Analyzing the Uses of a Software Modeling Tool

Xiaoming Li[1], Daryl Shannon[2], Jabari Walker[1]
Sarfraz Khurshid[2], Darko Marinov[1]

[1] *Dept. of Computer Science, University of Illinois, Urbana-Champaign, USA*
[2] *Dept. of Electrical & Computer Engineering, University of Texas, Austin, USA*
xli15@cs.uiuc.edu, dshannon@ece.utexas.edu, jlwalkr1@yahoo.com
khurshid@ece.utexas.edu, marinov@cs.uiuc.edu

Abstract

While a lot of progress has been made in improving analyses and tools that aid software development, less effort has been spent on studying how such tools are commonly used in practice. A study into a tool's usage is important not only because it can help improve the tool's usability but also because it can help improve the tool's underlying analysis technology in a common usage scenario. This paper presents a study that explores how (beginner) users work with the Alloy Analyzer, a tool for automatic analysis of software models written in Alloy, a first-order, declarative language. Alloy has been successfully used in research and teaching for several years, but there has been no study of how users interact with the analyzer. We have modified the analyzer to log (some of) its interactions with the user. Using this modified analyzer, 11 students in two graduate classes formulated their Alloy models to solve a problem set (involving two problems, each with one model). Our analysis of the resulting logs (total of 68 analyzer sessions) shows several interesting observations; based on them, we propose how to improve the analyzer, both the performance of analyses and the user interaction. Specifically, we show that: (i) users often perform consecutive analyses with slightly different models, and thus incremental analysis can speed up the interaction; (ii) users' interaction with the analyzer is sometimes predictable, and akin to continuous compilation, the analyzer can precompute the result of a future action while the user is editing the model; and (iii) (beginner) users can naturally develop semantically equivalent models that have significantly different analysis time, so it is useful to study manual and automatic model transformations that can improve performance.

1 Introduction

Alloy [4] is a first-order, declarative language suitable for expressing models of software systems. Alloy models are amenable to fully automatic analysis, using

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

the Alloy Analyzer [6]. The analyzer translates Alloy formulas to propositional formulas using a given *scope*, i.e., a bound on the universe of discourse, and uses off-the-shelf SAT solvers to find concrete instances or counterexamples for Alloy formulas. Alloy has been successfully used in research and teaching for several years and has assisted in finding and correcting design flaws in various systems [5,7]. So far, however, there has been no study into how users interact with the analyzer. It is important to study such interactions for at least the following reasons: (1) to improve the underlying analysis technology; (2) to point out how to make the tool more usable; and (3) to develop idioms for building Alloy models that enable more efficient analyses.

Two aspects of Alloy make such an investigation particularly worthwhile: the declarative nature of the language and the bounded-exhaustive checking performed by the analyzer. Declarative logic paradigms, in general, and Alloy, in particular, tend to elicit a pervasive use of conjunction. An Alloy model is often built by first defining sets and relations that represent the model and then defining formulas that constrain the representation appropriately, starting from a minimal representation and incrementally strengthening it until a sufficient level of detail is attained. The use of the analyzer in an interactive fashion assists the users in making the incremental changes and checking their validity. These incremental changes tend to be small, so the analyzer may exploit the differences introduced between consecutive executions [1] to provide a faster execution using the result of the previous execution.

The nature of Alloy's bounded-exhaustive checking implies that its results are valid with respect to the given scope only, i.e., if the analyzer fails to find an instance that satisfies an Alloy formula within a given scope (bound), an instance may still exist in a larger scope. For Alloy users, it is natural to increase their level of confidence in a model by increasing the scope and re-checking the model for which the analyzer previously failed to generate a desired instance in a smaller scope. Notice that in such a scenario, the only change in the model between two consecutive executions of the analyzer is the scope—once again, a situation arises where the analyzer may be able to provide a faster checking using the result of the previous execution.

This paper presents a study into how (beginner) users work with the Alloy Analyzer. We have modified the analyzer to log (some of) its interactions with the user. The modified analyzer saves a copy of the Alloy model every time the user compiles the model. To investigate the analyzer's usage, we asked students in two of our (graduate) classes to solve a problem set that required them to develop Alloy models using the logging-enabled analyzer. Our analysis of the resulting 68 logs from 11 students shows three key observations:

(i) Users often perform consecutive executions with models that differ only

[1] We use the term "execution" to refer to one checking of an Alloy model. The commonly used term for checking is "analysis", but in this paper, we use that term to refer to our study of Alloy executions.

slightly, which is as expected based on the two afore-mentioned aspects of Alloy;

(ii) User's interaction with the analyzer is sometimes predictable, e.g., that the user will compile and execute the model or that the user will ask for additional solutions to the model;

(iii) Users can naturally develop semantically equivalent models that have significantly different solving time.

