**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

José Manuel Costa Pereira

# A Web-based Social Environment for Alloy

**Dissertation**

October 2016

**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

José Manuel Costa Pereira

# A Web-based Social Environment for Alloy

**Dissertation**

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
**Professor Doutor Manuel Alcino Pereira da Cunha**
**Doutor Nuno Filipe Moreira Macedo**

October 2016

# ACKNOWLEDGEMENTS

I would mainly like to thank Prof. Alcino Cunha for the opportunity to develop such an interesting and fun project and also to the person that suggested it to me, my colleague and friend Eduardo Pessoa. These two persons as well as Dr. Nuno Macedo were invaluable contributions to the completion of this thesis trough their constant support, advice and enthusiasm.

I would also like to thank my colleagues for all the tips and advices that so many times helped me contour development obstacles.

Finally, i must express my profound gratitude to my parents and to my girlfriend Andreia for their encouragement and joy throughout my academic years. Thank you all.

This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project POCI-01-0145-FEDER-016826.

## ABSTRACT

Alloy is a declarative specification language which describes rules and complex structural behaviors. Alloy Analyzer is used to analyze this specifications, this tool generates concrete instances from the invariants specified in a model, it simulates sequences of defined operations and verifies whether properties introduced are valid or not. Currently, the tool is available as a runnable .jar and it contains a trivial GUI to interact with it. Being such, it requires JAVA installed. It's in the best interest of the community to achieve and easier access to this tool through a web platform that shall support it in real time and also allow sharing models developed in it by users. Formal methods of software development are growing and they would also benefit from the constructive feedback obtained through this platform regarding the Alloy language/tool.

## RESUMO

Alloy é uma linguagem de especificação declarativa que descreve regras e comportamentos de estruturas complexas. Para analisar estas especificações, utiliza-se o Alloy Analyzer, uma ferramenta que gera instâncias concretas a partir dos invariantes especificados num modelo, simula sequências de operações definidas e verifica se as propriedades introduzidas são satisfeitas. Atualmente, a ferramenta está disponível sob a forma de um `.jar` executável e contém uma *GUI* trivial para a sua utilização, requerendo assim o respetivo download e instalação do software adicional *JAVA* até estar operacional. É do interesse da comunidade conseguir um acesso facilitado a esta ferramenta através de uma plataforma web que a suporta em tempo real e permite a partilha simplificada de trabalhos elaborados. Também seria uma mais valia para os métodos formais na produção de software, que se encontram em constante crescimento, a extração de informação estatística acerca da utilização desta plataforma. Daqui poderia obter-se um feedback positivo face aos prós e contras da linguagem/ferramenta.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF LISTINGS

# 1 INTRODUCTION

The ever-increasing dependence of society on software impacts a higher demand for its quality. Formal methods for software development can be used as an auxiliary measure in the attempt to build such quality software. These are a set of mathematically rigorous techniques and tools applied to the specification, design, development and testing stages of software construction.

This report focuses on the first stages of software development, namely its specification and design, and on a specific formal specification language - *Alloy*, a language developed by the Software Design Group at MIT.

Alloy (Jackson, 2012) is a declarative specification language which describes rules and complex structural behaviours through a collection of constraints. *Alloy Analyzer* is a solver used to analyze such specifications. This tool generates concrete instances from the constraints specified in a model, it simulates sequences of defined operations, and verifies whether the specified properties are valid or not. Currently, this tool is available as a runnable `.jar` and it contains a simple GUI to interact with it, therefore requiring JAVA installed to be executed.

It is in the best interest of the community to obtain an easier access to this tool through a web platform that would support it in real time, allowing the sharing of models with other users and introduce competition through the completion of small challenges. Formal methods for software development are growing and they would also benefit from the constructive feedback obtained through the data generated from the user's interaction with this platform(Ball et al., 2013).

A high number of Massive Open Online Courses (MOOCs) exist nowadays as well as web platforms that organise and manage competitive programming contests. These provide useful information about the kind of features that a environment such as this should have to be successful.

The main objective of this project is to develop a web framework for Alloy, supporting most of the features currently available in the standalone *Alloy Analyzer*, and others specific to the web environment, in particular:

- The ability to perform model analysis server-side. This implies wrapping the *Alloy API* as a web-service.
- The ability to easily write and share Alloy models and instances. Features such as syntax highlighting or the creation of permalinks to allow the sharing of models and instances are essential towards this goal.
- An instance visualisation mechanism likewise to the one in the *Alloy Analyzer*, namely supporting user-defined themes, and possibly improving some of its drawbacks.
- Additionally, this endeavour aims at understanding the behaviour of users within the community when coding as well as their opinion regarding the *Alloy* notation and tool. Hence, a statistics gathering system should be embedded on the environment to collect information upon usage.

To accomplish this goal, several relevant technologies and related frameworks were researched, in particular:

- Explore possible web development technologies capable of supporting the desired environment in order to identify which best suit its implementation.
- Understand the relevance of the proposed features through the exploration of websites and tools with similar purposes.
- Gather information on graph visualization techniques and improvements previously proposed to the current *Alloy Analyzer*'s visualization facility.

This research was indispensable because not only it inspired some of the platform's features but also supported the development of a truly advantageous environment given the feedback gathered from the previous usage of the *Alloy Analyzer*.

Chapter 2 briefly explains the *Alloy* notation using a model example. It also gives insight on the *Alloy Analyzer*'s features and most common critics associated to its usage. Chapter 3, Related Work, presents some reviews of websites and tools that offer similar services relevant to the framework being developed here. Furthermore, it describes some related work regarding the improvement of the *Alloy Analyzer*'s graph visualisation facility, as well as generic approaches of the sort applied on similar tools. Chapter 4 is a description of most of the tools used to develop the environment as well as the reasons that led to their usage. Chapter 5 explains the implementation of the platform's components, how they connect, and their role in the overall solution. Finally, chapter 6 presents this thesis contributions and the work yet to be done on top of the built *web* environment.

# 2

# BACKGROUND

This chapter introduces the *Alloy* notation using a practical example of a puzzle's specification. Furthermore, the *Alloy Analyzer* is presented, highlighting its features and some related criticisms in order to better understand user's needs when faced with the tool.

## 2.1 ALLOY

The Alloy (Jackson, 2002) modeling language, first proposed in 1997, was idealised from a set of powerful and fitting features of previously existing tools. It gathered the simple and intuitive first-order logic semantics of Z notation (Spivey and Abrial, 1992) and the navigation style and type hierarchy of object modeling languages such as Object-Modeling Technique(OMT) (Rumbaugh et al., 1991). Also, the Z notation was adapted to ease the model analysis, featuring the usage of SAT solvers to automatically verify them within a bounded scope. Alloy was built to be a small, but powerful notation that provides users with an easy writing and reading.

### 2.1.1 *Applications and Framing*

Alloy spread trough a wide range of applications from finding bugs in security mechanisms to designing telephone switching networks, as well as used in many Formal Methods courses, and even introductory courses to logic and mathematical reasoning to high-school students (Ferreira et al., 2011). This latter usage is particularly relevant to this subject due to its feedback, considering we want to increase the human-tool interaction via the Web, which raises the question: What makes Alloy enticing?

Alloy can be considered a challenge because it promotes a depth of mathematical thinking, in fact one could sometimes consider it a puzzle as the following example will present it. This reaction was observed on some of the students of these Formal Method's courses which use the notation (Boyatt and Sinclair, 2008). Such information leads to believe that Alloy would benefit from a web environment where challenges could be easily created and shared.

### 2.1.2 *The Language*

To better demonstrate the language procedures, the example `frog.als` will be presented (`.als` is the extension given to *Alloy* notation files). This example specifies a popular puzzle available for solving at many websites. In this puzzle there are green frogs and brown frogs, three of each. There's also seven stones aligned in which the first three contain each a green frog. In the middle stands an empty stone and the three final stones all contain a brown frog each as shown in Figure 1.

The objective is to swap sides of both green and brown frogs, leaving the same empty stone in between them. Each frog can only move forward (green frogs to the right and brown frogs to the left). Also, frogs can only either move to an empty stone standing immediately next to them or jump over a single frog followed by an empty stone.

The idea is to specify all the constraints and bounds so that the *Alloy Analyzer* may solve the puzzle, and display sequences of possible frog moves to achieve the required victory conditions. The reader is not expected to fully understand the specification in Listings 2.1, as only some of it will be explained in detail.

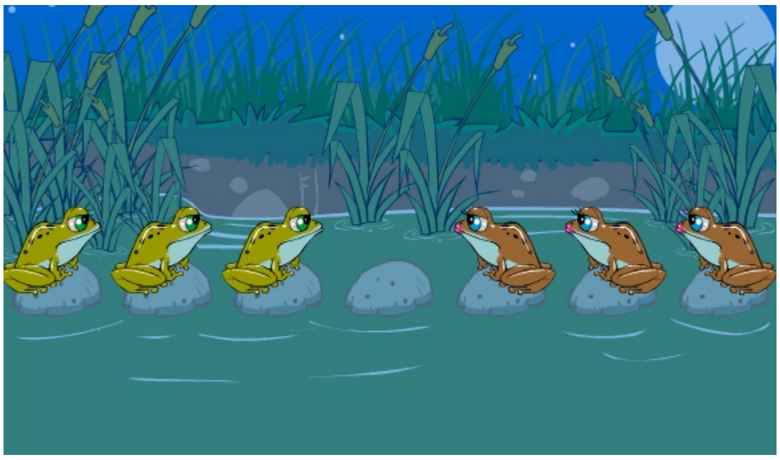


Figure 1: The frog puzzle

```
open util/ordering[Time]
open util/ordering[Stone]

sig Time {
  position : Frog -> one Stone
}

sig Stone {}

abstract sig Frog {}
sig Green, Brown extends Frog {}

fact {
  all t : Time, s : Stone | lone t.position.s
}

pred start[t : Time] {
  all g : Green | some g.(t.position).prev implies some (t.position).(g.(t.position).
      prev) & Green
  all b : Brown | some b.(t.position).next implies some (t.position).(b.(t.position).
      next) & Brown
}

pred end[t : Time] {
  all g : Green | some g.(t.position).next implies some (t.position).(g.(t.position).
      next) & Green
  all b : Brown | some b.(t.position).prev implies some (t.position).(b.(t.position).
      prev) & Brown
}

pred moveG [g : Green, t,t' : Time] {
  some g.(t.position).next
  t'.position = t.position - g->Stone + g->g.(t.position).next
}

pred jumpG [g : Green, t,t' : Time] {
  some (t.position).(g.(t.position).next) & Brown
  some g.(t.position).next.next
  t'.position = t.position - g->Stone + g->g.(t.position).next.next
}

pred moveB [b : Brown, t,t' : Time] {
  some b.(t.position).prev
  t'.position = t.position - b->Stone + b->b.(t.position).prev
}

pred jumpB [b : Brown, t,t' : Time] {
  some (t.position).(b.(t.position).prev) & Green
  some b.(t.position).prev.prev
  t'.position = t.position - b->Stone + b->b.(t.position).prev.prev
}

fact {
  start[first]

  all t : Time, t' : t.next {
    (some g : Green | moveG[g,t,t'] or jumpG[g,t,t'])
    or
    (some b : Brown | moveB[b,t,t'] or jumpB[b,t,t'])
  }
}

run {
  some t : Time | end[t]
} for exactly 3 Green, exactly 3 Brown, exactly 7 Stone, 16 Time
```

Listing 2.1: The frog puzzle in *Alloy*

At the top of the example, some signatures are declared using the keyword `sig`. These are sets of atoms and can be seen as the model's types or classes. The `extends` and `abstract` tokens whom resemble object-oriented notations, allow an hierarchy between signatures which is convenient for the matter in hand since it describes a natural order between the concepts presented.

Within the declaration of the `Time` type, there is the following statement: `position : Frog -> one Stone`. This declares a *field* named `position` that is a ternary relationship between signatures `Time`, `Frog` and `Stone`. The `one` keyword is one of Alloy's multiplicity constraints and translates to *exactly one*. Hence, the former statement can be read as *At each instance of time, each frog is positioned at one stone.*

Above the mentioned signatures, there are two `open` statements, these import an *Alloy*'s core module that introduces a notion of total order in a signature. In this case the specification states that both `Time` and `Stone` atoms have a relation of total order in their atoms. Hence the `next` and `prev` relations that appear further down allow the navigation between atoms of theses signatures. Additionally, the function `first` selects the initial atom of an order and this is useful to state things like "The initial instant of time" or "The first stone".

Functions define a way of getting relations, sets or atoms and they can take parameters that are used in getting its results. Although the model does not contain any, they could be defined using the keyword *fun* explained in the *Alloy*'s documentation. The declarations of predicates and facts can be observed instead at *frog.als* using the `pred` and `fact` keywords respectively.

Predicates define formulas which can be true or false. They can also take parameters and be invoked in facts and assertion declarations. There are six predicates declared at `frog.als`.

- `start` takes in an instant of time and verifies if at it, all the frogs are positioned accordingly to the initial disposition of the puzzle.

- `end` confirms if the puzzle has been solved by checking, analogously to `start`, if all the frogs are in their required positions.

- `moveG` and `jumpG` describe valid movements of green frogs while `moveB` and `jumpB` refer to the same movements of brown frogs.

Facts are used to describe more complex constraints which could not be expressed in the signature declarations. Take for example the second `fact` declared in `frog.als`. It initially states: `start[first]`. `start` is a predicate and `first` is the initial instance of a ordered type (`Time` in this case since `start` is defined as such) as mentioned above. Therefore, it specifies "At the initial instant of time, all the frogs are positioned accordingly to the initial disposition of the puzzle". Furthermore the fact declares that some frog must move or jump in between each instant of time assuring the liveness of the model.

The last model's declaration is a *run* and it useful to instruct the *Alloy Analyzer* to solve for the given constraints and return valid instances. Given that the puzzle contains 3 green frogs, 3 brown frogs and 7 stones, the bounds are set accordingly. The keyword `exactly` forces the solver to use exactly those quantities of signature atoms. In the case of the Time signature, here is simply specified a maximum number of atoms. If the solver can find a solution with less than 16 distinct atoms of `Time`, it will be valid.

### 2.1.3 *Dynamic Modeling*

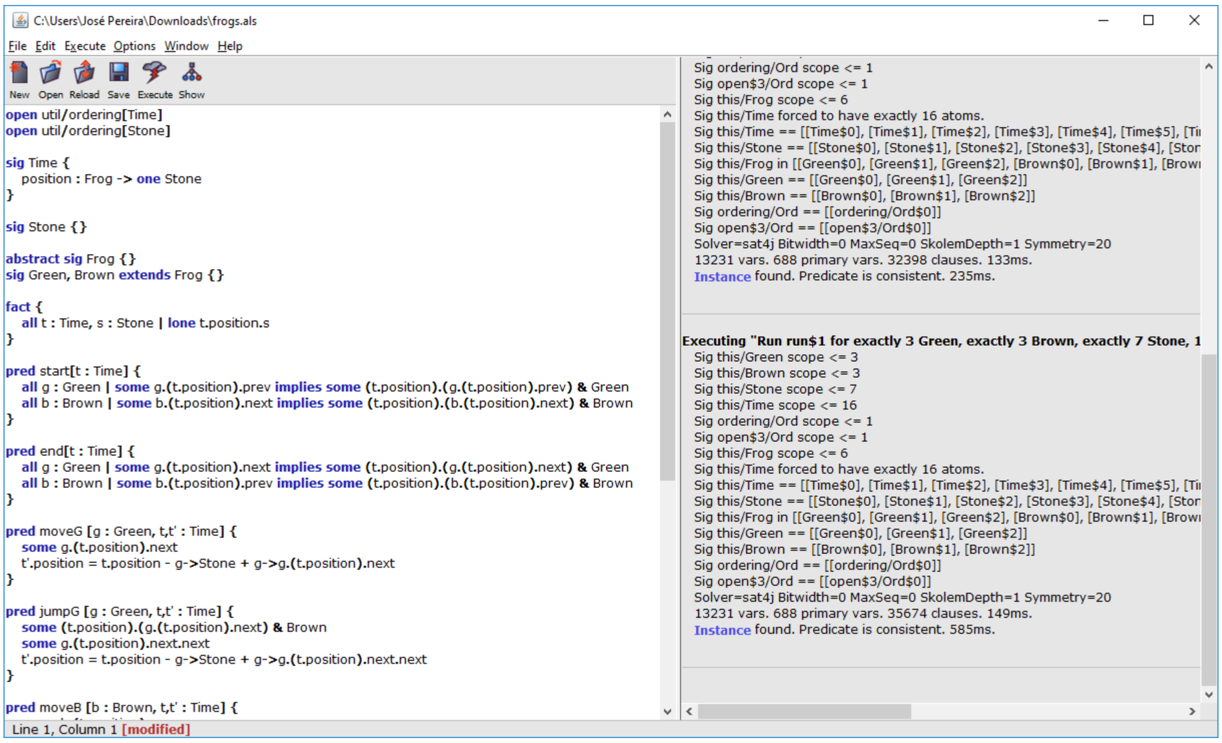
Some insight on dynamic modeling with *Alloy* is required in order to better understand further concepts and user concerns. In short, a structure can either be static or have dynamic behaviours. Considering the presented example, a static structure would be one specific instant of time, meaning the frogs wouldn't move from stone to another. In the *Alloy* notation, every relation is static and, as such, additional ordered signatures are used to explicitly represent states and therefore introduce dynamism. The `frog.als` uses the `Time` class for that purpose.

