Experiences on Teaching Alloy with an Automated Assessment Platform[⋆]

Nuno Macedo[b,a,∗], Alcino Cunha[c,a], José Pereira[c], Renato Carvalho[c,a], Ricardo Silva[c], Ana C. R. Paiva[b,a], Miguel Sozinho Ramalho[b,a], Daniel Silva[b]

[a] *INESC TEC, Porto, Portugal*
[b] *Faculty of Engineering of the University of Porto, Porto, Portugal*
[c] *University of Minho, Braga, 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. A data analysis library is also provided to support this process. Alloy4Fun has been used in formal methods graduate courses for 3 years and for the latest editions 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

[⋆]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.

[∗]Corresponding author

Email address: `nuno.m.macedo@inesctec.pt` (Nuno Macedo)

1. Introduction

Alloy [1] 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 of this work 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 property 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 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 [2] – 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, a process that becomes cumbersome in large classes with close interaction with the tutors. 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

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

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, with additional support for simple specification challenges in the form of duels where students attempt to discover a secret specified by the tutors. The deployment of such an 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 recording of every interaction with the (anonymous) user, information that is made available to the creator of the challenges for subsequent analysis. A data analysis library, with support for user-defined metric queries, is also provided to ease such analyses. Over the last three years, Alloy4Fun has been used in 5 editions of graduate courses on formal methods, 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. It starts with an overview of (and rationale for) the current features of Alloy4Fun in Section 2. Section 3 details how Alloy and Electrum were used in the formal design of Alloy4Fun, namely presenting a formalization of the internal data model of Alloy4Fun, essential to understand and develop mining procedures for the data available to challenge creators. The Alloy language is also presented in the process. Section 4 presents how challenges can be created using Alloy4Fun (Section 4.1), as well as the data analysis library that can

[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.

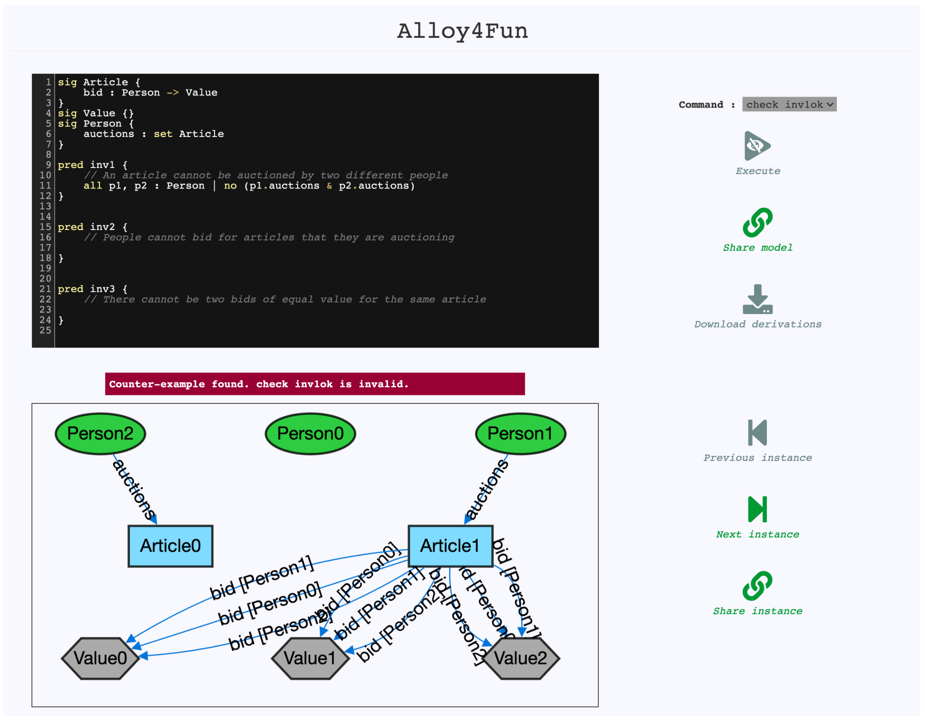
be used by challenge creators to gather insights about submitted resolutions (Section 4.2). Section 5 reports on the deployment of the platform in a formal methods graduate course (Section 5.1), results regarding usage and adoption of the platform (Section 5.2), and how to use the data analysis library for a more thorough analysis of specific exercises to help identify possible learning breakdowns (Section 5.3). It also shows how the collected data can be used to gather more general insights on Alloy usage patterns and learning pitfalls (Section 5.4). Finally, Section 6 concludes the paper and presents some ideas for future work.

This paper extends a conference paper [3] by *i*) formalizing the design of Alloy4Fun and more thoroughly presenting Alloy language (Section 3), so that the features of Alloy4Fun can be better appreciated; *ii*) presenting the novel library for collecting user-defined metrics (Section 4.2); *iii*) enhancing the lessons learned in Sections 5.2 and 5.4 with an additional year of experience and data; and *iv*) showing how the data analysis library can be used to thoroughly examine the results of specific challenges and identify learning breakdowns (Section 5.3).

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.

As an example, the Alloy4Fun screen capture shown in Fig. 1 shows a model of an online auctioning platform (`Auction`) platform, an example that was used as an exercise in classes. A distinctive feature of Alloy is that analysis commands can also be declared in a model: *run* commands verify the satisfiability of the declared facts and get witness scenarios, and *check* commands verify the validity of an assertion (assuming the facts to hold) and, if that is not the case, get counter-examples. In Alloy4Fun the topmost right button allows analysis commands to be executed: the command to be executed can be selected in



```
sig Article {
    bid : Person -> Value
}
sig Value {}
sig Person {
    auctions : set Article
}

pred inv1 {
    // An article cannot be auctioned by two different people
    all p1, p2 : Person | no (p1.auctions & p2.auctions)
}


pred inv2 {
    // People cannot bid for articles that they are auctioning

}


pred inv3 {
    // There cannot be two bids of equal value for the same article

}

```

```
Counter-example found. check inv1ok is invalid.
```

Figure 1: A failed attempt to solve a challenge in the `Auction` exercise.

the drop-down immediately above. If witnesses or counter-examples 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 solver used to perform a given analysis, not being able to display an unsatisfiable core, and the lack of support for Alloy's module system (except for the standard modules distributed with Alloy, which can be imported). In the specific case of Electrum, Alloy4Fun lacks the more sophisticated trace exploration options available in the Electrum Analyzer [4].

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 `inv1ok`, described in Section 4.1, 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 exactly 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 nature, in particular those used to procedural programming [5]. 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

[3]`http://alloy4fun.inesctec.pt/KhyrJNyr97soxNrXJ`

and online judges have become widely popular for learning how to program [6], 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, strips the original model of secrets and only shows public paragraphs (although secret commands can still be executed, their names being public). Whenever a user calls an analysis command, the server searches the original model for secrets and merges them with the submitted model. 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 5.1 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 [7]. 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 [8] (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 [9]. Pex [10], 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, Rise4Fun has some limitations in the interaction with the output of the tools, 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 a branch represents a point where a user generated a new *permalink* for a model which was subsequently accessed multiple times. The formalization of the Alloy4Fun data model in Section 3 provides further insights on this structure. 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. A data analysis library has also been developed to support the extraction of metrics from the derivation tree of a challenge given user-defined metrics, which is described in Section 4.2.

Implementation. Alloy4Fun was developed [11] 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. Formalizing Alloy4Fun with Alloy

Alloy itself (and Electrum) were used during the design phase of Alloy4Fun, to formalize and explore different design alternatives, and check that the operations made available via the web interface satisfy some of the expected invariants[4]. In this section we present this specification, including the Alloy4Fun internal data model, namely the derivation trees, thus providing a better insight on the data structures that can be mined by the users using the library presented in Section 4.2. A secondary goal of this presentation is to provide a more thorough overview of the Alloy language, and the Electrum extension, so that the results from Section 5 can be better appreciated.

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. Signature and field declarations can have *multiplicities* attached to impose cardinality constraints.

