}`, that has the purpose of retrieving any object where the field `"model_id"` is contained within the provided collection.

After specifying the query and its respective body, we finish by declaring how we want the information delivered to our application. We can choose to retrieve only the first object found (`firstResult()`), have it packaged as a Java Stream or List (`stream(), list()`), have it delivered across pages (`page()`), among others.

```
1  @ApplicationScoped
2  public class ChallengeRepository implements PanacheMongoRepository<Challenge> {
3
4      public Challenge findByModelIdAndCmdN(String model_id, String cmd_n) {
5        return find("model_id = ?1 and cmd_n = ?2", model_id, cmd_n).firstResult();
6      }
7
8      public Stream<Challenge> streamByModelIdIn(List<String> model_ids) {
9        return find(new Document("model_id", new Document("$in", model_ids))).stream();
10     }
11     ...
12 }
```

Listing 5.7: Challenge repository

6 Evaluation

The primary objective of any hint generating system is to provide valuable and contextually relevant clues or suggestions to assist individuals in solving problems, overcoming challenges, or improving their performance in various tasks. Our data-driven system fulfills this task by mining hint suggestions from historical data that describes the decisions taken by multiple users. As explained throughout the previous chapter, this process has some compromises between the system capability for presenting hints (availability) and the effectiveness of hints in helping the user understand and solve the problem at hand (quality). As a result, to evaluate our system we must evaluate both these aspects.

We can use a broad range of quantitative and qualitative assessments in order to precisely measure both the availability and quality of our system. Due to its complexity, the qualitative assessment was not planned to be conducted during the thesis and is expected to be performed in future work. As a result, our current evaluation focus mostly on the quantitative aspects.

More specifically, our evaluation aimed at answering the following research questions:

RQ1 Is SpecAssistant viable for hint generation?

RQ2 Is it viable to frequently update the submission graphs of SpecAssistant?

RQ3 To what extent does the incorporation of mutated formulas affects the performance?

RQ4 How does SpecAssistant performance stacks up against other alternatives?

To answer these questions, we employed a range of data-mining validation and testing techniques. These techniques encompassed data pre-processing, data-set splitting, cross-validation across various models, as well as a superficial overfitting and underfitting examination of ours models.

6.1 Data-set Partitioning

The first step in our evaluation was to divide the existing submission data into two distinct subsets: the training data-set, which was used to build the hint models, and the testing data-set, which was used to

measure their effectiveness. To obtain meaningful results, it is imperative that these samples generalize the entire system's behavior. If we fail to generalize the training data, the biased model will likely perform worse. Conversely, if we fail to generalize the testing data, our results may overlook some user behaviors, which would result in biased results.

The Alloy4Fun data-set [22], our targeted data-set, primarily consists of derivation trees where each sub-tree of the root corresponds to a distinct user attempt. These were generated by university students as direct result of their participation in academic courses teaching Alloy at the University of Minho (UM) and University of Porto (UP), and include nearly 70000 syntactically correct submissions. In particular, the Alloy4Fun data-set encompasses four academic years spanning from the fall of 2019 to the summer of 2023. For the academic years of 2019/2020 and 2020/2021, the data originated from a course that introduced a variety of challenges to two classes of around 20 students, including "Trash FOL", "Classroom FOL", "Trash RL", "Classroom RL", "Graphs", "LTS", "Production Line v1", and "CV". In contrast, during the academic years 2021/2022 and 2022/2023, the data resulted from a different course that exposed two classes of around 210 students to new challenges, namely "Production Line v2/v3", "Train Station", "Courses", and "Social Network". Despite the population size in the latter years being nearly 11 times larger than in the former years, the number of submissions was only the double, because in the latter course Alloy was taught for a short period only. One noteworthy aspect of the exercises is that it that they are publicly available. Therefore, it is possible that a small number of submissions could have also come from external users. This could potentially explain the data we have for older exercises, with submissions beyond their original class time-frames. The distribution of submissions can be observed in Figure 30.