There are two similar approaches to specify a dynamic model: the *local state idiom* and the *global state idiom*. In the former all mutable fields are defined in the global state signature. In the later, the state signature is added locally as an extra column at the end of each mutable field. The example model is an instance of the *global state idiom*. Naturally dynamic models will have higher order relations which can later compromise the readability of its instance's graph visualisation.

## 2.2 THE ALLOY ANALYZER

The Alloy Analyzer is a tool that supports fully automatic analysis of Alloy models. It translates Alloy formulas to boolean expressions which are then checked by SAT solvers. The user can mainly opt by *Simulation* or *Assertion* checking Bolton (2005). Both require bounds for each declared signature of the specified model and a default scope is assumed if the user does not provide such. When satisfiable constraints are declared, the *Simulation* generates random instances of the model according to the specification. On the other hand, an *Assertion* states a model's requirement and the tool attempts to find contradictory examples.

Figure 2 illustrates the *Alloy Analyzer*'s user interface.


C:\Users\José Pereira\Downloads\frogs.als
File Edit Execute Options Window Help
New Open Reload Save Execute Show

```
open util/ordering[Time]
open util/ordering[Stone]

sig Time {
   position : Frog -> one Stone
}

sig Stone {}

abstract sig Frog {}
sig Green, Brown extends Frog {}

fact {
   all t : Time, s : Stone | lone t.position.s
}

pred start[t : Time] {
   all g : Green | some g.(t.position).prev implies some (t.position).(g.(t.position).prev) & Green
   all b : Brown | some b.(t.position).next implies some (t.position).(b.(t.position).next) & Brown
}

pred end[t : Time] {
   all g : Green | some g.(t.position).next implies some (t.position).(g.(t.position).next) & Green
   all b : Brown | some b.(t.position).prev implies some (t.position).(b.(t.position).prev) & Brown
}

pred moveG [g : Green, t,t' : Time] {
   some g.(t.position).next
   t'.position = t.position - g->Stone + g->g.(t.position).next
}

pred jumpG [g : Green, t,t' : Time] {
   some (t.position).(g.(t.position).next) & Brown
   some g.(t.position).next.next
   t'.position = t.position - g->Stone + g->g.(t.position).next.next
}

pred moveB [b : Brown, t,t' : Time] {
```

```
Line 1, Column 1 [modified]
```

```
  Sig ordering/Ord scope <= 1
  Sig open$3/Ord scope <= 1
  Sig this/Frog scope <= 6
  Sig this/Time forced to have exactly 16 atoms.
  Sig this/Time == [[Time$0], [Time$1], [Time$2], [Time$3], [Time$4], [Time$5], [Ti
  Sig this/Stone == [[Stone$0], [Stone$1], [Stone$2], [Stone$3], [Stone$4], [Ston
  Sig this/Frog in [[Green$0], [Green$1], [Green$2], [Brown$0], [Brown$1], [Brow
  Sig this/Green == [[Green$0], [Green$1], [Green$2]]
  Sig this/Brown == [[Brown$0], [Brown$1], [Brown$2]]
  Sig ordering/Ord == [[ordering/Ord$0]]
  Sig open$3/Ord == [[open$3/Ord$0]]
  Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
  13231 vars. 688 primary vars. 32398 clauses. 133ms.
  Instance found. Predicate is consistent. 235ms.

Executing "Run run$1 for exactly 3 Green, exactly 3 Brown, exactly 7 Stone, 1
  Sig this/Green scope <= 3
  Sig this/Brown scope <= 3
  Sig this/Stone scope <= 7
  Sig this/Time scope <= 16
  Sig ordering/Ord scope <= 1
  Sig open$3/Ord scope <= 1
  Sig this/Frog scope <= 6
  Sig this/Time forced to have exactly 16 atoms.
  Sig this/Time == [[Time$0], [Time$1], [Time$2], [Time$3], [Time$4], [Time$5], [Ti
  Sig this/Stone == [[Stone$0], [Stone$1], [Stone$2], [Stone$3], [Stone$4], [Ston
  Sig this/Frog in [[Green$0], [Green$1], [Green$2], [Brown$0], [Brown$1], [Brow
  Sig this/Green == [[Green$0], [Green$1], [Green$2]]
  Sig this/Brown == [[Brown$0], [Brown$1], [Brown$2]]
  Sig ordering/Ord == [[ordering/Ord$0]]
  Sig open$3/Ord == [[open$3/Ord$0]]
  Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
  13231 vars. 688 primary vars. 35674 clauses. 149ms.
  Instance found. Predicate is consistent. 585ms.
```

Figure 2: Alloy Analyzer UI

The text editor on the left side contains the model. It possesses basic syntax highlight, stressing *Alloy*'s notation most significant keywords. It also highlights syntax errors, colouring red the error's position. *Simulations* and *Assertions* are both written within the model using the `run` and `check` statements, respectively. The user chooses which to perform on the *Execute* menu option bar on top. The tool's output is visible on the right side of the window.

### 2.2.1 *Visualization facility*

The *Analyzer* includes a visualisation facility that displays the instances found. Users can choose to inspect its textual description, a collapsible tree where the atoms and fields are grouped under their types, or a graph visualization of the instance.

The most sophisticated and overall used visualization method is the graph view of the instance. Examples are shown in Figures 3, 4, 5 and 6 corresponding to our previously presented `frog.als` model. Users can edit the diagram's appearance by changing its *Theme*. *Themes* can be saved and reused on other instances, saving users some time. Within the customisable options are: node and edge colours (limited to 7 colours), shapes of nodes (20 different node shapes), contours of nodes and edges (solid, dashed, dotted or bold) and node and edge names. Users can also opt between showing relationships as node attributes or edges and project the visualization over some type.

The most sophisticated feature on simplifying the readability of these dynamic model's graphs is projection. Projection over a type creates a frame (image) per each of its atoms, reducing the arity of related fields and consequently the usage of labeled edges (ternary relations) to represent them. This way users can navigate between frames and better understand the changes occurred.

Figure 3 illustrates a simulation of the `frog.als` for a minimal bound of one frog of each color and 3 stones. It is also limited to 2 `Time` atoms as shown. To represent the `position` field of type `Time`→`Frog`→`Stone`, labelled edges are used and its comprehension is not immediate. With a simple projection over the `Time` type, users can separate both states (`Time` atoms) and easily comprehend the instance, as shown in Figure 4.

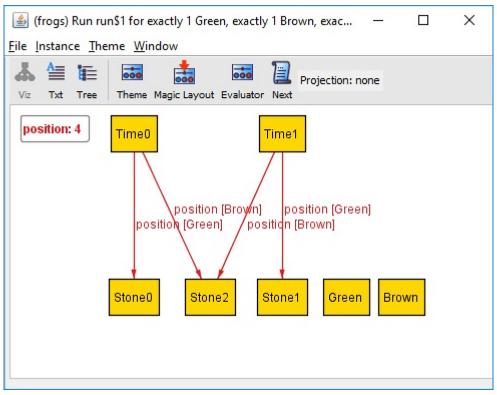

(frogs) Run run$1 for exactly 1 Green, exactly 1 Brown, exac...
File Instance Theme Window
Viz Txt Tree Theme Magic Layout Evaluator Next
Projection: none
position: 4
Time0
Time1
position [Brown]
position [Green]
position [Green]
position [Brown]
Stone0
Stone2
Stone1
Green
Brown

Figure 3: Frog puzzle with 1 of each `Frog`, 3 `Stone` and two `Time` atoms. No projection.

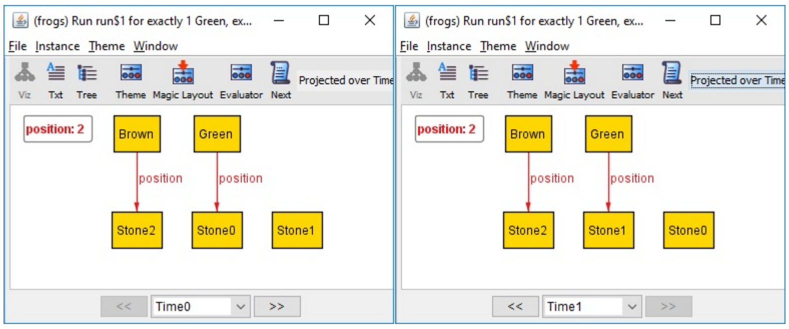

(frogs) Run run$1 for exactly 1 Green, ex...
File Instance Theme Window
Viz Txt Tree Theme Magic Layout Evaluator Next
Projected over Time
position: 2
Brown
Green
position
position
Stone2
Stone0
Stone1
<< Time0 >>
(frogs) Run run$1 for exactly 1 Green, ex...
File Instance Theme Window
Viz Txt Tree Theme Magic Layout Evaluator Next
Projected over Time
position: 2
Brown
Green
position
position
Stone2
Stone1
Stone0
<< Time1 >>

Figure 4: Frog puzzle with 1 of each `Frog`, 3 `Stone` and two `Time` atoms. With projection.

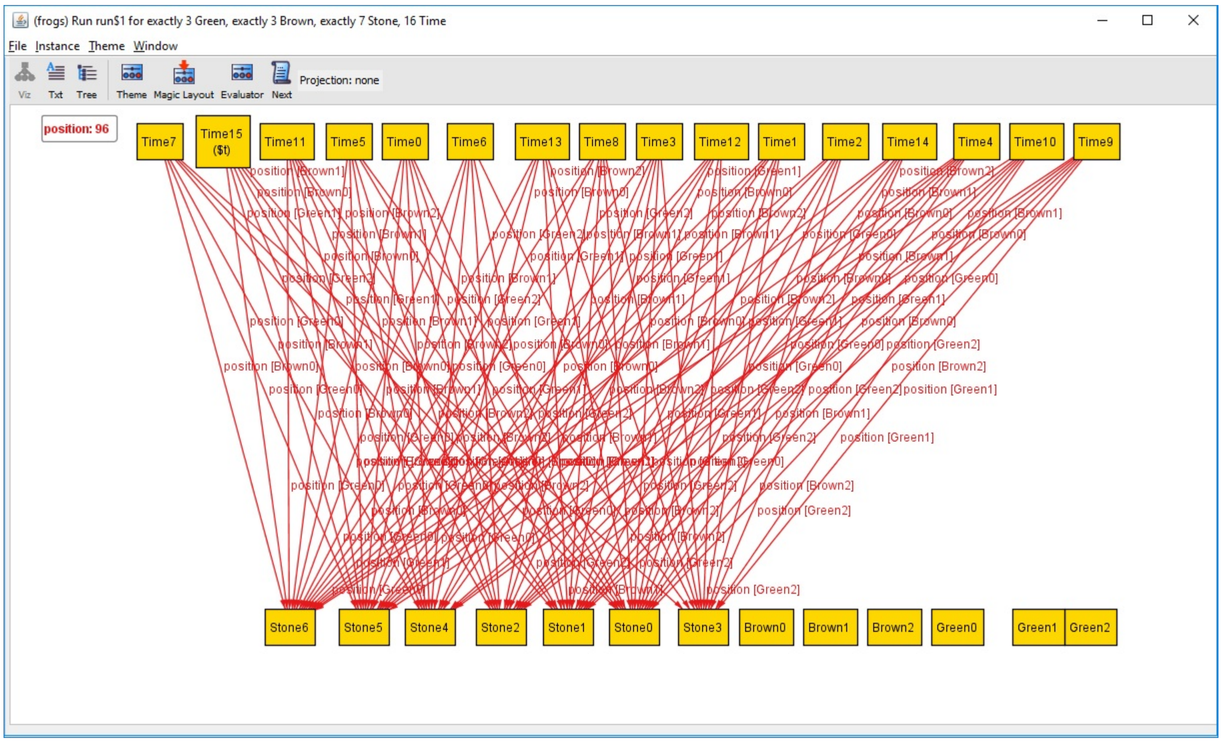

(frogs) Run run$1 for exactly 3 Green, exactly 3 Brown, exactly 7 Stone, 16 Time
File Instance Theme Window
Viz Txt Tree Theme Magic Layout Evaluator Next
Projection: none
position: 96
Time7 Time15 ($t) Time11 Time5 Time0 Time6 Time13 Time8 Time3 Time12 Time1 Time2 Time14 Time4 Time10 Time9
Stone6 Stone5 Stone4 Stone2 Stone1 Stone0 Stone3 Brown0 Brown1 Brown2 Green0 Green1 Green2

Figure 5: Visualization facility UI, frog.als solution

Figure 3 represents the default appearance of instances in the *Visualizer*. Making sense of the displayed information is not easy or fast, in fact it is very far from the kind of informal diagram that a user would draw by hand. The negative effect scales for instances of more complex models which may introduce higher-order relations or a much larger set of atoms. These appear confusing and illegible by default and sometimes drive users away from the *Visualization* feature, relying exclusively on the textual output of the tool. Also, many of them are put off by the seemingly difficult customisation of the *theme*.

Expert users perform systematic changes on their themes to make their graphs more intelligible. They often project over state, distinguish atom types through their colours and shapes and choose to display relationships as labels when too many edges pollute the viewing.

These behaviours were studied and applied in the *Alloy* graph visualization facility, improving it with a *Magic Layout* option that automatically infers visualization settings through a static analysis of the model, generating a theme superior to the default for most models Rayside et al. (2007). The frog puzzle solution, for example, is highly simplified by *Magic Layout* as it is visible in Figure 6.