[4]The design of Alloy4Fun actually followed a feature-oriented design approach, using an Alloy extension to model and analyze multiple variants [12]. In this section we present only the specification of the variant with the features that ended up being implemented.

```
sig Timestamp, Id, Object {}

sig Paragraph {}
sig Command extends Paragraph {}
sig Secret in Paragraph {}

abstract sig Entry {
  id : one Id }

abstract sig Model extends Entry {
  time   : one Timestamp,
  spec   : set Paragraph,
  parent : lone Id,
  root   : one Id }

sig Share extends Model {
  theme : one Object }

abstract sig Result {}
one sig Sat, Unsat, Error extends Result {}

sig Execution extends Model {
  command : one Command,
  result  : one Result }
sig HasWarnings in Execution {}

sig Instance extends Entry {
  model : one Id,
  graph : one Object }

abstract sig Link extends Entry {
  model : one Id }
sig Private, Public extends Link {}

pred inv {
  id in Entry lone → Id                                              // uniqueness of keys
  all i: Model.(parent+root) | some id.i&Model                    // referential integrity
  all i: Instance.model+Link.model | some id.i&Share              // referential integrity
  no iden&^(parent.~id)                                          // no circular derivations
  root.~id in *(parent.~id)                                     // root is ancestor or self
  all m: Model, i: ^(~id.parent).(m.root)&(m.id).*(~id.parent) |
    some (id.i).spec&Secret implies i = m.root                 // no secrets after the root
  all m: Execution | m.command in (m+m.root.~id).spec                        // valid command
  all m: Share | some Link<:model.(m.id)                       // all shares have some link
  all m: Instance | (m.model.~id).parent.~id.result = Sat // instances are from sat models
  Private.model in Public.model                         // public links are always generated
  all l: Private | some (l.model).~id.spec&Secret           // private links have secrets
}
```

Figure 2: Alloy structural model of Alloy4Fun

In the Alloy4Fun model presented in Fig. 2, abstract signature `Entry` represents all elements stored in its database, which have a single identifier assigned (field `id` with multiplicity **`one`**), and that are further divided as

- `Model` entries, that result from the `Execution` or `Share` of a model;

- `Instance` entries, that store shared model instances;
- `Link` entries, storing the *permalinks* of shared models and instances, either `Public` or `Private` (if secrets are defined).

Each `Model` has a timestamp `time`, the respective Alloy model `spec` as a set of `Paragraph`s, possibly a `parent` model (multiplicity **`lone`**) and a `root` of the derivation. Paragraphs are left uninterpreted except for `Command` elements, which are relevant for the design of Alloy4Fun. Any paragraph can also be marked as secret, identified through the sub-signature `Secret`. Relations `parent` and `root` are used to store the derivation tree of a model. Models are stored in the database both when *permalinks* are created for sharing or when some command is executed. `Share` models also store the respective `theme`, while `Execution` models register the executed `command` and the `result` of the execution, a value from signature `Result` (`Sat`, `Unsat` or `Error`); executions may also have reported warnings independently of the result, identified by sub-signature `HasWarnings`. `Instance` entries point to the identifier of the respective `model` and the `graph` representation of the instance so that the positioning of elements is preserved. Both `theme` and `graph` point to uninterpreted `Object` atoms whose content is not relevant for this analysis. Lastly, `Link` elements point to the identifier of the `model` they refer to.

Once the general structure of a model is declared, additional constraints can be defined in *Relational Logic* (RL), an extension of *First-Order Logic* (FOL) with operators that can be used to combine relations (*aka* predicates in FOL). Relational expressions combine the declared relations (and some constants like the universe of all atoms **`univ`**, or the binary identity relation **`iden`**) using typical set-theoretical (such as union `+`, intersection `&` or Cartesian product $\rightarrow$) and relational (such as the converse `~` and composition `.`) operations. The transitive `^` and reflexive-transitive $*$ closure of binary expressions can also be calculated. These operations allow the construction of complex expressions following a simple navigational style resembling that of object-oriented languages. Atomic relational formulas are build either from inclusion tests with **`in`**, or simple multiplicity tests. These are then composed through Boolean connectives (such as conjunction **`and`**, disjunction **`or`** or implication **`implies`**) or FOL quantifications (such as the universal **`all`** and existential **`some`** quantifiers).

Back to our example, a predicate `inv` has been declared which defines the structural invariants of the Alloy4Fun model, namely:

Referential integrity (ll. 36–38) Each identifier is unique, as stated through a simple multiplicity test over `id` that renders it injective. Moreover, identifier references must point to appropriate entries. Note that an expression like `Model.parent` retrieves all identifiers related to any `Model` through `parent`, and that for an identifier `i`, `id.i` retrieves the associated entries (in Alloy everything is a relation, including quantified variables which are singleton sets). Thus, a multiplicity test **`some`** `id.i` forces `i` to refer to some `Entry`, while **`some`** `id.i&Model` forces it to refer to some `Model` entry specifically.

Derivation tree (ll. 39–42) The parenthood relation must form a tree. Since field `parent` has type `Model` $\rightarrow$ `Id` and field `id` has type `Entry` $\rightarrow$ `Id`, expression `parent.~id` has type `Model` $\rightarrow$ `Entry` and denotes the parenthood relation at the level of entries. Thus, for a particular model `m`, `m.^(parent.~id)` retrieves all ancestor entries of `m` using the transitive closure, and **`no iden`**`&^(parent.~id)` guarantees that the `parent` relation is not circular. The root of a derivation is the ancestor model from which the secrets are inherited are merged during analysis, and can be used to determine which entries are attempts to solve a shared challenge (which, as will be described in Section 4.1, use secret commands for checking the correctness). The root must be one of the ancestors, and, since models accessed through public links are stored without the root secrets, between the root and the current model no other entry may contain secrets (otherwise it would be the root of a new derivation tree). A consequence of this is that, if a user following a public link to a challenge introduces secrets of its own in the model (for example, to create a new challenge),

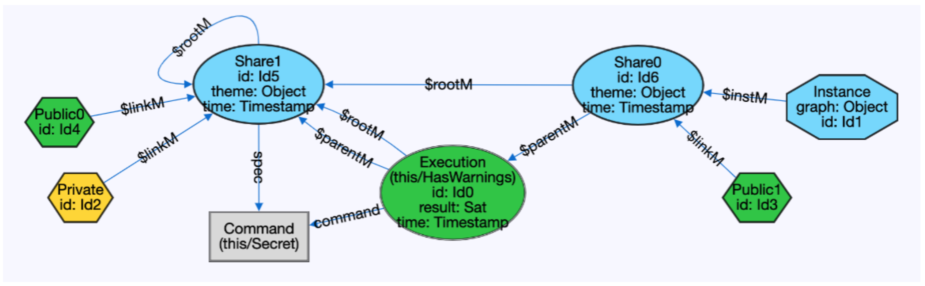


Figure 3: An instance of the Alloy4Fun structural model in Alloy4Fun.

the connection to the original challenge is removed, and the new entry is no longer part of the respective derivation tree. However, this does not force the root to be the closest ancestor with secrets: in particular, if the secrets of a model are deleted, the new derived model will become the root of its own derivation tree, since the previously existing secrets must not be inherited.

Execution data (l. 42) The command assigned to an execution entry must be among the commands defined in the respective Alloy specification or inherited from the root model.

Share data (l. 44) All share entries must have links referring to them.

Instance data (l. 45) Instance entries must refer to satisfiable executions.

Link data (ll. 46–47) If there is a private link to a model, there must also exist a public one (but not the other way around, in case there are no secrets). Moreover, private links must refer to models with secrets.

