ISR Institute for Software Research
University of California, Irvine

Titanium: Efficient Analysis of Evolving Alloy Specifications

Hamid Bagheri
University of California, Irvine
hamidb@uci.edu

Sam Malek
University of California, Irvine
malek@uci.edu

ISR Technical Report # UCI-ISR-15-1
September 2015

Institute for Software Research
ICS2 221
University of California, Irvine
Irvine, CA 92697-3455
isr.uci.edu

isr.uci.edu/publications

TITANIUM: Efficient Analysis of Evolving Alloy Specifications

Hamid Bagheri
Institute for Software Research
Department of Informatics
University of California, Irvine
hamidb@uci.edu

Sam Malek
Institute for Software Research
Department of Informatics
University of California, Irvine
malek@uci.edu

ABSTRACT

The Alloy specification language, and the corresponding Alloy Analyzer, have received much attention in the last two decades with applications in many areas of software engineering. Increasingly, formal analyses enabled by Alloy are desired for use in an on-line mode, where the specifications are automatically kept in sync with the running, possibly changing, software system. However, given Alloy Analyzer's reliance on computationally expensive SAT solvers, an important challenge is the time it takes for such analyses to execute at runtime. The fact that in an on-line mode, the analyses are often repeated on slightly revised versions of a given specification, presents us with an opportunity to tackle this challenge. We present Titanium, an extension of Alloy for formal analysis of evolving specifications. By leveraging the results from previous analyses, Titanium narrows the state space of the revised specification, thereby greatly reducing the required computational effort. We describe the semantic basis of Titanium in terms of models specified in relational logic. We show how the approach can be realized atop an existing relational logic model finder. Our experimental results show Titanium achieves a significant speed-up over Alloy Analyzer when applied to the analysis of evolving specifications.

Keywords

Formal Verification, Evolving Software, Relational Logic, Partial Models.

1. INTRODUCTION

Formal specification languages and the corresponding analysis environments have long been applied to a variety of software engineering problems. Most notably, the Alloy language and the corresponding analysis engine, Alloy Analyzer [21], have received a lot of attention in the software engineering community. Alloy provides a lightweight object modeling notation that is especially suitable for modeling structural properties of a software system. It has been used to solve a variety of software engineering problems, including software design [12, 14, 22], code analysis [8, 9, 31, 39], and test case generation [24, 28]. Given a model of a software system in Alloy, the Alloy Analyzer uses a SAT solver to automatically analyze the software system's properties, specified in the form of predicates and formulas.

Formal specifications have much in common with the complex software systems they represent; namely they are hard to develop and tend to evolve. Construction of formal specifications is a non-trivial task. Just like most complex software systems, formal specifications are developed iteratively [15], where in each iteration some elements of the model are modified, removed, and new ones are added, until the desired fidelity is achieved. In each iteration, the specification is analyzed to help the developer assess its utility, fix flaws, and plan the next set of changes.

In addition, as software systems tend to evolve over time, formal specifications representing them need to evolve as well. The evolution of specifications, however, is not limited to those that are constructed manually. In fact, automated means of generating formal specifications from software artifacts [11,24,31,33,39], often through some form of program analysis, have made it significantly easier to maintain an up-to-date specification for a changing software system. Such techniques have made it possible to verify an evolving specification of a software system, after the initial deployment of software, possibly at runtime, and as it changes. In such settings, the evolving specification is continuously analyzed in real-time to assess the properties (e.g., security [8, 9, 31]) of the corresponding software.

In spite of its strengths, Alloy Analyzer's reliance on computationally heavy SAT solvers means that it can take a significant amount of time to verify the properties of software. The ability to analyze the specifications efficiently is quite important, especially when they are developed through an iterative process. The development of a complex specification often involves repeated runs of the analyzer for debugging and assessment of its semantics. In an online mode, where the specifications are kept in sync with the changing software, and the analysis is performed at runtime, the time it takes to verify the properties of software is of even greater importance. There is, thus, a need for mechanisms that facilitate efficient analysis of evolving specifications in response to incremental changes. An opportunity to reduce the analysis time is presented by the fact that in the aforementioned scenarios, specifications are unlikely to change completely from one analysis to the next. It is, therefore, desirable to be able to leverage the results of analysis performed on a specification to optimize any subsequent anal-

Technical Report UCI-ISR-15-1
Revised 7/31/2016 .

yses on its revisions. However, Alloy Analyzer, as well as its other variants (e.g., Aluminum [32]), do not provide any mechanism to leverage the results of analysis across evolving specifications, even if they are substantially overlapping.

