Incremental Analysis of Evolving Alloy Models

Wenxi Wang[1](✉), Kaiyuan Wang[2](✉), Milos Gligoric[1](✉), and Sarfraz Khurshid[1](✉)

[1] The University of Texas at Austin, Austin, USA
{wenxiw,gligoric,khurshid}@utexas.edu
[2] Google Inc., Sunnyvale, USA
kaiyuanw@google.com

Abstract. Alloy is a well-known tool-set for building and analyzing software designs and models. Alloy's key strengths are its intuitive notation based on relational logic, and its powerful analysis engine backed by propositional satisfiability (SAT) solvers to help users find subtle design flaws. However, scaling the analysis to the designs of real-world systems remains an important technical challenge. This paper introduces a new approach, iAlloy, for more efficient analysis of Alloy models. Our key insight is that users often make small and frequent changes and repeatedly run the analyzer when developing Alloy models, and the development cost can be reduced with the incremental analysis over these changes. iAlloy is based on two techniques – a static technique based on a lightweight *impact* analysis and a dynamic technique based on solution *re-use* – which in many cases helps avoid potential costly SAT solving. Experimental results show that iAlloy significantly outperforms Alloy analyzer in the analysis of evolving Alloy models with more than 50% reduction in SAT solver calls on average, and up to 7x speedup.

1 Introduction

Building software models and analyzing them play an important role in the development of more reliable systems. However, as the complexity of the modeled systems increases, both the cost of creating the models and the complexity of analyzing these models become high [24].

Our focus in this paper is to reduce the cost of analyzing models written in Alloy [5] – a relational, first-order logic with transitive closure. The Alloy analyzer provides automatic analysis of Alloy models. To analyze the model, the user writes Alloy *paragraphs* (e.g., signatures, predicates, functions, facts and assertions), and the analyzer executes the *commands* that define constraint solving problems. The analyzer translates the commands and related Alloy paragraphs into propositional satisfiability (SAT) formulas and then solves them using off-the-shelf SAT solvers. We focus on successive runs of the analyzer as the model undergoes development and modifications. The key insight is that during model development and validation phases, the user typically makes many changes that

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 174–191, 2019.
https://doi.org/10.1007/978-3-030-17462-0_10

are relatively small, which enables the incremental analysis to reduce the subsequent analysis cost [1].

We introduce a novel technique called iAlloy that incrementally computes the analysis results. iAlloy introduces a two-fold optimization for Alloy analyzer. Firstly, iAlloy comes with a *static* technique that computes the *impact* of a change on commands based on a lightweight dependency analysis, and *selects* for execution a subset of commands that may be impacted. We call this technique *regression command selection* (RCS), since it shares the spirit of regression test selection for imperative code [4] and adapts it to declarative models in Alloy. Secondly, iAlloy comes with a *dynamic* technique that uses memoization to enable *solution reuse* (SR) by efficiently checking if an existing solution already works for a command that must be executed. SR uses a partial-order based on sets of parameters in predicate paragraphs to enable effective re-use of solutions across different commands.

To evaluate iAlloy we conduct experiments using two sets of Alloy models that have multiple versions. One set, termed *mutant version set*, uses simulated evolving Alloy models where different versions are created using the MuAlloy [21,27] tool for generating *mutants* with small syntactic modifications of the given base Alloy models. This set includes 24 base Alloy models and 5 mutant versions for each base model. The other set, termed *real version set*, uses base Alloy models that had real faults and were repaired using the ARepair [25,26] tool for fixing faulty Alloy models. For each faulty base model, its evolution is the corresponding fixed model. This set includes 36 base Alloy models and 2 versions for each model.

The experimental results show that iAlloy is effective at reducing the overall analysis cost for both sets of subject models. Overall, iAlloy provides more than 50% command execution reduction on average, and up to 7x speed up. In addition, SR performs surprisingly well in the real version set with 58.3% reduction of the selected commands, which indicates that our approach is promising for incrementally analyzing real-world evolving Alloy models.

This paper makes the following contributions:

- **Approach.** We introduce a novel approach, iAlloy, based on static analysis (regression command selection) and dynamic analysis (solution re-use) for incrementally analyzing evolving Alloy models, and embody the approach as a prototype tool on top of the Alloy analyzer.
- **Evaluation.** We conduct an extensive experimental evaluation of our approach using two sets of subject Alloy models, one based on syntactic mutation changes and the other based on fault fixing changes. The results show that iAlloy performs well on both sets.
- **Dataset.** We publicly release our subject Alloy models and their versions at the following URL: https://github.com/wenxiwang/iAlloy-dataset. Given the lack of common availability of Alloy models with evolution history, we believe that our dataset will be particularly useful for other researchers who want to evaluate their incremental analysis techniques for Alloy.

While our focus in this paper is the Alloy modeling language and tool-set, we believe our technique can generalize to optimize analysis for models in other declarative languages, e.g., Z [17] and OCL [2].

2 Background

In this section, we first introduce Alloy [5] based on an example which we use through the paper. Then, we describe MuAlloy [21,27] – a mutation testing framework for Alloy, which we apply to create different versions of an Alloy model to simulate model evolutions. Finally, we briefly describe regression test selection (RTS) for imperative code. Although our regression command selection (RCS) applies to declarative code, the two methods share similar ideas.

2.1 Alloy

Alloy [5] is a declarative language for lightweight modeling and software analysis. The language is based on first-order logic with transitive closure. Alloy comes with an analyzer which is able to perform a bounded exhaustive analysis. The input of the Alloy analyzer is an Alloy model that describes the system properties. The analyzer translates the model into conjunctive normal form (CNF) and invokes an off-the-shelf SAT solver to search for solutions, i.e., boolean instances. The boolean instances are then mapped back to Alloy level instances and displayed to the end user.

Figure 1 shows the Dijkstra Alloy model which illustrates how mutexes are grabbed and released by processes, and how Dijkstra's mutex ordering constraint can prevent deadlocks. This model comes with the standard Alloy distribution (version 4.2). An Alloy model consists of a set of *relations* (e.g., signatures, fields and variables) and constraints (e.g., predicates, facts and assertions) which we call *paragraphs*. A signature (`sig`) defines a set of atoms, and is the main data type specified in Alloy. The running example defines 3 signatures (lines 3–6), namely `Process`, `Mutex` and `State`.