Commands can then be defined to animate the model and inspect possible instances. Such 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 (which by default is 3 for all elements). For instance, a command `run { inv }` will generate instances that are consistent according to `inv` with at most 3 atoms per signature, but more interesting scenarios can be generated with additional restrictions, such as

```
run { inv and some Execution and some Instance } for 2 but 7 Entry, 7 Id
```

forcing some entries to exist, but to a maximum of 7. Figure 3 depicts one such possible instance in Alloy4Fun[5] with 3 model entries: a share with a secret command, an execution of that shared model with a satisfiable result, and an instance share of that result. A useful feature of Alloy is that auxiliary functions can be defined and depicted in instances to ease visualization, even without being considered during analysis. For instance, functions were defined to depict the derivation tree at directly the entry level, such as the following for the parenthood relation

```
fun parentM : Model → Model {
  parent.~id }
```

In Alloy every signature and field is immutable, which allows for the validation of structural properties. With the Electrum extension they can be declared as mutable, and formulas can also be specified with *Linear Temporal Logic* (LTL) operators (such as future **`always`** and **`eventually`** or past **`historically`** and **`once`**), which allow for the specification and verification of dynamic models. A typical analysis on such models is to check that the specified system operations preserve some desired invariants of the data. Given the complexity of Alloy4Fun operations and data model, we opted to conduct such an analysis at the design phase. A typical way to do such analysis with Electrum is to *i*) define the mutable elements and adapt the desired invariant accordingly; *ii*) specify the system operations using predicates that constrain the relation between the respective pre- and post-states; and *iii*) check whether traces built from the application of operations always preserve the invariant.

Figure 4 depicts an excerpt of the behavioural model of Alloy4Fun for an interactive user session. Here, singleton signature `A4F` represents the web ap-

[5]`http://alloy4fun.inesctec.pt/Z76PLPdYPBRxtS8er`

```
one sig A4F {
  var db       : set Entry,
  var time     : one Timestamp,
  var editor   : set Paragraph,
  var current  : one Id,
  followed     : lone Id } {
  some current.~id&db&Model
  some followed implies some followed.~id&db&Link }

pred execute {
  some i: Id, m: Execution {
    m not in A4F.db
    m.id = i
    m.time = A4F.time
    m.spec = A4F.editor
    m.parent = A4F.current
    m.root = ((some A4F.editor&Secret or A4F.followed.~id in Private)
      implies i else A4F.current.~id.root)

    A4F.db' = A4F.db + m
    A4F.current' = i

    A4F.time' = A4F.time
    A4F.editor' = A4F.editor } }

pred shareModel { ... }

pred shareInstance { ... }

pred user {
  current' = current
  db' = db }

pred init {
  inv                                                       // consistent database
  A4F.followed.~id.(Link<:model) = A4F.current              // id of the followed link
  A4F.editor = (A4F.followed.~id in Private implies    // show secrets if private link
    A4F.followed.~id.spec else A4F.followed.~id.spec - Secret)
}

fact trace {
  init
  always (user or execute or shareModel or shareInstance) }

check { always inv } for 3 but 5 Entry, 5 Id
```

Figure 4: Fragment of the Alloy dynamic model of Alloy4Fun

plication whose state is determinable by some mutable fields (marked as **`var`**). The most relevant field here is `db`, the set of `Entry` elements in the database at each moment. The current state of the `editor`, the identifier of the `current` entry and the current `time` are also stored as mutable fields. The link `followed` to start the session, if any, is also stored as a static field since it will not change during the session. The introduction of a mutable database requires adapting `inv` from Fig. 2 to this context: for instance, to preserve relational integrity at each state relations must point to `Entry` elements that exist in `db` in that state, which could be specified as:

```
all i: A4F.db&Model.(parent+root) | some id.i&A4F.db&Model
```

Next, four operations that make the system evolve were encoded: the execution of models, the sharing of models and instances, and the user interacting with the editor and visualizer. The first one is presented in Fig. 4 (ll. 10–24). Recall that primed expressions, such as `db'`, refer to the state of the expression in the succeeding state. Essentially, a new execution `m` is added to `db` with the appropriate data. The `root`, in particular, is reset if the current code has secrets or the session is editing a private link. Other than `db` and the `current` identifier, the state of the system does not evolve. That evolution is encoded by predicate `user` (ll. 30–32) which represents user interactions with the editor, which does not update the database.

Lastly, we want to check whether such operations effectively guarantee the preservation of the invariant. One way to do this is to force the system to start from a consistent state (determined by predicate `init` here) and evolve according to the declared operations, as in fact `trace` (ll. 41–43). A check command can then check whether the invariant is globally true at all traces (within a certain scope), which is true for the presented model. The modelling of the operations in a way that preserves the invariants actually drew our attention to some subtle corner cases prior to implementation (particularly, when should derivation trees be broken as secrets are added/removed).

4. Writing and mining challenges

As already stated, one of the main goals of Alloy4Fun is to be used in the educational domain, both by allowing the definition of challenges with auto-grading and the extraction of information regarding the students progress. This section describes how these two tasks can be performed by an Alloy instructor.

4.1. How to create challenges

The model secrets supported by Alloy4Fun can be used to create simple specification challenges in the form of duels, where the user (typically a 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). Note however, that although such exercises are 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 declare signatures and fields.

The model shown in Fig. 1 was obtained precisely by accessing the public *permalink* of the `Auction` exercise, which contains 3 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., `inv1`, whose desired property is that an article can be auctioned by at most one person), the student can check whether it is a valid solution (e.g., by running secret command `inv1ok`, for the case of `inv1`), 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 `inv1` is still incorrect, since the student attempt will fail when `p1=p2`). Figure 5 shows the secret implementation of challenge `inv1`: predicate `inv1oracle` specifies a correct solution for the challenge and command `inv1ok` checks the equivalence between both.

In more complex examples assertions may need to be further restricted, as in the `CV` exercise, which models an online *Curriculum Vitae* platform, for which the secret of `inv2` is presented in Fig. 6. Here several desired (and natural) properties of the model are solved in different challenges, but if they were solved independently the student would get many counter-examples where it would not be clear why their specification failed, since they would be "polluted" with dis-

```
pred inv1 {
  // An article cannot be auctioned by two different people

}
//SECRET
pred inv1oracle {
  all a : Article | lone auctions.a }
//SECRET
check inv1ok {
  inv1oracle iff inv1 }
```

Figure 5: The secret for the challenge `inv1` of `Auction` from Fig. 1.

```
//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 inv2oracle { all u : User | u.profile.source in Institution+u }
//SECRET
check inv2ok {
  ((some ShouldBeRejected iff (inv2 and not inv2oracle)) and
   inv1oracle and inv3oracle and inv4oracle) implies
     (inv2 iff inv2oracle) }
```

Figure 6: The secret for the challenge `inv2` of `CV`.

tracting problems corresponding to failures of other properties. Thus, we opted to check `inv2` assuming that the remaining properties hold. This conditional check is the reason to include `inv1oracle`, `inv3orable`, and `inv4oracle` as assumptions in the equivalence check `inv2ok`. Notice that in the preamble to this particular 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 first editions of the courses where Alloy4Fun was used 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, more complex challenges include two special atoms in the counter-example instance that signal whether an instance should have been rejected or accepted by a correct specification, meaning their solution is under- or over-specified, respectively. As seen in Fig. 6 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).

4.2. How to mine challenge statistics

The "owner" of an exercise can download the respective derivation tree through the Alloy4Fun interface as a JSON file. However, since analyzing such data can be cumbersome, we developed a very simple data analysis library to ease this process and that was used to mine the data presented in this paper. The library supports user-defined metric queries and it is written in Java, since it is the language in which the Alloy Analyzer is written and we want to exploit its parser and AST in order to collect metrics about the submitted models. It generates outputs both in a simple textual form and as a very basic HTML with data charts. As an example, the output for `Auction` is publicly available[6]. All data presented in the following sections – except for that regarding the conceptual classification of exercises from Section 5.4, which requires manual assignment – has been calculated using this library. A catalog of basic metrics is also provided in the repository.