In this paper, we introduce Titanium, an extension of Alloy Analyzer for efficient analysis of evolving Alloy specifications. The efficiency gain is due to a new method of determining the analysis bounds. In the conventional Alloy Analyzer, the user-defined bounds used to run the analysis determine the scope of search for models within a finite domain. The user-defined bound, however, is typically not the tightest bound for a problem. The tightest bound for a problem is effectively determined when all model instances are found. While this observation by itself is of little value when analyzing an Alloy specification from scratch, it is quite valuable when the analysis is targeted at evolving, substantially overlapping specifications, and forms the intuition behind our research.

Titanium first analyzes the structure of a revised specification, and identifies a set of relational variables that are shared with the originating specification. It then uses the instances produced for the original specification to potentially calculate tighter bounds for such relational variables in the revised specification. By tightening the bounds, Titanium reduces the search space, enabling the SAT solver to find the model instances at a fraction of time needed for the original bounds. The experimental results show that, with the proposed optimization, Titanium achieves significant gains in speed compared to the conventional Alloy Analyzer.

This paper makes the following contributions:

- *Efficient analysis of evolving specifications.* We propose a novel approach for analyzing evolving Alloy specifications that reduces the state space by adjusting the analysis bounds, thereby achieving significant speed-ups.

- *Formal description.* We formally describe the semantic basis of this approach in terms of models specified in relational logic, and demonstrate how it can be realized atop an existing relational logic model finder, without compromising soundness and completeness.

- *Implementation.* We have realized the analysis technique by modifying the Alloy Analyzer environment, resulting in a new version of the tool, which we have made publicly available [4].

- *Experiments.* We present empirical evidence of the efficiency gains using both Alloy specifications found in the prior work and those synthesized in a systematic fashion.

The remainder of this paper is organized as follows. Section 2 uses an illustrative example to describe the intuition behind our technique as well as the necessary background. Section 3 provides a formal description of our approach. Section 5 presents our experimental results obtained in our analysis of both real and synthesized specifications. Section 6 reviews the related research. Finally, Section 7 concludes the paper with a summary of our contributions and the avenues of future research.

2. ILLUSTRATIVE EXAMPLE

This section motivates our research and illustrates our optimization technique using a simple example. We describe the example using the Alloy [6] and Kodkod notations [41]. Section 3 presents a more detailed discussion of our approach.

```
 1  (a) a simple model of typing in Java
 2  abstract sig Type {
 3    subtypes: set Type
 4  }
 5  sig Class, Interface extends Type {}
 6  one sig Object extends Class {}
 7  sig Instance {
 8    type: Class
 9  }
10  fact TypeHierarchy {
11    // Object, root of subtype hierarchy
12    Type in Object.∗subtypes
13    // no self−subtyping
14    no t: Type | t in t.^subtypes
15    // subtype at most one class
16    all t: Type | lone t.~subtypes & Class
17  }
18  pred Show {
19    some Class − Object
20    some Interface
21  }
22  run Show for 2 but 3 Type
```

```
 1  (b) an updated version of the model
 2  sig Variable {
 3    holds: lone Instance,
 4    type: Type
 5  }
 6  fact TypeSoundness {
 7    all v: Variable | v.holds.type in v.type
 8  }
```

Listing 1: Alloy specification examples: (a) a specification describing Java typing and (b) new constructs added to the revised specification.

Consider the Alloy specification for a simplified model of typing in Java, shown in Listing 1. This specification is adopted from [6], and distributed with the Alloy Analyzer. Each Alloy specification consists of (1) data types, (2) formulas that define constraints over data types, and (3) commands to run the analyzer. Essential data types are specified in Alloy by their type signatures (sig), and the relationships between them are captured by the the declarations of *fields* within the definition of each signature. The running example defines 5 signatures (lines 2–9): Types are partitioned into Class and Interface types, with Object introduced as a singleton extending Class. Each Type may have a set of subtypes, and each Instance has a specific Class type.