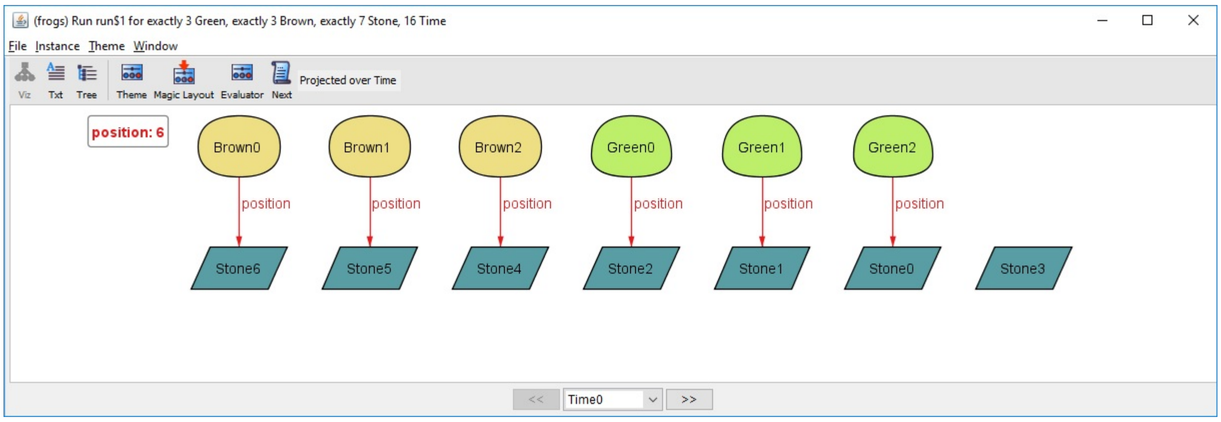

(frogs) Run run$1 for exactly 3 Green, exactly 3 Brown, exactly 7 Stone, 16 Time
File Instance Theme Window
Viz Txt Tree Theme Magic Layout Evaluator Next
Projected over Time
position: 6
Brown0
Brown1
Brown2
Green0
Green1
Green2
position
Stone6
Stone5
Stone4
Stone2
Stone1
Stone0
Stone3
<<
Time0
>>

Figure 6: Theme generated by the Magic Layout for the frog puzzle, projected at time 0.

Still, a few uncovered but pertinent concerns were raised like the inconsistency of node positions between frames of the same projection, the lack of stress of the state changes over time Zaman et al. (2013), and the impossibility of seeing multiple frames of a projection simultaneously in dynamic modelsZave (2015).

Such user's feedback is extremely valuable to this endeavour considering the new *Web* Visualization feature may be built from scratch enabling the best user experience driven implementation.

## 2.3 SUMMARY

This chapter introduced the *Alloy* notation using the frog puzzle dynamic model. *Alloy* is a small language with interesting descriptive capabilities and a rigorous mathematical background. *Alloy*'s graph visualization is an important feature to be implemented on this environment and therefore it is important to consider the current user's feedback on the matter and possibly improve the current implementation in the Alloy Analyzer.

# 3

# RELATED WORK

This chapter explores related tools and platforms in order to better understand the kind of features the *Alloy Web-based Environment* should/must have to be user friendly, entertaining and didactic. Additionally, it gives some insight on the graph visualization techniques for the kind of data outputed by *Alloy*.

## 3.1 SIMILAR PLATFORMS

When searching for terms like competitive programming, browser based *IDE* or interactive coding tutorials, many relevant and sophisticated platforms are found. Among these, a few with some impact in their respective communities were surveyed for their core features which, may be analogously applied in this work.

### 3.1.1 *Pex4Fun*

Pex4Fun[1] is a web-based educational gaming environment created by the Microsoft Research group for teaching and learning programming and software engineering using three different mainstream programming languages: C#, Visual Basic and F#. Visitors can learn a wide range of programming concepts from the basic to more advanced topics by solving puzzles proposed by other users (Tillmann et al., 2013). *Pex4Fun* was released to the public in June of 2010 and since then, the number of user attempts to solve puzzles has reached over 1.7 million as of December 2015.

The Microsoft Research group also launched a community site named *rise4fun* for showcasing software tools anywhere through a web browser in which *Pex4fun* is included, among many other tools. One can publish some tool easily using their *front end framework* simply by providing a *webservice* with the standard required defined *services*. With it, the developer's concerns regarding the *front end* implementation are removed since it is automatically generated and populated according to the *webservice's* output.

1 http://pex4fun.com/

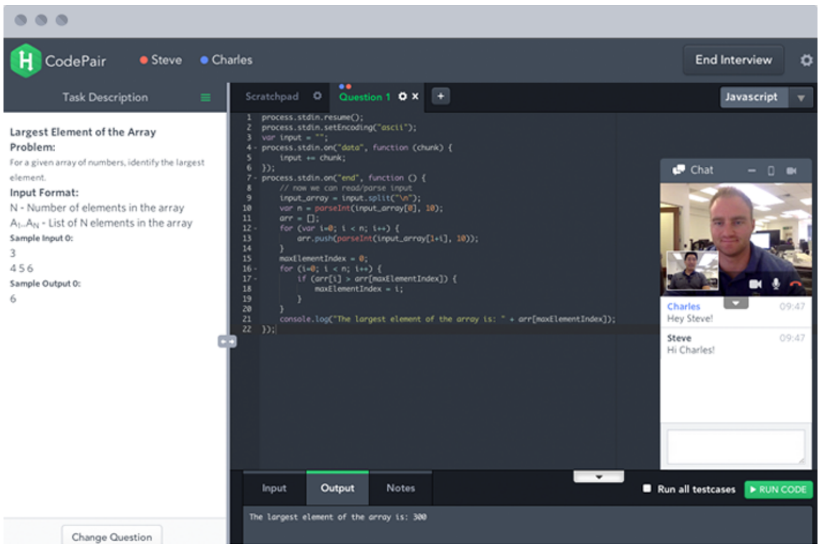
Considering the previous, it would be reasonable to consider such framework to implement the *Alloy* web-based environment. In fact, it supports many useful features such as syntax highlight, *permalinks*, embedded tutorials, etc. The deal breaker is in the tool's output display. It prints out static information provided by the *service's* result. This might be useful to print model interpretation's results textually but it won't support any kind of graph visualization feature essential to *Alloy*. Hence, *rise4fun's framework* is not an option to implement a good *web-based Alloy* environment.

### 3.1.2 *Hackerrank*

*Hackerrank*[2] is a company that focuses on promoting competitive programming challenges for an online community of over a million programmers already. There are challenges over several domains of knowledge in computer science and these can be solved on any browser using one of many available programming languages.

A challenge is composed by a program, a set of input test cases and a detailed description of what is expected of the implementation. Users write their solutions in-browser and the system cross-matches their outputs against the correct program's for each input test case.

The system scores them using the accuracy of their outputs as well as their efficiency through means of execution time. Participants earn badges and improve their position on the global leaderboard as they complete challenges.



**Largest Element of the Array**
**Problem:**
For a given array of numbers, identify the largest element.
**Input Format:**
N - Number of elements in the array
$A_1..A_N$ - List of N elements in the array
**Sample Input 0:**
3
4 5 6
**Sample Output 0:**
6

```
process.stdin.resume();
process.stdin.setEncoding("ascii");
var input = "";
process.stdin.on("data", function (chunk) {
    input += chunk;
});
process.stdin.on("end", function () {
    // now we can read/parse input
    input_array = input.split("\n");
    var n = parseInt(input_array[0], 10);
    arr = [];
    for (var i=0; i < n; i++) {
        arr.push(parseInt(input_array[1+i], 10));
    }
    maxElementIndex = 0;
    for (i=0; i < n; i++) {
        if (arr[i] > arr[maxElementIndex]) {
            maxElementIndex = i;
        }
    }
    console.log("The largest element of the array is: " + arr[maxElementIndex]);
});
```

Charles 09:47
Hey Steve!
Steve 09:47
Hi Charles!

```
The largest element of the array is: 300
```

Figure 7: *HackerRank*'s *CodePair* initiative for job interviews.

*HackerRank* attracts many programmers by offering hiring services to software companies like *facebook* and *yahoo!* among hundreds more. These companies organise *hackatons*

2 https://www.hackerrank.com/

and hire contestants through the platform based on their score. Also, interviews can be conducted online as shown in Figure 7 were recruiters can watch applicants code in real time.

### 3.1.3 *Mooshack*

Mooshack is a *Web-based client-server* application to deploy and manage programming contests (Leal and Silva, 2003). It is available for download at `https://mooshak.dcc.fc.up.pt/` and it requires a Linux OS, Apache server and Tcl to install.

The system provides multiple *views* according to the type of user accessing it, which can be contestants, judges or administrators. The contestant can access a challenge via his browser and submit a solution which is validated trough assertions of outputs between the submitted answer and a particular test case, both having the same input.

### 3.1.4 *Codechef*

*CodeChef*[3] is another competitive programming platform. It was created by *Directi* and it's aimed at education with no profit ends. It supports over 35 programming languages and possesses a large community of programmers whom help others with their skills, both beginners as well as more experienced users.

*CodeChef* hosts several contests, some rewarding the winners with cash prizes in order to attract more programmers. They have also initiatives aimed at the youngest in India to enable them to excel at international programming competitions.

### 3.1.5 *CodingGround*

[4] *TutorialsPoint* offers users the opportunity to code in-browser with almost every popular programming language. The initiative is called *CodingGround* and it contains online terminals and simple *IDEs* for several languages that allow the execution and sharing of code. Also, these environments contain in-bedded terminals which serve as input/output of the programs.

3 `https://www.codechef.com/`
4 `https://www.tutorialspoint.com/codingground.htm`

### 3.1.6 *Relevant Features*

Most of the previous presented platforms, among other in-browser programming websites share a set of characteristic features. The essential functionality these contain is some sort of code submission and further interpretation/compilation.

Furthermore these environments output information, it being the program's direct result or an assertion of it against the expected outcome in the case of competitive programming.

Additional features enhance the user's experience which is essential to capture their interest and therefore grow the community. Table 1 describes some of these functionalities visible at first glance in the previously introduced platforms.

Sophisticated text editors, user profiles and code sharing are features present in most of these established names in competitive programming and/or didactic programming. Hence these would be good investments to have in the *Alloy Web Environment*.

| | **Text Editor** | **Code Sharing** | **Competitive Programming** | **User Profile** | **Tutorials** |
|---|---|---|---|---|---|
| **Pex4Fun** | It contains a basic text editor to code in-browser. No syntax highlighting, code folding or suggestions included. | Users can share their code through permalink URL. | Introduces competition through coding duels where students must solve puzzles according to their teacher's specification. | Users may sign in and keep track of their duel's results and points through their profiles. | Contains tutorials on the subjects involved. |
| **HackerRank** | Advanced editor with syntax highlighting on several programming languages. Code folding, suggestions and inline error display. | There is no direct way of sharing code in this platform. Users may share code snippets on the site's discussion area for each specific problem. | HackerRank uses leaderboards, badges and priodic contests with prizes to stimulate competition. | Users have profiles that track their solved challenges history, earned badges, position on global leaderboard and more. | Offers the basic knowledge to solve some challenges. |
| **Mooshack** | No editor. Users must submit files containning the code. | User may not share their solutions with others trhough the platform. | Simply informs users of their solution's correctness. | Each contestant has an account and their answer submissions are linked to it. | No tutorials offered. |
| **CodeChef** | Possesses syntax highlighting, autocompletion, suggestion, code folding. Errors are not displayed inline. | Does not have permalink mechanism. Users share code snippets on the platform's general forum. | Hosts fortnight challenges with prizes. | Users can track their solved/on going challenges through their accounts. | Some specific problem solving oriented tutorials. |
| **CodingGround** | Offers syntax highlighting and code folding only. | Allows code sharing trough permalink URL. | Not aimed at competitive programming. | No such feature. | Lots of detailed tutorials regarding several languages and tools |

Table 1: Platform's features comparison

## 3.2 GRAPH VISUALIZATION

Graph display in-browser would be a valuable addition to the *Alloy*'s *Web* environment. To achieve a proper functionality it is important to consider graph simplification methods to make sense of examples like the one presented on Figure 3. An essential step is to consider the existing work on improving the *Alloy Analyzer* graph visualization which is limited to Zaman et al. (2013). Furthermore, some general techniques of graph visualization Herman et al. (2000) applied by existing tools will be presented.

### 3.2.1 *Improving Alloy Analyzer's graph visualization*

Zaman et al. (2013) suggest some improvements on the *Magic Layout* option described in the background by tackling the following issues: lack of state change highlighting on dynamic models, unconsidered similarities between node layouts of different but related atoms and consistency of node positioning between frames of a projection. Automatic multiple type projection was also explored to simplify graphs even further.

Colours were suggested to stress state changes on dynamic models as illustrated on Figure 9. Previously (Figure 8), colours were being used simultaneously with shapes to distinguish atom types. The idea is that shapes are distinctive enough and colours may be used to indicate state changes instead. The default *Magic Layout* colouring system would be applied on models without projection.

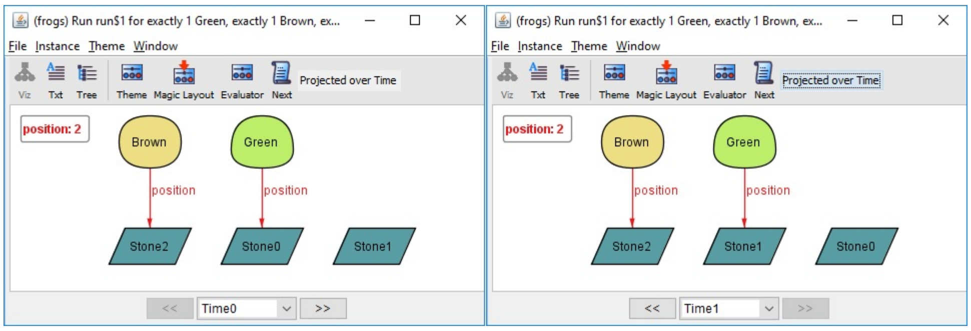

(frogs) Run run$1 for exactly 1 Green, exactly 1 Brown, ex...
File Instance Theme Window
Viz Txt Tree Theme Magic Layout Evaluator Next
Projected over Time
position: 2
Brown
Green
position
position
Stone2
Stone0
Stone1
<< Time0 >>
(frogs) Run run$1 for exactly 1 Green, exactly 1 Brown, ex...
File Instance Theme Window
Viz Txt Tree Theme Magic Layout Evaluator Next
Projected over Time
position: 2
Brown
Green
position
position
Stone2
Stone1
Stone0
<< Time1 >>

Figure 8: Current *Magic Layout* Theme appearance.

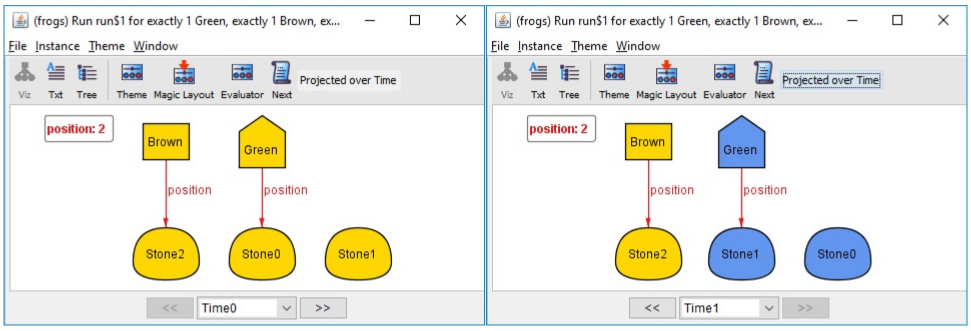

(frogs) Run run$1 for exactly 1 Green, exactly 1 Brown, ex...
File Instance Theme Window
Viz Txt Tree Theme Magic Layout Evaluator Next
Projected over Time
position: 2
Brown
Green
position
position
Stone2
Stone0
Stone1
<< Time0 >>
(frogs) Run run$1 for exactly 1 Green, exactly 1 Brown, ex...
File Instance Theme Window
Viz Txt Tree Theme Magic Layout Evaluator Next
Projected over Time
position: 2
Brown
Green
position
position
Stone2
Stone1
Stone0
<< Time1 >>

Figure 9: Suggested improvement on highlighting state changes.

Zaman et al. (2013) states that the previous *Magic Layout* could be improved by adding similarities between nodes belonging to the same super type. Previously, both top level types and types within the same hierarchy were being attributed shapes randomly for distinction purposes.

The new implementation introduced the term *haircuts* to distinguish nodes within the same hierarchy. In short all hierarchic types are represented by rectangles, which are the most adequate for node labelling and only the top edge shape (spiked, curvy, castle wall shaped, etc) varies from subtype to subtype, hence the *haircuts*. Figure 10 illustrates an example by applying similar *haircuts* to `Green` frogs and `Brown` frogs since they both extend the `Frog` super type.

