An Analysis of the Impact of Field-Value Instance Navigation in Alloy's Model Finding

César Cornejo[1,3](✉), María Marta Novaira[1], Sonia Permigiani[1], Nazareno Aguirre[1,3], Marcelo Frias[2], Simón Gutiérrez Brida[1,3], and Germán Regis[1,3]

[1] University of Río Cuarto, Río Cuarto, Argentina
ccornejo@dc.exa.unrc.edu.ar
[2] University of Texas at El Paso, El Paso, USA
[3] National Council for Scientific and Technical Research (CONICET), Buenos Aires, Argentina

Abstract. The use of SAT-based model finding for specification analysis is a crucial characteristic of Alloy, and a main reason of its success as a language for software specification. When a property of a specification is analyzed and deemed satisfiable, the user usually explores instances of the corresponding satisfiability, in order to understand the analysis outcome. The order in which instances are obtained during exploration can impact the efficiency and effectiveness with which specification analysis is carried out. This has been observed by various researchers, and different instance exploration strategies have been proposed, besides the standard SAT-solver driven strategy implemented with the Alloy Analyzer.

In this paper, we concentrate on a strategy recently proposed in the literature, that we refer to as "field-value" driven, and has been implemented in the tool HawkEye. The tool allows the user to interactively guide instance exploration, by enforcing constraints requiring fields to contain (resp., do not contain) specific values. We design an experiment involving faulty Alloy specifications featuring combinations of over constraints and under constraints, and perform a user study to analyze the impact of this instance exploration strategy, in comparison with the standard SAT-solver driven exploration. The study focuses on HawkEye's facility of interactive instance querying and how it may favor users, in its current realization, during Alloy model analysis and debugging. We perform an assessment of the evaluation, and summarize some of the reasons that may diminish the impact of field-value exploration in model finding.

1 Introduction

Formal methods are known to have a great potential to reduce bugs in software, and in general to improve the quality of software [15]. By helping developers to build abstract formal models of software, its design and domain constraints, formal specification allows for mathematical reasoning on these models, leading to early identification of various issues, including problem misconceptions,

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
S. Bonfanti et al. (Eds.): ABZ 2024, LNCS 14759, pp. 141–159, 2024.
https://doi.org/10.1007/978-3-031-63790-2_9

non-trivial contradictions, and the invalidity of expected software properties [8]. While formal methods traditionally employ (semi automated) deductive reasoning on formal specifications, the use of fully automated specification analysis, as realized with model checkers [7] and model finders [18,36], has significantly improved the practical applicability of formal methods. In this context, the Alloy formal specification language [18], and its automated specification analysis tool Alloy Analyzer, have received increasing attention by the formal methods and software engineering communities [2–5,10,13,14,23,28,35]. Besides the simplicity and expressiveness of the Alloy language, a main reason for the language's success is indeed the possibility of automatically analyzing specifications. Alloy Analyzer implements a bounded-exhaustive instance finding technique, that given an Alloy formula α and a bound n, is able to report satisfying instances of α bounded by n, or otherwise inform that α is unsatisfiable within bound n [18]. Alloy Analyzer resorts to SAT solving to implement instance finding, and when a formula is found satisfiable, it is able to sequentially explore *all* satisfying instances (up to isomorphism) within bound n [18,21]. Alloy exploits instance finding in two ways: to check *assertions* (checking that no violating instances exist within a bound n), and to query the satisfiability of *predicates* (used to model software operations or specific software properties, among other uses).

When a property of a specification is analyzed and deemed satisfiable, e.g., when an assertion is found to be invalid, or a predicate is confirmed to be consistent with the assumptions in the specification, the user usually explores instances of the corresponding satisfiability, as a way of understanding the analysis outcome. Such exploration of instances is a powerful mechanism that users employ to either confirm their expectations (obtained instances are what the developers expected), or help in the debugging process (e.g., when the satisfying instances are counterexamples of an intended property). During the instance exploration process, the order in which instances are provided to the user can impact the efficiency and effectiveness with which specification analysis is carried out: as satisfying instances can grow combinatorially, users typically explore a few before deciding to confirm a specification behaves as expected, or decide on what to modify when the outcome is incorrect. This has been observed by various researchers, and different instance exploration strategies have been proposed, including size-sorted instances [22], instances that favor minimality [31], and instances that feature further differences with respect to previously reported ones [29]. Alloy's standard instance finding strategy, implemented in the Alloy Analyzer, is driven by the underlying SAT solver, and thus is relatively unguided.

The importance of instance exploration, and the relevance of satisfying instance orderings, call for novel ways of reporting satisfying instances in Alloy, a topic that receives significant interest. A newly proposed mechanism to produce Alloy satisfying instances is the one put forward by A. Sullivan with her HawkEye tool [33]. HawkEye allows developers to interactively guide instance exploration. We call this instance exploration "field-value" driven, since the developer guides the exploration by enforcing constraints requiring fields to contain, or alternatively do not contain, specific values. This paper concentrates on HawkEye's instance exploration, and studies the impact of this interactive strategy in Alloy specification analysis and debugging. We design an experiment involving faulty Alloy specifications featuring combinations of over constraints and under constraints, and perform a user study to analyze the impact of this instance exploration strategy, in comparison with the standard SAT-solver driven exploration. The evaluation is based on various quantitative metrics, that aim at assessing whether field-value driven exploration provides a significant improvement in the debugging process or not. We also perform an assessment of the evaluation, and summarize some of the reasons that may diminish the impact of field-value exploration in model finding.

2 Preliminaries

In this section we briefly present Alloy and its main specification analysis approach, including instance exploration. We also present HawkEye, the instance exploration strategy that is the subject of our study.