Facts (fact) are formulas that take no arguments, and define constraints that every instance of a specification must satisfy, thus restricting the instance space of the specification. The formulas can be further structured using predicates (pred) and functions (fun), which are parameterized formulas that can be invoked. The TypeHierarchy fact paragraph (Listing 1, lines 10–17) states that Object is the root of the subtype hierarchy; that no Type is a subtype of itself

```
1  {T1,T2,O1,I1,I2}
2
3  Object:    (1,1) ::  {{O1},{O1}}
4  Class:     (0,2) ::  {{},{{T1},{T2}}}
5  Interface:(0,2) ::  {{},{{T1},{T2}}}
6  Instance:  (0,2) ::  {{},{{I1},{I2}}}
7  subtypes:  (0,9) ::  {{},{{O1,O1},{O1,T1},{O1,T2},{T1,O1},{T1,T1},{T1,T2},{T2,O1},{T2,T1},{T2,T2}}}
8  type:      (0,6) ::  {{},{{I1,O1},{I1,T1},{I1,T2},{I2,O1},{I2,T1},{I2,T2}}}
9
10 (all t: Object + Class + Interface | !(t in (t.^subtypes))) && ...
11 // ----------------------------------------------------------------------
12 // The upper bound for the subtypes relation in the updated specification is tightened by Titanium:
13 subtypes: (0,4) :: {{},{{O1,T1},{O1,T2},{T1,T2},{T2,T1}}}
```

Listing 2: Kodkod representation of the Alloy module of Listing 1.

(neither directly nor indirectly); and that each Type may be a subtype of at most one Class.

Analysis of specifications written in Alloy is completely automated, but bounded up to user-specified scopes on the size of data types. In particular, to make the state space finite, certain scopes need to be specified that limit the number of instances of each type signature. The run specification (lines 18–22) then asks for model instances that contain at least one Interface and one Class distinct from Object, and specifies a scope that bounds the search for model instances with at most two elements for each top-level signature, except for Type bounded to three elements.

In order to analyze such a relational specification bounded by the specified scope, both Alloy Analyzer and Titanium then translate it into a corresponding finite relational model in a language called Kodkod [41]. Listing 2 partially shows a Kodkod translation of Listing 1(a). A model in Kodkod's relational logic is a triple consisting of a universe of elements (also called *atoms*), a set of relation declarations including their lower and upper bounds specified over the model's universe, and a relational formula, where the declared relations appear as free variables [41].

The first line of Listing 2 declares a universe of five uninterpreted atoms.[1] In this section, we assume an interpretation of atoms, where the first two (T1 and T2) represent Type elements, the next one (O1) an Object element, and the last two (I1 and I2) Instance elements.

Lines 3–8 declare relational variables. Similar to Alloy, formulas in Kodkod are constraints defined over relational variables. While in Alloy these relational variables are separated into *signatures*, that represent *unary* relations establishing a type system, and *fields*, that represent non-unary relations, in Kodkod all relations are untyped, with no difference between unary and non-unary relational variables.

Kodkod further allows specifying a scope over each relational variable from both above and below by two *relational constants*. In principle, a relational constant is a prespecified set of tuples drawn from a universe of atoms. These two sets are called *upper* and *lower* bounds, respectively. Every relation in a model instance must contain all tuples in the lower bound, and no tuple that is not in the upper bound. That is, the upper bound represents the whole set of tuples that a relational variable may contain, and a lower bound a partial solution for a given model.

Consider the Object declaration (line 3), its upper and lower bounds both contain just one atom, O1, as it is defined as a singleton set in Listing 1. The upper bound for the variable $subtypes \subseteq Type \times Type$ (line 7) is a product of the upper bound set for its corresponding domain and codomain relations, taking every combination of an element from both and concatenating them.

```
// model instance 1
Object:    {{O1}}
Class:     {{T1}}
Interface:{{T2}}
Instance:  {{}}
subtypes:  {{O1,T2},{T2,T1}}
type:      {{}}

// model instance 2
Object:    {{O1}}
Class:     {{T1}}
Interface:{{T2}}
Instance:  {{I1},{I2}}
subtypes:  {{T2,T1},{O1,T1}}
type:      {{I1,T1},{I2,T1}}
```

Listing 3: Two arbitrarily selected instances for the specification of Listing 1(a).

The Kodkod's finite model finder then explores within such upper and lower bounds defined for each relational variable to find instances of a formula, which are essentially bindings of the formula's relational variables to relational constants in a way that makes the formula true. Listing 3 shows two different instances for the specification of Listing 1(a). A model instance can essentially be viewed as an *exact bound*, where the upper and lower bounds are equal.