Observation (i) points out that incremental constraint-solving techniques could improve the analyzer's performance. Alloy's use of SAT technology and recent advances in incremental SAT solvers [13] provide a natural way to start an exploration into optimizing the Alloy Analyzer. Observation (ii) points out that, similar to continuous compilation and continuous testing [10], the analyzer can continuously precompute the result of a future action while the user is editing the model or visually inspecting a solution. The first two observations shed light on how the Alloy tool-set may be improved. Observation (iii) shows that it would be worthwhile to study manual and automatic transformations of Alloy models that could result in improved execution time.

The rest of the paper is organized as follows. Section 2 presents an example interaction with the Alloy Analyzer. Section 3 presents how we collected and analyzed data from the students' interactions with the analyzer. Section 4 shows some preliminary results on improving the analyzer's performance using incremental solving and continuous execution. Section 5 concludes the paper.

2 Example

We next illustrate how users interact with the Alloy Analyzer to interactively develop a model. Through this example, we also introduce some key constructs of Alloy. More details of Alloy are available elsewhere [5]. As our running example, we use an interaction that a student had with the analyzer while solving a problem from the problem set. This problem considers modeling the abstract mathematical structure *tree*, i.e., a connected, acyclic, undirected graph. There are various (equivalent) definitions of a tree; we consider five definitions from a standard text-book [2]. Consider that we want to model them in Alloy to check their equivalence using the analyzer. In a typical scenario, the user of the analyzer starts from an empty model and develops it in the analyzer. To help the students, however, we provided a part of the model and asked them to provide the rest.

Let $G = (V, E)$ be an undirected graph, where V is a finite set (of vertices) and E is a binary relation on V. The following five statements are equivalent:

(i) G is a tree;

(ii) G is connected, but if any edge is removed from E, the resulting graph is disconnected;

(iii) G is connected, and $|E| = |V| - 1$;

(iv) G is acyclic, and $|E| = |V| - 1$;

(v) G is acyclic, but if any edge is added to E, the resulting graph has a cycle.

An Alloy model consists of *signature* declarations that introduce basic sets and relations, and of *formulas* that put constraints on these sets and relations. To model trees, we declare a `sig` (i.e., a set) of vertices and a binary relation on this set to represent the edges:

```
sig V {      // V is a set of vertices
  E: set V } // binary relation E: V <-> V for undirected edges
```

The cardinality operator `set` states that `E` is an arbitrary relation. (Operators `one` and `lone` respectively declare total and partial functions.) We represent an undirected edge from a vertex u to a vertex v as a pair of directed edges (u, v) and (v, u). Thus, `E` is a symmetric relation, which can be expressed using the transpose operator '`~`':

```
fact UndirectedGraph { E = ~E } // E is symmetric

fact NonEmpty { some V }        // consider non-empty graphs
```

A *fact* introduces a constraint that must be satisfied by any *instance* of the model, i.e., any satisfying assignment of values to sets and relations. The fact `NonEmpty` requires the instances to have at least one vertex[2]. The formula `some e` evaluates to true if the expression `e` evaluates to a non-empty set. (Similarly, `no e` evaluates to true when `e` evaluates to the empty set.)

We express Statement 1 using a *predicate*, i.e., a parameterized formula that can be *invoked* elsewhere:

```
pred Statement1() { Connected() && Acyclic() }

pred Connected() { all disj v1, v2: V | v1 in v2.^E }

pred Acyclic() { all v: V | not InCycle(v, E) }

pred InCycle(v: V, c: V -> V) { v in v.c ||
  some v': v.c | let c' = c - (v -> v') - (v' -> v) | v' in v.*c' }
```

Alloy provides the usual logical operators: '`&&`' (and), '`||`' (or), `not`, '`=>`' (implication), and '`<=>`' (bi-implication). The keyword `disj` requires `v1` and `v2` to be different (more precisely, disjoint singleton sets); `all` and `some` respectively represent universal and existential quantification; `in` represents subset (each expression is semantically a set [5], and thus `in` does not represent set membership); '`^`' denotes transitive closure, and '`*`' denotes reflexive transitive closure. The expression `v2.^E` thus denotes the set of all vertices reachable

[2] This condition is required for equivalence of Statements 1–5.

from `v2` following one or more traversals along the edges in `E`, and `Connected` states that there is a path between any two distinct vertices. The predicate `InCycle` states that a vertex `v` is a part of a cycle according to an edge relation `c` iff there is a self-loop at `v` or `v` has some neighbor `v'` such that even if we remove the edge(s) connecting `v` and `v'`, these two vertices are still connected. The operators '`->`' and '`-`' represent pairing (more generally, Cartesian product) and set difference, respectively.

Our problem set asked the students to extend the above Alloy model to represent each of statements 2–5.[3] The students also had to express the equivalence of the five statements in Alloy and to check them using the Alloy Analyzer. We next present the interaction that a student had with the analyzer to solve the above problem. We chose this particular interaction as a representative for the steps that the users typically perform while working with the Alloy Analyzer, going through a cycle of modifying the Alloy model and executing it.

