Experiences on Teaching Alloy with an Automated Assessment Platform

Nuno Macedo[1,2], Alcino Cunha[1,2(✉)], José Pereira[2], Renato Carvalho[1,2], Ricardo Silva[2], Ana C. R. Paiva[1,3], Miguel Sozinho Ramalho[1,3], and Daniel Silva[3]

[1] INESC TEC, Porto, Portugal
[2] University of Minho, Braga, Portugal
alcino@di.uminho.pt
[3] University of Porto, Porto, Portugal

Abstract. This paper presents Alloy4Fun, a web application that enables online editing and sharing of Alloy models and instances (including dynamic ones developed with the Electrum extension), to be used mainly in an educational context. By introducing secret paragraphs and commands in the models, Alloy4Fun allows the distribution and automated assessment of simple specification challenges, a mechanism that enables students to learn the language at their own pace. Alloy4Fun stores all versions of shared and analyzed models, as well as derivation trees that depict how they evolved over time: this wealth of information can be mined by researchers or tutors to identify, for example, learning breakdowns in the class or typical mistakes made by Alloy users. Alloy4Fun has been used in formal methods graduate courses for two years and for the latest edition we present results regarding its adoption by the students, as well as preliminary insights regarding the most common bottlenecks when learning Alloy (and Electrum).

Keywords: Teaching formal methods · Alloy · Automated assessment

1 Introduction

Alloy [6] is a popular formal specification language, accompanied by a toolkit, to describe and reason about software design. It is taught in several undergraduate and graduate courses in formal methods, including graduate courses taught by some of the authors at University of Minho (UM) and University of Porto (UP), in Portugal. One of the reasons for this popularity is the support for automated analysis provided by the Alloy Analyzer, an easy to download and install self-contained executable written in Java. The Analyzer also allows instances (either witness scenarios or counter-examples) to be graphically depicted using user-customized themes, a popular feature both for experienced users and students. Alloy is very effective in the specification and analysis of the static structures that pervade software design, but requires the employment of well-established

A. Raschke et al. (Eds.): ABZ 2020, LNCS 12071, pp. 61–77, 2020.
https://doi.org/10.1007/978-3-030-48077-6_5

idioms, that introduce an explicit notion of state or time, if mutability is to be considered and temporal properties analyzed. To avoid this cumbersome and error-prone process, several extensions to Alloy have been proposed, including one by authors of this paper – Electrum [7] – which extends the Alloy language with variable structures and linear temporal logic (including past operators), also adding bounded and unbounded model checking engines to the Analyzer.

Despite such streamlined toolkit, over the many years we taught and researched with Alloy we identified some missing features and functionalities that could further ease its adoption and its usage in an educational context. The first is the lack of a straightforward mechanism to *share* simple Alloy models, instances[1] and associated themes. This would be particularly useful for students trying to get feedback from the tutors about specific counter-examples, or to submit exercise resolutions for evaluation. The second is the absence of some *automated assessment* functionality or online judge system for students to independently check the correctness of their exercise resolutions. Due to some limitations of the visualizer packaged with the Analyzer, we also felt the need for a more decoupled infrastructure to test alternative instance *visualization* features.

To address these limitations we developed Alloy4Fun, a web application that enables online editing and sharing of Alloy and Electrum models[2] and instances, including simple specification challenges in the form of duels where students attempt to discover a secret specified by the tutors. Such online platform also provided us the opportunity to collect information regarding Alloy usage patterns from an extended user base: one of the features of Alloy4Fun is thus the ability to record every interaction with the (anonymous) user, information that is made available to the creator of the challenges for subsequent analysis. Over the last two years, Alloy4Fun has been used in 3 editions of graduate courses on formal methods and a tutorial at an international venue, which has allowed us to quickly obtain insight on how students use the language, namely identify typical mistakes or learning breakdowns in the class.

This paper presents Alloy4Fun and reports on its application in teaching Alloy, starting with an overview of (and rationale for) its current features in Sect. 2. Section 3 reports on its deployment in a formal methods graduate course (Sect. 3.1), including our experience on defining exercises, results regarding usage and adoption of the platform (Sect. 3.2), and some preliminary insights on Alloy usage patterns and learning pitfalls (Sect. 3.3). Finally, Sect. 4 concludes the paper and presents some ideas for future work. Knowledge of Alloy is not required to understand the paper, but can help better appreciate some of the features of Alloy4Fun.

[1] In Alloy literature, specifications are usually referred to as models, and the results of animation/verification commands as model instances.

[2] Electrum is retro-compatible with Alloy: models without temporal features are valid Alloy, apart from protected keywords. For readability we will simply refer to Alloy throughout the paper, unless some Electrum-specific feature is being discussed.