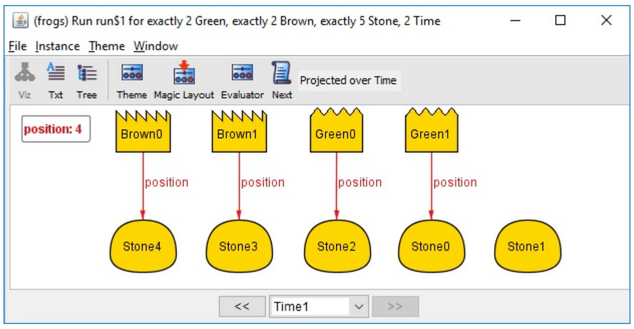

(frogs) Run run$1 for exactly 2 Green, exactly 2 Brown, exactly 5 Stone, 2 Time
File Instance Theme Window
Viz Txt Tree Theme Magic Layout Evaluator Next
Projected over Time
position: 4
Brown0
Brown1
Green0
Green1
position
Stone4
Stone3
Stone2
Stone0
Stone1
<<
Time1
>>

Figure 10: Usage of similar shapes on types belonging to the same hierarchy.

Some users complained about the constant changes of node positions between frames of a projection, causing a difficult tracking of the changes (Zave, 2015). To solve this inconvenience, a sophisticated algorithm is used to automatically calculate the node positioning which most reduces edge crossing. Then, those node positions are replicated in other frames. Still, some issues emerged with node labels as they increased in size causing a nodes area overflow. Some node's labels were covered by other nodes due to the impossibility of repositioning upon enlargement.

Even after projection, some graphs remain quite complex, specially if ternary or superior order relations remain. To further simplify the visualization of these, the new *Magic Layout* suggests automatic projection over multiple types, identifying them through a set of constraints.

Despite the improvement on visualization of 22 out of 24 models that come bundled with *Alloy 4.2* according to an quantitative and qualitative evaluation conducted by Zaman et al. (2013), these features have not been applied to the current *Magic Layout*.

### 3.2.2 *Graph properties*

Graph visualization efficiency is highly affected by the graph's size, therefore a good layout can optimise the usage of the viewing platform. A good practise is to keep the edge crossings to a minimum or if possible render a planar graph, *i.e.* a graph without edge crossings at all. Inevitably this raises some algorithmic issues: is the graph planar? If so, one must define constraints to find a planar layout for it.

Still, a low edge-crossing implementation is wise considering it has a far more positive effect on visualization than any other graph property like symetry or a reduced number of bends in edges according to usability studies (Purchase, 1997). Another regular concern of graph visualization tools is to preserve the "user's mental map" or graph's *predictability* The idea is that two distinct renderings of the same graph or similar graphs must not produce radical visual changes (Delest et al., 1998).

### 3.2.3 *Navigation and Interaction*

Simply adjusting the graph's layout is insufficient in achieving an intelligible representation of the information. Navigation and interaction with the graph is extremely helpful in better understanding it (Herman et al., 2000).

A good graph visualization tool allows users to explore with zoom and pan. These are useful to follow sequences of relations or better observe small clusters of nodes while abstracted from others. Still it is important to keep in mind possible aliasing problems related to excessive zooming.

Zooming can simply be an enlargement of the graph content or in the case of a more sophisticated tool it can display additional content as the user zooms in. These techniques are called *geometric zooming* and *Semantic zooming* (Hewlett and Selfridge-Field, 1998), respectively.

Despite its obvious utility, zoom creates issues of decontextualisation. The *focus+context* methods tackle this issue by offering a continuous display of the general graph context while zooming. One of these methods is *fisheye distortion* and it enlarges the required information while removing detail from the remaining, as illustrated on Figure 11 (Lamping et al., 1995).

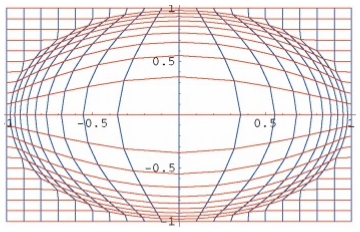


Figure 11: Fisheye distortion of a regular grid of the plane.

## 3.3 SUMMARY

This chapter detailed the features that the *Alloy* Environment should offer to stand out as a in-browser modeling environment. Some websites were surveyed in order to understand the detail and importance of their components to users. It was concluded that features such as syntax highlighting, code sharing and user's profiles were important add ups to the platform.

Additionally, a study on graph visualization tools and possible improvements on the *Alloy* graph visualization facility was conducted. Some interesting features were described by Zaman et al. (2013) which should be applied on the future browser graph visualization feature. It was possible to identify the usefulness of maintaining node positions over several frames of the same projection as a simplification on graph interpretation. The latest section of this chapter also described features like zooming and panning as standard requirements of graph visualization tools.

# 4 TECHNOLOGIES

The development of the *Alloy* platform involved several technologies. This chapter provides insight on them as well as the reasons which led to these specific frameworks, libraries, plug-ins, etc.

## 4.1 HOSTING

The web-based *Alloy* environment is currently online at `http://ec2-52-36-177-8.us-west-2.compute.amazonaws.com`. *Amazon EC2*[1] is the virtual server hoster of this page, hence the temporary uncanny public domain name, which follows the automatic standard naming for their server instances addresses. The *Linux t2.micro* minimal package with 1 memory gigabyte and one virtual CPU was effectively used to host the platform until now. Still, *Amazon* conveniently provides their clients an easy scaling of resources if later necessary.

This hosting service is presented as inexpensive, secure and reliable, among other qualities and the usage experience was indeed very pleasing. The whole instance manipulation is accessible and well supported by manuals and an huge community's feedback. Also, despite allowing users complete control over their instances, *Amazon* constantly enforces security measures which are extremely important for less experiences developers. *SSH* (Secure Shell) connections using `.pem` key pairs and security groups acting as virtual firewalls are examples of these demanded safety measures.

## 4.2 FROM ALLOY4.2.JAR TO A WEBSERVICE

*Alloy* is currently on version 4.2 and it is distributed at `http://alloy.mit.edu/alloy/` as a platform independent *.jar*. Its distributors also provide well documented examples of its application programming interface (*API*). One of these examples was particularly useful to build the model's compilation piece.

1 `https://aws.amazon.com/?nc1=h_ls`

Conditioned by *Java*, *Tomcat*[2] *Java servlet container* was chosen to host the *webservice* responsible for receiving models and returning instances of its executions. There were many options on running *JAX-WS* (*Java API* for *XML* Web Services) like *Glassfish* or *JBoss*. Still, *Tomcat* is described as a good approach to host small/medium applications which do not require the full set of features (*e.g.* high security, business logic oriented solutions) offered by other *Java Enterprise Edition* application servers.

A webservice's description language (*WSDL*) is an *XML* format language that describes communication endpoints abstractly allowing a platform independent data exchange.

### 4.2.1 *Eclipse*

The *Eclipse Juno*[3] development environment complemented by the Axis2 plug-in was used to aid in the making of the *webservice* as well as its deployment. Additionally to the advantages of constructing java code with this environment, it provides *JAX-WS* related features like the creation of *Dynamic Web Projects* which are specifically meant for such implementations. As mentioned above, the service contains a *WSDL*, in fact, it requires one to function. Eclipse automatically generates these files from the logics implemented with *Java*. The opposite is also possible, i.e., specifying the *WSDL* first and then generating the *Java* classes in which these webservice's logics need to be implemented.

Once these two components are created/generated, they can be exported through the environment into a `.war` single file. This file can then be automatically deployed to the intended *Tomcat* server by simply copying it to the right directory.

### 4.2.2 *WinSCP*

WinSCP[4] is a not only but mainly an open source, free, safe file transfer protocol (*SFTP*) client for `Windows`. It was helpful to transfer files like the `.war` previously detailed to the *Amazon EC2* instance using the provided internet protocol address and `.pem` certificate by *Amazon*. The whole connection process is nicely described in detail by both *Amazon*'s and *WinSCP*'s documentations.

## 4.3 METEOR

The *Alloy* Platform is essentially a web development effort, meaning the inevitable presence of frequently related concepts, tools and languages like *javascript*, *css*, *HTML*, relational or

2 http://tomcat.apache.org/
3 https://eclipse.org/juno/
4 https://winscp.net/eng/download.php

non relational databases, etc. This is often called the full-stack development, being the stack the whole *web* project, back-end and front-end.

Mastering such a variety of tools can be a challenge, hence the growing number of *web* development frameworks which speed up the production by generally packaging technologies in a single, more abstract one. *Javascript* frameworks in particular have become very popular of late to build *web* applications. This notoriety made us believe that using one of these frameworks would be beneficial to implement the *web* base *Alloy* environment.

*Meteor*[5] is a *open-source*, free, full-stack web development framework built using *Node.js*. With it, both server and client sides are developed using *Javascript* and they share the same APIs. In fact code runs both on client and server unless explicitly stated that it should only run in one of them.

Under the hood, Meteor contains *MongoDB*[6], *jQuery*[7] and provides a long list of nicely packaged libraries for an easy and fast integration. *MongoDB* is a *NoSQL* database, open-source application written mostly in *C++* and it's *JSON* (*JavaScript* Object Notation) oriented. *Jquery* is a *cross-browser JavaScript* library made to simplify client side interaction with *HTML* Perhaps the most appealing resource of Meteor is its Distributed Data Protocol (DDP) used to automatically synchronise data between the database and what's being displayed to the client. This feature allows the user interface to seamlessly reflect the true state of the world with minimal development effort.

### 4.3.1 *Why MeteorJS?*

As previously mentioned, there are several popular *Javascript* frameworks that one might use to build a functional web app. Meteor was chosen mostly for three reasons: it is well documented, it possesses a shallow learning curve and it is full-stack, meaning the same technology can be used to develop the front-end and back-end, unlike for example *AngularJs* which is front-end exclusively.

### 4.3.2 *Publish and Subscribe*

Meteor does not provide a direct connection between the client and the *MongoDB* instance, nor it should for obvious safety reasons. Instead, it uses the *Minimongo Javascript* library as an all in-memory client side cache. Still, much of the information contained in the databased is sensitive or redundant to the client and a filtration system is in order. *Meteor* solves this issue using a publish and subscription system.

5 https://www.meteor.com/
6 https://www.mongodb.com/
7 https://jquery.com/

In Meteor a publication is a named API on the server that constructs a set of data to send to a client. A client initiates a subscription which connects to a publication, and receives that data. That data consists of a first batch sent when the subscription is initialized and then incremental updates as the published data changes thanks to the *DDP*. This system ensures security and synchronisation using very simple concepts to grasp.

### 4.3.3 *Methods*

Often, web applications require some sort of mutation on their data. For example, the client may perform an *AJAX* request to run some data insertion routine in a secure environment (server side). Meteor refers to these events as Methods and additionally to the standard process described in the previous example they also simulate that routine client-side and reflect the results immediately. The advantage is that clients get a fast preview of the routine's result by avoiding the round trip delay. What if the result from the server comes back and is inconsistent with the simulation on the client? Then the user's interface is patched to reflect the actual state of the server. Meteor names this feature of *Optimistic UI*.

### 4.3.4 *Templating*

Another great feature offered by *Meteor* is templating. Like many other web templating systems it allows a faster and easier construction and content manipulation of either dynamic or static *HTML* pages by reusing its elements as templates. *Meteor* parses *HTML* files and identifies three top-level tags: *body*, *head* and *template*. Everything inside the *head* and *body* tags is added to the client's final *HTML head* and *body* elements respectively. *Template* tags content though is compiled into a *Meteor* template and can be used in other *HTML* files using *{{> templateName}}* or referenced in the *JavaScript* with *Template.templateName*.

Also, typically when building an HTML page one should add external dependencies like scripts or styling to the *HTML* document but *Meteor* handles this by itself linking everything and allowing clean and simple *HTML* files.

## 4.4 CYTOSCAPE

The report's related work chapter stressed the value of graphic instance visualisation. Hence, options were studied to better understand the state of the art in browser based graph visualisation.

Starting from scratch using technologies like *HTML* canvas drawing or Scalable Vector Graphics (*SVG*) would be extremely time consuming given the kind of functionality required. To effectively pursue a solution, existing *Javascript* libraries were explored, dedicated to the visualisation and manipulation of graphs. There are plenty of solutions on the matter, some dedicated to displaying large clusters of relational data, some for simpler concept representations. Some focused on graph creation and editing, others on visualisation and exploration of these.

The requirements implied by the current graphic instance representation in the *Alloy Analyzer* and the possible improvements researched in the previous chapter were considered. One concludes that ideally the technology should aim at the representation of relatively small amounts of data because of unavoidable confusion associated with huge graphs. As previously mentioned, users avoid this kind of instance representation in such cases. Also, the solution should also ideally offer graph navigation features as well as enable several kinds of aesthetic changes to make its interpretation easier.

The search of a library cross-matched with these requirements narrowed the findings to *Cytoscape*[8], a fully featured graph library written in pure *Javascript* (Franz et al., 2015). It offers selection and dragging of one or multiple nodes in the network, pan and zoom features and its *API* allows an easy manipulation and store/load of node positions which is helpful for context preservation and graph rearrangement according to the user's preference.

## 4.5 SUMMARY

This section briefly described the technologies used to build the *Alloy web* environment and why they were used. The most significant ones are *Meteor*, *Cytoscape* and Tomcat as most of the work was conducted in their respective contexts. The applied programming languages were essentially *Java* and *JavaScript* as it will be detailed in the next chapter.

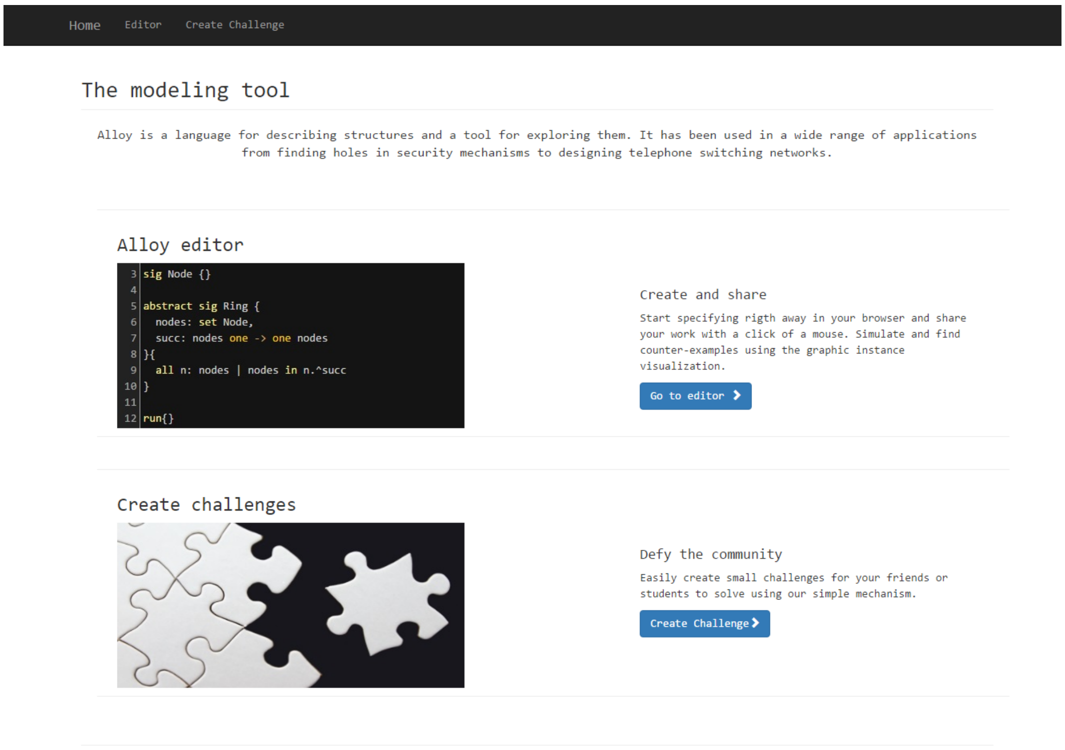
8 http://js.cytoscape.org/

# 5 DEVELOPMENT

The platform's development was focused on several key features. The priority was achieving an editor capable of basic model writing and interpretation since they're the pillars of the concept being developed. Another main concern was the model's visualisation component which would also be present in multiple functionalities throughout the project. Following, challenge creation, completion and sharing mechanisms were implemented and also some improvements on the visualisation feature.