After analyzing the specification, both Alloy Analyzer and Titanium produce the same instance set comprising 72 models (including symmetries), through enumerating all valid instances, in relatively the same amount of time, i.e., 421 and 419 ms, respectively. However, if the specification were to change, Titanium would leverage the instances produced for the original specification to potentially set a tighter bound for the shared relational variables, which in turn reduces the size of the state space, and improves the analysis time.

Figure 1 shows a simplified, schematic view of the Titanium approach, using an example consisting of 5 relational variables. As shown in Figure 1(a), the user-defined bounds scope the state space in the analysis of original specification. Each relational variable can be assigned a value within the

[1] Abbreviated atom names are chosen for readability, and do not indicate type, as in Kodkod all relations are untyped.

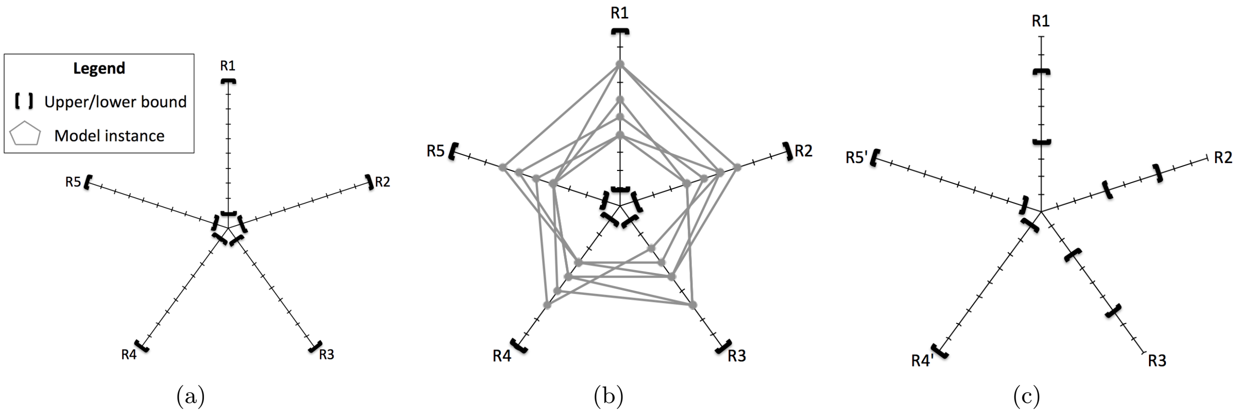


Figure 1: Simplified, schematic view of the Titanium approach, where the dimensions represent relational variables, in this case five hypothetical relational variables R1, R2, R3, R4 and R5: (a) user-defined bounds for the original specification, (b) instance set for the original specification, and (c) tightened bounds for the relations that remain unaffected in the revised specification, i.e., R1, R2 and R3.

user-defined bounds. A value assignment to all relational variables in such a way that do not violate constraints derived from the specification represents a satisfying model instance, and depicted as a pentagon in Figure 1b. The key observation is that once the satisfying model instances are found, we are able to tighten the bounds for a given specification, such that the same specification, or one that is substantially the same, can be solved significantly faster. Figure 1(c) shows a situation in which the revised specification does not affect R1, R2, and R3, and thus Titanium is able to set tighter bounds than those specified by the user for those relations. Of course, changed relations and those newly added or affected by changed relations (R4' and R5' in the case of this example) would maintain the user-defined bounds for the analysis.

Consider Listing 1(b), for example, where a new signature Variable is added, which may hold an Instance, and has a declared Type. The additional fact paragraph then states that each Instance held by a Variable should have types that are subtypes of the Variable's declared Type.

Given the updated specification, this time Titanium leverages the results of the previous run to set a tighter bound for relational variables that have not been changed and have no dependency on the other changed variables. In this particular example, the upper bound for the *subtypes* relation in the updated model is tightened by Titanium (Listing 1, lines 12–13). Out of 9 possible combination of $Type \times Type$ elements, just 4 pairs are valid, as the Object element (*O1*) is the root in the subtype hierarchy (line 11), and it cannot be a subtype for the other Type elements (*T1* and *T2*). This can be easily calculated by taking the union of satisfying model instances from the previous run. The exploration space thus would be reduced. As a result, Titanium is faster in finding a model instance, taking 51 ms for it compared to 275 ms that it takes for Alloy Analyzer to produce the first model instance. The time required to compute the whole instance set would also improve from 1574 ms to 1099 ms, in this simple example.