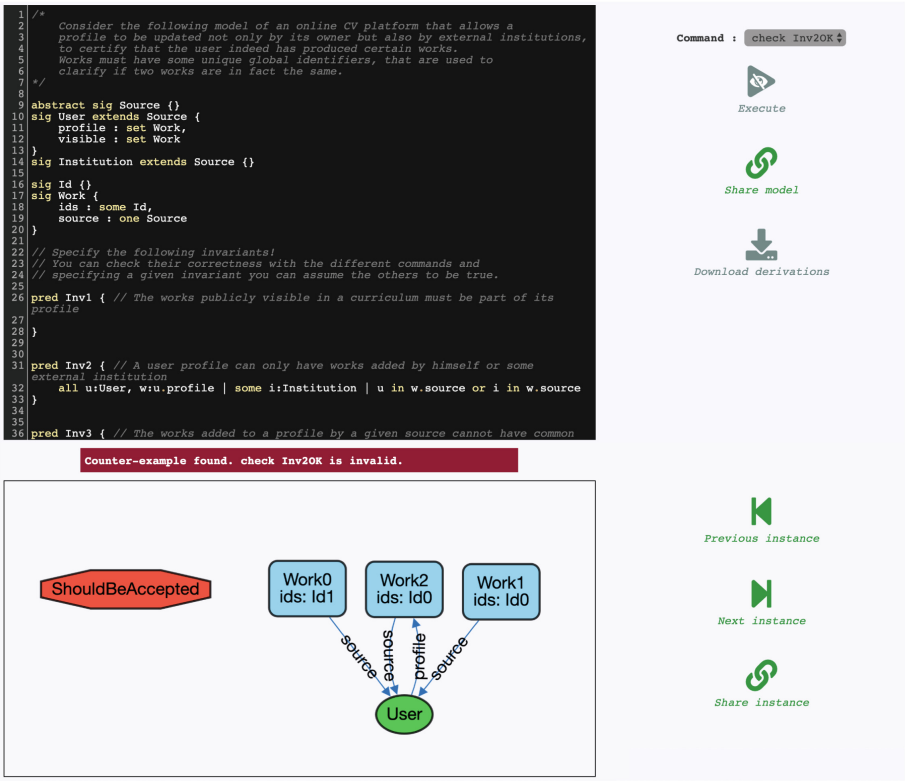


Fig. 1. A failed attempt to solve a challenge in the `CV` exercise.

2 Alloy4Fun Overview

The core of Alloy4Fun mimics in a web application the main features of the standalone Alloy Analyzer. After accessing alloy4fun.inesctec.pt (the URL where Alloy4Fun is currently deployed) the user gets an empty online editor (with syntax highlighting) where Alloy models can be written. An Alloy model consists of a sequence of paragraphs: each paragraph is either a *signature* (and the respective *fields*) declaration, a *fact* with a constraint that is assumed to hold, an *assertion* with a constraint to be checked, or an auxiliary *predicate* or *function* definition. Signatures introduce sets of elements (known in Alloy as *atoms*) and fields establish relations of arbitrary arity between those sets. Disjoint subset signatures can be declared by *extension*, and the parent signature can be marked as *abstract*, if it should only contain atoms present in its extensions. For example, the Alloy4Fun screen capture shown in Fig. 1 shows a model of an online *Curriculum Vitae* (`CV`) platform, an example that was used as an exercise in classes. This model declares a signature `Source` that is partitioned in two subsets, `User` and `Institution`. Two more signatures are declared in this example:

`Id` and `Work`. We also have several fields that relate atoms of these signatures. For example, `ids` is a binary relation that associates each atom of `Work` with its set of `Id`s. Signature and field declarations can have *multiplicities* attached to impose cardinality constraints. For example, the **`some`** in the declaration of field `ids` imposes that each `Work` should have at least one `Id`.

Formulas in facts, assertions, and predicates, are written in *Relational Logic* (RL), an extension of *First-Order Logic* (FOL) with operators that can be used to combine relations (*aka* predicates in FOL). The most frequently used one is the relational *composition* (written as `.`), an operator that allows us to "navigate" through a relation: for example, in predicate `Inv2` of Fig. 1, expression `u.profile` denotes the set of atoms of signature `Work` associated with `User u`. In Alloy every signature and field is immutable. With the Electrum extension they can be declared as mutable, and formulas can also be specified with *Linear Temporal Logic* (LTL) operators.

A distinctive feature of Alloy is that analysis commands can also be declared as paragraphs in a model. There are two kinds of commands: **`run`** commands, that verify the satisfiability of the declared facts and can be used to get witness scenarios; and **`check`** commands, that verify the validity of an assertion (assuming the facts to hold) and, if that is not the case, return a counter-example. All the analysis commands operate in a bounded domain: there is a user-defined *scope* imposed on every signature that limits the maximum number of elements that will be considered by the automatic verification procedures. In Alloy4Fun the topmost right button allows analysis commands to be executed: the command to be executed can be selected in the drop-down immediately above. If witnesses (in the case of **`run`**) or counter-examples (in the case of a **`check`**) are found, they are depicted below the editor as graphs that, likewise in Analyzer, can be customised with user-defined themes.

Besides these core functionalities, Alloy4Fun has some new features (and some improvements to existing ones) when compared to the Analyzer, as described in the sequel. Currently, it also has some limitations, most notably the inability to choose the underlying SAT solver used to perform a given analysis, not being able to display an unsatisfiable core, and lack of support for Alloy's module system (except for the standard modules distributed with Alloy, which can be used). In the specific case of Electrum, Alloy4Fun lacks the more sophisticated trace exploration options available in the Electrum Analyzer [3], as described next.