## 5.1 WEBSITE DESIGN

Currently the website can be divided in four different views: Homepage, editor, challenge creation and solving. Figure 12 is a screenshot of the website's homepage and, as displayed, it is possible to navigate through the upper menu or the links bellow to the editor and challenge creation pages. Note that there is no public list of challenges, therefore contestants must possess a specific URL to access a solvable challenge. The composing elements of the next figures will be explained further down this document. This section's objective is to give the reader a notion of the current *website*'s design.



The modeling tool

Alloy is a language for describing structures and a tool for exploring them. It has been used in a wide range of applications from finding holes in security mechanisms to designing telephone switching networks.

Alloy editor

```
sig Node {}

abstract sig Ring {
  nodes: set Node,
  succ: nodes one -> one nodes
}{
  all n: nodes | nodes in n.^succ
}

run{}
```

Create and share

Start specifying rigth away in your browser and share your work with a click of a mouse. Simulate and find counter-examples using the graphic instance visualization.

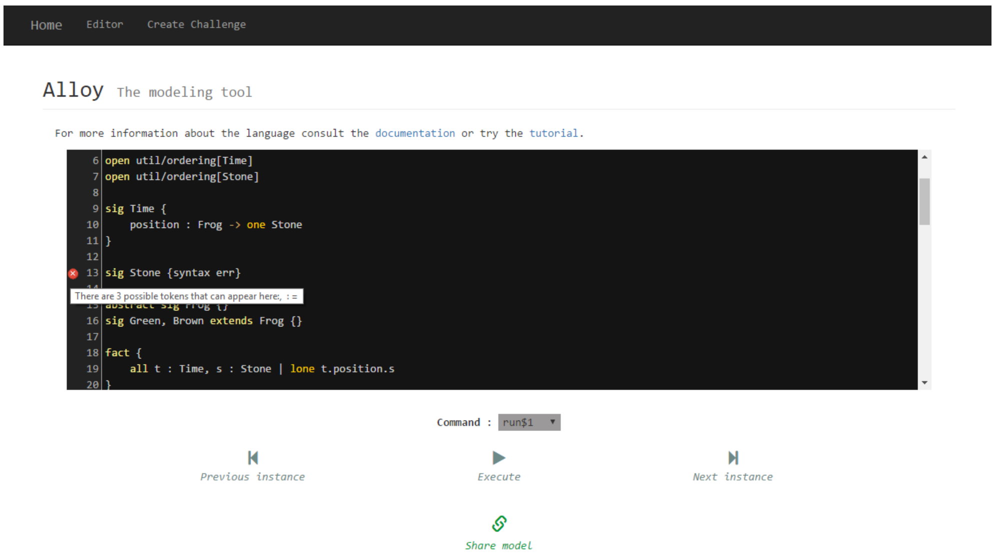
Create challenges

Defy the community

Easily create small challenges for your friends or students to solve using our simple mechanism.

Figure 12: The website's homepage

Figures 13 shows the text editor's where users can freely specify any system and share it using a simple and intuitive interface. The design, although important to the application's attractiveness, was not a main concern while building the platform, hence its improvement was delayed to future work.



Alloy The modeling tool

For more information about the language consult the documentation or try the tutorial.

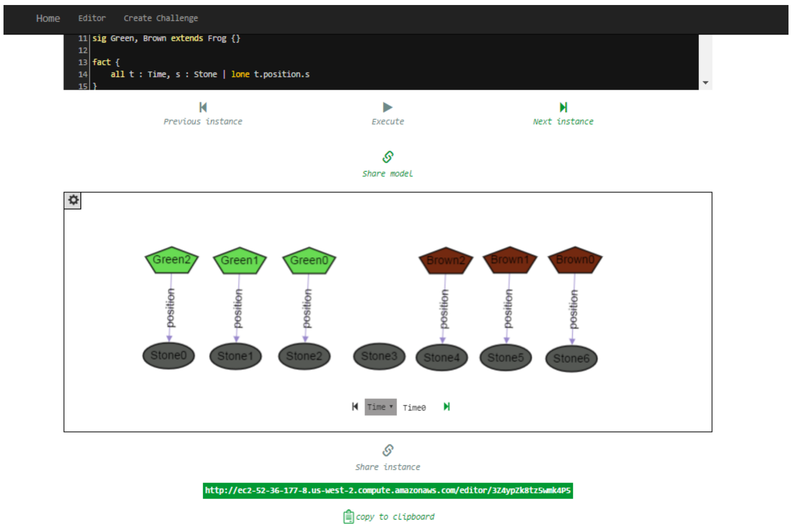
```
open util/ordering[Time]
open util/ordering[Stone]

sig Time {
    position : Frog -> one Stone
}

sig Stone {syntax err}

abstract sig Frog {}
sig Green, Brown extends Frog {}

fact {
    all t : Time, s : Stone | lone t.position.s
}
```

There are 3 possible tokens that can appear here:, : =

Command : run$1

Figure 13: The text editor

Figure 14 illustrates the platform's visualiser that pops up after some command is successfully executed, i.e., without any errors. The graph is rendered in an *HTML* canvas placed under the text area where the model is specified.

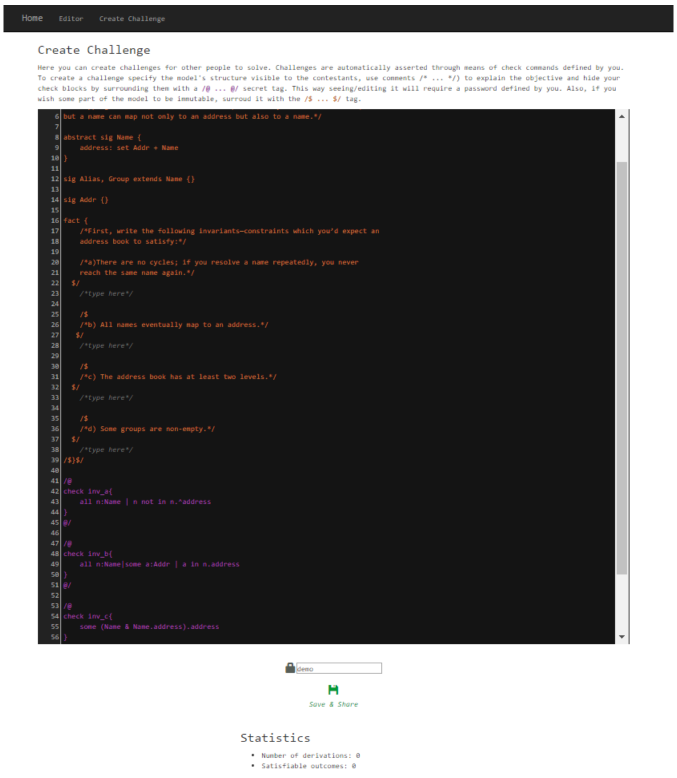


```
sig Green, Brown extends Frog {}

fact {
    all t : Time, s : Stone | lone t.position.s
}
```

http://ec2-52-36-177-8.us-west-2.compute.amazonaws.com/editor/3Z4ypZk8tz5wmk4P5

Figure 14: The visualizer

Finally, the last views are referent to the challenge creation and solving, respectively illustrated on Figures 15 and 16. The latter is also complemented with a visualizer (figure 14) so that contestants may observe counter-examples and better understand the challenge requirements.



Create Challenge

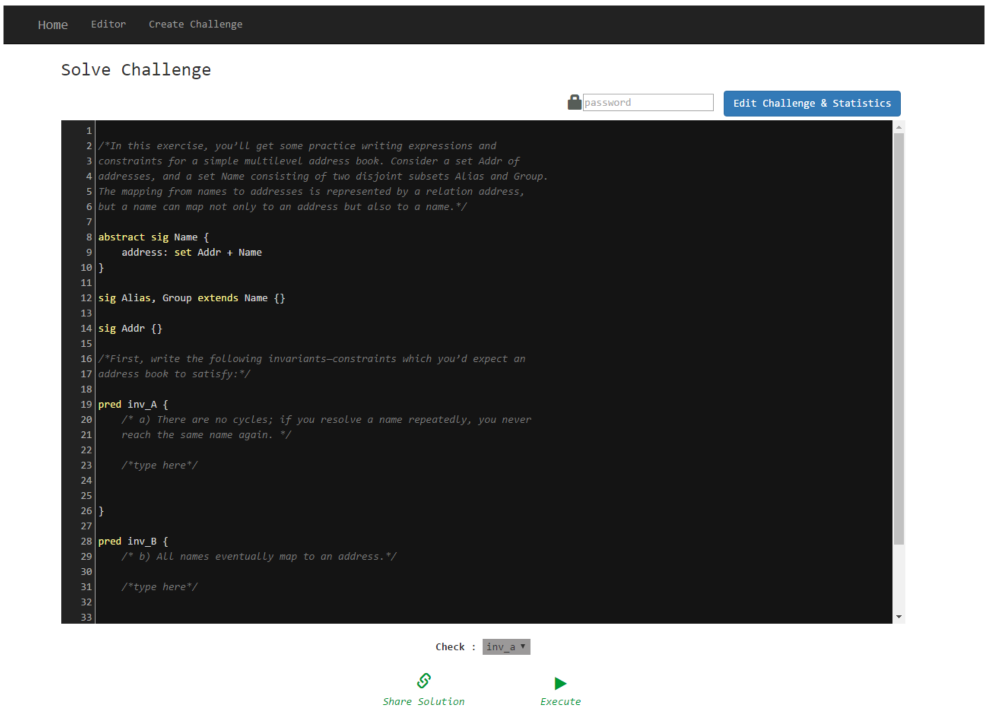
Here you can create challenges for other people to solve. Challenges are automatically asserted through means of check commands defined by you. To create a challenge specify the model's structure visible to the contestants, use comments /* ... */) to explain the objective and hide your check blocks by surrounding them with a /@ ... @/ secret tag. This way seeing/editing it will require a password defined by you. Also, if you wish some part of the model to be immutable, surroud it with the /$ ... $/ tag.

```
but a name can map not only to an address but also to a name.*/

abstract sig Name {
    address: set Addr + Name
}

sig Alias, Group extends Name {}

sig Addr {}

fact {
    /*First, write the following invariants—constraints which you’d expect an
    address book to satisfy:*/

    /*a)There are no cycles; if you resolve a name repeatedly, you never
    reach the same name again.*/
  $/
    /*type here*/

    /$
    /*b) All names eventually map to an address.*/
   $/
    /*type here*/

    /$
    /*c) The address book has at least two levels.*/
  $/
    /*type here*/

    /$
    /*d) Some groups are non-empty.*/
  $/
    /*type here*/
/$}$/

/@
check inv_a{
    all n:Name | n not in n.^address
}
@/

/@
check inv_b{
    all n:Name|some a:Addr | a in n.address
}
@/

/@
check inv_c{
    some (Name & Name.address).address
}
```

Statistics

- Number of derivations: 0
- Satisfiable outcomes: 0

Figure 15: Challenge creation page



Solve Challenge

```

/*In this exercise, you’ll get some practice writing expressions and
constraints for a simple multilevel address book. Consider a set Addr of
addresses, and a set Name consisting of two disjoint subsets Alias and Group.
The mapping from names to addresses is represented by a relation address,
but a name can map not only to an address but also to a name.*/

abstract sig Name {
    address: set Addr + Name
}

sig Alias, Group extends Name {}

sig Addr {}

/*First, write the following invariants—constraints which you’d expect an
address book to satisfy:*/

pred inv_A {
    /* a) There are no cycles; if you resolve a name repeatedly, you never
    reach the same name again. */

    /*type here*/


}

pred inv_B {
    /* b) All names eventually map to an address.*/

    /*type here*/

```

Figure 16: Challenge solving page

Some components are reused or slightly modified from view to view to perform their roles such as the text editor or the visualizer which are present in more than one of the application's pages. These individual components will be described in greater detail ahead in this chapter.

## 5.2 EDITOR ARCHITECTURE

The most basic action the platform must perform is the interpretation of some model. It should be able to inform if there are any possible instances by running some simulation or some counter-examples from an assertion check. The feedback for syntax errors is essential as well.

Figure 17 illustrates the architecture behind the editor feature. Users are presented with a simple interface where they may input the model's specification that they would otherwise write on the *Alloy Analyzer*. Then, an HTML *combo box* is dynamically filled with all the commands defined so that the user may choose which to execute.

When the execute button is pressed, the *Meteor Method*
`getInstance(args...)`
is invoked and its arguments are:

- The inputed specification as a string.
- The *ID* of the client's *HTTP* session which works as an identifier on the webservice's cached information for better performance.
- The instance number. The instance visualisation mechanism, similarly to the *Alloy Analyzer*, allows users to navigate between found instances. These instances are generated deterministically, *i.e.*, the same command executed twice over the same model produces the exact same instances with the same order when iterated. Hence, the instance number represents which of these instances needs to be returned.
- The name of the command to be executed.
- A flag that causes an override over the previously cached information in case the specification changed.

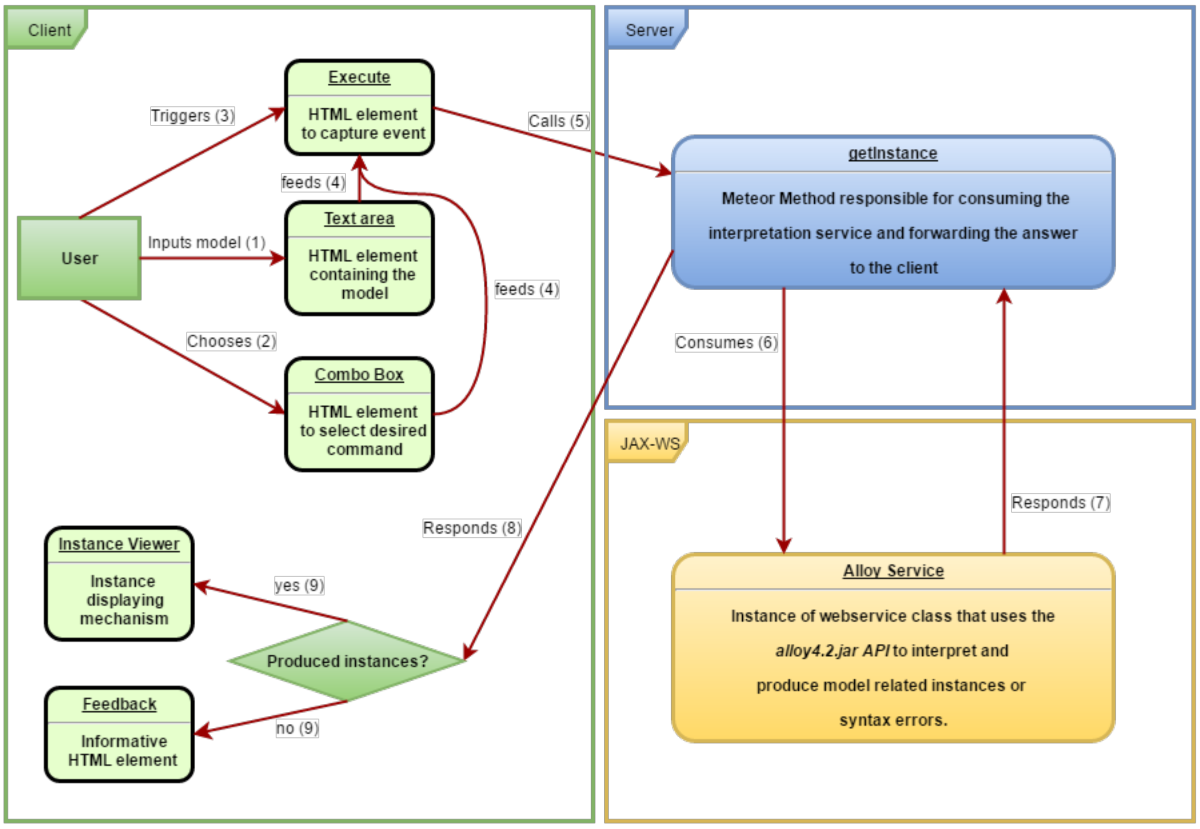

Client
Server
JAX-WS
Execute
HTML element to capture event
Triggers (3)
Calls (5)
feeds (4)
Text area
HTML element containing the model
User
Inputs model (1)
feeds (4)
Chooses (2)
Combo Box
HTML element to select desired command
getInstance
Meteor Method responsible for consuming the interpretation service and forwarding the answer to the client
Consumes (6)
Responds (7)
Alloy Service
Instance of webservice class that uses the *alloy4.2.jar API* to interpret and produce model related instances or syntax errors.
Responds (8)
Instance Viewer
Instance displaying mechanism
yes (9)
Produced instances?
no (9)
Feedback
Informative HTML element

Figure 17: Editor architecture

### 5.2.1 *Text editor's features*

To improve the usability of the model's specification text area, syntax highlighting, syntax error markings and aesthetic changes were introduced.