To ease the discovery and automatic execution of metric methods, we rely on Java annotations and reflection. Each metric method that is to be run should be annotated with `@MetricMethod` and provided with the group by operation (the typical count, sum, max, min, or average) and optionally some meta-data, such as the rule name and a longer description). The method's parameters must be annotated with which artifacts of the derivation tree of the exercise it is expecting to receive. The supported parameters are summarized in Table 1. Most of them correspond directly to the data model presented in Section 3. For

[6]`https://haslab.github.io/Alloy4Fun/SGfC5MjAijZfMQZPs.html`

annotation	type	inputs
`@ForAllSessions`	`Model`	all derivation branches of the root
`@ForAllModels`	`Model`	all model entries
`@ForAllExecutions`	`Execution`	all execution entries
`@ForAllShares`	`Share`	all share entries
`@ForAllLinks`	`Link`	all link entries
`@ForAllInstances`	`Instance`	all instance entries
`@ForAllSolutions`	`Solution`	all solutions of sat execution entries
`@ForAllErrors`	`Err`	all thrown error messages of model entries
`@ForAllWarnings`	`ErrorWarning`	all thrown warning messages of model entries
`@ForAllChallenges`	`String`	all challenges of the root

Table 1: Annotations for metric parameters

instance, a metric method with a parameter annotated with `@ForAllExecutions` will be called for all execution entries of the derivation tree of the exercise under analysis. For more detailed analysis, the re-execution of all entries can be requested to the library. In this case the byproducts of these procedures – the AST of the Alloy code of model entries, the full Alloy solution instances of satisfiable executions, and thrown error and warning messages – can also also be analyzed independently by metric methods. They also become accessible from the respective model entries. Lastly, the `@ForAllChallenges` allows iterating over the commands defined in the root model, useful to organize the reports according to the attempted challenge. Metric methods can have multiple annotated parameters, in which case they are called with the Cartesian product of all artifacts. Each method must return an array of results, which are subsequently aggregated and counted by the library during analysis.

As an example, consider the following method, which simply counts correct executions by testing their result:

```
@MetricMethod(groupby = COUNT, rule = "Correct␣executions")
public static Object[] correctExecs(@ForAllExecutions Execution exe) {
  if (!MetricRunner.isChallenge(exe.cmdName()) ||
        exe.result() != UNSAT) return null;
  else return new Object[] { exe.result() };
```

`MetricRunner` is the main class of the metric library, and provides several helper methods to support the analysis of challenges designed as presented in the previous section. Here, method *`isChallenge(String)`* tests whether the executed command was defined in the root model. The method tests if the execution was for a root command – ignoring the execution of commands defined by users – and if the result was unsatisfiable, which represents a correct resolution. Since this method either returns **`null`** or `UNSAT`, it is interpreted as a simple counter in the reports.

The next example acts instead at the session level, reporting the number of sessions in which a certain number of challenges have been solved correctly.

```
@MetricMethod(groupby = COUNT, rule = "Sessions␣by␣#␣of␣solved␣challenges")
public static Object[] sessionChallenge(@ForAllSessions Model der) {
  Set<String> unsats = sessionMetrics(der).unsatCommands();
  unsats.retainAll(MetricRunner.getChallenges());
  return new Object[] { unsats.size() }; }
```

Method *`sessionMetrics`*`(Model` is a helper that recursively calculates (and caches) statistics from a derivation tree, including all unsatisfiable commands that have been executed in children entries, retrieved by `unsatCommands()`. After filtering only for root challenges, it reports how many distinct challenges were solved in the session. After aggregation the result is the frequency of the number of solved challenges per session – including how many sessions solved all challenges. In the reports, this is visualized as a bar chart.

The next example reports on the execution results for each challenge of the exercise.

```
@MetricMethod(groupby = COUNT, rule = "Execution␣results␣by␣challenge")
public static Object[] execsChallenge(@ForAllExecutions Execution exe) {
  if (!MetricRunner.isChallenge(exe.cmdName())) return null;
  else return new Object[] { exe.cmdName(), exe.result() }; }
```

This metric reports a pair of values containing the attempted challenge and the execution result. After aggregation, we get the number of satisfiable, unsatisfiable and errored executions for each challenge. Reports depict this as a stacked bar chart.

Lastly we show a more complex metric that explores the AST of the submitted Alloy model, a metric reporting the execution results in reference to the number of quantified variables.

```
@MetricMethod(groupby = COUNT, rule = "Number␣of␣quantified␣variables")
public static Object[] numQuantVars(@ForAllExecutions Execution exe) {
  if (!MetricRunner.isChallenge(exe.cmdName())) return null;
    AggregateVisitor<Integer> vis =
      new AggregateVisitor<Integer>(Integer::sum, 0, MetricRunner.challengePreds()) {
      @Override
      public Integer visit(ExprQt exp) {
        return exp.count() + super.visit(exp); } };
  return new Object[] { exe.cmdName(), exe.formula().accept(vis), exe.result() }; }
```

The method relies on an aggregation visitor `AggregateVisitor` that, given a combining binary function and a default value for AST leaves, traverses the whole AST of an Alloy model. In this case, the visit method for quantifier expressions is overridden to count the number of quantified variables (method `count()`). If the full AST was traversed directly, however, the metric would not reflect the student's resolution, since the model contains the code from the root model – such as possible facts, or the oracle predicates. The metric library provides a helper method *`challengePreds()`* for this purpose, which retrieves the empty predicates from the root model, which we assume are those to be filled by students. The aggregator then only considers data after visiting one of those predicates. After analysis, the result is the number of correct, incorrect and errored executions per quantifier nested level, for each challenge. Reports split this into into several bar charts for visualization, one per challenge.

5. 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[7]. These first experiences also allowed us to identify which classes of exercises are better suited to be deployed in Alloy4Fun, as well as how the visualization features can be explored to provide more intuitive feedback to the students.

From this process resulted the first official release of Alloy4Fun, which at the time of writing registers around 58000 entries. It has since been used in graduate courses in the UM (2019/20 and 2020/21 academic years) and in the UP (2020/21), on an Alloy/Electrum tutorial at the World Congress on Formal Methods[8] (with a refined set of specification exercises with challenges), and in a series with more advanced specification challenges, entitled *GuessTheSpec*, that is being published in the official Alloy discussion forum[9]. The remainder of this section mainly reports on the usage of the platform by the students during these latest two editions of the UM graduate course. To illustrate the usefulness of the collected data for the Alloy community in general, it also presents preliminary results regarding common mistakes and difficulties when learning Alloy, in this case using all the data collected at both universities.

5.1. Alloy4Fun exercises

The challenges used in this course were based on 7 different problems:

- `Trash`, a model of a file system trash bin.

[7]Note that Alloy4Fun is only used for self-study and not for automated student grading.
[8]`http://haslab.github.io/TRUST/tutorial.html`
[9]`https://alloytools.discourse.group`

- `Classroom`, a model a classroom management system.
- `Graph`, a specification of several standard properties of unlabeled graphs.
- `LTS`, a specification of several standard properties of labeled transition systems.
- `Production`, a model of an automated production line in a factory.
- `CV`, a model of an online *Curriculum Vitae* platform.
- `Train`, a model of a station with moving trains (introduced only in 2020/21).

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 2. The table lists the *permalink* and total number of challenges of each exercise.

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 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 10 exercises described in Table 2.

5.2. 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

Table 2: Alloy4Fun exercises shared in the 2019/20 and 2020/21 editions of the UM course.

Id	**Exercise**	***Permalink***	**Chall.**
1	`Trash FOL`	`zA2MMSGy6iW8Mihep`	10
2	`Classroom FOL`	`Pdvipvrpr5hg7JKbs`	15
3	`Trash RL`	`WJdLnDL78m7mM7W4J`	10
4	`Classroom RL`	`i5u2pjKJt6Bz227QT`	15
5	`Graphs`	`28fwdmjL79X4SQ9EP`	8
6	`LTS`	`gqS3qTTn4B62NYmJX`	7
7	`Production`	`PKy7chamCieZyCix5`	4
8	`CV`	`X72J6js9fA3CKYQWX`	4
9	`Trash LTL`	`irRLJn7qbQq3xMFGp`	20
10	`Train`	`HDSYav6cKZ6ygy5N9`	18

taught for 4 additional weeks. In each week, a 1h lecture was followed by a 2h 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 Section 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.