Facts (`fact`) are formulas that take no arguments and define constraints that must be satisfied by every instance that exists. The formulas can be further structured using predicates (`pred`) and functions (`fun`) which are parameterized formulas that can be invoked. Users can use Alloy's built-in `run` command to invoke a predicate and the Alloy analyzer either returns an instance if the predicate is satisfiable or reports that the predicate is unsatisfiable. The `IsStalled` predicate (lines 12–14) is invoked by the `GrabMutex` predicate (line 16) and the `run` command (line 53). The parameters of the `IsStalled` predicate are `s` and `p` with signature types `State` and `Process`, respectively. An assertion (`assert`) is also a boolean formula that can be invoked by the built-in `check` command to check if any counter example can refute the asserted formula. Assertions does not take any parameter. The `DijkstraPreventsDeadlocks` assertion (lines 45–47) is invoked by the `check` command (line 60) with a scope of up to `6` atoms for each signature.

2.2 MuAlloy

MuAlloy [21,27] automatically generates mutants and filters out mutants that are semantically equivalent to the original base model. Table 1 shows the mutation operators supported in MuAlloy. *MOR* mutates signature multiplicity,

```
open util/ordering [State] as so
open util/ordering [Mutex] as mo
sig Process {}
sig Mutex {}
sig State { holds, waits: Process -> Mutex }
pred Initial [s: State] {
  no (s.holds + s.waits)
}
pred IsFree [s: State, m: Mutex] {
  no m.~(s.holds) // no process holds this mutex
}
pred IsStalled [s: State, p: Process] {
  some p.(s.waits)
}
pred GrabMutex [s: State, p: Process, m: Mutex, s': State] {
  !s.IsStalled[p] // a process can only act if it is not waiting for a mutex
  m !in p.(s.holds) // can only grab a mutex that is not yet hold
  all m': p.(s.holds) | mo/lt[m',m] // mutexes must be grabbed in order
  s.IsFree[m] => {
    p.(s'.holds) = p.(s.holds) + m // if the mutex is free, the process now holds it
    no p.(s'.waits) // the process is not stalled any more
  } else {
    p.(s'.holds) = p.(s.holds) // if the mutex is not free, the process still hold the same mutexes.
    p.(s'.waits) = m // and wait on the new mutex.
  }
  all otherProc: Process - p | { // other processes maintain the same state
    otherProc.(s'.holds) = otherProc.(s.holds)
    otherProc.(s'.waits) = otherProc.(s.waits)
  }
}
pred ReleaseMutex [s: State, p: Process, m: Mutex, s': State] {
  !s.IsStalled[p]
  ...
}
pred GrabOrRelease  {
  Initial[so/first] &&
  (all pre: State - so/last | let post =  so/next[pre] |  // for every pre and post state
  (post.holds = pre.holds && post.waits = pre.waits) ||  // either nothing happens
  (some p: Process, m: Mutex | pre.GrabMutex [p, m, post]) ||  // or a process grabs a mutex
  (some p: Process, m: Mutex | pre.ReleaseMutex [p, m, post]))  // or releases a mutex
}
pred Deadlock  {
  ...
}
assert DijkstraPreventsDeadlocks {
  GrabOrRelease => ! Deadlock
}
pred ShowDijkstra  {
 GrabOrRelease && Deadlock
 some waits
}
run Initial for 10
run IsStalled for 10
run IsFree for 10
run GrabMutex for 30
run ReleaseMutex for 35
run GrabOrRelease for 16
run Deadlock for 50 expect 1
run ShowDijkstra for 5 expect 1
check DijkstraPreventsDeadlocks for 6 expect 0
```

Fig. 1. Dijkstra Alloy model from standard Alloy distribution (version 4.2); the line written in red was absent from the faulty version

e.g., `lone sig` to `one sig`. *QOR* mutates quantifiers, e.g., `all` to `some`. *UOR*, *BOR* and *LOR* define operator replacement for unary, binary and formula list operators, respectively. For example, *UOR* mutates `a.*b` to `a.^b`; *BOR* mutates `a=>b` to `a<=>b`; and *LOR* mutates `a&&b` to `a||b`. *UOI* inserts an unary operator before expressions, e.g., `a.b` to `a.~b`. *UOD* deletes an unary operator, e.g., `a.* ~b` to `a.*b`.

Table 1. Mutation Operators Supported in MuAlloy

Mutation Operator	Description
MOR	Multiplicity Operator Replacement
QOR	Quantifier Operator Replacement
UOR	Unary Operator Replacement
BOR	Binary Operator Replacement
LOR	Formula List Operator Replacement
UOI	Unary Operator Insertion
UOD	Unary Operator Deletion
LOD	Logical Operand Deletion
PBD	Paragraph Body Deletion
BOE	Binary Operand Exchange
IEOE	Imply-Else Operand Exchange

LOD deletes an operand of a logical operator, e.g., `a||b` to `b`. *PBD* deletes the body of an Alloy paragraph. *BOE* exchanges operands for a binary operator, e.g., `a=>b` to `b=>a`. *IEOE* exchanges the operands of `imply-else` operation, e.g., `a => b else c` to `a => c else b`.

2.3 Regression Test Selection for Imperative Code

Regression test selection (RTS) techniques select a subset of test cases from an initial test suite. The subset of tests checks if the affected sources of a project continue to work correctly. RTS is *safe* if it guarantees that the subset of selected tests includes all tests whose behavior may be affected by the changes [4,32]. RTS is *precise* if tests that are not affected are also not selected. Typical RTS techniques has three phases: the *analysis phase* selects tests to run, the *execution phase* runs the selected tests, and the *collection phase* collects information from the current version for future analysis. RTS techniques can perform at different granularities. For example, FaultTracer [35] analyzes dependencies at the method level while Ekstazi [3] does it at the file level, and both tools target projects written in Java.