3. APPROACH

Editing an Alloy specification produces a new finite relational specification, i.e., a Kodkod problem, with finitely many distinct model instances. In this section, we show how solutions for the original specification can potentially be used to narrow the exploration space of the revised specification, and in particular, whether they constitute a partial solution, i.e., a lower bound, and/or a (partial) upper bound for variables in the new specification.

Our algorithm is described in two steps. First, we assume that the set of relations for the two models are equal. We then discuss the general algorithm where the universe of discourse, including relations, may change.

3.1 Basic Reasoning

Definition 1 (instance set)**.** *Let i be a model instance satisfying the model specification $\mathcal{S}$, $i \models \mathcal{S}$. We call $\mathcal{I}(\mathcal{S})$ an instance set for a model specification $\mathcal{S}$, where each model instance $i \in \mathcal{I}(\mathcal{S})$ satisfies the model specification $\mathcal{S}$:*

$$\mathcal{I}(\mathcal{S}) = \{i|\ i \in \mathcal{I}(\mathcal{S})\ \wedge\ i \models \mathcal{S}\})$$

Definition 2 (model specialization)**.** *Let $\mathcal{S}$ and $\mathcal{S}'$ be model specifications defined over the same set of relational variables. We say $\mathcal{S}$ is a specialization of $\mathcal{S}'$, $\mathcal{S} \leq \mathcal{S}'$, if and only if each model in the instance set of $\mathcal{S}$ is also a model instance for $\mathcal{S}'$, $\mathcal{I}(\mathcal{S}) \subseteq \mathcal{I}(\mathcal{S}')$.*

Four different scenarios are possible, depending on the model specialization relation between instance sets of the two specifications:

- **Superset**: $\mathcal{S} \leq \mathcal{S}'$, where the instance set of the revised specification includes the original specification's instance set. $\mathcal{I}(\mathcal{S})$ thus constitutes the lower bound for relations in $\mathcal{S}'$, as each model instance for $\mathcal{S}$ is also a valid instance for $\mathcal{S}'$. They essentially represent a priori known part, i.e., a partial instance, for the changed model specification, $\mathcal{S}'$, reducing the scope of the SAT problem to be solved to find potentially new instances, and thereby improving performance of the analysis.
- **Subset**: $\mathcal{S}' \leq \mathcal{S}$, where the instances of the revised specification are contained in the original specification's instance set. $\mathcal{I}(\mathcal{S})$ thus constitutes the upper bound for relations in $\mathcal{S}'$, as each model instance for $\mathcal{S}'$ is among model instances already found by solving $\mathcal{S}$, relieving the need to be rediscovered.

- **Equivalent**: $\mathcal{S} \leq \mathcal{S}' \wedge \mathcal{S}' \leq \mathcal{S}$, where a set of model instances for $\mathcal{S}'$ is equivalent to that of the model $\mathcal{S}$, or $\mathcal{I}(\mathcal{S}) = \mathcal{I}(\mathcal{S}')$.

- **Arbitrary**: No specialization relation exists, thus no efficiency gains are possible for the analysis of specification $\mathcal{S}'$.

Note that when the user specified scope has been increased in the analysis of the revised specification, Titanium is still able to adjust the lower bound, but not the upper bound. We can then reduce the problem of model specialization to the following propositional formula [40], where $\mathcal{P}(\mathcal{S})$ denotes the propositional formula for a model $\mathcal{S}$:

$$\mathcal{P}(\mathcal{S}) \leq \mathcal{P}(\mathcal{S}') \equiv (P(S) \Rightarrow P(S')) \tag{1}$$

Intuitively, it states that all satisfying solutions to $\mathcal{P}(\mathcal{S})$ are solutions to $\mathcal{P}(\mathcal{S}')$, or more formally, the set of model instances for $\mathcal{P}(\mathcal{S})$ is a subset of the instance set of $\mathcal{P}(\mathcal{S}')$, $\mathcal{I}(\mathcal{P}(\mathcal{S})) \subseteq \mathcal{I}(\mathcal{P}(\mathcal{S}'))$, if and only if $P(S)$ implies $P(S')$.