A lot of helpful Meteor packages of web development libraries are available under `https://atmospherejs.com/`. *Codemirror* is a versatile in-browser text editor implemented using *JavaScript* and an example of these packages. Syntax highlighting was achieved by simply writing a few regular expressions identifying the *Alloy* language keywords and symbols and assigning each a *css* class with a specific colour. With this, the editor dynamically colours the inputed data according to the language allowing a better reading and detection of typos.

Additionally the library simplifies the usage of gutters which are vertical bars along side the text area to point out the occurrence of syntax errors using characteristic symbols or identifying line numbers.

## 5.3 WEBSERVICE

The workflow of the *webservice* performing the model's interpretation is illustrated on Figure 18. Meteor uses *SOAP*, made available through another package, to communicate with the service using its *WSDL URL*.

```
@WebService
public class AlloyService {

  @Resource
  //The answers resource is shared between instances of the webservice.
  //answers represents cached solutions.
  //The String is the HTTP session id and the ClientSession is the solution.
  private static HashMap<String, ClientSession> answers;

  public AlloyService(){
    if (answers == null)
      answers = new HashMap<>();
  }

  @WebMethod
  public String getInstance(String model, final String sessionId, int
      instanceNumber, String commandLabel, boolean forceInterpretation) {

  if (answers.containsKey(sessionId) && !forceInterpretation) {
  //Get instance with number instanceNumber from the cached solution
      answers.get(sessionId).setIteration(instanceNumber);
      return answers.get(sessionId).getInstance();
    } else {
    //Parse, Type Check, Solve and get the indicated instance.
      ...
  return instance;
  }
}
```

Listing 5.1: Defining the webservice

The code snippet presented in Listing 5.1 illustrates the definition of the *JAX-WS*. Each service call of `getInstance` identified by the `@WebMethod` tag will launch an instance of the `AlloyService` class and return some result. The `@Resource` static `answers` variable is used as a caching system associating the client's session identifier with an interpretation of his model. This accelerates the process of requesting a new instance of the same model definition by avoiding re-interpretation.

The *Alloy API* referent to the interpretation methods is demonstrated on `http://alloy.mit.edu/alloy/code/ExampleUsingTheCompiler.java.html` and something very similar is used on the `AlloyService` class. The major time consumer on this algorithm's execution relates to the parsing, type checking and solution finding rather than iterating through the solution until the desired instance. Therefore, a great improvement was to cache the solution so that later it may be re-iterated for a following instance. This is implemented as follows: the webservice's routine checks for the force interpretation flag, if it's not `true`, it then searches for solutions associated with the given *HTTP* session identification. If some solution is found, it is simply iterated to the intended instance number and a response is formed. Otherwise, the standard procedure is performed, *i.e.*, the model is parsed, type checked and if possible solved and additionally to responding with some instance, the solution is cached so that it may be reused later on. If the specification suffers changes, the solution is recalculated using the force interpretation flag as explained above on the `getInstance()` arguments.

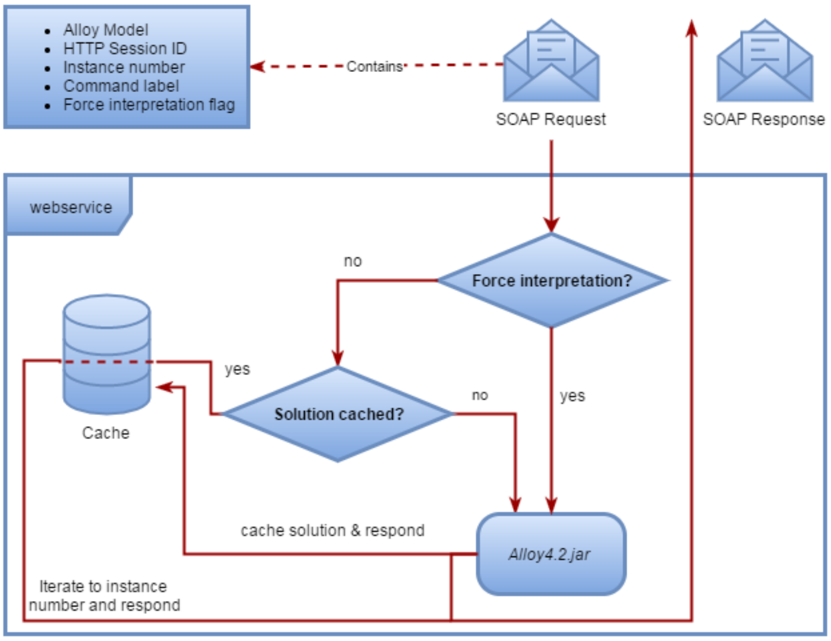


Figure 18: Webservice architecture

## 5.4 CHALLENGES

Challenges add great value to the website. Not only they increase the interactivity with the community but also enable a set of interesting use case scenarios like the dissemination of academic exercises through students by their teachers.

```
/$
/*In this exercise, you will get some practice writing expressions and
constraints for a simple multilevel address book. Consider a set Addr of
addresses, and a set Name consisting of two disjoint subsets Alias and Group.
The mapping from names to addresses is represented by a relation address,
but a name can map not only to an address but also to a name.*/

abstract sig Name {
  address: set Addr + Name
}

sig Alias, Group extends Name {}

sig Addr {}

/*First, write the following invariants-constraints which you would expect an
address book to satisfy:*/

pred inv_A {
  /* a) There are no cycles; if you resolve a name repeatedly, you never
  reach the same name again. */
  $/
    /*type here*/
/$
}

pred inv_B {
  /* b) All names eventually map to an address.*/
   $/
    /*type here*/
    /$
}$/
/@
pred inv_a{
    all n:Name | n not in n.^address
}
check inv_a {
  inv_A iff inv_a
}
pred inv_b{
    all n:Name|some a:Addr | a in n.^address
}
check inv_b {
  inv_B iff inv_b
}
@/
```

Listing 5.2: Defining a challenge

The code in Listing 5.2 demonstrates how a simple challenge can be created through the platform. The challenge creation page on the application suggests the usage of comment blocks to provide context about the model and the challenge's objectives. Through these example's comments, one can understand that it specifies some address book and it asks contestants to define two simple invariants: `inv_a` and `inv_b`.

Furthermore, two distinct block tags can be used, the /@ .. @/ and the /$ .. $/ which do not belong to the *Alloy* notation. The first serves to identify secret blocks of code, such as the example's check commands. These will serve to automatically assert the result and so will be hidden by a password defined by the challenger. The second tag can be used to state that some code is immutable. The contestants won't be allowed to edit the example's model signatures for instance.

Once the challenger is done creating, he can then share it through an *URL*, that once opened will originate something similar to Listing 5.3. Bear in mind that although the signatures aren't highlighted in any way in this documented, they cannot be edited according to the creator's placement of the /$ ... $/ markings. The application though, distinguishes locked blocks with a special highlight on such code.

```
  /*In this exercise, you willll get some practice writing expressions and
constraints for a simple multilevel address book. Consider a set Addr of
addresses, and a set Name consisting of two disjoint subsets Alias and Group.
The mapping from names to addresses is represented by a relation address,
but a name can map not only to an address but also to a name.*/

abstract sig Name {
  address: set Addr + Name
}

sig Alias, Group extends Name {}

sig Addr {}

/*First, write the following invariants-constraints which you would expect an
address book to satisfy:*/

pred inv_A {
  /* a) There are no cycles; if you resolve a name repeatedly, you never
  reach the same name again. */

    /*type here*/


}

pred inv_B {
  /* b) All names eventually map to an address.*/

    /*type here*/


}
```

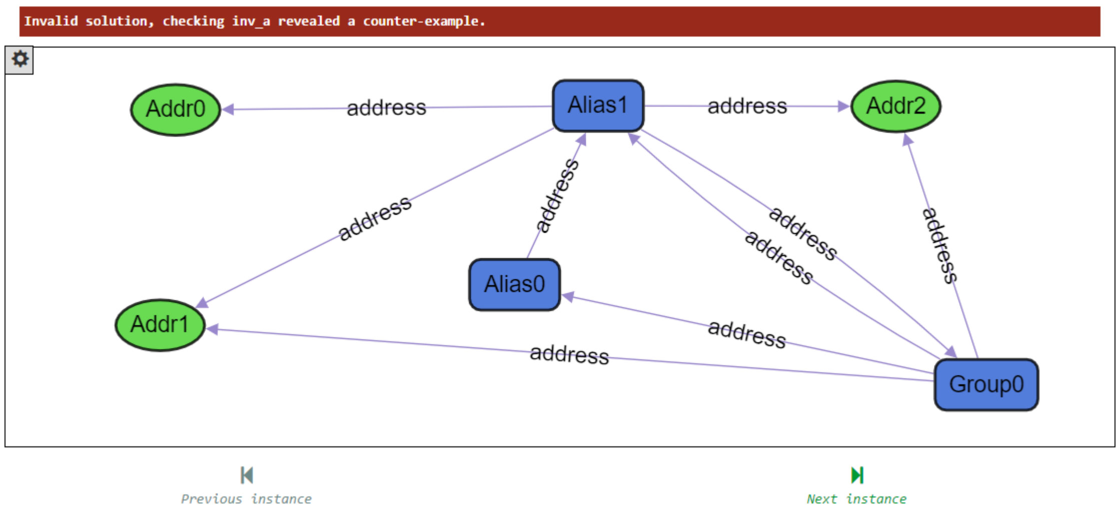
Listing 5.3: Solving a challenge

Solving this particular challenge consists in adding restraining facts for each of the invariants. Note that the stated rules within the predicates or facts may or may not be the same as the ones contained in the check commands. Actually something equal or equivalent is expected of the contestants, hence hiding these check commands from them. Say for instance that some contestant attempts to solve `inv_A` of the presented challenge on Listing 5.3 with a weak constraint as illustrated of Listing 5.4.

```
pred inv_A {
  /* a) There are no cycles; if you resolve a name repeatedly, you never
  reach the same name again. */
    all n:Name | n not in n.address

}
```

Listing 5.4: Wrong solution

The comment blocks explain that in order for `inv_A` to be satisfied, it must stated that there are no cycles when resolving a name's `address` repeatedly. This means that both `Alias` and `Group` atoms cannot be related to themselves trough multiple chaining of `address` relations. Analysing the contestant's specification, one can observe that he only states that names cannot be related with themselves trough `address` relation. Naturally, the *Alloy Analyzer* will find counter examples and display them so that this contestant may understand how to reinforce his constraints. One such example is illustrated on Figure 19.



Invalid solution, checking inv_a revealed a counter-example.

Figure 19: Counter example `inv_A`

Another handy, helpful and implementation free feature is using `run` commands to simulate instances under certain constraints. Since the challenge was created using named predicates, these can be used within the solution context to generate scenarios where the constraints are enforced. If our contestant were to add and execute `run{inv_a}` on his specification, he would then observe instances where there would be no cycles as specified by the challenger. Understanding the challenge requirements would come easy once comparing these simulations with the obtained counter-examples. One possible solution for the challenge is shown on Listing 5.5.

```
pred inv_A {
  /* a) There are no cycles; if you resolve a name repeatedly, you never
  reach the same name again. */
    no n:Name | n in n.^address
}

pred inv_B {
  /* b) All names eventually map to an address.*/
  all n:Name|some a:Addr | a in n.^address
}
```

Listing 5.5: A possible solution

This check mechanism provided by the Alloy Analyzer fits extremely well with the requirements of a challenge, automatically asserting solutions, even if they vary in syntax. Hence, reducing the effort related to implementing some other different solution verifying feature. Plus, this liberty offered to contestants, allows a variety of different approaches and coding styles, which may be interesting to analyse later on.

The system possesses a combo box for users to select which of the commands to run. Internally, an execution takes the contestant specification, adds the selected command and executes it. If no counter examples are found, it means the restrictions defined by the user are equivalent to the creator's and therefore a subheading is solved. Otherwise, i.e. if counter examples are found, they are presented for inspection to help understand what might be wrong.

## 5.5 STATISTICS

One of this platform's requirements is gathering useful data to produce statistics. The majority of data harvesting is made through challenges. Challenge creators can check their exercises regularly and understand how many attempts were made to solve their problem, how many were successful and not.

Although contestants may share their solutions, the platform doesn't distinguish them from the challenges themselves. Internally, a direct solution is a replica of the original challenge with altered content and a reference to it. This system allows building a derivations tree of an original challenge, i.e., one that does not derive from another.

This way, every time a contestant edits his solution and attempts to execute it, the whole specification is stored. Not only this feeds the information given to the challenge creator but also builds a good dataset to extract knowledge from. For instance, in the future one could try to understand how the majority of contestants progress to a solution since the solutions are chain linked through their derivations back to the original challenge as illustrated on Figure 20.

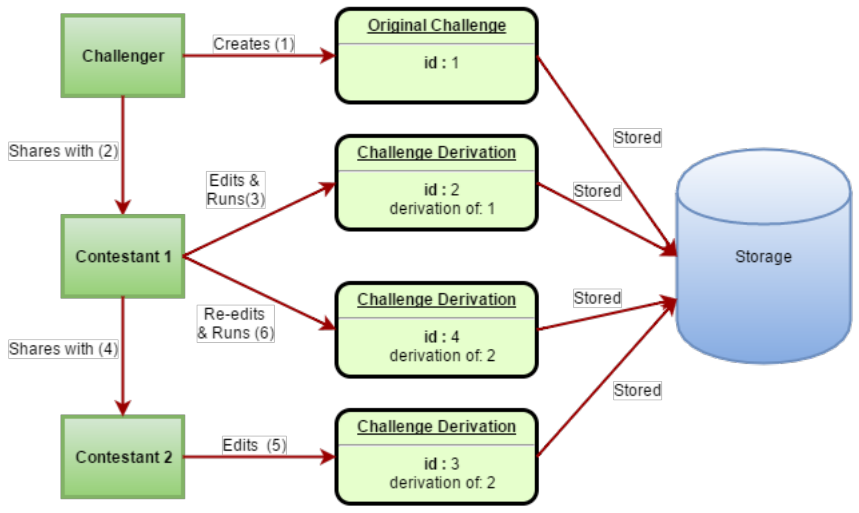


Figure 20: Challenges usage scenario

## 5.6 DATABASE

*Meteor* applications use *MongoDB* instances for backend support. Data stored in these instances is originally shapeless, *i.e.* without a strict specific structure. This may be advantageous, specially when there's uncertainty about how the application uses data or when such data can take multiple forms.