The 2020/21 edition was attended by 25 students and followed a very similar plan. Likewise the 2019/20 edition, Alloy was taught for 5 weeks and Electrum for 4 weeks, each week with a 1h lecture was followed by a 2h lab session. In this edition, Alloy4Fun was used in 6 lab sessions – 1 session focusing on the FOL exercises (`Trash FOL` and `Classroom FOL`), 3 sessions focusing on the RL exercises (`Trash RL`, `Classroom RL`, `Graphs`, `LTS`, `Production`, and `CV`), and 2 sessions focusing on the LTL exercises (`Trash LTL` and `Train`). However, unlike in the 2019/20 edition, students were encouraged to solve the shared exercises autonomously at home, with the lab sessions being used mainly to clarify doubts and to conduct more regular evaluations during the semester: the 3 main evaluation points (medium-size modeling project, individual exam, and supplementary exam) were complemented by 6 small evaluation challenges during the term, the first 4 done with Alloy4Fun (3 focusing on RL and 1 on LTL). Table 3 presents the 4 Alloy4Fun evaluation exercises[10]. Besides motivating the students to study and practice Alloy and Electrum more regularly during the term, these evaluation exercises also had the goal of helping the instructor evaluate the progress of the students regarding the expected learning outcomes and, in particular, identify possible learning breakdowns as early as possible. In Section 5.3 we will detail how the data analysis library presented in Section 4.2 can be used to help wit the latter task.

After these two editions of 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

[10]Unlike the practice exercises, most of these evaluation exercises are written in Portuguese.

Table 3: Alloy4Fun evaluation exercises shared in the 2020/21 edition of the UM course.

Id	**Exercise**	***Permalink***	**Chall.**
i	`RoomAllocation`	`MuqgwjouPRQ4PWHw5`	1
ii	`TrainStation`	`3vW9wNnPqNDC4cb4u`	3
iii	`Auction`	`34dtAqKXoYoRfbN4Q`	3
iv	`CreditCards`	`3N5BoPYnHf2cFrwgv`	3

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 of the 2019/20 edition and by 17 of the 25 students of the 2020/21 edition (in total, 30 out of 42), and, over the duration of both editions of the course, we collected almost 35000 interactions with the shared exercises (including evaluation exercises), most of them resulting from the execution of commands (checking the correctness of challenges) and a small portion from sharing of models. The dataset for both editions is freely available [13].

Concerning the first question, of the 30 students that answered the questionnaire, 25 (83.3%) said they used Alloy4Fun frequently outside classes, 4 (13.3%) only used it rarely, and 1 (3.3%) never used it[11]. Concerning the second question, all of the 29 (96.7%) students that used it outside classes answered that they used it to study for the exam. Of these, 22 (73.3%) 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. 7, seems to corroborate these answers. Chart 7a summarizes the attempts at solving the shared exercises over the 2019/20 semester, highlighting the classes where usage of Alloy4Fun was mandatory and the evaluation points (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*.

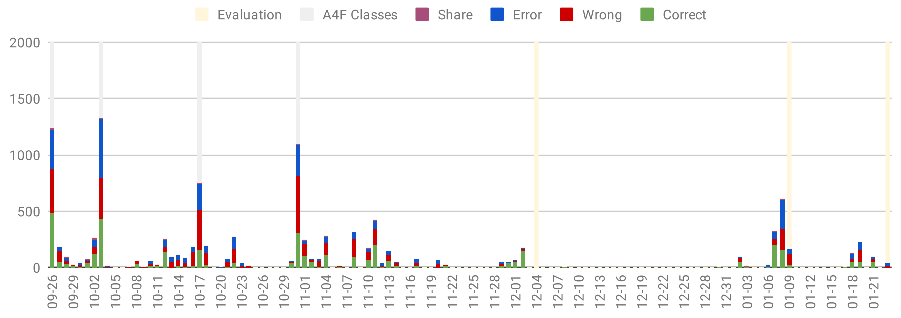
[11]Due to rounding, some percentage totals may not correspond with the sum of the separate figures.

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 exams. Part of the motivation for the evaluation challenges introduced in 2020/21 was to avoid these study peaks near only the main evaluations. Chart 7b summarizes the attempts at solving the shared exercises for that edition, highlighting the classes where Alloy4Fun was used, the 6 mini-evaluation challenges[12], the two exams late in the semester, and the project deadline, which coincides with the first exam). It is clear that the evaluation challenges have promoted more continuous, independent study (note that this data is only for the 10 shared challenges, and does not include the evaluation challenges). Likewise to the previous edition, students also relied on Alloy4Fun when studying for the exam.

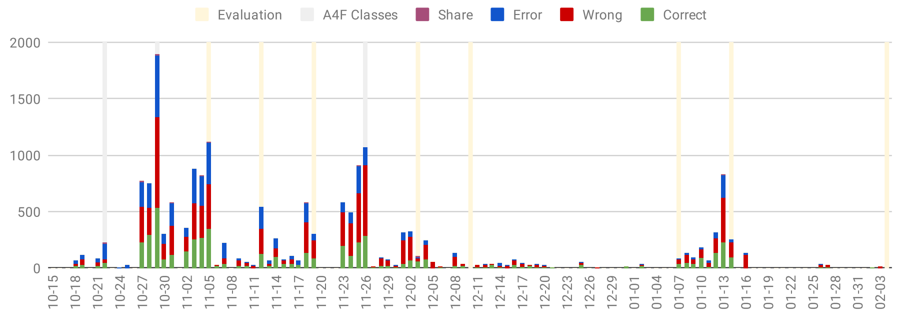
Figures 7c and 7d present statistics per exercise (below each exercise number we recall the number of challenges inside). Chart 7c presents the same execution information as Chart 7a and 7b (except shares), with the addition of the number of successful analyses (i.e., without error) that threw *warning* messages. 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. Considering exercises used in both editions[13] and averaging the executions per challenge and per student, we have a maximum of around 9 for exercise 6 and a minimum of around 4 for exercise 7, and an overall average of around 7 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

[12]Recall that the last two focused on using the Analyzer, thus an increase in Alloy4Fun activity near those was not expected.

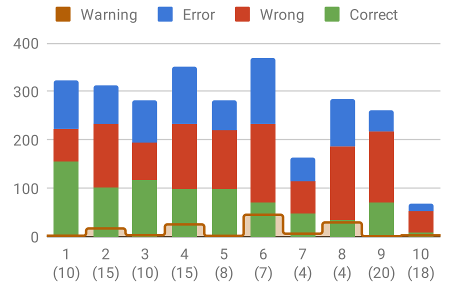
[13]`Train` was only introduced in the end of the 2020/21 semester, so it still counts fewer interactions.



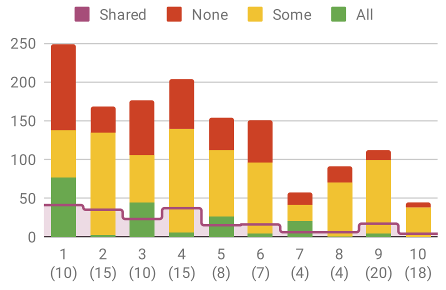
(a) Executions over the semester for the 2019/20 edition.



(b) Executions over the semester for the 2020/21 edition.



(c) Average challenge execution per exercise.



(d) Sessions per exercise.

Figure 7: Alloy4Fun usage statistics by 17 students over a semester for 9 exercises.

solve all exercises.

Chart 7d presents information regarding solving "sessions". Recall that a session is a branch in the derivation tree, typically recording the interaction of a student with Alloy4Fun while solving the challenges inside an exercise. For each exercise, we depict how many sessions 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 1407 sessions, with an average of 141 sessions per exercise (excluding evaluation exercises). 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` 76 sessions were recorded where all the challenges were solved, a strong indicator that most students should have solved it at least twice.