Note that finite relational models can be represented as propositional models using standard techniques [17, 21, 41]. Indeed, the Alloy Analyzer relies on Kodkod [41] that translates specifications in Alloy's relational logic into propositional formulas, which then can be solved by off-the-shelf SAT solvers. Specifically, a relational model, $\mathcal{S}$, in Kodkod translates into a formula, $\mathcal{P}(\mathcal{S})$, in propositional logic. There is a one-to-one correspondence between a relational model, $\mathcal{S}$, and its counterpart propositional model, $\mathcal{P}(\mathcal{S})$, notwithstanding the peripheral variables introduced as a byproduct of the translation process. The relationship between a model specification and its correspondence model instances defined above for propositional models are thus preserved for finite relational models under this mapping.

For each pair of specifications $\mathcal{S}$ and its revision $\mathcal{S}'$, we can check whether any *model specialization* holds between them with a SAT solver, by determining whether the formula $P(S) \Rightarrow P(S')$ is a tautology (i.e., whether its negation is not satisfiable). However, solving the negation formula $P(S) \wedge \neg P(S')$ can be expensive for large specifications of substantial systems. To alleviate this problem and render it more cost-effective, we adjusted the formula by leveraging the fact that the two specifications have many clauses in common, as one is derived from the other. Specifically, inspired by Thüm et al. [40], we state $P(S)$ and $P(S')$ as follows:

$$\mathcal{P}(\mathcal{S}) = p_S \wedge c \tag{2}$$

$$\mathcal{P}(\mathcal{S}') = p_{S'} \wedge c \tag{3}$$

where p_S and $p_{S'}$ denote the conjunction of clauses exclusive to $\mathcal{P}(\mathcal{S})$ and $\mathcal{P}(\mathcal{S}')$, respectively, and c the common clauses. Because $p_S \wedge c \wedge \neg c$ is a contradiction, the formula $P(S) \wedge \neg P(S') = (p_s \wedge c \wedge \neg p_{s'}) \vee (p_s \wedge c \wedge \neg c)$ then can be rendered as:

$$\mathcal{P}(\mathcal{S}) \wedge \neg p_{S'} \tag{4}$$

The formula can be further simplified as a disjunction of several easier to solve formulas, each one is represented as as a conjunction of $\mathcal{P}(\mathcal{S})$ and a sub-expression in the CNF representation of $p_{S'}$. Specifically, consider $p_{S'}$ with the CNF representation of $p_{S'_1} \wedge p_{S'_2} \wedge \ldots \wedge p_{S'_n}$. The formula $\mathcal{P}(\mathcal{S}) \wedge \neg p_{S'}$ then equals:

$$\bigvee_{1 \leqslant\ i\ \leqslant\ n} \mathcal{P}(\mathcal{S}) \wedge \neg p_{S'_i} \tag{5}$$

The problem of categorization of model changes per our specialization definition is now reduced to a disjunction of several sub-expressions, where each one is significantly smaller and easier to solve than the original formula. Instead of calling a SAT solver to determine satisfiability of a rather large formula, we can use multiple calls to a SAT solver, posing a more tractable sub-expression each time. If any sub-expression determines to be satisfiable, the entire model specialization evaluates to false, possibly before reasoning about all sub-expressions, further improving the effectiveness of the approach. Moreover, except for one clause ($p_{S'_i}$), multiple calls to a SAT solver solve exactly the same formula. This enables leveraging the incremental solving capabilities featured in many modern SAT solvers to make subsequent runs more efficient (each sub-expression is modeled as a separate addition of new constraints, $p_{S'_i}$, to an already solved formula, $\mathcal{P}(\mathcal{S})$).

Algorithm 1: ExtractBase

```
Input: $F : formulas, R : relations$
Output: $\langle\ bRelations,\ bFormulas\ \rangle$
$bFormulas \leftarrow \{\}$
$bRelations \leftarrow \{\}$
for $formula \in F$ do
    $rels = getRelationalVars(formula)$
    if $rels\ \subseteq\ R$ then
        $bFormulas \leftarrow bFormulas \cup formula$
        $bRelations \leftarrow bRelations \cup rels$
    end
end
return $\langle\ bRelations,\ bFormulas\ \rangle$
```

3.2 Extended Reasoning

In the following, we present our general approach for situations where both Alloy specifications are not defined over the same set of relational variables, i.e., relational variables may be added or removed as a result of the specification modification.