The user first checked that the provided model is consistent:

```
assert Test { !Statement1() }

check Test for 3
```

An Alloy *assertion* introduces a formula that should be checked, in this case that `Statement1` does not hold. The command `check` instructs the analyzer to find a counterexample to the given assertion using the specified scope, specifically 3. The analyzer proceeds by looking for satisfying assignments to the negation of the formula.[4] Each such assignment effectively gives a valuation to the set `V` and the relation `E` to satisfy the negation of `Test` (and implicitly all `fact` formulas).

When the analyzer finds a satisfying assignment, it can visually present it (as a user-customizable graph, where nodes represent specific objects and relations are represented with edges). The user can also choose to generate more satisfying assignments for the given formula. (This option in the analyzer exploits solution enumeration in SAT solvers such as mChaff [8] and relsat [1].)

The user further formulated Statement 3 and checked its equivalence with Statement 1:

```
pred Statement3() { Connected() && #E = #V - 1 }

assert Test { Statement1() <=> Statement3() }

check Test for 3
```

[3] More precisely, the problem also asked the students to write the predicate `Connected`.
[4] Besides `check`, the Alloy Analyzer also provides a command `run` that directly finds satisfying assignments for a given formula; `run Statement1 for 3` is equivalent to the above and would avoid the double negation, but the student likely forgot about the command `run`.

`Statement3` uses the set cardinality operator '`#`' to (incorrectly) represent the constraint $|E| = |V| - 1$ from the definitions. The Alloy Analyzer finds a counterexample for the above formula. The issue is that our Alloy model represents each undirected edge using two directed edges. Note that the counterexample would not have been found in the scope of one, which would allow only one element in the set `V`. Users typically start checking with the scope of three or four: smaller values can miss many counterexamples, and larger values lead to large execution time.

The user quickly realized the mistake and corrected the formula:

```
pred Statement3() { Connected() && #E = #V + #V - 2 }
```

This simple step illustrates the power of the analyzer: the users can automatically check the correctness of their models (within the given scope). Quick gaining of feedback helps users to correct their models while developing them. Indeed, it is the full automation of the execution that encourages the users to interactively develop the models in small steps and with frequent executions.

The user next used `check Test for 10` to check the model within the scope of ten. Although `Statement1` and (corrected) `Statement3` are equivalent for all graphs with up to three nodes, on the evidence so far, they may be non-equivalent for larger graphs. This increase in the bound from three to ten is somewhat unusual; users typically increase the value for one or two.

The user then added Statement 4:

```
pred EV1() { # E = #V + #V - 2 }

pred Statement3() { Connected() && EV1() }

pred Statement4() { Acyclic() && EV1() }

assert Test { Statement1() => Statement3()
              Statement3() => Statement4()
              Statement4() => Statement1() }

check Test for 4
```

Note that the user realized that the equivalence of several statements can be expressed using a circular implication.[5] (The lines without '`&&`' or any other connective are implicitly conjoined, so the three implications in `Test` are conjoined.) This check revealed no counterexample, so the user increased the scope from 4 to 5.

The user then proceeded by adding statements 2 and 5 and after a few more checks arrived at the following:

```
// connected but removing an edge makes it disconnected
pred Statement2() {
```

[5] That is also how the equivalence is proved in text-books.

```
  Connected()
  no E or
    all v1, v2: V | (v1 -> v2) in E =>
      let E' = E - (v1 -> v2) - (v2 -> v1) |
        some disj v3, v4: V | not v3 in v4.^E' }

pred Statement5() { // acyclic but if any edge is added, cyclic
  Acyclic()
  all v1, v2: V | not (v1 -> v2) in E implies
    let E' = E + (v1 -> v2) + (v2 -> v1) |
      some v: V | InCycle(v, E') }

assert Test { Statement1() => Statement2()
              Statement2() => Statement3()
              Statement3() => Statement4()
              Statement4() => Statement5()
              Statement5() => Statement1() }
```

The final model includes also the latest definition of the above formulas.

Using the Berkmin SAT solver [3], the Alloy Analyzer checks the final assertion for all graphs with up to 4 vertices and reports no counterexamples. The SAT solver completes its search in 4.1 seconds on a Pentium M 1.8GHz processor.

3 Study

This section presents the study that we performed to analyze how (beginner) users interact with the Alloy Analyzer. We first describe our experimental setup. We then present how we modified the analyzer to log its interaction. We next discuss an analysis of the resulting logs. We finally show that equivalent Alloy models can require significantly different solving times.

3.1 Experimental Setup

We collected the logs from the graduate students working on problem sets in two graduate seminars at the University of Texas at Austin and the University of Illinois at Urbana-Champaign. The students had no experience with Alloy prior to the classes but were given about two and a half lectures on Alloy. The problem set consisted of two problems. One problem was our running example on modeling tree definitions and checking their equivalence. The other problem was to model and solve a puzzle [14], given in English, to assign eight different jobs to four people subject to a list of constraints.