2.1 Alloy

Alloy is a formal specification language, that proposes to capture software properties using a logical language with a relational flavor [17,18]. Alloy puts a strong emphasis in automated analysis [21], and is accompanied by an efficient fully automated mechanism for bounded analysis of specification properties, that resorts to SAT solving to query for the bounded satisfiability or bounded validity of Alloy formulas [19]. Alloy features simple yet powerful syntax and semantics for specifications, or *models*, as they are typically called in the Alloy context. The usual way in which an Alloy specification is written starts by defining the model's data domains using *signatures*. Signatures describe data domains, and may include *fields*, which define relations between signatures. Signatures can also be extended by other signatures (akin to class extensions), meaning simply that the set of elements in the extending signature is a subset of the elements of the extended one. A signature can also be declared *abstract*, indicating that its corresponding elements are solely those of its extending signatures. As an example, consider a *secure laboratory* specification shown in Fig. 1 (this is an adaptation of a specification from [34]). The domain of *rooms* is defined by a corresponding signature `Room`. In the model a distinguished *secure room* is defined as `secure_lab`, a singleton (`one`) extension of *rooms*. Similarly, the laboratory's *administrative staff* and *researchers* are defined as extensions of the `Person` signature. The `Person` signature indicates that every person owns a set of *keys*. Signature `Key` captures the *keys* of the laboratory, and features fields to indicate which *person* is authorized to use a key, and which *room* a key opens.

Relations and constraints on model elements are specified by using *facts*, *predicates* and *functions*. These specifications are written in Alloy's relational logic, a first-order logic extended with relational operators, such as union (`+`), intersection (`&`), join (`.`), and most importantly, transitive closure (`^`). Alloy formulas use standard logical connectives (`and`, `or`, `not`, etc.), quantifiers (`all`, `some`, `no`, the latter corresponding to "there is no"), relational inclusion (in) and equality (=). Facts are formulas assumed to hold in the model. For instance, the laboratory specification indicates that every key that opens the secure lab, necessarily authorizes a researcher (each key authorizes a single person). Predicates are parameterized formulas, that can be used to state properties, and capture operations, among other things. For instance, CanEnter relates persons with rooms, specifying that a person p can enter a room r only if there exists a key that authorizes p, and opens room r. In order to state *intended* properties of the model, i.e., formulas that are expected to hold, Alloy allows developers to specify *assertions*. As an example, assertion OnlyResearchersInSecureLab states that any person that can enter the secure laboratory room must be a researcher.

Both predicates and assertions can be automatically analyzed, by searching for satisfying instances, in the case of predicates, and for counterexamples (witnessing property violations), in the case of assertions. Analyses are defined via *commands*: runs for predicates, and checks for assertions. The search exhaustively explores all possible instances bounded by a *scope*, the maximum number of elements to consider for each domain (associated with signatures). Scopes can be set as part of the command and can be different for each signature. As an example, the command:

```
run CanEnter for 3 Room, 5 Person, 5 Key
```

queries for instances containing up to three rooms, five people and five keys, that satisfy predicate CanEnter (as well as the model's facts). Alloy's bounded-exhaustive search for model instances is implemented by resorting to SAT solving, i.e., by translating Alloy analyses into propositional formulas in such a way that every satisfying valuation of the resulting formula corresponds to an instance of the predicate (resp., a counterexample of the assertion), within the corresponding scope. Alloy Analyzer, the tool implementing this technique, allows one to use a number of different off-the-shelf SAT solvers to analyze Alloy specification predicates and assertions.

Alloy has been recently extended to support linear-time temporal logic, among other features [6,24]. Since the instance exploration strategy that is the subject of this study is implemented only for "standard" Alloy, we do not consider here these newer characteristics of Alloy.

2.2 Alloy's Instance Exploration

When a command is executed using Alloy Analyzer and the corresponding formula is deemed satisfiable, the tool allows the developer to explore the satisfying instances of the formula. Instances can be sequentially inspected, one at a time, in a bounded-exhaustive fashion (all satisfying instances can be generated, up to isomorphism [20], using a mechanism known as *incremental solving* [16]). As satisfying instances correspond to satisfying valuations of the boolean formula that resulted from the translation from Alloy to propositional logic, and these are obtained by the execution of a SAT solver, the order in which satisfying

```
sig Room { }
one sig secure_lab extends Room { }

abstract sig Person {
    owns: set Key
}

sig Admin extends Person { }
sig Researcher extends Person { }

sig Key {
    authorizes: one Person,
    opens: one Room
}

fact {
    all k: Key | secure_lab = k.opens implies
                 k.authorizes in Researcher
}

pred CanEnter[p: Person, r:Room] {
    some k: Key | k.authorizes = p and r = k.opens
}
run CanEnter

assert OnlyResearchersInSecureLab {
    all p : Person | CanEnter[p, secure_lab] implies p in Researcher
}
check OnlyResearchersInSecureLab
```

Fig. 1. A simple Alloy model describing a secure laboratory.

instances are retrieved and presented to the developer depends on the underlying SAT solver. Thus, the developer has no direct control on the order in which these instances are obtained. This instance exploration order is of course important for model analysis. In particular, if a specification is weaker than intended, then the specification will lead to unwanted instances; how fast these are identified greatly depends on the order in which instances are retrieved. If, on the other hand, a specification is stronger than intended (but still satisfiable), then the developer will probably explore many specification instances until realizing that some expected "scenarios" are not being generated. This process can be labor-intensive, as the number of satisfying instances of Alloy formulas tends to grow combinatorially as the scope is increased. Specifications that are weaker than intended feature the so-called *underspecification* problem, i.e., not enough constraints are imposed and thus unwanted scenarios arise as a consequence. On the other hand, specifications that are stronger than intended have *overspecification* problems: the effect of stronger-than-intended assumptions is that some desired behaviors will be disallowed. Generally, underspecification problems are easier to identify via instance exploration than overspecification cases. The latter, as mentioned earlier, would in principle require one to explore all satisfying instances in order to acknowledge that there are missing scenarios.