It is possible however to define a ruling schema over the database's structure to control what is written into it. Choosing to use schemas or not is a tradeoff between flexibility and rigor. For documentation purposes and for an easier understanding from possible future contributors, the usage of a schema was preferred.

Figure 21 represents part of the data's structure currently being stored by the application. The `Model` and `Instance` collections support the editor feature. Every time a user chooses to share his model, it is stored and associated with an `_id` which concatenated with the platform's editor subdomain produces a valid *URL* displaying the specification. Furthermore, users can share instances associated with some model, hence the `Instance` collection containing a reference to the respective model's identification and a `graph` object to later redraw the targeted instance.

Regarding challenges, entries are composed by an `_id` that analogously to the editor's sharing mechanism can be used to propagate them. The `whole` field is a string containing the challenge as it was created with both tags and secrets. This is useful if the creator wishes to edit his challenge. He can simply use the password to unlock and edit this content. The `immutable` field holds the ranges of code containing immutable text as previously explained. The usage of lines and characters instead of simple character indexation is due to the text editor's library *API* requirements to block out the text. The `cut` field contains the challenge as it should be displayed to the contestant, without secrets or tags. Another array contained withing the `Challenge` collection is `challenges` and it holds the parsed content of the challenge's secret commands. The `name` key of these objects holds the check command label while the `value` contains the command itself. Although there's some data replication, this structure boosts performance and fits nicely with *Meteor*'s subscription/publication mechanism, preventing contestants from accessing the challenge's solution. Trivially, the `password` field refers to the challenge password defined by its creator. `derivationOf` points to another `Challenge` entry. There is no distinction between a solution and a challenge in terms of stored data. This solution supports the connection described above between states of a solution of derivations of the same. Finally, the `public` field is used to distinguish between intentionally shared solutions and automatically stored derivations of it through consecutive editing and running. This distinction dictated if the solution can be loaded through its *URL* or not, i.e., if a solution is intentionally shared by the user, the generated id can be used to access it through an *URL*, on the other hand, models stored after an execution are meant for statistic usage only and can't be reloaded by Users.

These private contents of executed models are stored in the `Run` collection. It only contains 3 fields apart from its entries `_id`: `sat` tells if the execution produced instances or not, i.e., if it was satisfiable. It also holds the name of the commands executed, so that information like the most executed command or number of successful attempts to solve a particular point of a challenge can be calculated. Lastly, the field `challenge` points to the challenge itself on `Challenge` collection. In practise, a completely solved challenge is a reference to some `Challenge` that derives from the original and has itself satisfiable entries in `Run` for each check command in the original challenge. The remaining collection, `ChallengeInstances`, allows users to share instances of some challenge's execution similarly to the editor's feature.

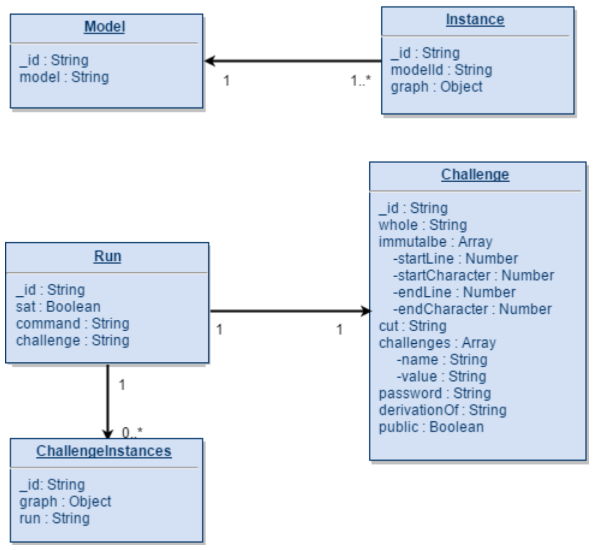


Figure 21: Database schema representation

## 5.7 VISUALIZER

Most of the project's effort was dedicated to the visualizer for multiple reasons. For once even using an existing open source graph rendering library, there are lots of essential features that must be implemented on it to shape it as an *Alloy* instance visualizer. This implied a vast search for *JavaScript* libraries flexible enough to fit in some of these features. Fortunately there is a vast range of this kind of tools, many of which are well developed and documented.

Apart from these difficulties there's the enhancement issues. As previously stated, there is some dissatisfaction among the community regarding some of the functionalities of the current *Alloy Analyzer's* visualizer version. Therefore, these issues had to be studied and rethought towards friendlier functionality.

### 5.7.1 *Data of an Alloy instance*

Back in the editor's architecture, it was demonstrated through Figure 17 how data flows from client to server to *webservice* and vice versa. Step 7 or the *Alloy* service's response is a string structured as a *JSON* document that is then parsed by the server and returned to the client. This document contains either details about syntax errors or a found *Alloy* instance. Once transformed into a *JavaScript* object, the data can be easily accessed for all the information required to build an informative graph.

```
{
  "sigs": [{
      "type": "STRING",
      "isPrimSig": "BOOLEAN",
      "isSubsetSig": "BOOLEAN",
      "parents": ["STRING"],
      "atoms": ["STRING"]
    }],
  "fields": [{
      "fieldName": "STRING",
      "arity": "INTEGER",
      "tuples": [["STRING"]]
    }],
  "skolems": [{
      "label": "STRING",
      "atom": "STRING"
    }]
}
```

Listing 5.6: Instance data representation

The *JSON* code listed on 5.6 outlines the information required to fill the visualizer. There are as many different signatures in the model as entries in `sigs`. Each entry contains the signature respective original name as stated in the specification, it informs if the signature is a subset signature or a primary one, i.e., if its contained in some other signature or not. If it's a primary signature, `parents` may hold its father's type name if one exist (the signature extends it's father). For subset signatures, `parents` will contain the name of the signature or signatures in which the former is contained. Finally `atoms` is a set of identifiers for each atom of a type in the given instance.

The `fields` attribute represents relations between the signatures described above. Each field defined in some signature specification is stored here stating its name, its arity and the tuples linking the involved atoms.

*Skolems* refer to reduced formulas without quantifiers which are equivalent to other quantified formulas. This is achievable using skolem constants or functions that capture the constraint of the quantified formula in their values (Jackson). These *skolems* are added to the rendered instance graph's elements as text making them more conspicuous.

### 5.7.2 *Rendering instances with CytoscapeJs*

As shown bellow on listing 5.7 *CytoscapeJs*'s initialisation mainly requires some components: The `container` or *HTML* element to be converted into a drawable canvas, an `elements` object containing all the graph's nodes and edges with their associated data, a `style` object dictating how those elements are to be drawn, i.e., colours, shapes, labels, etc. And finally a `layout` object ruling over the nodes positions.

The first step towards rendering an *Alloy* instance using this library was converting the previously structured data shown in Listing 5.6 into an `elements` array. Given the atoms names unique nature, these were chosen to act as identifiers, avoiding conflicts and simplifying the relations association.

Much like the *Alloy Analyzer*, relations with arity superior to two, i.e., involving more than two atoms were simplified into binary relations between the first and last atoms and the remainder atom names were concatenated with the relation name. For example a tuple with arity four relating `A$1->B$1->C$1->D$1` named `relation1` would be represented as an edge between `A$1` and `D$1` with the label `relation1[B$1->C$1]` by default. Additionally, atoms were complemented with the corresponding skolems as additional text to their labels.

Every time a node within the rendered graph is selected or dragged its style is recalculated and the graph is rerendered. Therefore, fields defined within the style array such as the ones referring to nodes `shape`, `background-color` and `content` in Listing 5.7 are automatically triggered. There are two possible approaches to set values to these fields: Store aesthetic features as data on the formerly described elements and apply them using `'data(field_name)'` or use functions to perform more complex tasks than simply accessing the element's data. The library is flexible enough so that node's colours, shapes, borders and sizes are manageable using this mechanism as well as edge's colours, contours and length.

Regarding the layout option object, *CytoscapeJs* offers six different other options as graph layouts like the displayed `'preset'`. :

- Preset - Node positions within the graph must be specified individually in its element data.
- Random - The random layout puts nodes in random positions within the viewport.
- Grid - The grid layout puts nodes in a well-spaced grid.
- Circle - Places nodes in a circle.
- Cocentric - The concentric layout positions nodes in concentric circles, based on a metric that you specify to segregate the nodes into levels.
- Breadthfirst - Puts nodes in a hierarchy, based on a breadthfirst traversal of the graph.
- Cose - The cose (Compound Spring Embedder) layout uses a physics simulation to lay out graphs.

```
cytoscape({
  container: document.getElementById('cy'),
  elements: [
    { group: 'nodes', // node n1
      data: {
        id: 'n1', // Mandatory for each element
        optionalData1 : "blue", //Any kind of relevant associated data
        optionalData2 : "rectangle"
      },
    },
    { group: 'nodes', // node n2
      data: { id: 'n2' }
    },
    { group: 'edges', // edge e1
      data: {
        id: 'e1',
        source: 'n1', // The source node id (edge comes from this node)
        target: 'n2'  // The target node id (edge goes to this node)
      }
    }
  ],
  layout: {name: 'preset'},
  style: [
    { //Style applied to every node
      selector: 'node',
      style: {
        'content': 'data(id)',//Node's label according to element data
        'shape': 'data(optionalData2)',//Node's shape according to element
           data
        'background-color' : function(ele){//Node's colour using a function
            return ele.data().optionalData1;
        }
      }
    }
  ]

});
```

Listing 5.7: *CytoscapeJS*'s initialisation components

### 5.7.3 *Theme settings*

At first glance, keeping the atom's individual features such as colour, shape, label, etc, stored in its data as *CytoscapeJs* elements seems appealing but unfortunately it would complicate supporting the current inheritance feature of *Alloy Analyzer*'s themes. . In practise, atoms belonging to some type should be able to inherit their father's appearance unless edited in some other way. To solve this issue, data structures holding theme settings and hierarchic information were defined and the node's appearance is mostly being set by the styling functions which traverse these structures and select the appropriate features as diagrammed on Figure 22.

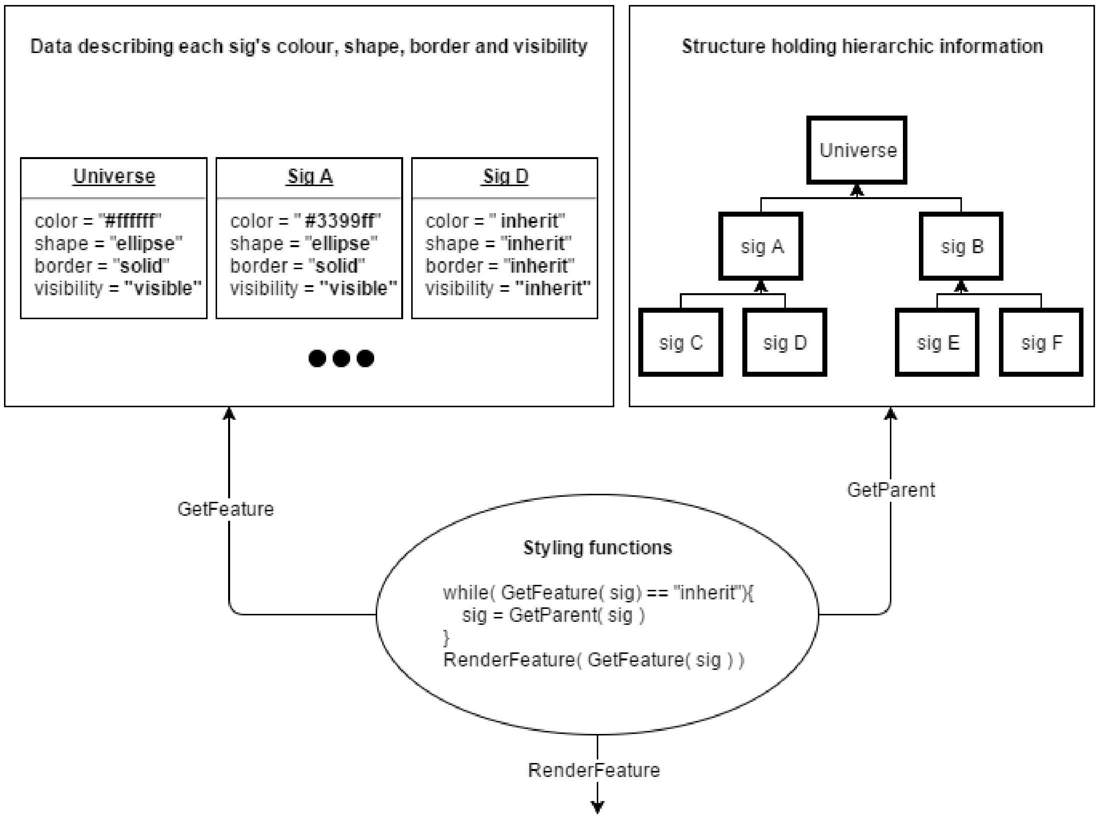


Figure 22: Styling procedure

Take for example the colour of an atom `D$1`, the function's behaviour is to continuously look for a valid colour throughout its ancestors. Hence, it starts by consulting the colour of atoms of type D and once it retrieves the `"inherit"` value, goes to the hierarchic tree to look for its father. Once again it looks for the colour value of `A` which `D` extends according to the example's tree and acquires a valid hexadecimal triplet to render as the `D$1`'s colour.

This nicely packaged theme data associated with each signature facilitates its storage and loading while sharing instances. By simply annexing these objects to their respective model entries on the database and reloading them once that same entry is reopened in the editor, sharing theme options is enabled.

Currently the platform supports the editing of the following theme settings:

- Atom colours using any hexadecimal colour triplet.

- Eleven kinds of atom shapes.
- Atom Borders between `dotted`, `dashed`, `solid` and `double`.
- Atom labels.
- Hiding unwanted atoms.
- Hiding unrelated atoms. Unlike the previous option, that hides all the atoms of some type, this option hides atoms of some type that are uninvolved in any relation present in the instance.
- Displaying atoms numbers.
- Use the original atoms names.
- Relations labels.
- Relations colours using any hexadecimal colour triplet.
- Relations edge style, i.e., either `dotted`, `dashed` or `solid`.
- Choosing to display relations as atom attributes or arcs(edges).

Furthermore atom positions can be automatically rearranged according to every layout made available by *CytoscapeJs* as previously detailed. Some of these layouts, `cose` in particular, attempt to ideally distribute graph's nodes throughout the canvas, maximising the space usage and improving the visualisation.

### 5.7.4 *Theme settings user interface*

Tweaking all the above settings is possible through the platform's visualizer interface. Likewise the *Alloy Analyzer* a sided tab was chosen to contain all the necessary inputs as illustrated on figure 23.

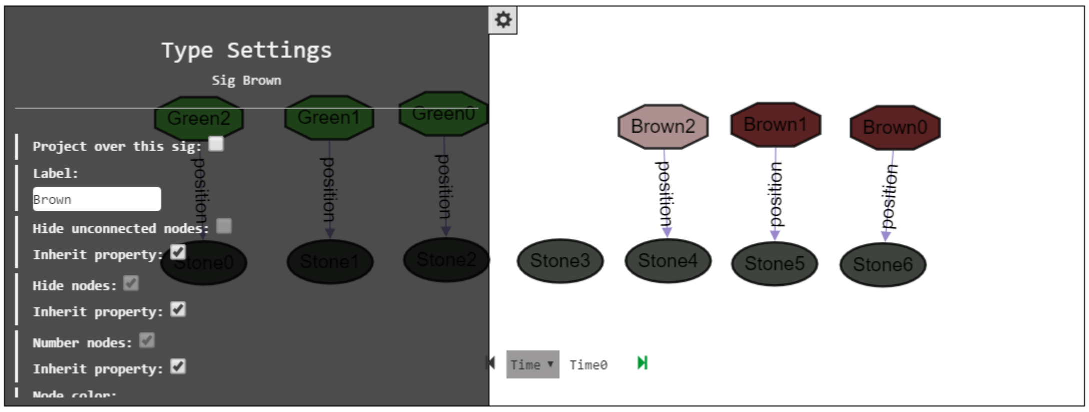


Figure 23: Theme settings user interface side panel

Users can open and close this panel using a top left button and scroll down through the options. Its content changes between three different views according to which element is selected in the *CytoscapeJs* canvas. Users can select elements by left clicking on them. If an atom is clicked, the panel's content will be referent to options about that atom's signature. If an edge is selected, analogously to the former, options regarding the specific relation will fill the side panel. Finally, options about the graph's layout and some other settings will be displayed if the canvas background is selected.