During the analysis phase, RTS tools commonly compute a checksum, i.e., a unique identifier, of each code entity (e.g., method or file) on which a test depends. If the checksum changes, we view its source code as changed, in which case the test is selected and executed; otherwise it is not selected. The execution phase is tightly integrated with the analysis phase and simply executes selected tests. During the collection phase, RTS either dynamically monitors the test execution [3] or statically analyzes the test [7] to collect accessed/used entities, which are saved for the analysis phase in the next run.

3 Motivating Example

This section describes how iAlloy works using two versions of the Dijkstra Alloy model. Line 18 (highlighted in red) in Fig. 1 was absent in a faulty version of the model which we denote as Version 1. The model in Fig. 1 is the correct version which we denote as Version 2.

First, we apply iAlloy to Version 1. iAlloy invokes commands `Initial` (line 52), `IsStalled` (line 53), `IsFree` (line 54) and `GrabMutex` (line 55) with the SAT solver. Before invoking command `ReleaseMutex` (line 56), iAlloy finds that the solution obtained from invoking `GrabMutex` can be reused as the solution of `ReleaseMutex`. Therefore, command `ReleaseMutex` is solved without invoking SAT. iAlloy continues to invoke the rest of the commands and finds that command `Deadlock` (line 58) can reuse the solution of `IsStalled`, and command `DijkstraPreventsDeadlocks` can reuse the solution of `ShowDijkstra`. Next, we apply iAlloy again to Version 2. iAlloy performs dependency analysis between Version 1 and Version 2, and only selects the commands that are affected by the change (Line 18 in Fig. 1), namely commands `GrabMutex`, `GrabOrRelease`, `ShowDijkstra` and `DijkstraPreventsDeadlocks`. iAlloy tries to reuse the solutions of previous runs when invoking the four selected commands and `GrabMutex` reuses the solution of command `GrabMutex` in Version 1.

Traditionally, Alloy analyzer needs to execute 18 commands with expensive SAT solving, which takes total of 103.01 seconds. In comparison, iAlloy only invokes 9 commands where 5 commands are saved by regression command selection and 4 commands are saved by solution reuse. In total, iAlloy takes 84.14 seconds. Overall, iAlloy achieves 1.22x speed-up with 18.87 seconds time saving. Section 5 evaluates more subjects and shows that iAlloy achieves 1.59x speed-up on average and reduces unnecessary command invocations by more than 50%.

4 Techniques

In an evolving Alloy model scenario, we propose a two-step incremental analysis to reduce the time overhead of command execution. The first step is regression command selection (RCS) based on static dependency analysis (Sect. 4.1). The second step is solution reuse (SR) using fast instance evaluation (Sect. 4.2). Note that RCS handles paragraph-level dependency analysis, while SR covers more sophisticated expression-level dependency analysis.

Algorithm 1 shows the general algorithm of our incremental analysis. For each version (m_v) in a sequence of model evolutions ($ModelVersionSeq$), iAlloy first applies RCS (*RCmdSelection*) to select the commands (*SelectCmdList*) that are affected since the last version. Then, for each command in *SelectCmdList*, iAlloy further checks whether the solutions of previous commands can be reused in the new commands (*CheckReuse*). Note that the solutions of commands in the same version can also be reused. However, if the signatures change in the current version, then SR is not applicable and all commands are executed. If none of the old solutions can be reused for the current command c, then iAlloy invokes the SAT solver (*Execute*) to find a new solution which may be used for the next run.

Algorithm 1. General Algorithm for Incremental Alloy Model Solving

Input: model version sequence $ModelVersionSeq$
Output: solution for each command

```
for $m_v \in ModelVersionSeq$ do
    $SelectCmdList$ = RCmdSelection($m_v$);
    for $c \in SelectCmdList$ do
        if Changed($c.Dependency.SigList$) then
            Execute($c$, $SolutionSet$);
        else if !CheckReuse($c$, $SolutionSet$) then
            Execute($c$, $SolutionSet$);
        end if
    end for
end for
```

Algorithm 2. Algorithm for Regression Command Selection

Input: one model version m_v
Output: selected command list

```
procedure RCmdSelection(Model $m_v$)
    List<Cmd> $SelectCmdList$;
    Map<Cmd, Nodes> $Cmd2DpdParagraphs$ = DpdAnalysis($m_v.AllCmd$);
    for $c \in m_v.AllCmd$ do
        $DpdParagraphs$ = $Cmd2DpdParagraphs$.get(c);
        if Exist($c.Dependency$) then                                ▷ old dependency
            $newDependency$ = CheckSum($DpdParagraphs$);
            if Changed($c.Dependency$, $newDependency$) then
                Update($c$, $newDependency$);
                $SelectCmdList$.add($c$);  ▷ update dependency and select commands
            end if
        else
            $dependency$ = CheckSum($DpdParagraphs$)
            Update($c$, $dependency$);
            $SelectCmdList$.add($c$);      ▷ update dependency and select commands
        end if
    end for
    return $SelectCmdList$;
end procedure
```

4.1 Regression Command Selection (RCS)

Algorithm 2 presents the algorithm for RCS. iAlloy first gets the dependent paragraphs of each command ($Cmd2DpdParagraphs$) based on the dependency analysis ($DpdAnalysis$). For each command c in model version m_v, iAlloy generates a unique identifier, as described in Sect. 2.3, for each dependent paragraph ($CheckSum$). If the checksum of any dependent paragraph changes, iAlloy selects the corresponding command as the command execution candidate ($SelectCmdList$) and updates the dependency with new checksum.