Instance Visualization and Navigation. When compared to the Analyzer, Alloy4Fun follows a more lightweight approach to the user interface, allowing the most common theme customizations (like changing the color of the atoms of a given signature) to be performed quickly through a right-click menu on atoms or edges. We also stripped down a bit theme features to a subset that we identified as those more commonly used. Alloy4Fun themes allow color, shape, stroke, and visibility parametrization for signatures and fields, signature projection, and the display of fields as attributes inside atoms. Among the unsupported features we have, for example, the customization of the atom labels for each signature or the

ability to hide only unconnected atoms of a particular signature. A new feature is the ability to select different layout algorithms to automatically organize nodes, which the user can then manually move. Unlike in the Analyzer, atom positions are preserved between the frames of projected instances, and when navigating the different states of a trace in the case of an Electrum (mutable) instance. In Fig. 1 a counter-example of a **`check`** command named `Inv20K` is being depicted with a user-defined theme. Unlike in the Analyzer, besides navigating to the next instance the user can also re-visit previously presented instances. In the case of Electrum, Alloy4Fun only allows one state of an instance trace to be visualised at a time (the Electrum Analyzer depicts two states side by side), and it is only possible to ask for a different next trace (the Electrum Analyzer has more sophisticated trace exploration options, for example it is possible to ask for trace with the same prefix up to the displayed state, but a different next state).

Sharing Models and Instances. The standard Alloy Analyzer provides limited support for model and instance sharing: they can be saved in separate files, which can then be shared using external tools (email, online repositories, etc), to be again opened at the destination for inspection or editing. When a visualization theme has been developed to ease the interpretation of instances, it must also be shared in an additional file. This sharing by saving/opening files rapidly becomes tedious and time consuming in some contexts, in particular for tutors of large classes that interact frequently with students (typically by email) to clarify doubts. Alloy4Fun provides the ability to easily share models and instances. After pressing the "share model" button a *permalink* is generated, that can later be used to access the model. Any theme defined by the user is also preserved when sharing, thus allowing instances of shared models to be depicted as intended by their creators. Concrete instances can also be shared via *permalinks*. The theme and positions of the depicted atoms and relations at the time of sharing are also preserved. This is a very handy feature since, likewise in the Analyzer, the positioning of atoms by the automatic layout mechanism is often not ideal, requiring manually rearrangement for better comprehension. For instance, the instance presented in Fig. 1 can be shared as depicted[3].

Anonymous Interaction. In Alloy4Fun there are no user accounts nor means to recover the *permalinks* of previously shared models and instances. The user is responsible for keeping track of relevant *permalinks* using some external mechanism (Alloy4Fun provides a "copy to clipboard" button to ease this task). The anonymity, namely the absence of user accounts, was a design choice made in order to keep the interaction with the web application as simple as possible, to maximize user exposure, and also to avoid dealing with privacy and security issues, namely the hassle of storing and managing user credentials and of implementing mandatory regulations concerning data protection.

Automatic Assessment. Although the Alloy specification language has very neat and simple syntax and semantics, many students struggle with its declarative

[3] http://alloy4fun.inesctec.pt/8Q4Sbjqj4KzHuvuNC.

nature, in particular those used to procedural programming [2]. One way to overcome this difficulty is by independently solving exercises proposed by tutors, but, even with automated analysis and visual feedback, it is often difficult for students to assess whether they reached the correct answer, and tutors are required to inspect and interpret the solutions (something not scalable for large classes). These problems could be mitigated with automatic assessment functionalities, allowing students to solve exercises at their own pace and without the constant need for face-to-face time with tutors. In recent years, auto-graders and online judges have become widely popular for learning how to program [10], and we believe this success could be replicated in the learning of formal methods in general, and Alloy in particular.

With this in mind, the user in Alloy4Fun has the ability to mark any paragraph of a model as *secret*, by adding the special comment `//SECRET` immediately before. When sharing a model with secret paragraphs two *permalinks* are generated: a private one that, when accessed, reveals the full model, including secrets; and a public one that, when accessed, only shows public paragraphs, but internally still considers the secret in analyses and still allows the execution of secret commands (whose names are public). Using a comment instead of a new keyword to mark secret paragraphs ensures compatibility with Alloy's default syntax, allowing users to copy and paste models from Alloy4Fun to the standalone Analyzer, and vice versa. Section 3.1 will describe how this feature can be used to create simple specification exercises in the form of duels, where the user/student tries to reach a secret specification. The instance shown in Fig. 1 was obtained precisely by accessing the public *permalink* of an exercise, and failing to solve a challenge, for which a counter-example was returned.

Mining Derivation Trees. A possible way to gain insight about the students' learning process is to have access to their attempts at solving the proposed exercises, and tool support to mine this corpus for useful data [8]. Again, such feature would also be useful for research, and was one of the reasons that led Microsoft to develop the www.rise4fun.com web service, that allows researchers to easily deploy their tools on the web and collect human-tool interactions for posterior mining [1] (besides other advantages of web tools, like increased exposure, since the need for downloading and installing is eliminated, and promoting reliability given the large amount of test cases that can be collected). One of the most popular examples available via Rise4Fun, and the inspiration for developing Alloy4Fun, is www.pex4fun.com, a web-based educational gaming environment for learning programming, where students can engage in coding duels where they attempt to write code equivalent to a tutor's secret implementation [12]. Pex [11], an advanced white box test-generation tool, is used on the background to find inputs that show discrepancies between the student's code and the secret implementation. However, the interaction with the outcome of the tools has limitations in Rise4Fun, which would prevent the implementation of key Alloy features such as instance iteration and customization. This has led us to implement our own solution rather than integrate Alloy in this service.