Additionally, users can navigate between options of different signatures or relations on the bottom of all the panel's views. This is useful to edit previously hidden atoms (not clickable) or to select abstract signatures not represented on the graph. Editing these signatures may affect their descendants appearances if the environment is set so, which it is by default.

Besides using the side panel to edit all these settings, users can right click nodes which will pop open a menu to edit their colours, shapes and to project over their signature as displayed on Figure 24.

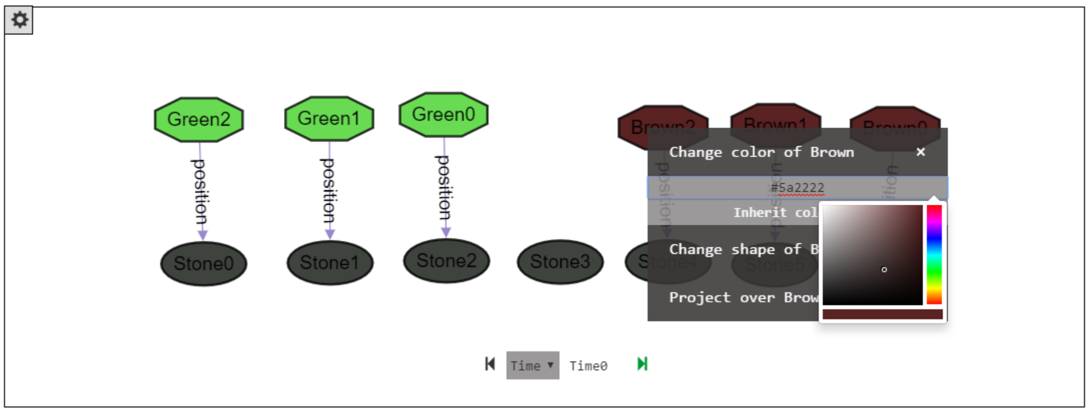


Figure 24: Theme settings user interface with right click event

This option provides a faster access to three of the most significant aspect settings in an intuitive way. Something that the *Alloy Analyzer*'s visualizer is not equipped with. Plus, its worth mentioning that every aspect change or changes made in the *Alloy Analyzer*'s theme editor must be manually applied (clicking an apply button) before being observable. Our platform automatically applies every change immediately after it's made which is convenient.

## 5.8 PROJECTION

As explained in the background chapter, the *Alloy Analyzer*'s projection feature is a very useful tool to mostly make dynamic model's instances more intelligible. Still, there are some complaints about it, specially regarding the highlight of state changes. (**?**) in particular suggests the visualiser should display more than a state at a time and the positions of the atoms should be consistent between them. This feedback was taken into account when building the web visualiser and two possible improvements were implemented and studied.

### 5.8.1 *Multiple frame visualisation*

The first idealised solution of an improved projection functionality for the *web* visualizer was representing projections as a single graph, i.e, without explicit frames representing states. Take for example Figure 25, illustrating in practise how the solution would represent the frog puzzle instance. Using a *CytoscapeJs* feature called compound nodes, state atoms being projected over would engulf their respective frames as their parent nodes. Plus, users would be able to adjust atom positions which would affect all the same atoms inside each frame. Lets say the user dragged the example's `Stone$0` atom to the right. All the other `Stone$0` atoms within the `Time` compound nodes would move to the right as well. This way, not only multiple frame visualisation would be possible simultaneously, but atoms would also maintain their positions throughout the states which would facilitate immensely identifying changes between them, thus improving the visualiser.

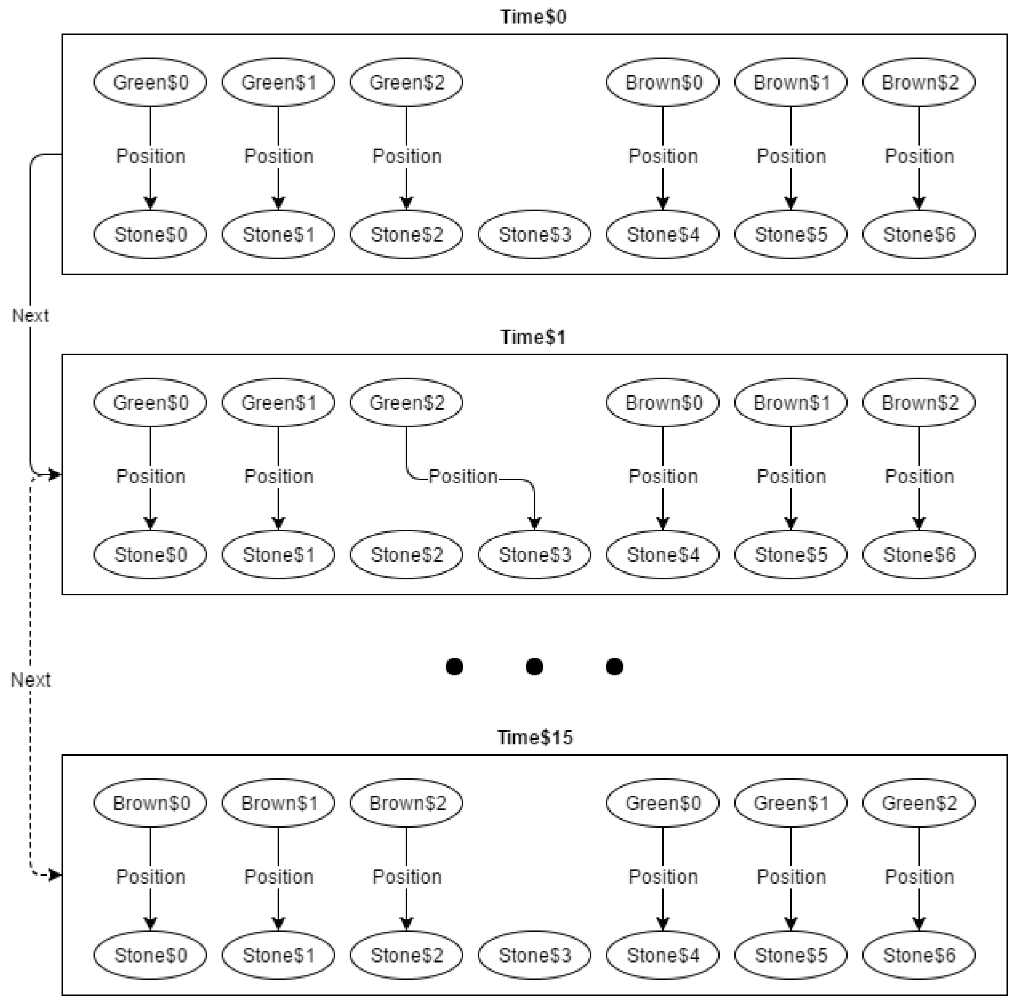


Figure 25: Multiple frame visualization projection

Once the solution was implemented though, two distinct problems arose. Although not frequently used, its possible to project over multiple signatures simultaneously. Well, lets say someone required projecting over `Time` and `Stone` signatures in the previous example. Considering the instance contains fifteen `Time` atoms and seven of type `Stone`, the canvas should render fifteen times seven compound nodes at once. The first problem was the positioning the atoms in a clear and automatic way. Eventually with complex enough instances we would be back to the illegibility of unprojected instances. Furthermore, calculating these huge instances would become impractical even for examples such as the one presented regarding execution times. Given the circumstances a different approach was attempted in order to obtain a viable solution.

### 5.8.2 *Current projection*

Currently the application allows projection alike the *Alloy Analyzer* through frame by frame inspection as shown on Figure 26. Still, the concerns previously explained weren't ignored. The issue about atom positioning was solved by storing and reapplying atom positions on the canvas between frame navigations. The simultaneous inspection of multiple frames though was thought of, but not implemented so far. The idea behind it is to have as many *CytoscapeJs* canvases as intended by the user, allowing him to choose which frame to render on each canvas. Given the implementation effort required, its developments was postponed to future work.

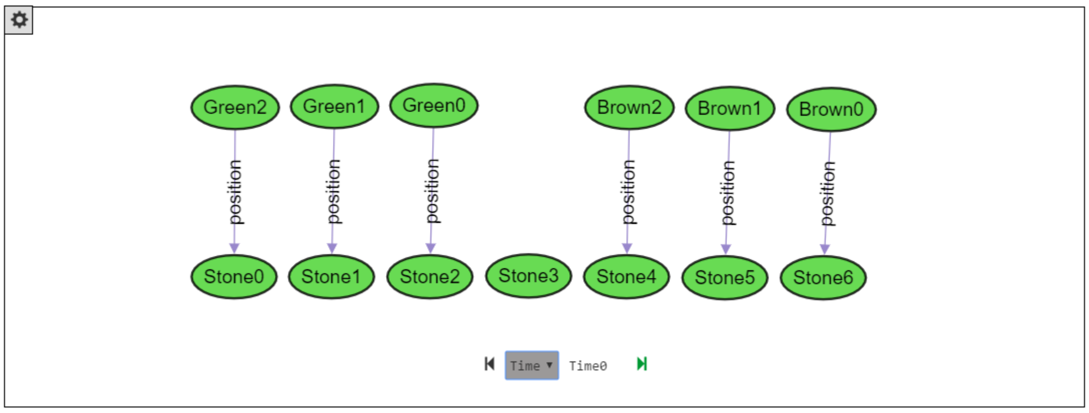


Figure 26: Current projection feature

In case of multiple signature projection, users can choose in which type they'd like to navigate by selecting it in a combo box between the next and previous frame buttons.

## 5.9 SUMMARY

This chapter described the platform workflows and most outstanding details of its functionality. Here, the editor, visualizer challenge creation, solving and data collection mechanisms were explained to give the reader some insight of whats under the hood of the *Alloy* web environment. The next chapter presents the conclusions taken from this project's development and research as well as possible future developments.

# 6 CONCLUSIONS AND FUTURE WORK

This thesis describes the research and development carried out to build a web framework for *Alloy*, supporting, among others, most of the features currently available in the standalone *Alloy Analyzer*. Lots of web platforms perform similar roles for other languages either having recreational, didactic or competitive purposes. We started by studying these platforms in order to determine the features that are fundamental for a successful tool, capable of empowering the language and growing the community. Our contributions were essentially:

- An environment accessible through any web browser, were users can write *Alloy* specifications and analyse them, producing instances and inspecting them, as they would using the *Alloy Analyzer* stand alone. This removes the necessity for downloading the executable `.jar` and installing additional software (*Java*), thus providing an easier access to it.
- A mechanism for easy sharing of models or instances in the community. Instead of sending *.als* files or entire specifications, users disseminate their work through *URL*s. Additionally, we implemented a similar feature to share instances, something new in the community and that will simplify a lot the understanding of (and discussion about) concrete models.
- Bringing the competition element to the the language through the creation of challenges with the capability of automatically verifying solutions. We expect this feature to be a great contribution not only but mainly for academic environments. It empowers contestants/students by providing immediate feedback on their solutions through meaningful counter-examples.
- An infrastructure capable of gathering all sorts of data about the platform's usage. Mining such data, would enable the extraction of profitable knowledge like profiling challenge contestants, understanding their methods and difficulties.

## 6.1 FUTURE WORK

Its easy to imagine many new features given the prototype stage of the platform or identify less significant missing ones from the *Alloy Analyzer* stand alone. Still, some are specially worth mentioning as they would greatly contribute to the environment's robustness:

- Enabling the inspection of instances as text rather than using the platform's graphical visualiser, considering a great number of users prefers so. Even the most sophisticated graph manipulation/inspection tool would be ineffective given large enough instances leaving text as one viable option.
- Further develop the instance visualiser, enabling the inspection of multiple frames in dynamic models.

## 6.2 EVALUATION

The greatest point in this project's future work is its evaluation. Ideally, the platform should be used extensively by real users in order to get some feedback on several aspects of it, as well as understand its true potential. The most immediate requirement is the usability experiences to better determine what features should be improved and added. Additionally, the usage would help understand the true potential of the data exploration and what complementing information should be gathered to build more valuable knowledge. Finally, studying the application scalability will only as well be possible under such circumstances.

# BIBLIOGRAPHY

Thomas Ball, Peli de Halleux, Nikhil Swamy, and Daan Leijen. Increasing human-tool interaction via the web. In *Proceedings of the 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 49–52. ACM, 2013.

Christie Bolton. Using the Alloy Analyzer to verify data refinement in Z. *Electronic Notes in Theoretical Computer Science*, 137(2):23–44, 2005.

RC Boyatt and JE Sinclair. Experiences of teaching a lightweight formal method. *Electronic Notes in Theoretical Computer Science*, pages 71–80, 2008.

M Delest, I Herman, and G Melançon. Tree visualization and navigation clues for information visualization. In *Computer Graphics Forum*, volume 17, pages 153–165, 1998.

Joao F Ferreira, Alexandra Mendes, Alcino Cunha, Carlos Baquero, Paulo Silva, Luís Soares Barbosa, and José Nuno Oliveira. Logic training through algorithmic problem solving. In *Tools for Teaching Logic*, pages 62–69. Springer, 2011.

Max Franz, Christian T Lopes, Gerardo Huck, Yue Dong, Onur Sumer, and Gary D Bader. Cytoscape. js: a graph theory library for visualisation and analysis. *Bioinformatics*, page 557, 2015.

Ivan Herman, Guy Melançon, and M Scott Marshall. Graph visualization and navigation in information visualization: A survey. *Visualization and Computer Graphics, IEEE Transactions on*, 6(1):24–43, 2000.

Walter B Hewlett and Eleanor Selfridge-Field. *Melodic similarity: Concepts, procedures, and applications*, volume 11. Mit Press, 1998.

Daniel Jackson. Alloy documentation about skolemization relations. URL `http://alloy.mit.edu/alloy/documentation/quickguide/skolem.html`.

Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

John Lamping, Ramana Rao, and Peter Pirolli. A focus+ context technique based on hyperbolic geometry for visualizing large hierarchies. In *Proceedings of the SIGCHI conference on*

*Human factors in computing systems*, pages 401–408. ACM Press/Addison-Wesley Publishing Co., 1995.

Jose Paulo Leal and Fernando Silva. Managing programming contests with mooshak. *Software—Practice & Experience*, 33(6):567–581, 2003.

Helen Purchase. Which aesthetic has the greatest effect on human understanding? In *Graph Drawing*, pages 248–261. Springer, 1997.

Derek Rayside, Felix Chang, Greg Dennis, Robert Seater, and Daniel Jackson. Automatic visualization of relational logic models. *Electronic Communications of the EASST*, 7:14, 2007.

James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William E. Lorensen, et al. *Object-oriented modeling and design*, volume 199. Prentice-hall Englewood Cliffs, 1991.

J Michael Spivey and JR Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

Nikolai Tillmann, Jonathan de Halleux, Tao Xie, and Judith Bishop. Pex4fun: A web-based environment for educational gaming via automated test generation. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 730–733. IEEE, 2013.

Atulan Zaman, Iman Kazerani, Medha Patki, Bhargava Guntoori, and Derek Rayside. Improved visualization of relational logic models. *University of Waterloo, Tech. Rep*, 2013.

Pamela Zave. A practical comparison of alloy and spin. *Formal Aspects of Computing*, 27(2): 239–253, 2015.