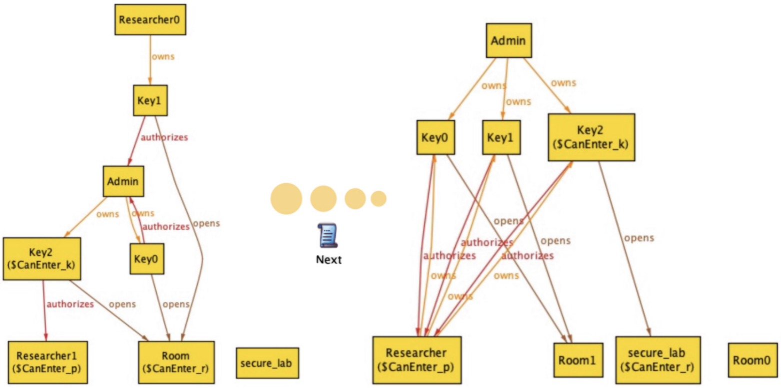
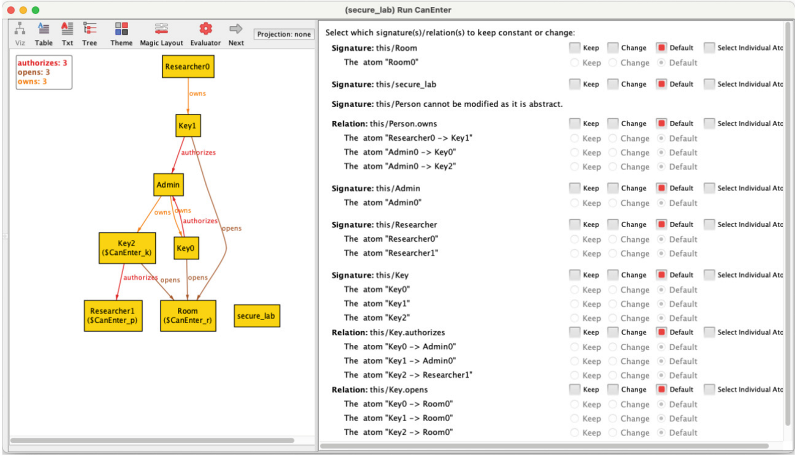


Fig. 2. Two satisfying instances of predicate `CanEnter`.

As an example of the impact of the instance visualization ordering provided by Alloy Analyzer, consider Fig. 2. The figure shows the first instance generated for the `CanEnter` predicate (with default scope 3), and the first instance of this predicate where, contrary to the expectations of the developer, an administrative staff member has access to the secure lab (meaning that the administrative person owns a key that opens the secure lab). This first unwanted instance is the 657th obtained instance, meaning that, by exploring instances of `CanEnter`, the developer would realize of this unwanted behavior after observing 656 previous instances. Moreover, assertion `OnlyResearchersInSecureLab` has no counterexamples in this model, so the analysis of this other assertion does not help us in identifying the defects present in this specification. Let us remark that the issue here, that leads to the unwanted scenarios, is associated with the fact that key ownership and authorization are different concepts, captured by different fields (`owns` and `authorizes`, respectively). The unwanted behaviors arise because no constraint enforces that a person may only own keys that authorize himself/herself, thus allowing admin staff to access the secure lab in some scenarios.

2.3 HawkEye

As explained above, the order in which instances are obtained during exploration can impact the efficiency and effectiveness with which specification analysis is carried out. Thus, having more control on how these instances can be visited or generated has the potential of significantly increasing the efficiency and effectiveness with which an Alloy model is analyzed. Precisely this hypothesis is what led to the development of HawkEye [33]. HawkEye is a technique put forward by A. Sullivan and implemented over the Alloy Analyzer that allows the developer to have more control on instance exploration, by interactively guiding the exploration, through the enforcement of constraints requiring to keep (resp., change) previously obtained values for signatures and relations. Therefore, we call this



(secure_lab) Run CanEnter

Select which signature(s)/relation(s) to keep constant or change:

Signature: this/Room ☐ Keep ☐ Change ☑ Default ☐ Select Individual Atc
The atom "Room0" ○ Keep ○ Change ● Default

Signature: this/secure_lab ☐ Keep ☐ Change ☑ Default ☐ Select Individual Atc

Signature: this/Person cannot be modified as it is abstract.

Relation: this/Person.owns ☐ Keep ☐ Change ☑ Default ☐ Select Individual Atc
The atom "Researcher0 -> Key1" ○ Keep ○ Change ● Default
The atom "Admin0 -> Key0" ○ Keep ○ Change ● Default
The atom "Admin0 -> Key2" ○ Keep ○ Change ● Default

Signature: this/Admin ☐ Keep ☐ Change ☑ Default ☐ Select Individual Atc
The atom "Admin0" ○ Keep ○ Change ● Default

Signature: this/Researcher ☐ Keep ☐ Change ☑ Default ☐ Select Individual Atc
The atom "Researcher0" ○ Keep ○ Change ● Default
The atom "Researcher1" ○ Keep ○ Change ● Default

Signature: this/Key ☐ Keep ☐ Change ☑ Default ☐ Select Individual Atc
The atom "Key0" ○ Keep ○ Change ● Default
The atom "Key1" ○ Keep ○ Change ● Default
The atom "Key2" ○ Keep ○ Change ● Default
Relation: this/Key.authorizes ☐ Keep ☐ Change ☑ Default ☐ Select Individual Atc
The atom "Key0 -> Admin0" ○ Keep ○ Change ● Default
The atom "Key1 -> Admin0" ○ Keep ○ Change ● Default
The atom "Key2 -> Researcher1" ○ Keep ○ Change ● Default
Relation: this/Key.opens ☐ Keep ☐ Change ☑ Default ☐ Select Individual Atc
The atom "Key0 -> Room0" ○ Keep ○ Change ● Default
The atom "Key1 -> Room0" ○ Keep ○ Change ● Default
The atom "Key2 -> Room0" ○ Keep ○ Change ● Default

Fig. 3. HawkEye visualization interface.

strategy for instance exploration *field-value* guided. As depicted in Fig. 3, HawkEye allows the user to manually instruct *how* to generate the next satisfying instance. It offers three possible actions on signature atoms, fields, and field tuples: one can *keep* an element of the current instance (enforcing it to be maintained as is in the next instance), *change* an element, causing it to be different from its current value in the next instance, or leave it unconstrained (the default) for the next instance.