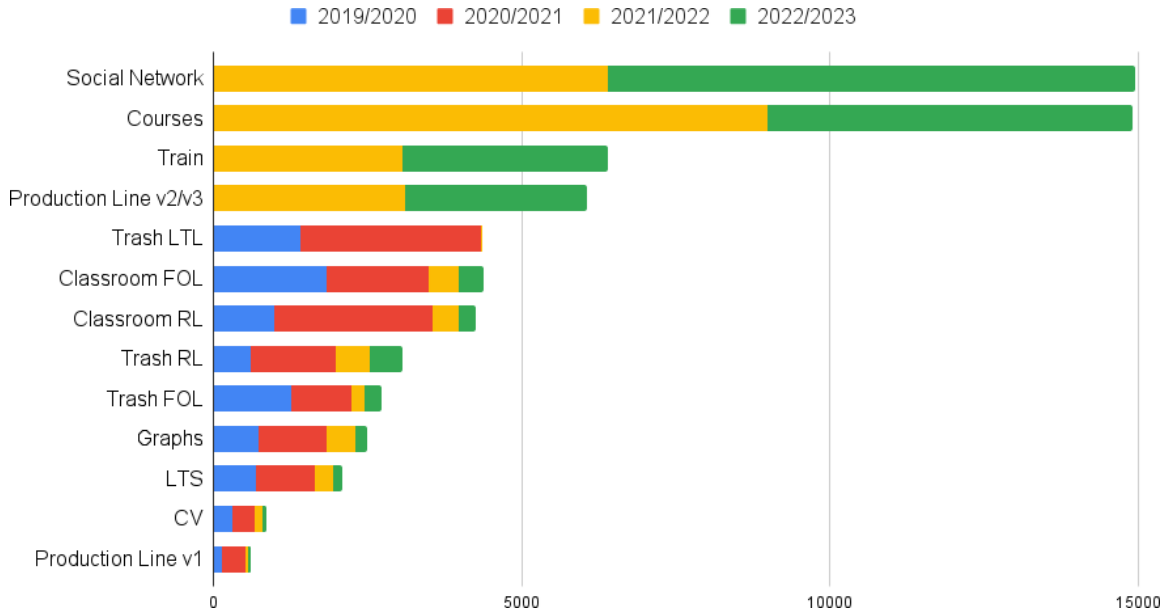


Figure 30: Number of syntactically correct submissions in each exercise

We focused on randomly splitting the user attempts, i.e. the direct sub-trees of the root of each derivation tree, based on the number of valid submissions each one can contribute to each submission

graph. The partitioning process begins by calculating the number of valid submissions per challenge in each sub-tree of an exercise. Then, it randomly selects sub-trees for the smallest partition (in our case, the testing data-set), continuing this selection process until the sum of its valid submissions surpasses the predefined split ratio. In our implementation, we designated 70% of data for training and 30% for testing. Because each sample is inherently random, we believe that our results will be less biased and can be generalized.

6.2 Benchmarking

After establishing each partition, every challenge of the training data-set was put through our hint model computation process. As a result, our system generated a hint model for every challenge defined across each exercise. Subsequently, we individually tested each invalid submission within the testing data-set against our hint generation technique and TAR, the previous hint generation system of Alloy4Fun. Each result was stored separately in a database collection named "Test" described in Table 14. This collection outlines specifications on each tested submission based on the type of test performed, which is described by the respective tool (SpecAssistant, TAR), policy rules and potential augmentations, such as the incorporation of mutated formulas. The specifications themselves include a number of useful data points, such as the test's success, the elapsed time, the chosen state, among others. After the benchmarking it accommodated more than 200000 distinct entries. This procedure to handle benchmark data gave us flexibility in conducting the evaluation. From an execution standpoint, it allowed us to consistently validate and possibly repeat tests without compromising the remaining data, and from an analytical standpoint, it allows us to precisely tailor our data view to each research question without losses of information.

Field	Format	Optional	Example	Description
_id.model_id	String	NO	`uCFT6ci2br98QhWF7`	Identification of the request's model
_id.type	String	NO	`"TED-SPEC"`	Type of benchmark preformed
data.success	Boolean	NO	true	Whether the benchmark generated a hint
data.time	Double	NO	`0.2409658`	Elapsed Time in seconds
data.match	ObjectId	Yes	`64c14e91129a556f2555d0023`	Reference for the matched state
data.next	ObjectId	Yes	`64c14e91129a556f25558450`	Reference for the next state
data.edge	ObjectId	Yes	`64c14e91129a556f25551182`	Reference for the corresponding transition object between the $match$ and $next$
graphId	ObjectId	Yes	`65043b32364fce46dcb96123`	The benchmark's submission model

Table 14: Dictionary of the collection *Test*

Each individual test was given a timeout of one minute. The rationale behind this decision was based on the expectation that any hint generation technique exceeding this time would severely compromise the

user experience, rendering the tool unusable. However, the ideal upper limit for the elapsed time before experiencing any degradation in the quality of the user's experience is 10 seconds, according to [20]. Every benchmark was ran on a virtualized Docker deployment using `mongodb` image version `6.0.5`[1] and RedHat's `ubi8/openjdk-17` image version `1.16`[2] on top of Docker Engine version `24.0.2`. The host of this virtualized setup integrated a minimalistic setup of Windows 11 Pro version `10.0.1928` on top of a Intel Core i5-13600KF processor, 32 gigabytes of RAM and a $1TB$ SSD with an approximate Read/Write speed of $530MB/s$.

With our benchmark results we aimed at answering the above research questions.

RQ1 Is SpecAssistant viable for hint generation?

We will begin evaluating our system by analyzing its performance across each exercise. This can be achieved by analyzing the elapsed times and success rates for each of them. A summary of these results is presented in Table 15.