Every shared model and instance is stored by Alloy4Fun in its database. However, to enable the proponents of challenges to mine the submissions for useful information, every model for which a command was executed is also stored, along with the respective result (e.g., whether satisfiable or not, or whether errors were thrown). Moreover, for each model, the identifier of the model from which it derives and a time-stamp are also stored. This means that all the models that are developed after accessing a shared *permalink* end up forming a *derivation tree*. In the case of a *permalink* with secrets/challenges, a branch in this tree typically corresponds to an interactive session where one user/student is trying to solve the different challenges defined inside, and can be analyzed to determine, for example, how many challenges were solved or how many attempts were needed to solve each one. Every fork in branch represents a point where a user generated a new *permalink* for a model which was subsequently accessed multiple times. Alloy4Fun allows anyone in possession of the secret *permalink* of a model to download the respective derivation tree in an easy to process JSON format.

Implementation. Alloy4Fun was developed [9] with Meteor, a full-stack isomorphic JavaScript framework for developing web applications based on Node.js. The client uses CodeMirror as text editor and the Cytoscape.js graph visualization library to depict instances. Models and instances are stored in a MongoDB document-oriented database at the server. To execute commands, we encapsulated the Alloy Analyzer in a RESTful web service implemented in Java. Seamless deployment of both the application and the service in a server is performed using Docker. All the Alloy4Fun code is open-source and available at github.com/haslab/Alloy4Fun.

3 Experiences on Teaching with Alloy4Fun

In the first semester of the 2018/19 academic year we did a preliminary evaluation of Alloy4Fun in two graduate formal methods courses at UM and UP. The former taught Alloy for 6 weeks and had 22 students enrolled, and the latter for 4 weeks and had 156 students enrolled. Both courses had one weekly lecture and one weekly lab session. This experiment – which recorded almost 5000 interactions – allowed us to test a beta version of the application in a medium-sized audience to detect and fix bugs and identify possible design improvements. One major identified design improvement regarded a special "lock" comment available in the beta version to prevent the accidental editing of certain paragraphs that could render the challenges unsolvable (or trivially solvable). However, we noticed students rarely tried to change the model outside of the challenge predicates, and opted to remove this feature for simplicity and efficiency[4]. These first experiences also allowed us to identify which classes of exercises are better suited to be explored in Alloy4Fun, as well as how the visualization features can be explored to provide more intuitive feedback to the students.

[4] Note that Alloy4Fun was only used for self-study and not for student grading.

From this process resulted the first official release of Alloy4Fun, which has been used in the 2019/20 academic year in the UM graduate course and on an Alloy/Electrum tutorial at the World Congress on Formal Methods[5], with a refined set of specification exercises with challenges. The remainder of this section reports on the usage of the platform by the students during this latest instance of the UM graduate course, including preliminary results regarding the most common mistakes and difficulties when learning Alloy.

3.1 Alloy4Fun Exercises

The model secrets supported by Alloy4Fun can be used to create simple specification challenges in the form of duels, where the user/student tries to reach a secret specification. Such models – which we refer to as *exercises* – can have a public predicate that the student must fill-in, together with a secret **`check`** command that asserts (for a given scope) that such predicate is equivalent to the desired specification (typically in a separate secret predicate). Although useful for practicing the usage of logic (either relational or temporal) in the specification of properties, there are certain classes of problems for which the approach based on secret specifications is not well-suited, namely modeling exercises where the student is expected to freely declare signatures and fields.

The model shown in Fig. 1 was obtained precisely by accessing the public *permalink* of the `CV` exercise, which contains 4 challenges (in this case, simple problems where a natural language description of a desired property of the model is given for each of them). After filling the empty predicate (e.g., `Inv2`), the student can check whether it is a valid solution (e.g., by running secret command `Inv2OK`, for the case of `Inv2`), which will either return a "no counter-example found" message, meaning the challenge is solved, or a counter-example otherwise (as is the case in Fig. 1, showing that the specification of `Inv2` is still not correct).

Figure 2 shows the secret implementation of challenge `Inv2`: predicate `Inv2o` specifies a correct solution for the challenge and command `Inv2OK` checks the equivalence between both. In exercises such as `CV` where several desired (and natural) properties of the model are solved in different challenges, we opted to check this equivalence assuming that the remaining properties hold: if that was not the case the student would get many counter-examples where it would not be clear why their specification failed, since they would be "polluted" with distracting problems corresponding to failures of other properties. This conditional check is the reason to include `Inv1o`, `Inv3o`, and `Inv4o` as assumptions in the equivalence check `Inv2OK`. Notice that in the preamble to the exercise the students are warned that they can assume the properties in the remaining challenges to be true when solving a particular challenge.