We told the students how to enable the analyzer to collect the logs of their model developments. We also told them that we may use the (anonymized) models that they develop as case studies in an investigation of how users work with the analyzer and how to develop *incremental* techniques to provide faster

solving. We did not tell the students the specific experiments that we wanted to perform. Submission of logs was voluntary and did not affect the student's grade, either positively or negatively. Their solutions were graded only based on the final models that they sent.

3.2 Logging

We design our logging facility to provide the Alloy developers with usage data that may help further improve the Alloy tool-set. The current logging facility logs compilation, execution, and user-interface events. All the information required to replay an event is stored, together with the time stamps that record when the event begins and ends. Besides the time stamps, the information stored includes the configuration of the Alloy Analyzer and the SAT solver, the source file being compiled (and any source files referenced), and the string representation of the command. This paper focuses on two types of events: (1) compile, which record how users compile a model, and (2) execute, which record how users executes commands after a successful compilation.

Although this paper only uses two types of events, our logging facility records other usage data that might improve further understanding of the usage pattern of the Alloy Analyzer. For example, the user interface events may help to streamline the workflow of the analyzer.

3.3 Analysis

We next present our analysis of the logs collected in our two classes. In total, we collected logs from 11 students. (Many students either worked offline or used logging incorrectly and thus didn't provide us with useful logs.) There were a total of 68 UI sessions and 2308 compilations in these logs. Of these compilations, 391 (or 16.9%) failed with compile errors and 452 (or 19.5%) were successful compilations but without any execution. Unfortunately, our logging did not record the models of failed compilations as we had not expected them to be useful for improving the Alloy Analyzer. However, their relatively large percentage suggests that the beginner users may have problems learning some constructs of Alloy. We plan to record failed models in the future; it would be interesting to analyze them to identify potential improvements in the language or its documentation to avoid common mistakes. Apparently, the users quickly learned how to deal with the compile errors: the users started compiling models more frequently to catch errors early in the recently changed parts of the models.

We next analyze the 1465 logged successful compilations and executions to detect what changes the users made to the models between the consecutive executions. We call a change an *event*. We first introduce the types of events that our analysis detects. We then describe how our analysis detects these events using a level of syntactic and semantic comparisons. (These are not full syntactic and semantic comparisons, as explained later.) We finally present

the analysis results.

3.3.1 Events

Recall that Alloy models consist of signatures (which correspond to data in programs), formulas (which correspond to code in programs), and commands (which correspond to the inputs in program runs). An important part of the command is specifying the scope, i.e., the bounds for the basic sets in the model. We define the following events to track the changes in the parts of the model:

- For signatures: SN (sig_new) adding a new sig; SD (sig_delete) deleting a sig; SM (sig_mod) modifying an existing sig.
- For formulas: FN (formula_new) adding a new formula; FD (formula_delete) deleting a formula; FM (formula_mod) modifying an existing formula.
- For commands: CN (command_new) adding a new command; CD (command_delete) deleting a command; CM (command_mod) modifying an existing command.
- For scope: OS (only_scope) the only change in the model is changing the scope in the command; ND (non_decrementing) the scope was increased only; OO (one_one) only one bound was increased for exactly one.
- Summary events: SS (single_sig) only one sig was changed in the model; SF (single_formula) only one formula was changed in the model; CR (consecutive_repeat) two *consecutive* executions have the same signatures, formulas, command, and scope; ER (execution_repeat) an execution is repeated but not necessarily consecutively.

We have written a program that traverses a given log (or a set of logs) and counts the number of events. The program proceeds as follows. It first removes from the model semantically unnecessary syntactic elements such as comments and white spaces. It then parses parts of the model and uses two types of comparisons: syntactic comparison (for SN, SD, SM, FN, FD, FM, CN, CD, CM, SS, and SF events) and semantic comparison (for OS, ND, OO, CR, and ER events).

3.3.2 Syntactic Comparison

The syntactic comparison in our program uses the concept of *text similarity*. The program splits each semantic unit—signature, formula, command—into two parts, the declaration of the unit and the content of the unit. The program compares these units separately. For example, the following sig definition:

```
one sig V extends W {
  E: set V }
```

is represented as a 3-element tuple (`V`, `E: set V`, `one, extends W`), where `one` and `extends` are Alloy keywords that specify singleton sets and subsets, respectively. The similarity of two signatures is then a weighted sum of the

similarities of the three components; our current implementation uses equal weights for these three components.

Our program uses the *edit distance* [9,12] as the metric for similarity between two component strings. The edit distance between two strings is the number of keystrokes required to change one string to the other string. We use the edit distance as our goal is to find what the user changed between consecutive executions. The edit distance is normalized to the lengths of the two strings. When the edit distance is 0, it means the two strings are identical. When the edit distance is 1, it means the two strings are totally different. The values between 0 and 1 represent the similarity of the two strings. The lower the edit distance, the higher the similarity. For example, when the edit distance is 0.5, we need to change about 50% of the one string to get the other string.