Exercise	Historical Submissions	Hint Requests	Misses	Hits	Elapsed Time (s)	
					Avg.	Std. Dev.
Courses	10431	2418	1606 (66%)	812 (34%)	0.014	0.005
Social Network	10428	2793	1528 (55%)	1265 (45%)	0.008	0.005
Train Station	4394	1331	802 (60%)	529 (40%)	0.015	0.006
Production Line	4156	1102	729 (66%)	373 (34%)	0.013	0.004
Trash LTL	2788	890	533 (60%)	357 (40%)	0.011	0.003
Classroom FOL	2702	663	332 (50%)	331 (50%)	0.010	0.003
Classroom RL	2474	687	448 (65%)	239 (35%)	0.016	0.006
Trash RL	1530	347	152 (44%)	195 (56%)	0.016	0.004
TrashFOL	1425	194	94 (48%)	100 (52%)	0.018	0.006
Graphs	1281	370	208 (56%)	162 (44%)	0.019	0.008
LTS	995	393	319 (81%)	74 (19%)	0.016	0.006
CV	596	218	174 (80%)	44 (20%)	0.012	0.004
Production Line v1	424	120	85 (71%)	35 (29%)	0.017	0.008

Table 15: Performance summary across each exercise

Our system excels in the performance aspect, by consistently providing hints within milliseconds after each request and consequently negating any waiting times from the user requests. On the other hand, we can also see that for most possible cases at least a third of all requests had an hint, and overall our system has an average availability of 38%. In a full deployment, as the hint graph accumulates more submissions from multiple academic years, we anticipate a consistent improvement in these results.

Another very important factor in the system's performance is also the effectiveness of each hint in guiding the student. Although the quality of each hint is subjective to each individual, a quantifiable factor we can use to evaluate each policy rule is the number of steps that the system expects an "ideal" user (one

[1] hub.docker.com/_/mongo

[2] catalog.redhat.com/.../ubi8/openjdk-17

that always guesses the next state implied by the provided hints) would take to reach a correct solution on each incorrect submission. This metric changes in accordance with the underlying policy rules, providing an insight into their quality. In Figure 31, we present for each policy of Section 5.1.4 the distribution of incorrect submissions of the testing data-set based on the number of steps the systems expects of its users.

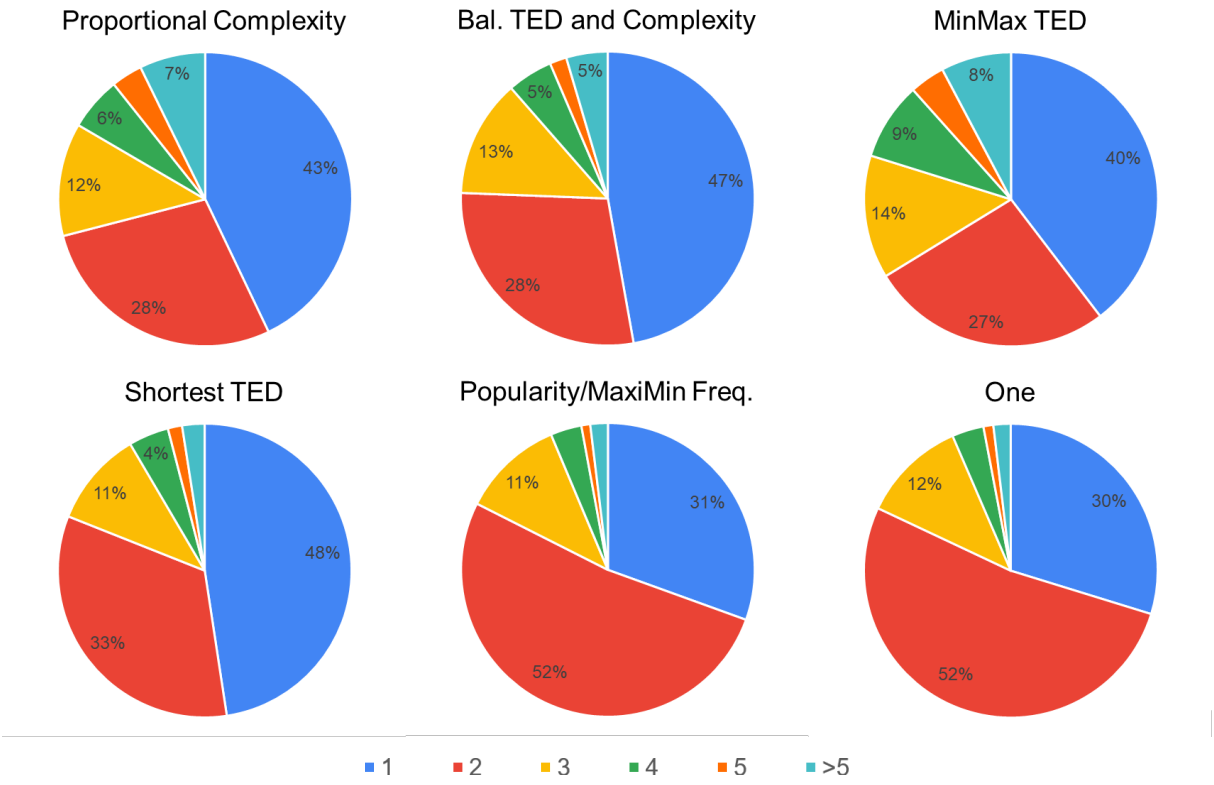


Figure 31: Distribution of incorrect submissions by expected steps across multiple policies

Overall, despite variations in the rules, our policies consistently needed a maximum of two hints to fix 70% of invalid submissions. These findings enable us to infer that, in general, our system will be efficient in providing hints to the user. This imposes a baseline level of quality on hints, since the reduced number of steps, and consequently hints, will help the user understand and solve the exercise faster. However, it is worth noting that we have some difference between policies. Notably, the "Popularity", "MaxiMin Frequency", and "One" policies tend to provide more hints than the other policies, since they generally prefer using two steps over one.