Concerning *permalinks*, 14 (46.7%) students mentioned that they generated them frequently to store their own solutions for later access, 8 (26.7%) did it rarely, and, somehow surprising, 8 (26.7%) never did it. Generating *permalinks* for the purpose of sharing with colleagues and tutors was even less common: only 9 (30.0%) students did it frequently, 9 (30.0%) rarely, and 12 (40.0%) never. Figure 7d 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. In total, models were shared 390 times (including the evaluation exercises). Surprisingly, the share instance feature has barely been used: there were only 4 generated *permalinks* for instances for these exercises. 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, 24 (80.0%) students mentioned that counter-examples were frequently useful to help find the correct answer, but of these 14 (46.7%) 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. The questionnaire also included a free filling suggestions question, and 2 students explicitly asked for mechanisms to suggest more clear counter-examples.

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 24 (80.0%) strongly agreeing.

5.3. In-depth analysis of learning outcomes

To exemplify how the data analysis library introduced in Section 4.2 can be used to assess the progress of the students regarding the expected learning outcomes, and in particular identify possible learning breakdowns, we will illustrate its application in the analysis of the results of the third evaluation exercise of the 2020/21 edition, identified as `Auction` in Table 3. The exercise shared with the students was written in Portuguese, but is equivalent to the English version depicted in Fig. 1. This exercise focused on RL and was shared in the week immediately after the 5 weeks that taught Alloy, which included 4 lab sessions of practice with Alloy4Fun. At this point the students were already expected to have a good knowledge about Alloy in general and RL in particular. The exercise included 3 challenges:

- The first was a very simple challenge that required students to enforce the binary relation `auctions` to be injective, a property that was several times specified in different challenges in the previous lab sessions (although with many different formulations in natural language).
- The second required imposing a restriction on both `auctions` and `bid`, the latter being a ternary relation, whose manipulation our study presented

in the next section showed to be potentially problematic.

- The third also required imposing a kind of injectivity restriction, but now on the ternary relation `bid`.

All the 25 students attempted to solve this exercise during the class it was shared: 17 students managed to solve all challenges, 6 solved two, and 2 solved none. This positive result was in line with the expected at that point in the semester, namely taking into account the number of previous lab sessions. By applying metric `execsChallenge` (presented in Section 4.2) we computed the success rate of each challenge (the ratio between correct and total number of attempts): the success rate for the first and third challenges was very similar (around 27%), with the success rate for the second a bit higher (around 38%). This result was a bit surprising, since at first glance the second property seemed to be the most difficult, namely requiring more logic and relational operators. To understand whether this result is due to some learning breakdown further analysis is needed.

Notice that the same property can be specified in many different styles with RL. On one extreme, it is possible to use a pure FOL style, where atomic formulas only check if specific tuples belong to a relation (*aka* predicate) and formulas typically end up using many quantifiers. On the other extreme, we can use the so-called *point-free* style, where properties are expressed without any quantifiers or logic connectives, by resorting just to inclusion checks between (many times complex) relational expressions defined by composition using the relational operators. In Alloy, the most natural style sits somewhere between these two extremes. A few quantifiers are used to frame the main entities that are subject to a restriction, from which we navigate using mainly the composition operator in order to compute sets of related entities that are relevant to the formula. This intermediate style is usually known as *navigational* [1]. For example, the injectivity property required in the first challenge could be specified in the pure FOL style as

```
all p,q: Person, a: Article | p→a in auctions and q→a in auctions implies p=q
```

and in the pure point-free style just as

```
auctions.~auctions in iden
```

None of these is ideal: while the former is too verbose, the latter is quite difficult to understand for non RL experts. Using the intermediate navigational style we could, for example, specify this constraint as

```
all p,q: Person | p!=q implies no (p.auctions & q.auctions)
```

Here we directly require that any two different people have disjoint sets of auctions. Or even better, using just one quantification we could almost directly transliterate the natural language requirement to logic, requiring that every article has at most one person that auctions it, resulting in

```
all a: Article | lone auctions.a
```

Alloy also has a special syntax to impose multiplicity constraints on the domain or range of a relation, that can also be used to easily solve this challenge, namely as

```
auctions in Person lone → Article
```

These two formulations seem the easier way to solve the proposed challenge, provided students understood that relations can be navigated backwards or knew the special multiplicity arrow syntax. Both concepts were taught in the classes, but maybe not well explained or practiced enough, hence the poor results in the first and third challenges.

It is relatively simple to mine which style was used by the students by counting the number of quantified variables in the submitted solution. This can be done with metric `numQuantVars` presented in Section 4.2. Among the almost 300 registered attempts to solve the first challenge, 19% used the pure point-free style or multiplicity check without any quantifiers, 42% used one quantification, 22% used two, 15% used three, and a residual 2% used four. The number of attempts with the expected single quantifier was surprisingly low, an indication that indeed many students still did not understood the usefulness of backwards navigation in specifying this kind of properties. To check whether students used the special multiplicity syntax we can easily define the following metric using the library presented in Section 4.2.

```
@MetricMethod(groupby = COUNT, rule = "Number_of_multiplicity_checks")
public static Object[] numMultArrows(@ForAllExecutions Execution exe) {
  if (!MetricRunner.isChallenge(exe.cmdName()) return null;
  AggregateVisitor<Integer> qnt =
    new AggregateVisitor<Integer>(Integer::sum, 0, MetricRunner.challengePreds()) {
    @Override
    public Integer visit(ExprBinary exp) {
      return (isMultArrow(exp.op)?1:0) + super.visit(exp); } };
  return new Object[] { exe.cmdName(), exe.formula().accept(vis), exe.result() }; }
```

where `isMultArrow()` is a method that simply tests whether the binary operator is one of the 15 multiplicity arrows. Running this metric on the data revealed that no student used it, which is very surprising, and a clear indicator that this special syntax needs to be taught more extensively.

Concerning the success rate, among those that used no quantifiers it was around 14%, among those that used 1 quantified variable it was 41%, with 2 variables it was 6%, and with 3 variables 35% (for 4 variables the available data is not statistically relevant). This data seems to confirm the intuition that the formulation with one quantified variable presented above was indeed the easiest way to solve the challenge, although the success rate of the students that used a pure FOL style was still quite high. The biggest surprise is however the extremely low success rate of the attempts with two quantified variables, something that might point out to further learning breakdowns. One potential issue could be that, instead of checking that the sets of auctions of different persons are disjoint, some students are simply checking whether they are different, a common mistake we already observed in previous challenges. Another potential issue we previously noticed is that many students forget that when quantifying over two variables of the same signature they can be instantiated with the same atom, hence the need for checking that `p!=q` in the above version with two quantified variables. Given the recurrence of this pattern, Alloy syntax has the special keyword **`disj`** that can be used in quantification to force the quantified variables to be different. With this keyword the version of the property with two quantified variables could be written more simply as

```
  all disj p,q: Person | no (p.auctions & q.auctions)
```

To further investigate this issue we defined the following metric that counts the number of uses of **disj**, whose full definition we omit since it is similar to those already presented.

```
@MetricMethod(groupby = COUNT, rule = "Number␣of␣uses␣of␣disj")
public static Object[] numDisjQuants(@ForAllExecutions Execution exe) { ... }
```

Applying this metric revealed that only 4% of the attempts used **disj** (albeit with a very high success rate of 70%), an indication that the latter issue was perhaps not discussed enough in classes nor **disj** presented as a simple way to address it.

5.4. Insights on learning Alloy

One of the goals of Alloy4Fun is to collect data to support empirical studies on learning and understanding formal specifications, which currently are virtually non-existent. In fact, the dataset that accompanies this work has already been used by others to evaluate a technique on the area of model repair [14].