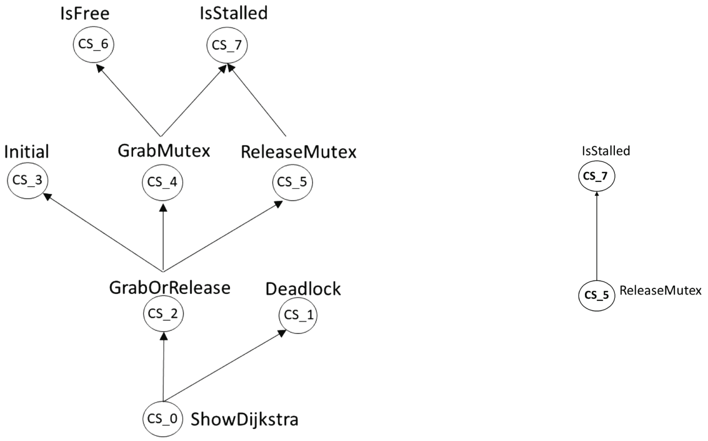

Fig. 2. Dependency graph for ShowDijkstra (left) and ReleaseMutex (right) command in the Dijkstra model

The dependency information of each command is the key for RCS. The dependency analysis for Alloy models can be either at the paragraph level or at the expression level. For safety reasons as we mentioned in Sect. 2.3, we do dependency analysis on the paragraph level in RCS. And we address further fine-grained expression level analysis in SR to achieve a better precision. To filter out the changes in comments and spaces, we traverse the AST of each paragraph and output the canonicalized string of the paragraph. The canonicalized string is hashed into a checksum which represents the unique version of the paragraph.

We take the Dijkstra Alloy model in Fig. 1 as an example. The dependency graph of command `ShowDijkstra` is shown in Fig. 2 (left), including transitively dependent Alloy paragraphs and their corresponding checksums `CS_i`. Since the checksum `CS_4` of predicate `GrabMutex` is changed (line 18 in Fig. 1) and `GrabMutex` is in the dependency graph of command `ShowDijkstra`, command `ShowDijkstra` is selected. In comparison, the dependency graph of command `ReleaseMutex` is shown in Fig. 2 (right). Since the checksums of both `IsStalled` and `ReleaseMutex` do not change, command `ReleaseMutex` is not selected.

4.2 Solution Reuse (SR)

Algorithm 3 illustrates how iAlloy checks if a solution can be reused by the current command. The input to Algorithm 3 is each selected command (c) from RCS and a solution set containing all the previous solutions ($SolutionSet$). If the solution s from $SolutionSet$ includes valuations of parameters of the Alloy paragraph (represented as $CheckList$ which includes implicit Alloy facts) invoked by c (Sect. 4.2.1), and $CheckList$ is satisfiable under s (Sect. 4.2.2), then s can be reused as the Alloy instance if c is invoked and c need not be invoked with expensive SAT solving (return true). Otherwise, SAT solving is involved to generate a

Algorithm 3. Algorithm for Solution Reuse Checking

Input: one command and the solution set
Output: if the command can reuse any solution in the solution set

```
procedure CheckReuse(Cmd $c$, Set<Solution> $SolutionSet$)
    List<Nodes> $CheckList$;
    $CheckList$.add($c.Dependency.FactList$);
    if CheckCmd($c$) then                                ▷ $c$ is check command
        $CheckList$.add($c.Dependency.Assert$);
    else                                                 ▷ $c$ is run command
        $CheckList$.add($c.Dependency.Pred$);
    end if
    for $s \in SolutionSet$ do
        if $c.param \subseteq s.cmd.param$ && $s.sol$.evaluator($CheckList$) = true then
            return true;
        end if
    end for
    return false;
end procedure
```

new solution (if there is any) which is stored for subsequent runs (Algorithm 4, Sect. 4.2.3).

Note that SR not only filters out the semantically equivalent regression changes, but also covers the sophisticated expression-level dependency analysis. For example, suppose the only change in an Alloy model is a boolean expression changed from `A` to `A || B` where `||` stands for disjunction and `B` is another boolean expression, the old solution of the corresponding command is still valid and can be reused. Besides, SR allows solutions from other commands to be reused for the current command, which further reduces SAT solving overhead.

4.2.1 Solution Reuse Condition

As described in Sect. 2, each command invokes either a predicate or an assert. Each predicate has multiple parameter types which we denote as *parameter set* for simplicity in the rest of the paper. The parameter set of any assertion is an empty set ($\varnothing$). As shown in the following equation, we define the parameter set of a command c (`c.param`) as the parameter set of the directly invoked predicate (`ParamSet(c.pred)`) or assertion ($\varnothing$).

$$c.param = \begin{cases} ParamSet(c.pred), & c \text{ is run command} \\ \varnothing, & c \text{ is check command} \end{cases}$$

A command that invokes an Alloy paragraph with parameters implicitly checks if there exists a set of valuations of the corresponding parameters that satisfies the paragraph. We observe that command c_2 can reuse the solution s_1 obtained by invoking c_1 if the parameter set of c_2 is a subset of that of c_1, namely $c_2.param \subseteq c_1.param$. The solution reuse complies to a partial order based on

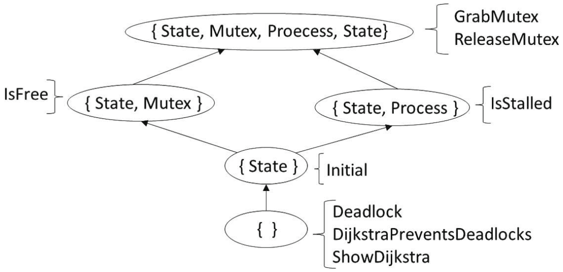

Fig. 3. Parameter relations of commands in the Dijkstra model

Algorithm 4. Algorithm for Command Execution

Input: one command and the solution set
Output: save the solution if it is SAT or print out UNSAT

```
procedure CmdExecute(Cmd $c$, Set<Solution> $SolutionSet$)
    A4Solution $sol$ = Alloy.solve($c$);
    if sol.IsSat() then                              ▷ if the solution is SAT;
        Solution $s$;
        $s.sol = sol$;              ▷ store the instance and corresponding command;
        $s.cmd = c$;
        $SolutionSet$.add($s$);
    else
        print UNSAT
    end if
end procedure
```

the subset relation of command parameters. On the other hand, solution s_1 cannot be reused by c_2 if $c_2.param \subsetneq c_1.param$, in which case we do not know all the valuations of c_2's parameters.