RQ2 Is it viable to frequently update the submission graphs of SpecAssistant?

In order to update a hint model we must recompute it. This process demands a significant amount of resources, and despite being an offline operation that can run concurrently with the other functionalities, it is significantly impeded by the number of models to update and the duration of the training process of each.

As detailed in Section 5.2.4, our hint model computation process comprises three distinct phases: Model Ingestion, Attribute Computation, and Policy Computation. The Attribute Computation procedure exhibits near-instant execution times. In contrast, the number of iterations of the other two stages grow significantly with the number of inputs. Notably, the Model Ingestion's performance is tied to the size of the derivation trees, while Policy Computation's performance depends on the size of the submission graph. However, the format of the derivation trees allows our implementation of the Model Ingestion process to be parallelized, being capable of simultaneously parsing as many models as there are available workers. On the other hand, the Policy Computation employs a semi-parallel computing approach, where it will process as many graph nodes as it is permitted, as a result of the existing variable dependencies between its iterations. As a result, to perform a thorough evaluation we imposed limits on our CPU scheduler and tested the performance of each procedure under three distinct resource scenarios: one with 18 available concurrent threads, a second with 12, and finally, one using only 6 concurrent threads. The average results are presented in Table 16.

Exercise	**Number of Submissions**	**Number of States**	**Model Ingestion (s)**			**Policy Computation (s)**		
			6 **THD**	12 **THD**	18 **THD**	6 **THD**	12 **THD**	18 **THD**
Courses	10431	4104	312.9	274.6	215.3	1.271	1.009	0.883
Social Network	10428	3605	218.8	180.6	165.5	2.262	0.91	0.99
Train Station	4394	1874	95	88.7	69.8	1.14	1.198	0.953
Production Line	4156	1862	138.6	125.1	113.6	1.279	1.154	0.542
Trash LTL	2788	1219	68.8	58.8	52.2	0.517	0.392	0.403
Classroom FOL	2702	903	60.9	53.2	45.2	0.5	0.402	0.431
Classroom RL	2474	985	66.7	55.5	46.7	0.475	0.493	0.391
Trash RL	1530	446	43.4	37	31.5	0.43	0.41	0.27
Trash FOL	1425	343	38.4	31.1	28.6	0.263	0.277	0.214
Graphs	1281	599	35.1	32.7	27.7	0.649	0.364	0.522
LTS	995	489	33.1	25.5	22.8	0.373	0.707	0.61
CV	596	324	13	12.4	11.7	0.24	0.318	0.403
Production Line v1	424	176	10.3	8.6	7.7	0.412	0.327	0.291

Table 16: Hint Model computation performance

The Model Ingestion process demonstrates a noticeable improvement as the number of processor units increases. However, accurately predicting its behavior with a speedup formula, such as Amdahl's Law [16], is challenging. This challenge arises from the fact that although the algorithm excels at scheduling each model for parsing, the inconsistent performance of the employed SAT solver, along with the file system calls wait times, introduces a significant degree of randomness and uncertainty into the total execution time of the parsing task. The Policy Computation time also exhibits a slight improvement in certain cases, but generally seems to be unaffected by the different number of threads. Nevertheless, considering its reduced execution time when compared to the entire process, it can be concluded that its performance impact is negligible.

As we can see, even in our worst resource case scenario the hint model computation processes took around 19 minutes for the 13 exercises with a total of 43000 submissions, which means we could possibly

update this daily if the number of newly added submissions justified the process.

RQ3 To what extent does the incorporation of mutated formulas affects the performance?

As visible in Table 17, our mutation technique undeniably delivers a significant boost to our system's availability. It provides a 55% average increase in the amount of hints in the older exercises and a 45% average increase in the newer ones, resulting in an overall 50% average increase in the availability. Regrettably, this enhancement comes at the expense of a slightly prolonged execution time, being nearly one hundred times slower than the technique without the mutations. Despite this, it still provides its answers within a fraction of a second, a period which is well bellow our target quality threshold of 10 seconds.

Exercise	**Hint Requests**	**No mutations Hint Rate**	**With mutations Hint Rate**	**Gain**	**Elapsed Time** (s)	
					Avg.	**Std. Dev.**
Courses	2418	812 (34%)	1298 (54%)	486 (60%)	0.659	0.367
Social Network	2793	1265 (45%)	1740 (62%)	475 (38%)	0.666	0.344
Train Station	1331	529 (40%)	751 (56%)	222 (42%)	0.811	0.463
Production Line	1102	373 (34%)	525 (48%)	152 (41%)	0.720	0.401
Trash LTL	890	357 (40%)	569 (64%)	212 (59%)	0.416	0.141
Classroom FOL	663	331 (50%)	463 (70%)	132 (40%)	0.455	0.167
Classroom RL	687	239 (35%)	330 (48%)	91 (38%)	0.306	0.147
Graphs	370	162 (44%)	251 (68%)	89 (55%)	0.242	0.127
Trash RL	347	195 (56%)	260 (75%)	65 (33%)	0.205	0.114
TrashFOL	194	100 (52%)	154 (79%)	54 (54%)	0.348	0.191
LTS	393	74 (19%)	141 (36%)	67 (91%)	0.242	0.098
CV	218	44 (20%)	48 (22%)	4 (9%)	0.676	0.390
Production Line v1	120	35 (29%)	38 (32%)	3 (9%)	0.449	0.230

Table 17: Performance summary of the mutation technique across each exercise

RQ4 How does SpecAssistant performance stacks up against other alternatives?