An advantage of syntactic comparison is that it traces certain changes more closely than the semantic comparison. For example, if the user changes the definition from `sig V` to `one sig V`, the internal Alloy representations for the two versions are quite different, while syntactic comparison can easily find that the user just changed a single sig definition. However, the limitation of syntactic comparison is that it requires further parsing when difference in smaller granularity is desired, for example to detect if only scopes are changed in the commands.

We can define two semantic units to be equal, modified, or different based on the similarity between their components. Two semantic units are equal if the edit distance between them is 0. Our program uses an empirically selected threshold of 25% to determine if two units are modified versions or simply different. (We determined the threshold by a detailed manual inspection of comparisons for several randomly selected examples.) If the edit distance is below the threshold, the units are treated as modified versions. Otherwise, they are different. Our threshold is pretty high such that the confidence of counting two versions as modified is high, i.e., the two versions have only minor differences. In other words, the data shown in Figure 1 slightly underestimates the true number of modification cases and thus the potential that incremental solving can bring to the Alloy Analyzer.

3.3.3 Semantic Comparison

The semantic comparison in our program parses the model and performs a level of semantic analysis to detect the changes that the user made. Specifically, our program detects, based on the scope, the bound for each basic signature. Our running example with trees had only one signature, `V`, but in general there can be several signatures in the model. The scope is then a vector of the bounds for each of these signatures. The user can specify the scope in several ways in the commands. The full details are elsewhere [5], and we provide here only a few examples: `check Eq for 5` specifies that the bounds for all signatures should be five, `check Eq for 4 V` specifies that `V` should have bound four while other

ses.	SN	SD	SM	FN	FD	FM	CN	CD	CM	SS	SF	OS	ND	OO	CR	ER	#C
1	0	0	0	19	7	50	20	18	23	0	47	17	12	5	3	14	92
2	0	0	0	2	2	21	8	9	14	0	23	13	9	4	4	16	50
3	0	0	0	1	1	34	14	11	3	0	25	5	4	1	1	8	48
4	0	0	0	0	0	18	7	7	5	0	18	4	4	2	1	5	35
5	0	0	0	0	0	19	10	13	11	0	19	8	6	5	1	6	43
6	0	0	1	0	0	34	0	0	0	1	34	6	6	0	6	9	41
7	0	0	0	1	0	23	2	2	20	0	18	4	4	1	0	4	36
8	0	0	0	2	0	35	8	2	12	0	19	5	4	0	2	4	39
9	7	8	3	25	17	35	5	4	16	3	38	10	6	2	1	3	60
10	0	0	0	0	0	48	6	6	10	0	42	9	5	0	0	0	57
11	0	0	0	2	1	42	6	6	18	0	36	18	12	5	1	1	63
12	0	0	0	1	0	35	9	9	20	0	28	18	13	6	2	3	57
13	0	0	0	0	1	21	9	9	6	0	20	4	2	1	1	1	34
14	0	0	0	3	2	25	4	2	3	0	28	4	2	0	1	6	34
15	0	0	0	0	0	29	1	0	17	0	25	12	10	6	0	0	43
16	8	13	0	32	22	29	14	9	12	0	32	5	5	0	3	7	52
17	0	0	0	1	0	38	0	0	3	0	26	8	7	0	2	4	37
18	2	8	2	11	2	58	5	5	15	2	57	9	6	0	2	20	76
19	19	11	8	31	15	17	7	7	4	4	27	6	6	0	0	0	42
sum	36	40	14	131	70	611	135	119	212	10	562	165	123	38	31	111	939

Fig. 1. Number of changing events the users performed while modifying the models during 19 UI sessions.

signatures should have the default values (currently three), and `check Eq for 5 but 3 V` specifies that all signatures should have bound five except that `V` should be three. Our program analyzes the command and the signatures to build the entire scope vector. It then compares these scope vectors between different executions of Alloy models.

3.3.4 Number of Events

Figure 1 shows the number of events that the users performed while changing the Alloy models. For each of the 19 UI sessions with most compilations, we tabulate the total number of events performed. In 59.9% (562/939) of the cases, the modifications between two consecutive executions involve only one formula, and 17.6% (165/939) of the consecutive executions differ only in their scopes. These numbers highlight the importance of incremental solving in the Alloy Analyzer. Moreover, 11.8% (111/939) of all model executions are identical to some previous execution in the same UI session. (While similar consecutive models are akin to spatial locality, repeated models are akin to temporal locality.) This suggests that the analyzer could cache the results of executions and compare each new model with the previously executed models.

3.4 Equivalent Models, Different Performance

We next show that semantically equivalent but syntactically different Alloy models can require significantly different solving times. While it is clear that in (almost) any reasoning system the solving time depends on the specific formulation of the problem, our results show that beginner Alloy users naturally create models that take different solving time. Our result thus provides

evidence against the claim made by Sullivan et al. [11][page 140]:

> TestEra, because it employs the Alloy Analyzer's translation to SAT, is largely insensitive to the constraint's logical structure.

While one could artificially construct equivalent Alloy models that have different solving times, we consider the models that the students actually submitted as solutions to the problem set. Specifically, we consider the solutions submitted for the problem on tree equivalence, used as our running example.