HawkEye implements its instance exploration strategy by a more sophisticated use of incremental solving: it encodes the user-provided constraints as additional propositional clauses, that are conjoined with the formula being analyzed, in order to generate the "next" satisfying instance [33]. This is rather efficient, as incremental solving allows one to incorporate these extra clauses without having to restart the solving process [16]. The mechanism has been shown to allow for a faster convergence to violating scenarios in underspecification contexts, and to more quickly converge to unsatisfiability outcomes in over specification cases [33].

3 The Experiment

The instance exploration strategy proposed with HawkEye has been assessed using some case studies, and studying expressiveness possibilities, as well as convergence to identifying underconstraint and overconstraint issues in the considered case studies [33]. Our objective in this paper is to complement previous evaluations with a controlled user study, in which we analyze the impact of HawkEye's instance exploration, compared to standard Alloy, in debugging various Alloy specifications. Our study comprises two similar experiments with undergraduate students who have had some instruction on Alloy specification, using both Alloy Analyzer and HawkEye. These experiments were carried out in two consecutive years (we will refer to these as the *editions* of the study), and assessed the performance of the students in debugging four faulty Alloy models taken from the literature. The four models are:

- *Java Inheritance* (J.Inh): the model was taken from [34] and describes how class inheritance behaves in the Java programming language. This model contains an under constraint issue. Both editions of the experiment used exactly the same version of this faulty model.
- *Linked List* (L.List): the specification was taken from [34] and captures a formal specification of singly linked lists, with some associated properties such as sortedness. There were some slight changes to this model across the two editions of the experiment: the first edition used a faulty model with two bugs (an underconstraint bug and an overconstraint bug), while the version used for the second experiment had two overconstraint bugs. The model and one of the bugs were adapted from a case study in [33].
- *Secure Lab* (S.Lab): this model describes a security protocol of a laboratory with a secure room, which is meant to be accessed only by researchers (we used a variant of this model for illustration purposes in the preliminaries section). The specification is adapted from one originally introduced in [34]. The version used in the first edition contained an overconstraint bug, while the version for the second edition was improved, crafting an underconstraint bug instead.
- *Graph*: this specification was taken from [26], and captures graphs and some of their properties: undirectedness, transitivity and strong connectivity. The model used for the first edition of the experiment had an over constraint bug. The model was simplified for the second edition, and had no bugs.

In all cases, the specifications were complemented with corresponding detailed narratives, explaining what each specification was intended to express. Each student participating in the study received these specifications and their narratives. Students were also informed that the specifications may be correct or incorrect; in case a specification was considered correct, they had to simply explicitly state it and leave the specification as is; if a specification was considered incorrect, they had to debug it, and provide a modified specification with the identified issues fixed. No time limit was imposed, but the experiment was designed with an expected duration of two hours (this is also the reason why the experiment is limited to four faulty specifications). Time was not taken into account as a metric during the evaluation.

In both editions of the experiment, students were split into two groups: one of the groups was assigned the standard Alloy Analyzer, the other group was assigned HawkEye. The work of each student in each group was individual. The first edition of the experiment was carried out several months after students had instruction on Alloy in the context of an undergraduate course. For this reason, before the experiment, students received a brief summary of Alloy's syntax and semantics, as well as a brief demo on the use of Alloy Analyzer and HawkEye; students in HawkEye's group were also presented with the specific characteristics of HawkEye's instance exploration, and its mechanism to enforce constraints for "next instance" generation. The second edition of the experiment was performed in the context of the undergraduate course featuring Alloy; HawkEye was included in the course as an additional tool for specification analysis, and thus the students did not need any notation summary or tool demo prior to the experiment.

When the students finished their corresponding assignments, they were asked to fill in a questionnaire. Questions in the questionnaire asked about problems found in the specifications, opinions on the user-friendliness of the corresponding tool, issues that may have emerged during the assignments, and their overall opinions on the tools, the specification language, and possible suggestions to improve modeling and analysis in the Alloy context.

The interaction of the students with the corresponding tools during specification debugging was measured along the following general metrics:

- *Number of instances* explored during the analysis/debugging of each model.
- *Number of edits* made to the specification in order to fix it. Some standard "interaction" edits such as adding a `run show` command were not taken into account.

For the students that received HawkEye, we also measured the *number of guided instances* visited during instance exploration. Moreover, when guided instance exploration was used, we also assessed the complexity of the added constraints, measured in terms of the number of constraints used, their type (change or keep), and their granularity (whole field vs. specific field values, for instance). A total of 30 students participated in the experiment, 10 students in the first edition, and 20 students in the second edition. In both editions the participating students were split into two groups of the same size, and assigned a tool to each group.

3.1 Recording Student Interactions

Students were informed that their interactions with the corresponding tools was going to be logged, and assessed anonymously. It is worth mentioning that the participation in the experiment was optional, and thus all participating students voluntarily did so. The logging of user actions was realized via implementations of interaction logging into the Alloy Analyzer and HawkEye. The logging was completely transparent to the users, and generated logging data in a folder, that the corresponding student submitted when the assignment concluded. All relevant actions and events were logged: edits to a model and the corresponding modified lines; syntactic checks on models; command executions and their corresponding outcomes; instance exploration activity, including the visualized instances, their number, etc.; and of course the specific restrictions in the interactive instance generation for the case of HawkEye. All this information and the faulty models are available in [1].

4 Evaluation

We now present the results of the experiment, and an evaluation of the information collected from the interaction of the participating students with the tools. Our first part of the analysis performs a higher level comparison of the two tools, based on the metrics described in the previous section. The second part concentrates more deeply on the specific usage of HawkEye, tries to examine how the users exploited the instance constraints provided by HawkEye, and how this feature facilitated specification debugging.