Our primary emphasis was on comparing our system with TAR. This choice stemmed from the fact that it is the only Alloy hint generation systems that currently supports Alloy version 6, besides being the only developed specifically for Alloy4Fun. Table 18 compares the results of running the benchmarks on TAR and our system. It shows the percentage of hints provided by each tool, as well as the percentage of submissions for which they can both provide hints. It also displays the execution times of TAR. As we can see, there is a significant performance gap between the two techniques. As previously seen, our system excels in delivering rapid responses, while TAR semantic equivalence checks hinder its execution time. The average response time of TAR is approximately 30 seconds, with a standard deviation of 20 seconds. This average is well above the threshold of 10 seconds, and thus it could make some users start to disregard the tool. Additionally, the exceptionally high standard deviation is indicative that for a significant number of cases the performance could be much worse.

Exercise	Hint Requests	SpecAssistant Hint Rate	TAR Hint Rate	Duplicate Rate	Elapsed Time (s)	
					Avg.	Std. Dev.
Courses	2418	812 (34%)	502 (21%)	30 (5%)	38.564	26.041
Social Network	2793	1265 (45%)	539 (19%)	20 (2%)	44.615	23.290
Train Station	1331	529 (40%)	320 (24%)	9 (2%)	38.203	26.533
Production Line	1102	373 (34%)	297 (27%)	15 (4%)	38.544	25.146
Trash LTL	890	357 (40%)	771 (87%)	23 (4%)	29.504	20.198
Classroom FOL	663	331 (50%)	182 (27%)	13 (5%)	26.748	22.974
Classroom RL	687	239 (35%)	159 (23%)	7 (4%)	35.898	23.803
Graphs	370	162 (44%)	175 (47%)	0 (0%)	20.998	21.882
Trash RL	347	195 (56%)	237 (68%)	3 (1%)	20.915	22.424
TrashFOL	194	100 (52%)	167 (86%)	5 (4%)	20.950	21.806
LTS	393	74 (19%)	37 (9%)	0 (0%)	18.226	19.195
CV	218	44 (20%)	61 (28%)	2 (4%)	48.209	16.811
Production Line v1	120	35 (29%)	106 (88%)	0 (0%)	20.091	20.211

Table 18: Performance summary of TAR across each exercise

Concerning availability, both TAR, which has the potential to offer hints to any submission, and our system, which can only yield results for submissions it has seen before (without the mutations), exhibit similar outcomes by providing hints to approximately 40% of the requests, on average. What is particularly intriguing is the low overlap between both techniques – when we compare the number of requests for which both tools provided hints, we can see that it is rather small. As a result we can conclude that both tools complement each other, and thus it is expected that a system that combines both could significantly enhance the overall availability, possibly increasing the availability to around 80%.

7

Conclusions and Future Work

7.1 Conclusions

In contemporary software development it is imperative to ensure the proper functioning of software applications. Among other techniques, this can be achieved by building robust models during the early development phases. The most effective models are formal specifications which rely on mathematical concepts to remove ambiguities. However, these require a high proficiency from the developers. Our thesis aimed at easing the learning of the Alloy formal modeling language, hopefully making it more accessible to any developer.

To be more precise, our goal was to conceive a new hint generation system that could improve the Alloy learning experience, intended to be deployed on the Alloy4Fun online platform. There were already some existing Alloy automated repair and hint generating techniques, such as TAR [10], BeAFix [8] and FLACK [50], which mainly rely on semantic comparison techniques to produce outputs. These techniques suffered from poor performance, which rendered them unusable in a platform such as Alloy4Fun where users expect near instant feedback. In light of this, we have decided to explore the application of data-driven techniques, which have been relatively underexplored within the context of Alloy, and that could potentially solve the performance problem, by mining solution patterns from a pool of historical submissions which could then be translated into hints.

The developed system, known as SpecAssistant, compiles the historical submission data into a "Submission Graph". This structure enables SpecAssistant to compute "Hint Models", which can be used to guide users towards valid answers to the specification challenges. When presented with an invalid Alloy formula, the system processes it and then uses the computed policy to suggest the next action. The hints are delivered as code highlights and textual explanations. To evaluate SpecAssistant, we primarily focused on quantifying its availability and performance. This process involved partitioning a historical submission data-set [22] into training and testing data-sets, which were subsequently used to run some benchmarks and compare our system with TAR, the hint system previously developed for Alloy4Fun. As anticipated, SpecAssistant consistently provides hints within milliseconds, effectively eradicating any user waiting times. Our findings also indicate that hint availability is still good, since it can provide hints for at

least 33% of invalid submissions in the testing data-set, a ratio that can be improved to 50% through the integration of formula mutation techniques into the process.