Recall that the problem asked the students to model five definitions of a tree and to express their equivalence. While the students came up with several different formulas to express the definitions, they also came up with five different formulas to express the equivalence. This was much to our surprise, as we expected that the students may use only two or three formulas to express equivalence. This diversity points out that the students likely did not copy the solutions. More seriously, the diversity shows how beginning users can surprise expert users (or even tool developers) by using the tools in a way that was not anticipated. Finally, the diversity provides an additional motivation to understand how users work with the Alloy Analyzer.

To present the students' approaches to checking equivalence, we use $\texttt{S}_i$ to stand for $\texttt{Statement}_i\texttt{()}$, where $1 \leq i \leq 5$. The students used four different formulas for equivalence:

```
assert Eq1 { S1 => S2 && S2 => S3 && S3 => S4 && S4 => S5 &&
             S5 => S1 }
assert Eq2 { S1 <=> S2 && S1 <=> S3 && S1 <=> S4 && S1 <=> S5 }
assert Eq3 { S1 <=> S2 && S2 <=> S3 && S3 <=> S4 && S4 <=> S5 &&
             S5 <=> S1 }
assert Eq4 { S1 <=> S2 && S1 <=> S3 && S1 <=> S4 && S1 <=> S5 &&
             S2 <=> S3 && S2 <=> S4 && S2 <=> S5 &&
             S3 <=> S4 && S3 <=> S5 &&
             S4 <=> S5 }
```

One student used a rather different approach for checking equivalence; instead of representing the equivalence of all statements in one formula, the student used four formulas:

```
//Uncomment one line at a time to check equivalence.
//assert Eq5 { S1 <=> S2 }
//assert Eq5 { S1 <=> S3 }
//assert Eq5 { S1 <=> S4 }
assert Eq5 { S1 <=> S5 }
```

We next compare the performance of the analyzer for checking the above assertions of equivalence. Note also that all those assertions are equivalent among themselves; as a matter of fact, they are all equivalent to true. Thus, the negation of the assertions is unsatisfiable, and the analyzer cannot find any solution for the negation (and a counterexample for the formula). We use each assertion with the same model for expressing tree definitions and check

the assertion for the scope of four. For `Eq5`, we check all four assertions separately and sum all four times. Checking for scope 4, the Alloy Analyzer takes 12.6 seconds for `Eq1`, 10.7 seconds for `Eq2`, 16.2 seconds for `Eq3`, 28.6 seconds for `Eq4`, and 20.1 seconds for (all four) `Eq5`. Thus, the best formulation gives a 2.67X speed-up in the solving time over the worst formulation. When we check the same assertions for scope 5, the speed-up increases to 5.75X.

This result points out that the users should be made aware that different models can result in greatly different solving time. Actually, expert Alloy users gain this through experience and do rewrite their models in order to speed up the execution. We leave it as a future work to study these rewrites to generate a set of guidelines for (re)writing the models to obtain efficient executions.

4 Potential Improvements

This section shows some *preliminary* results that illustrate potential for improving the Alloy Analyzer's performance. We present two types of improvements: (i) improvements that can be obtained by using *incremental SAT solving* to speed the execution of (similar) consecutive models, and (ii) improvements that can be obtained by computing some results while the user is editing the model or visually inspecting a solution.

4.1 Incremental Solving

As shown earlier, Alloy users often execute similar models one after the other. This leads us to consider the use of incremental SAT solvers to improve the analyzer's execution time. Incremental SAT solvers work as follows [13]. They first take one SAT problem, as usual, and find whether it is satisfiable or not. Additionally, they track how the inference steps that they perform depend on the input clauses. After that, the next SAT problem can be presented to the solver not from scratch, but as a *delta* from the previous SAT problem, which describes what old clauses to remove and what new clauses to add. The solver can then use this information to invalidate the inference steps that depended on the removed clauses and to perform the search only for the new clauses. Solving only the delta often results in a much improved SAT solving time, compared to the SAT solving time of the new problem from scratch.

We present results that exploit incremental SAT solving in two steps performed with the Alloy Analyzer: (i) adding only one fact to the model and (ii) increasing the scope by exactly one. These are only preliminary results for evaluating the potential of incremental execution. To obtain the exact results, we would need to modify the entire analyzer to perform incremental execution.

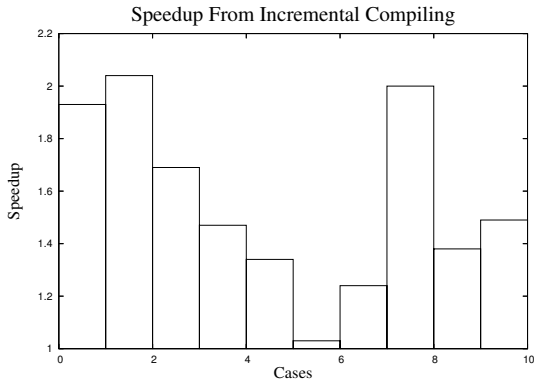

Fig. 2. Speed-up of incremental execution for consecutive models that differ only in addition of one fact.