4.1 Higher Level Comparison of the Tools

Let us first summarize the general results, in terms of bugs found and models fixed, corresponding to the two analyzed tools. Table 1 provides this information. Each row corresponds to a bug in a specific model, indicated by its abbreviation, and provides information about the model the bug corresponds to (using v1 and v2 for distinguishing the models used in the two editions of the experiment), and the type of bug ('O' for overconstraint, 'U' for underconstraint, and 'correct' for the non-buggy model). In each row we indicate how many participants found the corresponding bug (identified the correct bug location), and how many of these were able to correctly fix the specification (notice then that the "model fixed" column value is always smaller or equal to the corresponding "Bug Found" column value), separated in the two tool groups. Regarding incorrect bug identification, we had only a few cases: one student (HawkEye group) reported a bug in Graph v1, but did not correctly locate the bug in the model; two students (both from the HawkEye group) reported bugs in Graph v2, which is a correct model; and finally a student (Alloy group) reported a bug in Java Inheritance v2, without correctly locating it. The bugs in models on Java Inheritance and Linked List were the ones in which participants showed a better performance. The only overconstraint bug in this group is LinkedList v2 bug 2, which is identifiable by a provided assert. The remaining overconstraint bugs in Linked List, Secure Lab and Graph showed a significantly worse overall performance. In general, in terms of these high level metrics, HawkEye did not provide a clear advantage in debugging, over Alloy Analyzer. Overconstraint bugs seem more difficult to spot via instance exploration than underconstraint bugs (with the exception of Linked List v2 bug 2, as mentioned above), as expected.

Let us further analyze the corresponding interactions with the tools. Table 2 shows the total numbers of edits that participants made to the models during debugging, and the instances explored during analysis (for HawkEye users, the table also reports the number of constraints set during instance exploration, i.e., the overall use of guided instance exploration). Overall, participants using Alloy Analyzer explored more instances than those using HawkEye. However, we did not observe a correlation between the use of constraints in HawkEye with exploring fewer instances (it is worth noting that the numbers in the table are total numbers, but only 8 out of 15 students in this group used constraints). Notice that some cells in Table 2 have two reported numbers (one of them within parentheses). These

Table 1. Summary of bugs found and models fixed with the analyzed tools

Model	Alloy		HawkEye	
	Bug found	Model fixed	Bug found	Model fixed
Graph v1 bug (O)	0	0	0	0
J.Inh v1 bug (U)	5	4	5	4
L.List v1 bug 1 (U)	5	4	4	2
L.List v1 bug 2 (O)	3	2	1	1
S.Lab v1 bug (O)	4	4	2	1
Graph v2 correct	–	–	–	–
J.Inh v2 bug (U)	7	5	8	3
L.List v2 bug 1 (O)	1	1	2	1
L.List v2 bug 2 (O)	5	4	7	5
S.Lab v2 bug (U)	4	2	6	1
Total underconst. (U)	21	15	23	10
Total overconst. (O)	13	11	12	8

Table 2. User Interactions, in terms of editions to models and instances explored during analysis.

Model	Alloy		HawkEye		
	Explored instances	Edits	Explored instances	Constraints	Edits
Graph v1	40	0	54	2	0
J.Inh v1	71	11	109 (169)	17	25 (96)
L.List v1	95 (257)	15	126	8	25
S.Lab v1	188	17	79	1	9
Graph v2	372	27	432	0	37
J.Inh v2	366	79	285	12	108
L.List v2	1890	92	530	44	113
S.Lab v2	341	41	88	0	24
Total	3363	282	1703	84	341

are depicted to highlight some outliers in the analysis. In the Java Inheritance v1 model, a HawkEye user interpreted a "check" command as a "run" command, and since the student expected instances which were not obtained (the assertion had no counterexamples), many subsequent edits were performed on the model after having repaired it. Value 25 in this case corresponds to the edits until the model was fixed, whereas 96 is the total number of edits (including the unnecessary edits after the model was first fixed). Similarly, in Linked List v1, an Alloy user bounded exhaustively checked *all* instances of the "sorted" predicate, a behavior that is uncommon compared with the usual. We report between parentheses the total number of explored instances, including this unusual case.

Table 3. Average user interactions, in terms of editions to models and instances explored during analysis.

Model	Alloy		HawkEye	
	Explored instances	Edits	Explored instances	Edits
Graph v1	8.0	0.0	10.8	0.0
J.Inh v1	14.2	2.2	22.0	5.0
L.List v1	19.0	3.0	25.2	5.0
S.Lab v1	37.6	3.4	15.8	1.8
Graph v2	37.2	2.7	43.2	3.7
J.Inh v2	36.6	7.9	20.8	9.9
L.List v2	189.0	9.2	53.0	11.3
S.Lab v2	34.1	4.1	8.8	2.4
Total	224.2	18.8	113.5	22.7

We also analyzed in more depth the impact of HawkEye constraints in model edits and repairs. Overall, in eight opportunities users performed a model edit immediately after querying for an instance under a manually specified HawkEye constraint. In terms of model fixing, in one case, a model was repaired immediately after querying for an instance under a manually specified constraint. To better understand the "per user" interaction with the corresponding tools, Table 3 reports the average number of edits and instances explored, for each of the case studies, and the analyzed tools. Again, in general Alloy Analyzer users explored more instances (roughly twice as many compared to HawkEye users) and performed slightly fewer model edits during debugging (about 17% fewer edits, compared with HawkEye users).