Figure 3 shows how solution reuse is conducted based on the subset relations of command parameter set in the Dijkstra model. For instance, since the parameter set `{}` ($\varnothing$) is the subset of all parameter sets above it, the corresponding commands `Deadlock`, `DijkstraPreventsDeadlocks` and `ShowDijkstra` with parameter set `{}` can reuse all solutions of commands whose parameter sets are the super set of `{}`, namely `Initial`, `IsFree`, `IsStalled`, `GrabMutex` and `ReleaseMutex`. Since any parameter set is a subset of itself, a solution $s1$ of command c_1 can be reused by the command c_2 which has the same parameter set as c_1.

4.2.2 Solution Reuse Evaluation

Once a solution s can be reused for command c, we need to further check if s is actually the solution of c that satisfies the corresponding constraints. As described in Sect. 2, the constraints of a command come from all facts and the transitively invoked predicate/assertion. To reuse s in the old version, s must be

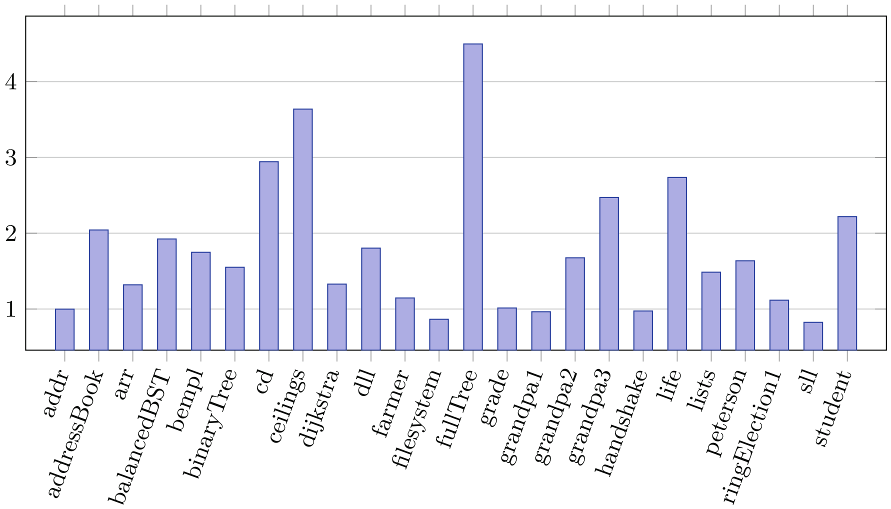

Fig. 4. Speedup results on Mutant Version Set

satisfiable for c in the new version. If c is unsatisfiable under the valuations of s, it does not imply that c is unsatisfiable in the solution space and thus c must be invoked with SAT solving. The satisfiability of command c is determined by the Alloy built-in evaluator under the valuation of s.

4.2.3 Command Execution

If none of the solutions can be reused by command c, iAlloy executes the command as described in Algorithm 4. If a solution *sol* is found (`Sol.IsSat()`), the solution *sol* together with the command c is saved for subsequent runs. To avoid saving too many solutions as the model evolves (which may slow down the SR and reduce the overall gain), we only keep the most recent solution for each command. In future work, we plan to evaluate how long a solution should be kept.

5 Experimental Evaluation

In this paper, we answer the following research questions to evaluate iAlloy:

- RQ1: How does iAlloy perform compared to traditional Alloy Analyzer (which we treat as the baseline)?
- RQ2: How much reduction of the commands executed does Regression Command Selection and Solution Reuse contribute in the two subject sets?
- RQ3: What is the time overhead of Regression Command Selection, Solution Reuse and command execution in iAlloy, respectively?

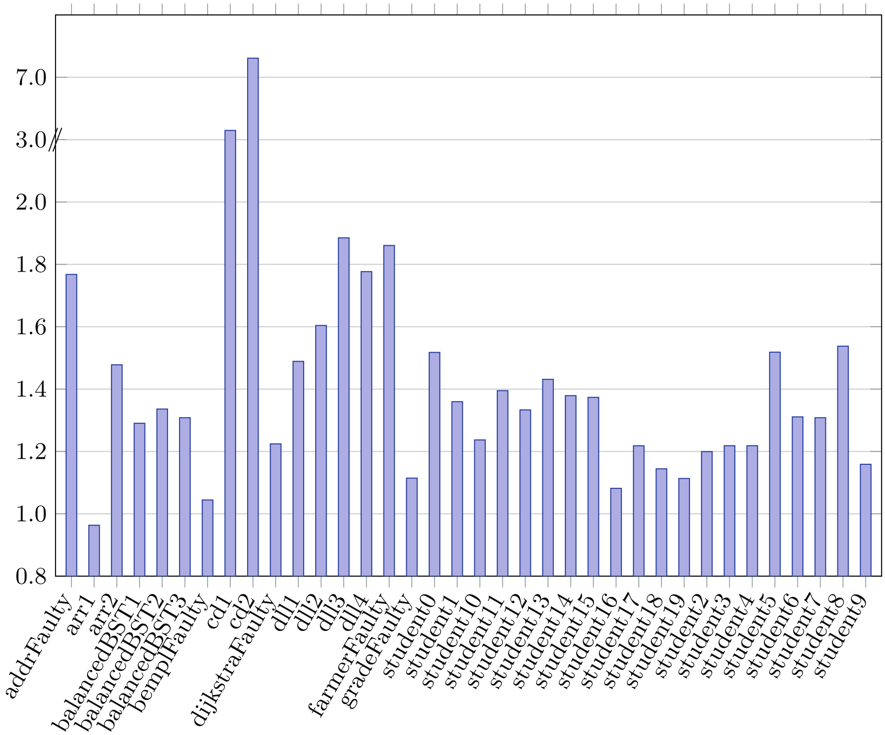

Fig. 5. Speedup results on Real Version Model Set

5.1 Experimental Setup

Subjects: There are two subject sets in the experiment. The first set of subjects is the simulated evolving Alloy model version sets, which we call Mutant Version Set. In this set, we take 24 Alloy models from the standard Alloy distribution (version 4.2) and use them as the first version. For each model in version 1, we use MuAlloy [27] to generate several mutants and randomly select one as version 2. This process continues until we get the fifth version. Thus, each subject in the Mutant Version Set includes five versions. The second subject set is called Real Version Set. Each subject in this set consists of two model versions: the real faulty model (version 1) from the ARepair [26] distribution and the correct model after the fix (version 2). There are 36 subjects in this set.