Although we have not yet developed such a principled study, this section presents some preliminary results that, taking advantage of the collected data in both universities, provide some insights about how students learn Alloy. Namely, we will explore how the the success rate of students relates to: *i*) the complexity of the formulas, including the complexity of the temporal component; *ii*) some specific features of the language which our anecdotal evidence suggests pose more difficulties to students. We also analyze which are the most common errors/warnings thrown by the analyses. Of all the 35943 executions of the shared challenges in both universities without errors, 16232 were correct (45%), meaning that in average each challenge required two attempts to be solved (after fixing possible errors). Of those, 2468 threw a warning during analysis. Additionally, 13752 submissions threw an error during analysis and were not executed.

To this end, we started by classifying a normalized version[14] of each challenge according to a set of required concepts that in our experience have proven difficult for students to assimilate. Namely, we categorize each challenge regarding:

- the number of logic or relational operators (OC) and maximum nesting level (ON)
- the number of quantifiers (QC) and maximum quantifier nesting level (QN) (including comprehensions)
- the number of temporal operators (TC) and maximum temporal nesting level (TN)
- whether it requires manipulating ternary relations (TE)
- whether it requires transitive closures over fields (CF)
- whether it requires transitive closures over expressions (either relational expressions or relations defined by comprehension) (CE)
- whether it requires binary temporal operators (BT)

For each exercise, Table 4 presents the average value per challenge for numeric classifiers, and the number of challenges that fall into each of the (non-exclusive) Boolean categories.

Figure 8 presents the success rate of the challenges in relation to the (normalized) complexity of the expected solution (also listing the total number of challenges for each). Perhaps not surprisingly, our results show that the success rate of students correlates better with the nesting level of the expected solutions (ON) than with their actual size (OC). In the former, success rates are above average up to nesting level 5 (44%). This means that the size of the normalized expressions is not the best predictor for the students' difficulties –

[14]The normalized specifications were obtained by expanding into almost pure FOL (or FO-LTL when temporal logic was required), using no relational operators except for closures.

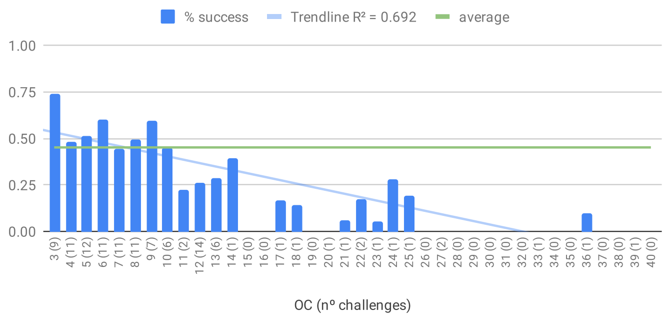
Table 4: Classification of shared Alloy4Fun exercises, average per challenge for numeric values, challenge count for Boolean ones.

Id	**Chall.**	**OC**	**ON**	**QC**	**QN**	**TC**	**TN**	**TE**	**CF**	**CE**	**BT**
1	10	5.9	3.9	1.2	1.2	0.0	0.0	0	0	0	0
2	15	8.9	4.9	1.9	1.7	0.0	0.0	4	0	0	0
3	10	5.9	3.9	1.2	1.2	0.0	0.0	0	0	0	0
4	15	7.8	4.9	1.9	1.7	0.0	0.0	4	1	0	0
5	8	6.9	4.5	1.3	1.3	0.0	0.0	0	2	1	0
6	7	10.9	5.6	2.9	2.4	0.0	0.0	6	0	2	0
7	4	7.8	5.0	1.8	1.8	0.0	0.0	0	1	0	0
8	4	17.0	6.5	2.0	1.8	0.0	0.0	0	0	1	0
9	20	5.1	4.4	1.1	1.1	1.7	1.7	0	0	0	3
10	18	16.6	7.4	2.3	1.7	2.2	1.7	0	1	0	5
i	1	36.0	8.0	7.0	3.0	0.0	0.0	0	0	0	0
ii	3	11.0	6.3	2.0	1.7	0.0	0.0	0	0	0	0
iii	3	10.3	5.0	1.0	1.0	0.0	0.0	2	0	0	0
iv	3	11.3	7.0	1.7	1.3	1.7	1.7	0	0	0	0

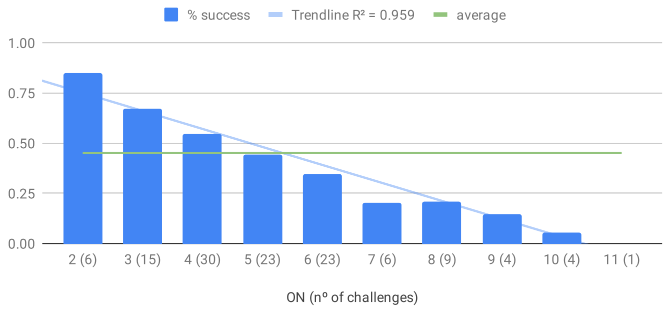
either because "horizontal" complexity is not as impactful as "vertical" one, or because that horizontal complexity is artificially caused by the expansion into the normalized version, while the nesting level remains mostly unchanged by that transformation. Finer studies could be conducted on the individual challenges, using strategies such as the one presented in Section 5.3, comparing the correctness of student submissions with their complexity.

Figure 9 presents the success rate of the challenges in relation to the number of quantifications in the expected (normalized) specifications, one of the main sources of complexity in first-order logic. Again, the nesting level (QN) seems to be a better predictor than the overall number of expected quantifications (QC), and with 2 nested quantifications success rate is already below average (40%). In fact, the success rate for solutions with any nested quantifier (QN $>$ 1) is 34%, well below average, while for those without nesting it stands at 53%.

Lastly, Fig. 10 presents the success rate of the challenges in relation to the (normalized) temporal complexity of the required specifications. As expected, challenges that require any number of temporal operators (TC $>$ 0) have a success rate well below those that do not require temporal operators (27% against

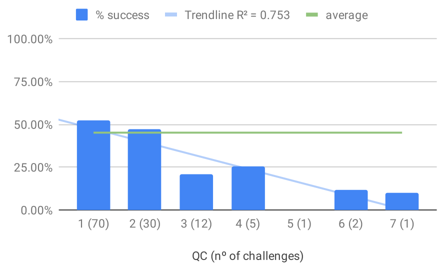


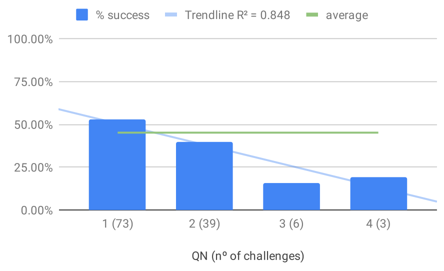
(a) Success rate over number of operators.



(b) Success rate over nesting level of operators.

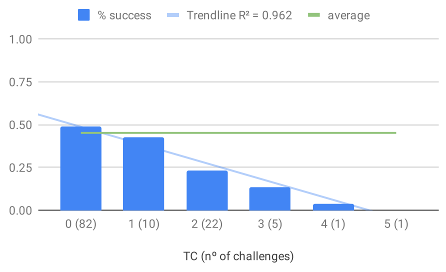
Figure 8: Success rate compared with expected normalized formula complexity (also showing the number of challenges with each complexity).



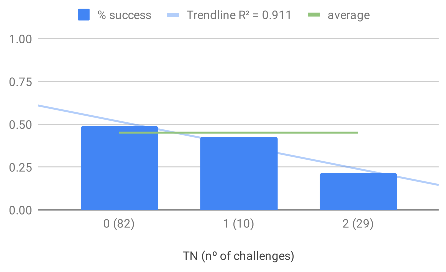


(a) Success rate over number of quantifiers. (b) Success rate over nesting level of quantifiers .

Figure 9: Success rate compared with expected normalized quantifier complexity (also showing the number of challenges with each complexity).



(a) Success rate over number of temporal operators.



(b) Success rate over nesting level of temporal operators.

Figure 10: Success rate compared with expected normalized temporal complexity (also showing the number of challenges with each complexity).