4.1.1 Adding a Fact

Alloy users can add constraints to their model by adding new facts. A user may add a fact while building the model or while tuning the generation of instances (or counterexamples). For example, recall the following assertion from Section 2:

```
assert Test { !Statement1() }
check Test for 3
```

The analyzer's execution generates a tree with one vertex and no edges. Assume that we instead want to see larger trees, say a tree with exactly two vertices. We can add the following fact to express this requirement:

```
fact { #V = 2 }
```

Re-running the analyzer at present requires translating the modified Alloy model in its entirety into a CNF formula and then solving the formula. Using an incremental SAT solver instead, we can simply provide the solver the delta using an incremental translation that produces only the boolean formula that represents the new constraint. Doing so not only allows the solver to generate a desired tree more efficiently but also eliminates the need to translate the whole Alloy model into CNF.

In our experiments, we modified the Alloy compiler so that the compiler can translate the delta fact into the delta boolean formula. By comparing two consecutive models, we can find the new fact in the later model. Our modified compiler renames this fact such that the original Alloy compiler generates the CNF formula only for the new fact. Our modifier compiler also instructs the SAT solver to reuse the solving trace of the previous model. The translation between the previous model and the boolean variables is maintained when generating the delta boolean formula, so that the user can visualize how the new fact affects the original solution. Our modification ensures the correctness of delta solutions, i.e., we improve the performance by reusing the previous solution, but the new solution satisfies both the models with and without the delta fact.

Figure 2 shows the performance gain of using our incremental compiling. We present 10 cases from the logs where the user only adds one fact to a model between two consecutive executions. These 10 cases have significant solving time for the original model and thus the speed-up can be observed; if the original model takes little solving time, we do not need incremental solving. We measure the time to solve the original model (t_{orig}), the time to solve the later model with the delta fact in its entirety ($t_{orig+delta}$), and the time to solve the later model incrementally, i.e., to solve only the delta fact (t_{delta}). The speed-up is given by $\frac{t_{orig+delta}+t_{orig}}{t_{delta}+t_{orig}}$. The speed-up of incremental compiling ranges from 1.03X to 2.04X with an average of 1.56X. Note that we consider the sum of times for two executions in both cases; the speed-up of just the second execution is $\frac{t_{orig+delta}}{t_{delta}}$, which is even higher.

4.1.2 Increasing Scope

Incremental solving need not be performed only using incremental SAT solvers. The analyzer's result from a previous execution can be used instead to re-write the current model so that it induces faster execution even when the whole model is re-compiled and the SAT solver is executed from the beginning.

Consider checking an assertion with the analyzer. If it fails to find a counterexample, Alloy users are likely to increase their level of confidence in their model by increasing the scope and re-executing the analyzer, as illustrated by following consecutive executions from a student log:

(i) `check Test for 4`

(ii) `check Test for 5`

Since Alloy's execution is *scope monotonic* (i.e., if the analyzer fails to find a counterexample using scope i, no counterexample exists for scope $j \leq i$), when the user executes the check for scope i after executing the same check for scope $i-1$, we can use the fact that no counterexample was found during the first execution to direct the analyzer to check for *exactly* i vertices. To illustrate, the time to check the equivalence formulation in Section 2 for scope 5 reduces from 8 min 43 sec to 8 min 24 sec. Although in this case the improvement is only 3.5%, we believe that a better technique could yield higher speed-ups.

The logs show that 17.6% (165/939) of all executions only increase the scopes compared with the previous execution. Of those, 23.0% (38/165) commands increase the scope only by one on exactly one signature. The data indicates that users frequently increase scopes, usually with small step. Thus, incremental solving for increasing scope might improve the performance of the Alloy Analyzer.

4.2 Continuous Execution

Continuous compilation is a method for reducing the latency time in Integrated Development Environments: while the programmer is editing the pro-

gram, the machine is compiling it in the background, and thus when the programmer wants to execute the program, it is already compiled. A similar technique has been recently proposed for testing. Continuous testing [10] runs the unit tests in the background as the programmer is editing the program; if a test fails, the programmer is warned that his recent change may be breaking some regression tests.

We propose to use a similar approach in the Alloy Analyzer: it can continuously execute the model to prepare the results that the user will (likely) ask for next. One situation in which this naturally applies is while the user is editing a model. The analyzer can then be translating the model into SAT and running it on the underlying SAT solver. After asking for a solution, the user would then be presented with it faster. Another, somewhat surprising, situation where continuous solving applies is while the user is visually inspecting one solution for a model: the analyzer can then be instructing the SAT solver to generate the next solution in the background. Our results show that (beginner) users are not very likely to check for the next solution, but if they do check the next solution, they tend to repeat this operation a few times in a row. Repeatedly looking at the next solution is something that we have anecdotally observed in expert Alloy users as well.