In conclusion, we believe SpecAssistant has fulfilled the objectives for this thesis. We leveraged data-driven methods to create a hint generation system that is capable of instantly and effectively offer assistance to users by leveraging data collected from peers in the past.

This development of SpecAssistant was challenging, mostly because of the nature of the available historical data. The primary objective of a hint system is to offer suggestions that indeed help users reach valid solutions and have a positive impact on learning. However, there is no guaranty that historical peer data is the best source of hints for that. We attempted to mitigate this by allowing challenge developers to customize the policy selection rules, however the problem may still persist. On a more technical perspective, Alloy's Analyzer also proved difficult to work with, as its compiler and AST formats are not specially suited for our normalization and parsing procedures.

7.2 Future Work

We have identified three aspects in which our work can be improved: the evaluation process, the implementation, and the hint computation and delivery technique.

In order to improve our evaluation, it is essential that we undertake user studies in the future. While these studies were not included in the thesis plan, they can offer invaluable insights into user behaviors when exposed to our hints. This information is essential for a comprehensive assessment of the hints' quality, i.e. their ability to aid users in solving challenges and improve their learning.

To improve SpecAssistant's implementation, we should start by improving the underlying AST structures. As mentioned earlier, the use of Alloy Analyzer has presented us with a series of problems during the normalization process, particularly in dealing with invalid AST nodes, which led to some undesirable workarounds. A redevelopment of this structure could remove all these issues, consequently enhancing the outcomes of our AST normalization and differencing algorithms. It would also streamline other needed system improvements, such as the implementation of an Alloy parser suited for our model parsing procedures.

Another possible improvement would be enhancing the way the system computes and delivers hints. Improving hint computation requires improving our current data-models. This can be achieved by either adding new attributes to be used in policy rules or by shifting from our state-based data-model to a session-based one (i.e. a model that generate hints from the full array submissions of a session). Improving hint deliver, requires improving the text messages that are provided with the hints. To do this we could improve our text generation technique using the rapidly evolving deep learning language models.

Bibliography

[1] Anne Adam and Jean-Pierre H. Laurent. “LAURA, A System to Debug Student Programs”. In: *Artif. Intell.* 15.1-2 (1980), pp. 75–122 (cit. on p. 36).

[2] Elena N. Akimova, Alexander Yu. Bersenev, Artem A. Deikov, Konstantin S. Kobylkin, Anton V. Konygin, Ilya P. Mezentsev, and Vladimir E. Misilov. “A survey on software defect prediction using deep learning”. In: *Mathematics* 9.11 (2021), p. 1180 (cit. on p. 34).

[3] Frances E. Allen. “Control flow analysis”. In: *Symposium on Compiler Optimization, Urbana- Champaign, Illinois, USA, July 27-28, 1970*. Ed. by Robert S. Northcote. ACM, 1970, pp. 1–19 (cit. on p. 48).

[4] Paolo Antonucci, H.-Christian Estler, Durica Nikolic, Marco Piccioni, and Bertrand Meyer. “An Incremental Hint System For Automated Programming Assignments”. In: *ACM Conference on Innovation and Technology in Computer Science Education*. Ed. by Valentina Dagiene, Carsten Schulte, and Tatjana Jevsikova. ACM, 2015, pp. 320–325 (cit. on p. 38).

[5] Tiffany Barnes and John C. Stamper. “Toward Automatic Hint Generation for Logic Proof Tutoring Using Historical Student Data”. In: *Intelligent Tutoring Systems, 9th International Conference*. Ed. by Beverly Park Woolf, Esma Aïmeur, Roger Nkambou, and Susanne P. Lajoie. Vol. 5091. Lecture Notes in Computer Science. Springer, 2008, pp. 373–382 (cit. on p. 39).

[6] Sahil Bhatia and Rishabh Singh. “Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks”. In: *CoRR* (2016) (cit. on p. 42).

[7] Simón Gutiérrez Brida, Germán Regis, Guolong Zheng, Hamid Bagheri, ThanhVu Nguyen, Nazareno Aguirre, and Marcelo F. Frias. “BeAFix: An Automated Repair Tool for Faulty Alloy Models”. In: *36th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2021, pp. 1213–1217 (cit. on p. 36).

[8] Simón Gutiérrez Brida, Germán Regis, Guolong Zheng, Hamid Bagheri, ThanhVu Nguyen, Nazareno Aguirre, and Marcelo F. Frias. “Bounded Exhaustive Search of Alloy Specification Repairs”. In: *43rd*

IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021. IEEE, 2021, pp. 1135–1147 (cit. on pp. 36, 75).

[9] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems 33*. Ed. by Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin. 2020 (cit. on p. 43).

[10] Jorge Cerqueira, Alcino Cunha, and Nuno Macedo. "Timely Specification Repair for Alloy 6". In: *Software Engineering and Formal Methods - 20th International Conference*. Ed. by Bernd-Holger Schlingloff and Ming Chai. Vol. 13550. Lecture Notes in Computer Science. Springer, 2022, pp. 288–303 (cit. on pp. 2, 3, 36, 38, 51, 75).