During the course we also noticed that the students found it hard to distinguish whether the provided counter-example represents a scenario where their solution was over-specified or under-specified. For this reason, in the challenges

[5] http://haslab.github.io/TRUST/tutorial.html.

```
//SECRET
abstract one sig RejectedBy {}
//SECRET
sig ShouldBeRejected, ShouldBeAccepted extends RejectedBy {}
...
pred Inv2 { // A user profile can only have works added by himself or some external institution

}
//SECRET
pred Inv2o { all u : User | u.profile.source in Institution+u }
//SECRET
check Inv20K {
  (Inv1o and Inv3o and Inv4o and (some ShouldBeRejected iff (Inv2 and not Inv2o))) implies
    (Inv2 iff Inv2o) }
```

Fig. 2. The secret for the challenge `Inv2` of `CV` from Fig. 1.

used later in the course we opted to include two special atoms in the counter-example instance that signal whether an instance that should have been rejected or accepted by a correct specification, meaning their solution is under- or over-specified, respectively. As seen in Fig. 2 this can be achieved by introducing a singleton signature whose possible values are either `ShouldBeRejected` or `ShouldBeAccepted` and through a simple trick in the equivalence check, namely making the verification conditional to the existence of the `ShouldBeRejected` atom when the student solution incorrectly holds (or vice-versa).

The challenges used in this course were based on 6 different problems:

- `Trash`, a model of a file system trash bin.
- `Classroom`, a model a classroom management system.
- `Graph`, a specification of several standard properties of unlabeled graphs.
- `LTL`, a specification of several standard properties of labeled transition systems.
- `Production`, a model of an automated production line in a factory.
- `CV`, the *Curriculum Vitae* model used as running example in this paper.

For some of these problems we developed more than one variant (or *exercise*) focusing on different features of the language. Each variant was provided as a shared model to students and contained multiple challenges, as summarized in Table 1. The table lists the *permalink* and total number of challenges of each exercise (the columns F1 to F9 will be discussed in Sect. 3.3).

Challenges in these exercises range from trivial (e.g., asking to enforce simple inclusion dependencies or multiplicities), to more complex ones requiring the use of nested quantifiers or closures. As expected, the introduction of the Alloy (and Electrum) language and underlying logics in classes was gradual: FOL constructs were first presented, followed by the full set of RL operators, and finally the LTL operators specific to Electrum. To try to understand the impact of using relational operators, we introduced two variants of the first two problems: one where challenges were to be solved using only the FOL subset of Alloy, and another, introduced when students already had knowledge of RL, where they

Table 1. Alloy4Fun exercises shared for the 2019/20 year.

Id	Exercise	*Permalink*	Chall.	F1	F2	F3	F4	F5	F6	F7	F8	F9
1	Trash FOL	zA2MMSGy6iW8Mihep	10	0	1	2	0	0	0	0	0	0
2	Classroom FOL	Pdvipvrpr5hg7JKbs	15	6	6	9	4	0	0	0	0	0
3	Trash RL	WJdLnDL78m7mM7W4J	10	0	1	2	0	0	0	0	0	0
4	Classroom RL	i5u2pjKJt6Bz227QT	15	5	6	9	4	1	0	0	0	0
5	Graphs	28fwdmjL79X4SQ9EP	8	1	0	0	0	2	1	0	0	0
6	LTS	gqS3qTTn4B62NYmJX	7	4	2	6	6	0	2	0	0	0
7	Production	PKy7chamCieZyCix5	4	1	1	3	0	1	0	1	0	0
8	CV	X72J6js9fA3CKYQWX	4	3	0	3	0	0	1	0	0	0
9	Trash LTL	irRLJn7qbQq3xMFGp	20	0	0	1	0	0	0	0	5	14

could use all the standard Alloy operators to solve the challenges. For the `Trash` problem we also created a mutable variant, where challenges required the usage of the LTL operators of Electrum to be solved. Hence the total of 9 exercises described in Table 1. As an example, exercise `CV` (containing 4 challenges) is the one shown in Fig. 1.

3.2 Student Usage and Adoption

In the 2019/20 edition 17 students attended the UM course. Alloy was taught for 5 weeks and, for the first time in this course, Electrum was also taught for 4 additional weeks. In each week, a 1 h lecture was followed by a 2 h lab session. Alloy4Fun was used in the lab sessions that followed the lectures that introduced FOL, RL, and LTL, mainly as a way to practice the usage of these logics to specify natural language requirements.

In the lab sessions that addressed other aspects of the Alloy language and analysis not amenable for automated assessment, such as solving problems that required the development of a full model from scratch, students were expected to still use the Alloy Analyzer and locally manage their models. In principle, they could also have used Alloy4Fun to develop most of the problems addressed in those sessions, but we also wanted students to gain some experience in using the standard Analyzer, particularly since the current limitations of Alloy4Fun (presented in the beginning of Sect. 2, such as the lack of module support or the lack of sophisticated trace exploration options in the case of Electrum) might prove problematic for some more realistic problems. Thus, Alloy4Fun was only used in 4 lab sessions, each introducing a particular set of exercises – 1 session with `Trash FOL` and `Classroom FOL` after the FOL lecture, 2 sessions with `Trash RL`, `Classroom RL` and `Graphs` after the RL lecture, and 1 session with `Trash LTL` after the LTL lecture. Extra exercises (namely `LTS`, `Production`, and `CV`) were made available in the course website for the students to freely explore. Moreover, all exercises were kept available throughout the semester so that students could independently practice outside of the classes. During the course there were 3 evaluation points involving Alloy: a medium-size modeling

project (developed with the standard Analyzer outside of the classes in groups of two students), an individual written exam, and finally a supplementary exam for students failing the first attempt.