Baseline: The baseline in this experiment is the traditional Alloy Analyzer, which executes each command for each version.

Platform: We conduct all our experiments on Ubuntu Linux 16.04, an Intel Core-i7 6700 CPU (3.40 GHz) and 16 GB RAM. The version of Alloy we did experiments on is version 4.2.

Table 2. RCS, SR and Command Execution Results in Mutant Version Set

Model	cmd	select	reuse	execute	T_select (%)	T_reuse (%)	T_execute (%)
addr	5	5 (100%)	0 (0%)	5 (100%)	4.2	0.0	95.8
addressBook	10	9 (90%)	3 (33.3%)	6 (66.7%)	0.3	53.5	46.2
arr	5	5 (100%)	2 (40%)	3 (60%)	3.6	1.9	94.5
balancedBST	20	16 (80%)	13 (81.3%)	3 (18.7%)	12.3	23.7	64.0
bempl	10	10 (100%)	4 (40%)	6 (60%)	1.4	1.8	96.8
binaryTree	5	5 (100%)	3 (60%)	2 (40%)	1.7	0.9	97.4
cd	20	13 (65%)	9 (69.2%)	4 (30.8%)	0.7	0.8	98.5
ceilings	30	18 (60%)	13 (72.2%)	5 (27.8%)	2.9	5.3	91.7
dijkstra	30	23 (76.7%)	9 (39.1%)	14 (60.9%)	0.6	36.3	63.2
dll	20	14 (70%)	9 (64.3%)	5 (35.7%)	11.4	14.8	73.9
farmer	15	15 (100%)	3 (20%)	12 (80%)	0.3	1.6	98.1
filesystem	15	11 (73.3%)	3 (27.3%)	8 (72.7%)	27.9	17.4	54.7
fullTree	15	13 (86.7%)	11 (84.6%)	2 (15.4%)	1.6	2.3	96.1
grade	10	10 (100%)	0 (0%)	10 (100%)	1.2	0.9	97.9
grandpa1	15	15 (100%)	0 (0%)	15 (100%)	0.6	0.0	99.4
grandpa2	10	7 (70%)	3 (42.9%)	4 (57.1%)	1.2	1.0	97.8
grandpa3	25	16 (64%)	6 (37.5%)	10 (62.5%)	0.3	0.5	99.2
handshake	20	20 (100%)	0 (0%)	20 (100%)	0.5	0.0	99.5
life	15	7 (46.7%)	1 (14.3%)	6 (85.7%)	0.9	2.2	96.9
lists	20	20 (100%)	9 (45%)	11 (55%)	0.2	0.4	99.4
peterson	85	69 (81.2%)	41 (59.4%)	28 (40.6%)	0.8	7.8	91.5
ringElection1	30	30 (100%)	7 (23.3%)	23 (76.7%)	0.4	1.7	97.9
sll	5	5 (100%)	0 (0%)	5 (100%)	29.9	6.2	63.9
student	25	23 (92%)	20 (87.0%)	3 (13.0%)	9.2	21.5	69.3
Overall	460	379 (82.4%)	169 (44.6%)	210 (55.4%)	4.7	8.4	86.8

5.2 RQ1: Speed-up Effectiveness

Figures 4 and 5 show the speedup of iAlloy compared to the baseline on Mutant Version Set and Real Version Set, respectively. The x-axis denotes the subject names and the y-axis denotes the speed up. In Mutant Version Set, iAlloy achieves speed-up for 19 subjects (75% of the subject set), with up to 4.5x speed-up and 1.79x on average. The reason iAlloy did not speed up on the remaining 5 subjects is that either the change is in the signatures or many commands are unsatisfiable under the previous solutions, where the analysis time overhead in iAlloy (RCS and SR) is larger than the savings. In Real Version Set, we observe that iAlloy achieves a speedup of up to 7.66x and 1.59x on average over all subjects except one (97% of the subject set). iAlloy does not save any time on `arr1` because there exists a single command in the subject and the command is unsatisfiable (in which case neither RCS nor SR can save any command executions).

5.3 RQ2: Command Selection and Solution Reuse Effectiveness

Columns 2–5 in Tables 2 and 3 show the total number of commands in each subject (cmd), the number of the selected commands and their percentage compared to the total number of commands (select), the number of solution reuse