We next present results that estimate the decrease in the latency for getting next solution with and without continuous execution. We cannot obtain the precise result, because our logs do not record the entire user's interaction with the Alloy Analyzer and thus we do not know the precise time when the users performed all actions. Our logs record when a user begins checking for the next solution and when the SAT solver returns the result. We use $begin_i$ to denote the beginning time of the ith checking and end_i for the time when the results are returned to the user. Thus, $begin_{i+1} - end_i$ is the period when the user inspects the ith solution, which is also the potential decrease in the latency for getting the $i + 1$th solution if we can overlap the computation for the $i + 1$th solution and the user inspection.

We examined 84 cases in which the users checked for the next solution in the logs. On average, the users spent 8 seconds examining the returned solution and waited less than 1 second for the SAT solver to return the next solution. In all 84 cases, the time that the users spent on visual inspection was longer than the time that the SAT solver took to generate the next solution. If the analyzer instructs the SAT solver to search for the next solution immediately after the previous solution is returned, the user can get the next solution instantaneously when the next solution is desired.

5 Conclusions and Future Work

We have presented an analysis of the use of the Alloy Analyzer, a tool for automatic checking of software models written in Alloy, a first-order, declarative language. Although there has been a lot of prior work on Alloy, there

has been no study of how users interact with the analyzer. We analyzed the interactions that 11 graduate students had with the tool while developing two models for a problem set. Our results show that: (i) users often perform consecutive executions with slightly different models, and thus incremental checking could speed up the interaction; (ii) users' interaction with the analyzer are sometimes predictable, and the analyzer can precompute the result of a future action while the user is editing the model; and (iii) (beginner) users can naturally develop semantically equivalent models that have significantly different execution time, and it is worthwhile to study model transformations that can improve execution time.

Our results provide an encouraging starting point for the further analysis of the Alloy Analyzer. We are planning to collect more logs and analyze them to detect potential further improvements for the analyzer. Implementing full incremental execution and continuous execution in the analyzer would be an important step in realizing the potential improvements. More generally, the Alloy Analyzer is only one example tool used in software development. We believe that studies of tool usage are important, and we are planning to explore usage of other tools in the future.

Acknowledgments

We would like to thank the 11 students from our classes for submitting the logs of their interactions with the Alloy Analyzer. We would also like to thank anonymous reviewers for their comments on a previous version of this paper. We are thankful to Derek Rayside and Greg Dennis for their help with Alloy logging.

References

[1] Bayardo Jr., R. J. and R. C. Schrag, *Using CSP look-back techniques to solve real world SAT instances*, in: *Proceedings of the National Conference on Artificial Intelligence*, 1997, pp. 203–208.

[2] Cormen, T. H., C. E. Leiserson and R. L. Rivest, "Introduction to Algorithms," The MIT Press, Cambridge, MA, 1990.

[3] Goldberg, E. and Y. Novikov, *BerkMin: A fast and robust SAT-solver*, in: *Proceedings of the Design, Automation, and Test in Europe (DATE)*, 2002, pp. 142–149.

[4] Jackson, D., *Micromodels of software: Modelling and analysis with Alloy* (2001), http://sdg.lcs.mit.edu/alloy/book.pdf.

[5] Jackson, D., "Software Abstractions: Logic, Language, and Analysis," The MIT Press, Cambridge, MA, 2006.

[6] Jackson, D., I. Schechter and I. Shlyakhter, *ALCOA: The Alloy constraint analyzer*, in: *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, 2000, pp. 730–733.

[7] Khurshid, S. and D. Jackson, *Exploring the design of an intentional naming scheme with an automatic constraint analyzer*, in: *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE)*, Grenoble, France, 2000, pp. 13–22.

[8] Moskewicz, M. W., C. F. Madigan, Y. Zhao, L. Zhang and S. Malik, *Chaff: Engineering an efficient SAT solver*, in: *Proceedings of the 39th Design Automation Conference (DAC)*, 2001, pp. 530–535.

[9] Myers, E. W., *An O(ND) difference algorithm and its variations*, Algorithmica **1** (1986), pp. 251–266.

[10] Saff, D. and M. D. Ernst, *An experimental evaluation of continuous testing during development*, in: *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Boston, MA, 2004, pp. 76–85.

[11] Sullivan, K., J. Yang, D. Coppit, S. Khurshid and D. Jackson, *Software assurance by bounded exhaustive testing*, in: *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Boston, MA, 2004, pp. 133–142.

[12] Ukkonen, E., *Algorithms for approximate string matching*, Information and Control **64** (1985), pp. 100–118.

[13] Whittemore, J., J. Kim and K. Sakallah, *SATIRE: A new incremental satisfiability engine*, in: *Proceedings of the 38th Conference on Design Automation (DAC)*, Las Vegas, NV, 2001, pp. 542–545.

[14] Wos, L., R. Overbeek, E. Lusk and J. Boyle, "Automated Reasoning (2nd ed.): Introduction and Applications," McGraw-Hill, Inc., New York, NY, USA, 1992.