Our approach leverages *declarative slicing* [45], which is a program slicing technique applicable to analyzable declarative languages. Declarative slicing partitions a declarative formula specification, such as Alloy and Kodkod, into two slices of *base* and *derived*, where each slice is a disjoint subset of the formula constraints. A base slice is defined by a set of relational variables, called *slicing criterion*, to be constrained by the formula constraints specified in the base slice, and an instance of which represents a partial solution that can be extended by an instance of a derived slice for the entire specification.

More formally, let $S = \langle R, F \rangle$ be a specification, consisting of a set of relational variables R and a set of relational formulas, F, defined over R. Let $S_b.r \subseteq R$ and $S_d.r \subseteq R$ partition R, and $S_b.f \subseteq F$ be the formulas that only involve relations in $S_b.r$. We call $S_b.r$ a base slice for $\mathcal{S}$ if and only if:

$$\forall\ i_b\ \in\ I(S_b) \mid \exists\ i_d\ \in\ I(S_d)\ .\ i_b\ \times\ i_d\ \in\ I(S) \tag{6}$$

```
Algorithm 2: Superset
  Input: $F : S'.formulas, R : S.relations \cap S'.relations$
  Output: $lb$ //adjusted lower bound set
1 $\langle$ bRelations, bFormulas $\rangle \leftarrow ExtractBase(F, R)$
2 $lb \leftarrow S'.lb$
3 if $bFormulas \subseteq S.formulas$ then
4     for $r \in bRelations$ do
5         $lb(r) \leftarrow \bigcap_{i \in I(s)} i.val(r)$
6     end
7 end
8 return $lb$
```

```
Algorithm 3: Subset
  Input: $F : S.formulas, R : S.relations \cap S'.relations$
  Output: $ub$ //adjusted upper bound set
1 $\langle$ bRelations, bFormulas $\rangle \leftarrow ExtractBase(F, R)$
2 $ub \leftarrow S'.ub$
3 if $bFormulas \subseteq S'.formulas$ then
4     for $r \in bRelations$ do
5         $ub(r) \leftarrow \bigcup_{i \in I(s)} i.val(r)$
6     end
7 end
8 return $ub$
```

To derive a base slice from the revised specification in a way that its relations are shared with those of the original specification, we use a constraint partitioning algorithm.

Algorithm 1 outlines the partitioning process. It gets as input a set of relational variables, R, and a set of relational formulas, F. Without loss of generality, we assume that F is a conjunction of several sub-formulas, i.e., $F = \wedge formulas$. As an example, the formula in Listing 2, line 10, represents this form for the constraints specifications in our running example (Listing 1). The algorithm then iterates over each such sub-formulas, extracts the set of relational variables, *rels*, constrained by the given formula, and evaluates it against the given set of relational variables, R. If the formula's variable set, *rels*, is a subset of R, it is added to the formulas in the base slice, *bFormulas*, and *rels* to the base slice relation set, *bRelations*, whose bounds will potentially be adjusted.

As a result of calling *ExtractBase* with the formulas of a specification and the shared relational variables of another specification, the algorithm produces a bases slice. We thus can reason about and update the bounds of the revised specification base slice according to the model specialization relations described in Section 3.1.

Algorithm 2 computes the lower bound for the *superset* scenario outlined in Section 3.1. It first calls *ExtractBase* with a a set of relational formulas defined in S' and a set of relational variables shared between the two specifications as inputs. Note that the base slice relation set, *bRelations*, is a subset of those relations shared between the two specifications, and they are not necessarily equal. The algorithm then evaluates a set of formulas in the base slice against those specified in S. If the extracted formulas form a subset of $S.formulas$, that means S_{base} is a model specialization of S'_{base}, or $S_{base} \leq S'_{base}$, as *formula (1)* is a tautology. The instance set of S_{base} thus constitutes the lower bound for the corresponding relational variables in S'. Recall from Section 2 that a lower bound for a relational variable represents the tuples that the variable must contain. The intersection of values assigned to a relation in all instances thus constitutes a lower bound for that relational variable.