Table 3. RCS, SR and Command Execution Results in Real Version Set

Model	cmd	select	reuse	execute	T_select (%)	T_reuse (%)	T_execute (%)
addr	2	2 (100%)	1 (50%)	1 (50%)	24.9	4.7	70.4
arr1	2	2 (100%)	0 (0%)	2 (100%)	7.4	0.0	92.6
arr2	2	2 (100%)	1 (50%)	1 (50%)	7.2	1.4	91.4
bBST1	8	8 (100%)	6 (75%)	2 (25%)	13.4	15.2	71.4
bBST2	8	8 (100%)	6 (75%)	2 (25%)	14.0	15.0	70.9
bBST3	8	8 (100%)	6 (75%)	2 (25%)	13.5	14.9	71.5
bempl	4	4 (100%)	0 (0%)	4 (100%)	1.8	0.4	97.8
cd1	8	7 (87.5%)	5 (71.4%)	2 (28.6%)	1.1	0.9	97.9
cd2	8	7 (87.5%)	6 (85.7%)	1 (14.3%)	3.5	3.0	93.5
dijk	12	10 (83.3%)	5 (50%)	5 (50%)	0.7	23.2	76.2
dll1	8	8 (100%)	6 (75%)	2 (25%)	13.2	16.0	70.8
dll2	8	8 (100%)	6 (75%)	2 (25%)	12.7	17.0	70.3
dll3	8	8 (100%)	7 (87.5%)	1 (12.5%)	16.3	22.3	61.3
dll4	8	8 (100%)	7 (87.5%)	1 (12.5%)	17.6	22.3	60.1
farmer	7	7 (100%)	2 (28.6%)	5 (71.4%)	0.7	2.5	96.8
grade	4	4 (100%)	1 (25%)	3 (75%)	3.6	1.7	94.8
stu0	10	10 (100%)	7 (70%)	3 (30%)	8.1	11.0	80.9
stu1	10	10 (100%)	6 (60%)	4 (40%)	5.8	8.3	85.9
stu10	10	10 (100%)	5 (50%)	5 (50%)	6.7	10.1	83.2
stu11	10	10 (100%)	7 (70%)	3 (30%)	7.6	10.4	81.9
stu12	10	10 (100%)	7 (70%)	3 (30%)	7.6	9.2	83.2
stu13	10	10 (100%)	7 (70%)	3 (30%)	6.4	9.5	84.1
stu14	10	10 (100%)	6 (60%)	4 (40%)	6.6	8.7	84.8
stu15	10	10 (100%)	6 (60%)	4 (40%)	6.9	6.7	86.4
stu16	10	10 (100%)	4 (40%)	6 (60%)	9.4	13.3	77.4
stu17	10	10 (100%)	5 (50%)	5 (50%)	6.7	8.0	85.3
stu18	10	10 (100%)	4 (40%)	6 (60%)	7.7	10.5	81.8
stu19	10	10 (100%)	4 (40%)	6 (60%)	6.1	9.8	84.1
stu2	10	10 (100%)	4 (40%)	6 (60%)	6.2	8.6	85.2
stu3	11	11 (100%)	5 (45.5%)	6 (54.5%)	5.3	8.9	85.8
stu4	10	10 (100%)	4 (40%)	6 (60%)	7.1	9.6	83.3
stu5	10	10 (100%)	7 (70%)	3 (30%)	8.1	8.2	83.7
stu6	10	10 (100%)	6 (60%)	4 (40%)	7.0	9.1	84.0
stu7	10	10 (100%)	6 (60%)	4 (40%)	6.6	8.9	84.5
stu8	10	10 (100%)	7 (70%)	3 (30%)	6.7	7.4	85.9
stu9	10	10 (100%)	4 (40%)	6 (60%)	7.1	11.0	81.9
Overall	306	302 (98.7%)	176 (58.3%)	126 (41.7%)	8.1	9.7	82.2

and their percentage in selected commands (reuse), and the number of actually executed commands and their percentage in selected commands (execute), for the Mutant and Real Version Set respectively. We can see that, both RCS and SR help reduce command execution in both subject sets, but to different extent. A smaller portion of commands are selected in Mutant Set (82.4%) than in Real Set (98.7%). This is due to the fact that there are more changes between versions in Real Set than in Mutant Set. However, smaller portion (41.7% vs. 55.4%) of the selected commands are executed and a larger portion (58.3% vs. 44.6%) of selected commands successfully reuse solutions in Real Set, comparing with Mutant Set. Besides, there are 54.3% command execution reduction $(\frac{cmd - execute}{cmd})$ in Mutant Set and 58.8% in Real Set. The result shows that iAlloy is promising in reducing the command executions in analyzing real world Alloy models as they evolve.

5.4 RQ3: Time Consumption

Columns 6–8 in Tables 2 and 3 present the percentage of time consumption in RCS (T_select), SR (T_reuse), and command execution (T_execute) in the Mutant Version Set and Real Version Set, respectively. We can see that in both subject sets, execution takes most of the time while RCS and SR are lightweight.

6 Related Work

A lot of work has been done to improve [20,22,24] and extend [10–13,16,19,25, 28–31,33] Alloy. We discuss work that is closely related to iAlloy.

Incremental Analysis for Alloy. Li et al. [9] first proposed the incremental analysis idea for their so-called consecutive Alloy models which are similar to the evolving models. They exploit incremental SAT solving to solve only the *delta* which is the set of boolean formulas describing the changed part between two model versions. Solving only the delta would result in a much improved SAT solving time than solving the new model version from scratch. Titanium [1] is an incremental analysis tool for evolving Alloy models. It uses all the solutions of the previous model version to potentially calculate tighter bounds for certain relational variables in the new model version. By tightening the bounds, Titanium reduces the search space, enabling SAT solver to find the new solutions at a fraction of the original solving time. These two approaches are the most relevant to our work that both focus on improving solving efficiency in the translated formulas. Whereas our incremental approach is to avoid the SAT solving phase completely, which is fundamentally different from existing approaches. In addition, Titanium has to find all the solutions in order to tighten the bounds, which would be inefficient when only certain number of solutions are needed.

Regression Symbolic Execution. Similar to the SAT solving applications such as Alloy analyzer, symbolic execution tools also face the scalability problems, in which case a lot of work has been done to improve the performance [6,14,23,34]. The most closely related to our work is regression symbolic execution [14,15,34]. Similar to our RCS, symbolic execution on the new version is guided through the changed part with the previous versions. In addition, there is also work on verification techniques that reuses or caches the results [8,18].

7 Conclusion and Future Work

In this paper, we proposed a novel incremental analysis technique with regression command selection and solution reuse. We implemented our technique in a tool called iAlloy. The experimental results show that iAlloy can speed up 90% of our subjects. Furthermore, it performs surprisingly well in models of the real faulty versions with up to 7.66 times speed up and above 50% command execution reduction. This indicates that iAlloy is promising in reducing time overhead of analyzing real-world Alloy models. In the future, we plan to extend iAlloy to support changes that involve Alloy signatures and perform a more fine-grained analysis to improve command selection.

Acknowledgments. We thank the anonymous reviewers for their valuable comments. This research was partially supported by the US National Science Foundation under Grants Nos. CCF-1566363, CCF-1652517, CCF-1704790 and CCF-1718903.