49%, respectively). However, students seem to understand well the usage of a single temporal operator (43% success rate, close to the average), but success quickly deteriorates as more temporal operators are required. Comparing with overall complexity, here the number of required temporal operators actually correlates well with the success rate of the students. Unfortunately, formulas with temporal nesting level of more than 2 are uncommon and were not available in our pool of challenges, so we are unable to properly compare TC with TN.

Figure 11 compares the results of challenge execution classified under each Boolean category (also listing the total number of challenges for each, which may overlap). For each of the 4 categories, the number of correct (green) and wrong (red) executions are presented. Additionally, entry All presents the results for all challenges.

As expected, the manipulation of ternary relations (TE), which is frequent in Alloy specifications, proved problematic (30% success rate). The result is aligned with our anecdotal evidence, and we already had special care with higher arity relations in lectures. Concerning closures, usage of a closure operator over a relation (CF) was not very problematic (46% success rate, slightly above average), but challenges that required applying a closure operator to a relational expression (CE) were the most difficult to solve among our categories (18% success rate). We had some anecdotal evidence that closures were difficult for

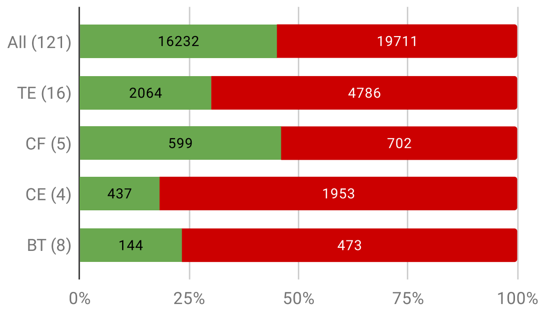


Figure 11: Executions per class of challenge (also showing the number of challenges under each category).

students, but this discrepancy between the two cases was rather surprising, meaning that special attention should be given to the latter case in lectures. Interestingly, for challenges that require binary temporal operators (BT), the success rate (23%) is close to the one for challenges requiring nested temporal operators shown in Fig. 10 (21%), so both concepts appear to be a bottleneck for students.

We also collected statistics about typical errors and warnings, with Tables 5 and 6 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, 3, 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 2, 5, and 7), in total 36% of the errors, and simple typing errors related to arity (messages 4, 6, 9, and 10), in total 19% 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 rec-

Table 5: Most common error messages.

	Message	#
1	There are ... possible tokens that can appear here.	2088
2	This must be a formula expression.	1216
3	The name ... cannot be found.	1087
4	`in` can be used only between 2 expressions of the same arity.	761
5	This must be a set or relation.	678
6	This cannot be a legal relational join.	661
7	This expression failed to be typechecked.	269
8	The `"all x"` construct is no longer supported.	248
9	`^` can be used only with a binary relation.	158
10	`~` can be used only with a binary relation.	144

Table 6: Most common warning messages.

	Message	#
1	The join operation here always yields an empty set.	646
2	This variable is unused.	377
3	Subset operator is redundant, because the left & right subexpressions are always disjoint.	319
4	`^` is redundant since its domain and range are disjoint.	110
5	Subset operator is redundant, because the right subexpression is always empty.	92
6	Subset operator is redundant, because the left subexpression is always empty.	74
7	`&` is irrelevant because the two subexpressions are always disjoint.	43
8	`<:` is irrelevant because the result is always empty.	35
9	`=` is redundant, because the left & right expressions always have the same value.	22
9	`=` is redundant, because the left & right expressions are always disjoint.	22
9	The value of this expression does not contribute to the value of the parent.	22

ommend using that alternative instead. Concerning warnings, all but the second most common message (unused variables, 22% 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 [15].

6. Concluding remarks and future work

This paper 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 graduate formal methods courses 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. A data analysis library was developed to simplify the mining of useful data from the derivation trees, and we have shown how this library can be used to write metrics that help identify possible learning breakdowns in the class.

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. We also intend to start running some of the metrics server-side shortly, with the resulting data and graphics available to owners of secret links at the click of a button, further simplifying the process of timely identifying learning breakdowns.

We still believe that the ability to simply share models and instances to permalinks is useful – even in contexts other than the educational, for instance to share models in publications. However, due to the feedback and usages results regarding that feature, we are wondering whether the full anonymity decision was too strict. A hybrid approach could be followed, for instance like the one followed in the CodeWorld platform[15], where sessions can be fully anonymous, but accounts can be created to keep track of the created and shared models. This would also be a step towards a future integration with learning management systems in order to automatically grade students.

Another open research line, also highlighted in the questionnaire results, is how to provide more intuitive feedback to students regarding incorrect submissions. One possibility is to incorporate in Alloy4Fun an alternative instance visualizer more amenable for dynamic systems [16]. Another one is to explore fault localization and repair techniques to identify possibly issues and suggest fixes.

[15] `https://code.world/`

Acknowledgements

We would like to thank Daniel Jackson for the helpful comments and suggestions about the design of Alloy4Fun, and also all the students that were beta testers.

References

[1] D. Jackson, Software Abstractions: Logic, Language, and Analysis, 2nd Edition, The MIT Press, 2012.

[2] N. Macedo, J. Brunel, D. Chemouil, A. Cunha, D. Kuperberg, Lightweight specification and analysis of dynamic systems with rich configurations, in: SIGSOFT FSE, ACM, 2016, pp. 373–383.

[3] N. Macedo, A. Cunha, J. Pereira, R. Carvalho, R. Silva, A. C. R. Paiva, M. S. Ramalho, D. C. Silva, Experiences on teaching Alloy with an automated assessment platform, in: ABZ, Vol. 12071 of LNCS, Springer, 2020, pp. 61–77.

[4] J. Brunel, D. Chemouil, A. Cunha, N. Macedo, Simulation under arbitrary temporal logic constraints, in: F-IDE, Vol. 310 of EPTCS, 2019, pp. 63–69.

[5] R. Boyatt, J. Sinclair, Experiences of teaching a lightweight formal method, in: FORMED, 2008, pp. 71–80.

[6] A. A. Sioson, Experiences on the use of an automatic C++ solution grader system, in: IISA, IEEE, 2013, pp. 1–6.

[7] K. Mangaroska, M. N. Giannakos, Learning analytics for learning design: A systematic literature review of analytics-driven design to enhance learning, IEEE Transactions on Learning Technologies 12 (4) (2019) 516–534.

[8] T. Ball, P. de Halleux, N. Swamy, D. Leijen, Increasing human-tool interaction via the web, in: PASTE, ACM, 2013, pp. 49–52.

[9] N. Tillmann, J. de Halleux, T. Xie, J. Bishop, Pex4Fun: A web-based environment for educational gaming via automated test generation, in: ASE, IEEE, 2013, pp. 730–733.

[10] N. Tillmann, J. de Halleux, Pex – White box test generation for .NET, in: TAP, Vol. 4966 of LNCS, Springer, 2008, pp. 134–153.

[11] J. Pereira, A web-based social environment for Alloy, Master's thesis, Universidade do Minho, Escola de Engenharia (2016).

[12] C. Liu, N. Macedo, A. Cunha, Simplifying the analysis of software design variants with a colorful alloy, in: SETTA, Vol. 11951 of LNCS, Springer, 2019, pp. 38–55.

[13] N. Macedo, A. Cunha, A. C. R. Paiva, Alloy4fun dataset for 2020/21 (Apr. 2021). `doi:10.5281/zenodo.4676413`.

[14] S. G. Brida, G. Regis, G. Zhengz, H. Bagheriz, T. Nguyenz, N. Aguirre, M. Frias, Bounded exhaustive search of Alloy specification repairs, in: ICSE, IEEE, 2021, pp. 1135–1147.

[15] J. Edwards, D. Jackson, E. Torlak, A type system for object models, in: SIGSOFT FSE, ACM, 2004, p. 189–199.

[16] R. Couto, J. C. Campos, N. Macedo, A. Cunha, Improving the visualization of Alloy instances, in: F-IDE, Vol. 284 of EPTCS, 2018, pp. 37–52.