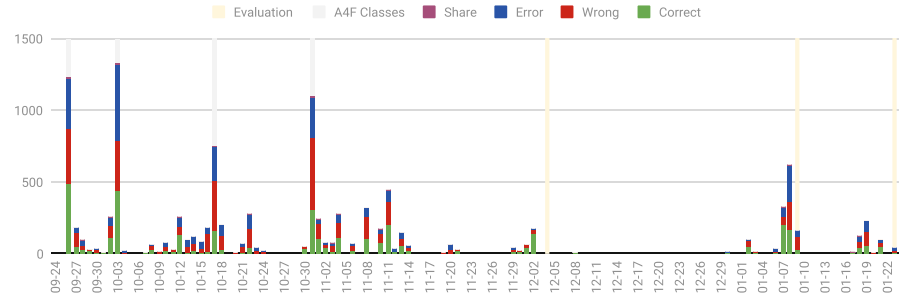
After concluding the course, the main question we tried to answer was whether students found Alloy4Fun useful as an automated assessment platform while learning Alloy. More specifically: 1) have the students used Alloy4Fun regularly outside classes? 2) in particular, have they used it when studying for the exams? 3) have they found the sharing feature useful? 4) were the counter-examples useful to reach the correct solution? To answer these question we used two methods: an anonymous questionnaire and analysis of the data collected by Alloy4Fun. The questionnaire was answered by 13 of the 17 students, and, over the duration of the course, we collected almost 11000 interactions with the exercises, most of them resulting from the execution of commands (checking the correctness of challenges) and a small portion from sharing of models[6].

Concerning the first question, of the 13 students that answered the questionnaire, 9 said they used Alloy4Fun frequently outside classes, 3 only used it rarely, and 1 never used it. To the second question all of the 12 students that used it outside classes answered that they used it to study for the exam. Of these, 9 mentioned that when studying for the exam they actually repeated some of the exercises they had already solved before. The data collected throughout the semester, shown in Fig. 3, seems to corroborate these answers. Figure 3a depicts the usage of the platform over time, highlighting the classes where Alloy4Fun was mandatory and the evaluation points (first the project deadline, and later in the semester the two exams). Each entry in the dataset is either a *correct* (unsatisfiable) check, a *wrong* (satisfiable) check, an analysis that threw an *error* (e.g., parsing) or a model stored for *sharing*. Despite the peak of usage during the Alloy4Fun classes, we can see that the students have indeed relied on Alloy4Fun outside the classes, and in particular when studying for the written exam.

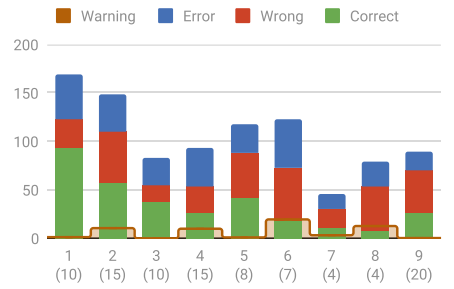
Figures 3b and 3c present statistics per exercise (below each exercise number we recall the number of challenges inside). Figure 3b presents the same execution information as Fig. 3a (except shares), with the addition of the number of successful analyses (i.e., without error) that threw a *warning*. This information is normalised taking into account the number of challenges in each exercise (i.e., the graph shows the average number of executions per challenge). This chart provides some evidence that most of the students attempted to solve all exercises, including some of those not used in class. For example, averaging the executions per challenge and per student, we have a maximum of around 10 for exercise 1 and a minimum of around 3 for exercise 7, and an overall average of around 6 attempts per challenge per student. Even taking into account failed attempts and repeated attempts to solve exercises already previously solved, it is relatively safe to infer that such numbers can only have resulted from having most of the class attempting to solve all exercises.

Figure 3c presents information regarding solving "sessions". Recall that a session is a branch in the derivation tree, typically recording the interaction of a

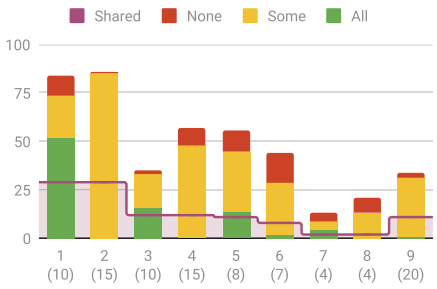
[6] This dataset is freely available in the Alloy4Fun GitHub repository.



(a) Executions over time.



(b) Average challenge execution per exercise.



(c) Sessions per exercise.

Fig. 3. Alloy4Fun usage statistics by 17 students over a semester for 9 exercises.

student with Alloy4Fun while solving the challenges inside an exercise. For each exercise we depict how many session solved all its challenges, some of its challenges, or none. Of course, some students might have multiple sessions recorded for each attempt to solve an exercise, since they might not solve all the challenges in a single continuous session and access the original shared *permalink* several times, instead of generating a new *permalink* of a partial resolution for later resuming the work. Overall we identified 430 sessions, with an average of 48 sessions per exercise. Even with all the uncertainty, it is safe to say that indeed most students should have used Alloy4Fun frequently outside the classes (from our observation, during classes students mainly used a single session per exercise), including repeated attempts to solve exercises already previously solved (as reported in the questionnaire): for example, for `Trash FOL` around 50 sessions were recorded where all the challenges were solved, a strong indicator that each student should have solved it at least twice.