References

1. Bagheri, H., Malek, S.: Titanium: efficient analysis of evolving alloy specifications. In: International Symposium on Foundations of Software Engineering, pp. 27–38 (2016)
2. Rational Software Corporation: Object constraint language specification. Version 1.1 (1997)
3. Gligoric, M., Eloussi, L., Marinov, D.: Practical regression test selection with dynamic file dependencies. In: International Symposium on Software Testing and Analysis, pp. 211–222 (2015)
4. Graves, T.L., Harrold, M.J., Kim, J.-M., Porter, A., Rothermel, G.: An empirical study of regression test selection techniques. Trans. Softw. Eng. Methodol. **10**(2), 184–208 (2001)
5. Jackson, D.: Alloy: a lightweight object modelling notation. Trans. Softw. Eng. Methodol. **11**(2), 256–290 (2002)
6. Jia, X., Ghezzi, C., Ying, S.: Enhancing reuse of constraint solutions to improve symbolic execution. In: International Symposium on Software Testing and Analysis, pp. 177–187 (2015)
7. Legunsen, O., Shi, A., Marinov, D.: STARTS: STAtic regression test selection. In: Automated Software Engineering, pp. 949–954 (2017)
8. Leino, K.R.M., Wüstholz, V.: Fine-grained caching of verification results. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, Part I, vol. 9206, pp. 380–397. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_22
9. Li, X., Shannon, D., Walker, J., Khurshid, S., Marinov, D.: Analyzing the uses of a software modeling tool. Electron. Notes Theoret. Comput. Sci. **164**(2), 3–18 (2006)
10. Montaghami, V., Rayside, D.: Extending alloy with partial instances. In: Derrick, J., et al. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 122–135. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30885-7_9
11. Montaghami V., Rayside D.: Staged evaluation of partial instances in a relational model finder. In: Ait Ameur Y., Schewe KD. (eds) Abstract State Machines, Alloy, B, TLA, VDM, and Z. ABZ 2014. LNCS, vol. 8477, pp. 318–323. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43652-3_32

12. Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: principled scenario exploration through minimality. In: International Conference on Software Engineering, pp. 232–241 (2013)
13. Nijjar, J., Bultan, T.: Bounded verification of ruby on rails data models. In: International Symposium on Software Testing and Analysis, pp. 67–77 (2011)
14. Person, S., Yang, G., Rungta, N., Khurshid, S.: Directed incremental symbolic execution. SIGPLAN Not. **46**(6), 504–515 (2011)
15. Ramos, D.A., Engler, D.R.: Practical, low-effort equivalence verification of real code. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 669–685. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_55
16. Regis, G., et al.: DynAlloy analyzer: a tool for the specification and analysis of alloy models with dynamic behaviour. In: Foundations of Software Engineering, pp. 969–973 (2017)
17. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice-Hall Inc., Upper Saddle River (1989)
18. Strichman, O., Godlin, B.: Regression verification - a practical way to verify programs. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 496–501. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69149-5_54
19. Sullivan, A., Wang, K., Khurshid, S.: AUnit: a test automation tool for alloy. In: International Conference on Software Testing, Verification, and Validation, pp. 398–403 (2018)
20. Sullivan, A., Wang, K., Khurshid, S., Marinov, D.: Evaluating state modeling techniques in alloy. In: Software Quality Analysis, Monitoring, Improvement, and Applications (2017)
21. Sullivan, A., Wang, K., Zaeem, R.N., Khurshid, S.: Automated test generation and mutation testing for alloy. In: International Conference on Software Testing, Verification, and Validation, pp. 264–275 (2017)
22. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_49
23. Visser, W., Geldenhuys, J., Dwyer, M.B.: Green: reducing, reusing and recycling constraints in program analysis. In: International Symposium on the Foundations of Software Engineering, pp. 58:1–58:11 (2012)
24. Wang, J., Bagheri, H., Cohen, M.B.: An evolutionary approach for analyzing Alloy specifications. In: International Conference on Automated Software Engineering, pp. 820–825 (2018)
25. Wang, K., Sullivan, A., Khurshid, S.: ARepair: a repair framework for alloy. In: International Conference on Software Engineering, pp. 577–588 (2018)
26. Wang, K., Sullivan, A., Khurshid, S.: Automated model repair for alloy. In: Automated Software Engineering, pp. 577–588 (2018)
27. Wang, K, Sullivan, A., Khurshid, S.: MuAlloy: a mutation testing framework for alloy. In: International Conference on Software Engineering, pp. 29–32 (2018)
28. Wang, K., Sullivan, A., Koukoutos, M., Marinov, D., Khurshid, S.: Systematic generation of non-equivalent expressions for relational algebra. In: Butler, M., Raschke, A., Hoang, T.S., Reichl, K. (eds.) ABZ 2018. LNCS, vol. 10817, pp. 105–120. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91271-4_8
29. Wang, K., Sullivan, A., Marinov, D., Khurshid, S.: ASketch: a sketching framework for alloy. In: Symposium on the Foundations of Software Engineering, pp. 916–919 (2018)

30. Wang, K., Sullivan, A., Marinov, D., Khurshid, S.: Fault localization for declarative models in alloy. eprint arXiv:1807.08707 (2018)
31. Wang, K., Sullivan, A., Marinov, D., Khurshid, S.: Solver-based sketching of alloy models using test valuations. In: Butler, M., Raschke, A., Hoang, T.S., Reichl, K. (eds.) ABZ 2018. LNCS, vol. 10817, pp. 121–136. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91271-4_9
32. Wang, K., Zhu, C., Celik, A., Kim, J., Batory, D., Gligoric, M.: Towards refactoring-aware regression test selection. In: IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp. 233–244 (2018)
33. Wang, W., Wang, K., Zhang, M., Khurshid, S.: Learning to optimize the alloy analyzer. In: International Conference on Software Testing, Verification and Validation (2019, to appear)
34. Yang, G., Păsăreanu, C.S., Khurshid, S.: Memoized symbolic execution. In: International Symposium on Software Testing and Analysis, pp. 144–154 (2012)
35. Zhang, L., Kim, M., Khurshid, S.: Localizing failure-inducing program edits based on spectrum information. In: International Conference on Software Maintenance and Evolution, pp. 23–32 (2011)