[11] Rui Couto, José Creissac Campos, Nuno Macedo, and Alcino Cunha. "Improving the Visualization of Alloy Instances". In: *4th Workshop on Formal Integrated Development Environment*. Ed. by Paolo Masci, Rosemary Monahan, and Virgile Prevosto. Vol. 284. EPTCS. 2018, pp. 37–52 (cit. on p. 17).

[12] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L. Thomas van Binsbergen. "Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback". In: *Int. J. Artif. Intell. Educ.* 27.1 (2017), pp. 65–100 (cit. on p. 38).

[13] Alex Gerdes, Johan Jeuring, and Bastiaan Heeren. "Using strategies for assessment of programming exercises". In: *41st ACM technical symposium on Computer science education*. Ed. by Gary Lewandowski, Steven A. Wolfman, Thomas J. Cortina, and Ellen Lowenfeld Walker. ACM, 2010, pp. 441–445 (cit. on p. 38).

[14] Rahul Gupta, Aditya Kanade, and Shirish K. Shevade. "Deep Reinforcement Learning for Programming Language Correction". In: *CoRR* (2018) (cit. on p. 42).

[15] Andrew Head, Elena L. Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. "Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis". In: *4th ACM Conference on Learning @ Scale*. Ed. by Claudia Urrea, Justin Reich, and Candace Thille. ACM, 2017, pp. 89–98 (cit. on p. 38).

[16] Mark D. Hill and Michael R. Marty. "Retrospective on Amdahl's Law in the Multicore Era". In: *Computer* 50.6 (2017), pp. 12–14 (cit. on p. 72).

[17] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2012 (cit. on pp. 1, 2, 5).

[32] Finn V. Jensen and Thomas D. Nielsen. “Bayesian Networks and Decision Graphs”. In: *Knowl. Eng. Rev.* 23.4 (2008), p. 413 (cit. on p. 48).

[18] W. Lewis Johnson and Elliot Soloway. “PROUST: Knowledge-Based Program Understanding”. In: *IEEE Trans. Software Eng.* 11.3 (1985), pp. 267–275 (cit. on p. 36).

[19] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. “Semi-supervised verified feedback generation”. In: *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Ed. by Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su. ACM, 2016, pp. 739–750 (cit. on p. 38).

[20] Chao Liu, Ryen W. White, and Susan T. Dumais. “Understanding web browsing behaviors through Weibull analysis of dwell time”. In: *33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. Ed. by Fabio Crestani, Stéphane Marchand-Maillet, Hsin-Hsi Chen, Efthimis N. Efthimiadis, and Jacques Savoy. ACM, 2010, pp. 379–386 (cit. on p. 70).

[21] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. “Lightweight specification and analysis of dynamic systems with rich configurations”. In: *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Ed. by Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su. ACM, 2016, pp. 373–383 (cit. on p. 17).

[22] Nuno Macedo, Alcino Cunha, and Ana C. R. Paiva. *Alloy4Fun Dataset for 2022/23*. Version EM 2022/23. Zenodo, 2023-07 (cit. on pp. 2, 3, 32, 68, 75).

[23] Nuno Macedo, Alcino Cunha, José Pereira, Renato Carvalho, Ricardo Silva, Ana C. R. Paiva, Miguel Sozinho Ramalho, and Daniel Castro Silva. “Experiences on teaching Alloy with an automated assessment platform”. In: *Sci. Comput. Program.* 211 (2021), p. 102690 (cit. on pp. 2, 25).

[24] Mehak Maniktala, Christa Cody, Amy Isvik, Nicholas Lytle, Min Chi, and Tiffany Barnes. “Extending the Hint Factory for the assistance dilemma: A novel, data-driven HelpNeed Predictor for proactive problem-solving help”. In: *CoRR* (2020) (cit. on p. 40).

[25] Michael L. Mauldin. “Semantic Rule Based Text Generation”. In: *10th International Conference on Computational Linguistics and 22nd Annual Meeting of the Association for Computational Linguistics*. Ed. by Yorick Wilks. ACL, 1984, pp. 376–380 (cit. on p. 43).

[26] Jessica McBroom, Irena Koprinska, and Kalina Yacef. “A Survey of Automated Programming Hint Generation: The HINTS Framework”. In: *ACM Comput. Surv.* 54.8 (2022), 172:1–172:27 (cit. on pp. 3, 34).

[27] Martin Monperrus. “Automatic Software Repair: A Bibliography”. In: *ACM Comput. Surv.* 51.1 (2018), 17:1–17:24 (cit. on pp. 2, 34).

[28] H. Hernan Moraldo. “An Approach for Text Steganography Based on Markov Chains”. In: *CoRR* (2014) (cit. on p. 43).

[29] Weili Nie, Nina Narodytska, and Ankit Patel. “RelGAN: Relational Generative Adversarial Networks for Text Generation”. In: *7th International Conference on Learning Representations*. OpenReview.net, 2019 (cit. on p. 43).

[30] Andy Oram and Greg Wilson, eds. *Making Software - What Really Works, and Why We Believe It*. Theory in practice. O’Reilly, 2011 (cit. on p. 1).