Concerning *permalinks*, 7 students mentioned that they generated them frequently to store their own solutions for later access, 3 did it rarely, and, somehow surprising, 3 never did it. Generating *permalinks* for the purpose of sharing with colleagues and tutors was even less common: only 5 students did it frequently, 4 rarely and 4 never. Figure 3c also depicts how many session had at least one *permalink* generated, and indeed we can see that, for most of the exercises,

the number of *permalinked* sessions is clearly less than the number of students. Surprisingly, the share instance feature has not been used: there were only 2 generated *permalinks* for instances. These results seem to suggest that one of our main goals for Alloy4Fun – to simplify the sharing of models and instances – may actually not be that popular in an educational setting, but of course a more comprehensive study must be conducted to clarify that.

Concerning the last question, 10 students mentioned that counter-examples were frequently useful to help find the correct answer, but of these 4 only found them useful if they had the atoms that signal whether the shown counter-example should have been rejected or accepted by a correct specification. Unfortunately we have no data to corroborate this, but in principle Alloy4Fun could be used to check whether those atoms are indeed helpful or not, for example by giving two different versions of an exercise to different sets of students and then analyzing the results. This is one of the studies we intend to conduct in the near future.

Finally we also asked the students the overall question of whether they found Alloy4Fun useful for learning Alloy and Electrum: all of them agreed that was the case, with 8 of the 13 strongly agreeing.

3.3 Insights on Learning Alloy

Taking advantage of the collected data, we also tried to get some insights about how students learn Alloy, and in particular determine which features of the language pose more difficulties and should thus be addressed more carefully in lectures. To this end, we started by classifying a normalized version[7] of each challenge according to a set of required concepts, namely whether it requires:

F1 using more than 10 logic or relational operators
F2 a simple restriction of the multiplicity of a relation
F3 nested quantifications (ignoring multi-variable quantifications)
F4 manipulating ternary relations
F5 transitive closure over fields
F6 transitive closure over expressions (either relational expressions or relations by comprehension)
F7 reasoning about total orders (i.e, using the `ordering` module)
F8 a single temporal operator
F9 nested temporal operators

For each exercise, Table 1 presents the number of challenges that fall into each of these (non-exclusive) categories. Figure 4 compares the results of challenge execution classified under each category (also listing the total number of challenges for each). For each of the 9 categories, the number of correct (green) and wrong (red) executions are presented. Additionally, entry F0 collects the results of challenges that require none of the above concepts, and category All the results for all challenges. Of all the 7689 executions without errors, 3682 were correct (48%), meaning that in average each challenge required two attempts to be solved (after solving possible errors).

[7] Normalized specifications were expanded into almost pure FOL (or FO-LTL when temporal logic was required), using no relational operators except for closures.

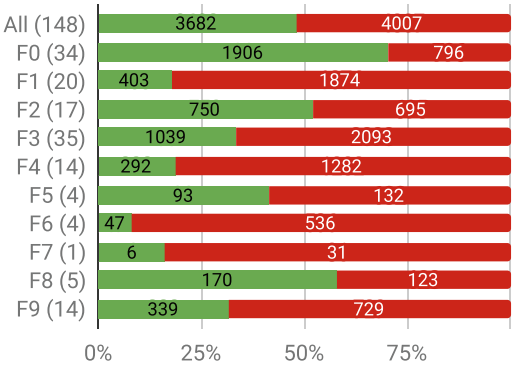


Fig. 4. Executions per class of challenge. (Color figure online)

As expected, challenges requiring none of the listed concepts (F0) were simpler (71% success rate), and those requiring more than 10 operators (F1) were notoriously more difficult (18% success rate). Contrary to our expectations, given that Alloy has special syntax for that purpose, challenges that required restricting the multiplicity of relations (F2) were only slightly easier than average (52%). As expected, the need to use nested quantifiers (F3) increases the difficulty of challenges (33% success rate). Concerning closures, usage of a closure operator over a relation (F5) was not very problematic (41% success rate), but challenges that required applying a closure operator to a relational expression (F6) were the most difficult to solve (8% success rate). We had some anecdotal evidence that closures were difficult for students, but this discrepancy between the two cases was rather surprising, meaning that special attention should be given to the later case in lectures. Other problematic concepts were the manipulation of ternary relations (F4) (19% success rate), and usage of the standard `ordering` module (F7) (16% success rate), both frequently used in Alloy specifications. The first result is aligned with our anecdotal evidence, and we already had special care with higher arity relations in lectures. The second is a bit more surprising, meaning that, likewise to closures of relational expressions, we should invest more lecture time in explaining how to use this module. Concerning Electrum, students seem to understand well the usage of a single temporal operator (F8) (58% success rate), but, as expected and likewise quantifiers, specifications requiring nesting of several temporal operators (F9) were more difficult (32% success rate).

We also collected statistics about typical errors and warnings, with Tables 2 and 3 presenting the 10 most commonly found error and warning messages, respectively. Concerning errors, as expected, the most frequent are basic parsing errors (corresponding to messages 1, 2, and 8, and including, for example, parenthesis problems or misspelled identifiers), totaling around 44% of the errors. Of the remaining, the most frequent are incorrectly applying logic operators to relational expressions and vice-versa (messages 3, 5, and 7), in total 28% of

Table 2. Most common error messages.