4.2 Detailed Analysis of HawkEye Usage

We now turn our attention to a more detailed analysis of how HawkEye's instance exploration is used/exploited. First, it is worth remarking that the instance exploration facility of HawkEye may be mimicked by standard Alloy users by manually writing specific "specializations" of existing predicates or introducing additional facts (these specializations would enforce the constraints that a HawkEye user would set during instance exploration). However, none of the Alloy users exhibited this behavior. Although it is difficult to understand why users did not do this kind of modification, we conjecture that users do not do so because of the associated cost of writing additional predicates or facts. On the other hand, HawkEye users, who had this facility "built-in", made use of it, indicating that it is indeed a facility that users find useful (8 out of 15 participants used HawkEye constraints in at least one model).

Regarding the specific constraints used during instance exploration with HawkEye, Table 4 summarizes the observations. In the table, Atom refers to constraints that enforce changes in specific atoms, whereas All refers to the coarser

Table 4. Types of constraints used in HawkEye's instance explorations

Model	Signatures		Relations		Keep	Change
	Atom	All	Atom	All		
Graph v1	0	1	1	1	1	2
J.Inh v1	10	9	11	6	32	4
L.List v1	1	10	0	13	15	9
S.Lab v1	3	4	4	1	11	1
Graph v2	0	0	0	0	0	0
J.Inh v2	7	6	10	2	25	0
L.List v2	0	40	12	57	100	21
S.Lab v2	0	0	0	0	0	0
Total	21	70	38	80	184	37

granularity constraint that enforces a change (or a keep) on the whole signature or field. Overall, there were more "keep" constraints than "change" constraints, and users favored coarser granularity in the constraints (whole signatures/fields, over specific elements or tuples in these).

Let us now focus on how the use of HawkEye constraints evolved as the experiments progressed. Figure 4 shows the number of explored instances, with and without the use of constraints, as users progressed from one model to the next, during the experiment. Regarding which models the participants chose to work on, 14 participants started with Java Inheritance and the rest chose Graph; 13 participants chose Linked List as the second model, while one student worked on each of the other models as their second model; the third model was Secure Lab for 11 participants, 2 worked on Linked List, and 2 on Graph; and finally 11 students worked on Graph as their last model and the rest on Secure Lab. As the Figure shows, the use of constraints decreased as time progressed (and so did the proportion of constrained instances vs unconstrained ones). More precisely, for the first model, 4.52% of the instance explorations corresponded to "constrained" ones (25 out of 553); for the second model, the percentage was 8.04% (51 out of 583 instance explorations); for the third model, this percentage decreased to 1.24% (3 out of 239), and finally for the last model the percentage was 0.35% (1 out of 282 explorations).

It is worth turning our attention back to the Secure Lab (v1 and v2) case study, where, as we have seen earlier in this section, instance exploration by Alloy users was significantly greater than the number of instance explorations by HawkEye users. It is worth noting however that almost no guiding constraints were used for this case study by HawkEye users (only one participant set exactly one constraint for instance exploration). This leads us to believe that, for this case study, HawkEye participants used the tool mostly in the style of Alloy Analyzer, i.e., without guided instance exploration. On the other hand, LinkedList v2 is a case study in which HawkEye seemed to have provided more benefits:

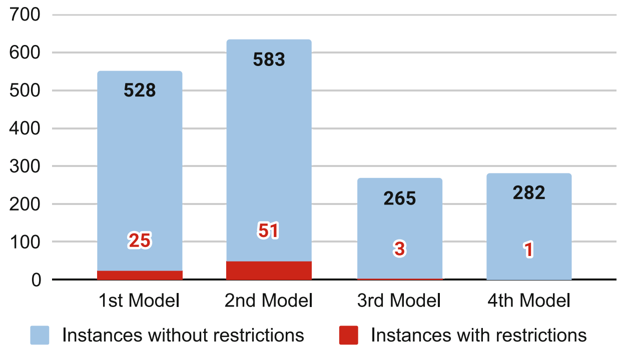


Fig. 4. Use of constraints vs standard instance exploration, by HawkEye users, as the experiment progressed from model to model.

significantly fewer instances explored compared to Alloy users, which correlates in this case with an important use of constraints.

4.3 Discussion

Overall, our experiment does not indicate that field-value guided instance exploration significantly improves model analysis and debugging. Guided instance exploration received attention by the HawkEye users in our experiments, i.e., users actually exercised this functionality. But the use of guided instance exploration was not maintained as the experiment progressed, and had no significant impact in debugging and fixing, as far as the metrics considered allowed us to assess.

We believe that guided instance exploration is worthwhile, but the results may be affected by issues that have to do with the current tool support, i.e., the current implementation, and lack of clarity of the results of the analyses when guided constraints are employed. More precisely, during our questionnaire after the experiment, users valued positively the constraint editor, and said that it allowed them to more easily converge to specific instances they were interested in observing. However, they also reported some limitations. Concretely, two users mentioned that HawkEye does not save previous constraints, and sets everything back to default after querying for a "next" instance. Also, when subsequent constraints are set, users reported that it is not clear whether newly produced instances also satisfy the first constraints set or not, making it more difficult to interpret the results. There is no clear way of knowing which constraints one has accumulated during a use session of instance exploration with HawkEye. In other words, having a clearer picture of the accumulated constraints and their potential contradictions, is useful information that is currently not present during HawkEye's exploration sessions.

We asked users to rank the tool usability and clarity from extremely easy (1) to extremely difficult (10). HawkEye had an average of 5.2 (standard deviation

1.55) vs. the 5 (standard deviation 1.63) average of Alloy Analyzer, showing in general terms a relatively similar usage effort across the tools. It must also be mentioned that, in the first edition of our experiment, all participants had some previous experience with Alloy Analyzer, but none with HawkEye specifically. Thus, it seems reasonable to assume that users were more accustomed to Alloy instance exploration concepts, compared to the more novel exploration offered by HawkEye. Our second edition corrected this issue, but except for LinkedList v2, the use of constraints was, in general terms, maintained. An extended study, with further participants and different levels of expertise, and perhaps with more models with different characteristics, is definitely a must to generalize the results of our experiment, and try to find a correlation between the use of constraints and specification characteristics. In our opinion, based on the observations of the participants and our own experience with HawkEye, the explicit information of the accumulated constraints is crucial. Moreover, it seems relatively natural to think of constraints during instance exploration as a query *refinement* process, that may be more inherently tree-like, rather than linear, as in standard Alloy's instance exploration.