[31] Benjamin Paaßen, Barbara Hammer, Thomas William Price, Tiffany Barnes, Sebastian Gross, and Niels Pinkwart. “The Continuous Hint Factory - Providing Hints in Vast and Sparsely Populated Edit Distance Spaces”. In: *CoRR* (2017) (cit. on p. 40).

[33] Mateusz Pawlik and Nikolaus Augsten. “Efficient Computation of the Tree Edit Distance”. In: *ACM Trans. Database Syst.* 40.1 (2015), 3:1–3:40 (cit. on pp. 60, 62).

[34] Mateusz Pawlik and Nikolaus Augsten. “Tree edit distance: Robust and memory-efficient”. In: *Inf. Syst.* 56 (2016), pp. 157–173 (cit. on pp. 53, 60, 62).

[35] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas J. Guibas. “Autonomously Generating Hints by Inferring Problem Solving Policies”. In: *2nd Second ACM Conference on Learning @ Scale*. Ed. by Gregor Kiczales, Daniel M. Russell, and Beverly P. Woolf. ACM, 2015, pp. 195–204 (cit. on p. 39).

[36] Thomas W. Price, Yihuan Dong, Rui Zhi, Benjamin Paaßen, Nicholas Lytle, Veronica Cateté, and Tiffany Barnes. “A Comparison of the Quality of Data-Driven Programming Hint Generation Algorithms”. In: *Int. J. Artif. Intell. Educ.* 29.3 (2019), pp. 368–395 (cit. on p. 2).

[37] Thomas W. Price, Rui Zhi, and Tiffany Barnes. “Hint Generation Under Uncertainty: The Effect of Hint Quality on Help-Seeking Behavior”. In: *Artificial Intelligence in Education - 18th International Conference*. Ed. by Elisabeth André, Ryan S. Baker, Xiangen Hu, Ma. Mercedes T. Rodrigo, and Benedict du Boulay. Vol. 10331. Lecture Notes in Computer Science. Springer, 2017, pp. 311–322 (cit. on p. 39).

[38] Kelly Rivers and Kenneth R. Koedinger. “Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor”. In: *Int. J. Artif. Intell. Educ.* 27.1 (2017), pp. 37–64 (cit. on pp. 2, 40).

[39] Higor Amario de Souza, Marcos Lordello Chaim, and Fabio Kon. “Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges”. In: *CoRR* (2016) (cit. on p. 36).

[40] John Stamper, Tiffany Barnes, Lorrie Lehmann, and Marvin Croy. “The hint factory: Automatic generation of contextualized help for existing computer aided instruction”. In: *9th International Conference on Intelligent Tutoring Systems Young Researchers Track*. 2008, pp. 71–78 (cit. on pp. 39, 44).

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems 30*. Ed. by Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett. 2017, pp. 5998–6008 (cit. on p. 43).

[42] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. “Automated model repair for Alloy”. In: *33rd ACM/IEEE International Conference on Automated Software Engineering*. Ed. by Marianne Huchard, Christian Kästner, and Gordon Fraser. ACM, 2018, pp. 577–588 (cit. on p. 36).

[43] Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. “Fault Localization for Declarative Models in Alloy”. In: *31st IEEE International Symposium on Software Reliability Engineering*. Ed. by Marco Vieira, Henrique Madeira, Nuno Antunes, and Zheng Zheng. IEEE, 2020, pp. 391–402 (cit. on p. 37).

[44] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. “Automatic program repair with evolutionary computation”. In: *Commun. ACM* 53.5 (2010), pp. 109–116 (cit. on p. 2).

[45] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. “A Survey on Software Fault Localization”. In: *IEEE Trans. Software Eng.* 42.8 (2016), pp. 707–740 (cit. on p. 36).

[46] W. Eric Wong and J. Jenny Li. “An Integrated Solution for Testing and Analyzing Java Applications in an Industrial Setting”. In: *12th Asia-Pacific Software Engineering Conference*. IEEE Computer Society, 2005, pp. 576–583 (cit. on p. 36).

[47] Franz Wotawa. “Fault Localization Based on Dynamic Slicing and Hitting-Set Computation”. In: *10th International Conference on Quality Software*. Ed. by Ji Wang, W. K. Chan, and Fei-Ching Kuo. IEEE Computer Society, 2010, pp. 161–170 (cit. on p. 36).

[48] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. “A brief survey of program slicing”. In: *ACM SIGSOFT Softw. Eng. Notes* 30.2 (2005), pp. 1–36 (cit. on p. 36).

[49] Michihiro Yasunaga and Percy Liang. “Break-It-Fix-It: Unsupervised Learning for Program Repair”. In: *38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Machine Learning Research. PMLR, 2021, pp. 11941–11952 (cit. on p. 42).

[50] Guolong Zheng, ThanhVu Nguyen, Simón Gutiérrez Brida, Germán Regis, Marcelo F. Frias, Nazareno Aguirre, and Hamid Bagheri. “FLACK: Counterexample-Guided Fault Localization for Alloy Models”. In: *43rd IEEE/ACM International Conference on Software Engineering*. IEEE, 2021, pp. 637–648 (cit. on pp. 37, 75).

[51] Wanrong Zhu, Zhiting Hu, and Eric P. Xing. “Text Infilling”. In: *CoRR* (2019) (cit. on p. 43).