	Message	#
1	There are ... possible tokens that can appear here.	747
2	The name ... cannot be found.	444
3	This must be a formula expression.	432
4	**in** can be used only between 2 expressions of the same arity.	277
5	This must be a set or relation.	277
6	This cannot be a legal relational join.	220
7	This expression failed to be typechecked.	117
8	The "**all** x" construct is no longer supported.	85
9	~ can be used only with a binary relation.	58
10	This must be a unary set.	50

Table 3. Most common warning messages.

	Message	#
1	The join operation here always yields an empty set.	213
2	Subset operator is redundant, because the left & right subexpressions are always disjoint.	123
3	This variable is unused.	121
4	^ is redundant since its domain and range are disjoint.	25
5	= is redundant, because the left & right expressions always have the same value.	11
6	<: is irrelevant because the result is always empty.	10
7	& is irrelevant because the two subexpressions are always disjoint.	8
8	= is redundant, because the left & right expressions are always disjoint.	8
9	The value of this expression does not contribute to the value of the parent.	6
10	Subset operator is redundant, because the right subexpression is always empty.	3

the errors, and simple typing errors related to arity (messages 4, 6, 9, and 10), in total 26% of the errors. The reader unacquainted with Alloy could find the frequency of the former rather surprising, but this is a rather frequent error due to the syntactic similarity between some logical and relational operators (for example, **not** for negation vs. **no** for emptiness check or && for conjunction vs. & for intersection). Fortunately, Alloy has alternative syntax for many logic operators (for example, **and** for conjunction) and maybe instructors should recommend using that alternative instead. Concerning warnings, all but the third most common message (unused variables, 23% of the total warnings) are warnings about potentially irrelevant expressions – formulas that are trivially true or false or expressions that always denote an empty set – a testimony to the usefulness of Alloy's sophisticated type system [5].

4 Concluding Remarks and Future Work

We briefly presented Alloy4Fun, a web application for online editing and sharing of Alloy models and instances, that also allows the automated assessment of simple specification challenges. Its main intended use is in an educational context, and our preliminary evaluation in a graduate formal methods course provided evidence that students found the automated assessment feature useful for learning Alloy and Electrum (and the sharing feature less so). We also collected evidence that some features of the Alloy language are particularly problematic for students, and should be addressed with particular care by tutors.

We intend to continue using Alloy4Fun in our formal methods courses in the upcoming years, collecting more data to support more detailed and informed analyses about the language usage. Concerning the application itself, we intend to develop tools to simplify the mining of useful data from the derivation trees, possibly to be run server-side at the click of a button (with results visualized in the browser), to enable the timely identification of learning breakdowns. We also intend to incorporate in Alloy4Fun an alternative instance visualizer more amenable for dynamic systems [4].

Acknowledgements. We would like to thank Daniel Jackson for the helpful comments and suggestions about the design of Alloy4Fun. This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020. The third and forth authors were 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. The second author was also supported by the FCT sabbatical grant with reference SFRH/BSAB/143106/2018.

References

1. Ball, T., de Halleux, P., Swamy, N., Leijen, D.: Increasing human-tool interaction via the web. In: Proceedings of the 11th ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 49–52. ACM (2013)
2. Boyatt, R., Sinclair, J.: Experiences of teaching a lightweight formal method. In: Proceedings of the 1st Workshop on Formal Methods in Computer Science Education, pp. 71–80 (2008)
3. Brunel, J., Chemouil, D., Cunha, A., Macedo, N.: Simulation under arbitrary temporal logic constraints. In: Proceedings of the 5th Workshop on Formal Integrated Development Environment, EPTCS, vol. 310, pp. 63–69 (2019)
4. Couto, R., Campos, J.C., Macedo, N., Cunha, A.: Improving the visualization of Alloy instances. In: Proceedings 4th Workshop on Formal Integrated Development Environment, EPTCS, vol. 284, pp. 37–52 (2018)
5. Edwards, J., Jackson, D., Torlak, E.: A type system for object models. In: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 189–199. ACM (2004)
6. Jackson, D.: Software Abstractions: Logic, Language, and Analysis, 2nd edn. The MIT Press, Cambridge (2012)
7. Macedo, N., Brunel, J., Chemouil, D., Cunha, A., Kuperberg, D.: Lightweight specification and analysis of dynamic systems with rich configurations. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 373–383. ACM (2016)
8. Mangaroska, K., Giannakos, M.N.: Learning analytics for learning design: a systematic literature review of analytics-driven design to enhance learning. IEEE Trans. Learn. Technol. **12**(4), 516–534 (2019)
9. Pereira, J.: A web-based social environment for Alloy. Master's thesis, Universidade do Minho, Escola de Engenharia (2016)

10. Sioson, A.A.: Experiences on the use of an automatic C++ solution grader system. In: Proceedings of the 4th International Conference on Information, Intelligence, Systems and Applications, pp. 1–6. IEEE (2013)
11. Tillmann, N., de Halleux, J.: Pex–white box test generation for.NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79124-9_10
12. Tillmann, N., de Halleux, J., Xie, T., Bishop, J.: Pex4Fun: a web-based environment for educational gaming via automated test generation. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, pp. 730–733. IEEE (2013)