5 Related Work

The order in which satisfying instances of a formula are reported by a model finder is generally agreed to have an important impact in model analysis and debugging. Thus, different instance exploration approaches have been proposed. Aluminum [31] favors *minimality*, i.e., it attempts to first generate the smallest instances that satisfy a particular Alloy formula. REACH [22] follows a related policy, by generating instances in ascending order by size. Bordeaux [29] exploits partial max satisfiability [12] to generate pairs of satisfying and non-satisfying instances, that are as similar as possible, thus helping to understand what is being captured by a formula. N. Macedo et al. [25] also proposed using partial max satisfiability to visit satisfying instances with minimal or maximal differences with respect to previously generated satisfying instances. In this same line, Amalgam [30] complements instance finding with explanations of why (or why not) specific components are part (or are not part) of a satisfying instance. More recently, Ringert and Sullivan [32] also studied how instances are presented to the user, concentrating on the relevant parts of instances, in relation to the property being analyzed. They proposed a notion of *abstract instance*, which summarizes the important aspects in the satisfaction (or, more precisely, in the violation) of a property, and abstracts away the irrelevant aspects that are common to many instances. HawkEye [33], the subject of our study, is the only instance exploration strategy that allows for an interactive exploration of instances, by allowing users to essentially build some specialized queries for subsequent instances during instance exploration. Other works such as [11,27] involve users studies. The former examines how users interact cognitively with model-finder output, and explores how those tools can be enhanced to assist them more effectively. The latter explores how both non-novice and novice users employ the Alloy Analyzer to detect bugs and create models based on natural language specifications.

The impact of different strategies for Alloy instance exploration has been studied in the past [9], including comparisons with various techniques such as Aluminum and Amalgam. These studies are prior to the development of HawkEye, and thus our work complements them. We focused specifically in the facility of interactive instance querying, as implemented in HawkEye, and how it may favor developers, in its current realization, during Alloy model analysis and debugging.

6 Conclusion

The exploration of satisfying instances of a specification component, e.g., counterexamples of intended properties or sample instances of predicates of a specification, is known to be a very powerful mechanism that users employ to validate as well as to debug specifications. How these instances are explored, or more precisely the order in which these are reported to developers, is very important, and can improve the effectiveness and efficiency of specification analysis. This is confirmed by the various different instance exploration strategies that have been proposed for the Alloy language [22,29,31,33].

In this paper, we have studied HawkEye [33], one of these strategies, the first, as far as we are aware of, that allows for an *interactive* mechanism to explore instances. HawkEye provides a mechanism that allows the user to enforce constraints on subsequent instances, during specification instance exploration. We proposed a user study, with the aim of assessing the impact of this dynamic instance exploration strategy in specification debugging and analysis. Our experiment involved voluntary students, with certain experience with Alloy and HawkEye, who undertook the task of debugging a number of faulty Alloy specifications, some using Alloy Analyzer, some using HawkEye. Our results indicate that the interactive instance exploration offered by HawkEye, while attractive and found useful by users, has no substantial impact in model debugging and repairability. We remark however that our study is limited both in the number of participating users, and their expertise level (in total, the study only involved 30 novice users). Finally, our observations also suggest that further development and refinement of HawkEye's underlying concepts and implementation are needed to match the maturity level of traditional Alloy instance exploration, including a better handling of constraint relationships during exploration sessions.

Acknowledgements. We would like to thank the anonymous reviewers of this paper for their detailed feedback. This work is supported by Argentina's ANPCyT through grants PICT 2019-3134, 2019-2050, 2020-2896, and 2021-0601; by an Amazon Research Award; and by EU's Marie Sklodowska-Curie grant No. 101008233 (MISSION).

Competing Interests. The author(s) has no competing interests to declare that are relevant to the content of this manuscript.

References

1. Replication package. https://sites.google.com/view/field-value-evaluation
2. Abad, P., et al.: Improving test generation under rich contracts by tight bounds and incremental SAT solving. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, 18–22 March 2013, pp. 21–30. IEEE Computer Society (2013)
3. Alhanahnah, M., Stevens, C., Bagheri, H.: Scalable analysis of interaction threats in IoT systems. In: Khurshid, S., Pasareanu, C.S. (eds.) ISSTA 2020: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, 18–22 July 2020, pp. 272–285. ACM (2020)
4. Bagheri, H., Kang, E., Malek, S., Jackson, D.: A formal approach for detection of security flaws in the android permission system. Formal Aspects Comput. **30**(5), 525–544 (2018)
5. Bagheri, H., Sadeghi, A., Behrouz, R.J., Malek, S.: Practical, formal synthesis and automatic enforcement of security policies for android. In: 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, 28 June–1 July 2016, pp. 514–525. IEEE Computer Society (2016)
6. Brunel, J., Chemouil, D., Cunha, A., Macedo, N.: The electrum analyzer: model checking relational first-order temporal specifications. In: Huchard, M., Kästner, C., Fraser, G. (eds.) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, 3–7 September 2018, pp. 884–887. ACM (2018)
7. Clarke, E.M., Grumberg, O., Kroening, D., Peled, D.A., Veith, H.: Model Checking, 2nd edn. MIT Press, Cambridge (2018)
8. Clarke, E.M., Wing, J.M.: Formal methods: state of the art and future directions. ACM Comput. Surv. **28**(4), 626–643 (1996)
9. Danas, N., Nelson, T., Harrison, L., Krishnamurthi, S., Dougherty, D.J.: User studies of principled model finder output. In: Cimatti, A., Sirjani, M. (eds.) SEFM 2017. LNCS, vol. 10469, pp. 168–184. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66197-1_11
10. Dennis, G., Chang, F.S.-H., Jackson, D.: Modular verification of code with SAT. In: Pollock, L.L., Pezzè, M. (eds.) Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, 17–20 July 2006, pp. 109–120. ACM (2006)
11. Dyer, T., Nelson, T., Fisler, K., Krishnamurthi, S.: Applying cognitive principles to model-finding output: the positive value of negative information. Proc. ACM Program. Lang. **6**(OOPSLA1), 1–29 (2022)
12. Fu, Z., Malik, S.: On solving the partial MAX-SAT problem. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 252–265. Springer, Heidelberg (2006). https://doi.org/10.1007/11814948_25
13. Galeotti, J.P., Rosner, N., Pombo, C.G.L., Frias, M.F.: TACO: efficient sat-based bounded verification using symmetry breaking and tight bounds. IEEE Trans. Softw. Eng. **39**(9), 1283–1307 (2013)
14. Galeotti, J.P., Rosner, N., Pombo, C.L., Frias, M.F.: Analysis of invariants for efficient bounded verification. In: Tonella, P., Orso, A. (edis.) Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, 12–16 July 2010, pp. 25–36. ACM (2010)

15. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering, 2nd edn. Prentice Hall (2003)
16. Hooker, J.N.: Solving the incremental satisfiability problem. J. Log. Program. **15**(1&2), 177–186 (1993)
17. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2), 256–290 (2002)
18. Jackson, D.: Software Abstractions - Logic, Language, and Analysis. MIT Press, Cambridge (2006)
19. Jackson, D.: Alloy: a language and tool for exploring software designs. Commun. ACM **62**(9), 66–76 (2019)
20. Jackson, D., Jha, S., Damon, C.: Isomorph-free model enumeration: a new method for checking relational specifications. ACM Trans. Program. Lang. Syst. **20**(2), 302–343 (1998)
21. Jackson, D., Schechter, I., Shlyakhter, I.: Alcoa: the alloy constraint analyzer. In: Ghezzi, C., Jazayeri, M., Wolf, A.L. (eds.) Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, 4–11 June 2000, pp. 730–733. ACM (2000)
22. Jovanovic, A., Sullivan, A.: REACH: refining alloy scenarios by size (tools and artifact track). In: IEEE 33rd International Symposium on Software Reliability Engineering, ISSRE 2022, Charlotte, NC, USA, 31 October–3 November 2022, pp. 229–238. IEEE (2022)
23. Khalek, S.A., Yang, G., Zhang, L., Marinov, D., Khurshid, S.: TestEra: a tool for testing java programs using alloy specifications. In: Alexander, P., Pasareanu, C.S., Hosking, J.G. (eds.) 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, 6–10 November 2011, pp. 608–611. IEEE Computer Society (2011)
24. Macedo, N., Brunel, J., Chemouil, D., Cunha, A., Kuperberg, D.: Lightweight specification and analysis of dynamic systems with rich configurations. In: Zimmermann, T., Cleland-Huang, J., Su, Z. (eds.) Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, 13–18 November 2016, pp. 373–383. ACM (2016)
25. Macedo, N., Cunha, A., Guimarães, T.: Exploring scenario exploration. In: Egyed, A., Schaefer, I. (eds.) FASE 2015. LNCS, vol. 9033, pp. 301–315. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46675-9_20
26. Macedo, N., et al.: Experiences on teaching alloy with an automated assessment platform. Sci. Comput. Program. **211**, 102690 (2021)
27. Mansoor, N., Bagheri, H., Kang, E., Sharif, B.: An empirical study assessing software modeling in alloy. In: 11th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE 2023, Melbourne, Australia, 14–15 May 2023, pp. 44–54. IEEE (2023)
28. Mirzaei, N., Garcia, J., Bagheri, H., Sadeghi, A., Malek, S.: Reducing combinatorics in GUI testing of android applications. In: Dillon, L.K., Visser, W., Williams, L.A. (eds.) Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, 14–22 May 2016, pp. 559–570. ACM (2016)
29. Montaghami, V., Rayside, D.: Bordeaux: a tool for thinking outside the box. In: Huisman, M., Rubin, J. (eds.) FASE 2017. LNCS, vol. 10202, pp. 22–39. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_2
30. Nelson, T., Danas, N., Dougherty, D.J., Krishnamurthi, S.: The power of "why" and "why not": enriching scenario exploration with provenance. In: Bodden, E., Schäfer, W., van Deursen, A., Zisman, A. (eds.) Proceedings of the 2017 11th Joint

Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, 4–8 September 2017, pp. 106–116. ACM (2017)

31. Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: principled scenario exploration through minimality. In: Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) 35th International Conference on Software Engineering, ICSE 2013, San Francisco, CA, USA, 18–26 May 2013, pp. 232–241. IEEE Computer Society (2013)
32. Ringert, J.O., Sullivan, A.: Abstract alloy instances. In: Chechik, M., Katoen, J.P., Leucker, M. (eds.) FM 2023. LNCS, vol. 14000, pp. 364–382. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-27481-7_21
33. Sullivan, A.: Hawkeye: user-guided enumeration of scenarios. In: Jin, Z., et al. (eds.) 32nd IEEE International Symposium on Software Reliability Engineering, ISSRE 2021, Wuhan, China, 25–28 October 2021, pp. 569–578. IEEE (2021)
34. Wang, K., Sullivan, A., Khurshid, S.: Automated model repair for alloy. In: Huchard, M., Kästner, C., Fraser, G. (eds.) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, 3–7 September 2018, pp. 577–588. ACM (2018)
35. Zave, P.: Reasoning about identifier spaces: how to make chord correct. IEEE Trans. Softw. Eng. **43**(12), 1144–1156 (2017)
36. Zhang, H., Zhang, J.: MACE4 and SEM: a comparison of finite model generators. In: Bonacina, M.P., Stickel, M.E. (eds.) Automated Reasoning and Mathematics. LNCS (LNAI), vol. 7788, pp. 101–130. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36675-8_5