Algorithm 3 computes the upper bound for the *subset* scenario outlined in Section 3.1. It first calls *ExtractBase* with a set of relational formulas defined in S and a set of relational variables shared between the two specifications as inputs, because this time we want to see whether S'_{base} is a model specialization of S_{base}. It then evaluates a set of formulas in the base slice against those specified in S' (rather than S as done in Algorithm 2). If the extracted formulas is a subset of $S'.formulas$, that means $S'_{base} \leq S_{base}$. The instance set of S_{base} thus constitutes the upper bound for their corresponding relational variables in S'. Recall that an upper bound for a relational variable represents the tuples that a relational variable may contain. The union of values assigned to a relation in all instances thus constitutes an upper bound for that relational variable.

Finally, in the case of *equivalent* relation (recall Section 3.1), both upper and lower bounds need to be tightened. In essence, the equivalent relation implies the existence of both superset and subset relations (i.e., $\mathcal{S} \leq \mathcal{S}' \wedge \mathcal{S}' \leq \mathcal{S}$). Titanium calculates the bounds for equivalent relations by making consecutive calls to Algorithms 2 and 3.

Titanium, similar to Alloy, is both sound and complete for the given bounds, yet more efficient for analysis of evolving specifications. Space constraints prevented us from including a proof. But they follow naturally from the algorithms in Section 3.2 and the discussions, backed with mathematical notations in Section 3.1.

Note that because the problem of reasoning about Alloy model edits has been reduced to a satisfiability problem, as presented in Section 3.1, it could still have an exponential run-time in the worst case. Therefore, in the implementation (line 4 in Algorithms 2 and 3), we have taken a computationally effective approach, in which whenever $p_{S'_i}$ in formula (5) is not equal to true, we conclude that the formula $P(S) \Rightarrow P(S')$ is not a tautology, and skip the adjustment of the corresponding bound (either lower or upper). In the next section, we evaluate the execution time of our algorithm empirically and show that the proposed optimization technique achieves significant improvement compared to the Alloy Analyzer.

4. TOOL IMPLEMENTATION

We have implemented Titanium as an extension to the Alloy relational logic analyzer. To implement the algorithms presented in the previous sections, Titanium modifies both the Alloy Analyzer and its underlying relational model finder, Kodkod [41]. The differences between Alloy Analyzer and Titanium lie in the translation of high-level Alloy models into low-level bounded relational models, and the facility to effectively determine the scopes of the updated models given the instance set of the original specification. The Titanium tool is publicly available at the project website [4]

5. EXPERIMENTAL EVALUATION

This section presents the experimental evaluation of Titanium. Our evaluation addresses the following research questions:

- **RQ1** What is the performance improvement achieved by Titanium's incremental analysis compared to Alloy Analyzer?

Table 1: Results for publicly available and automatically extracted Alloy specifications.

		Alloy Analyzer 4.2			Titanium			
Specification	#Rels	Vars	Clauses	Analysis Time (S)	Vars	Clauses	Adj. Time (S)	Analysis Time (S)
Decider	47	2,384	3,526	2.549	1,819	2,657	0.507	0.533
Wordpress	54	2,419	3,479	3.718	1,385	1,615	0.245	1.198
Moodle	39	1,054	1,477	2.370	1,034	1,453	0.014	1.598
Ecommerce	70	3446	4705	5.462	1,618	1,682	0.133	0.711
DBLP	80	385	498	1.967	279	352	0.507	1.174
Library	78	254	341	7.411	129	177	0.616	0.510
Coach	86	275	369	0.260	21	24	0.032	0.031
WebML	47	122	178	0.738	119	172	0.050	0.592
GMF Graph	36	350	439	6.186	186	223	0.275	2.488
App Bundle 1	1,288	1,570,839	79,332,736	427.405	5,589	11,822	11.344	16.202
App Bundle 2	1,313	1,874,758	81,500,102	673.887	5,690	12,657	11.070	25.052
App Bundle 3	1,175	1,233,843	63,408,733	168.225	4,835	10,278	10.461	4.235
App Bundle 4	1,064	1,195,927	49,633,639	242.387	4,756	10,082	8.788	8.431
App Bundle 5	1,185	1,454,029	55,447,247	451.082	5,360	11,489	10.990	17.527

- **RQ2** What is the overhead of Titanium? How does the Titanium's overhead relate to the size of the original instance set?
- **RQ3** How do the efficiency gains in Titanium relate to the extent of change in the specification?

Our experimental subjects are Alloy specifications drawn from a variety of different sources and of different problem domains. These specifications further vary much in terms of size and complexity. We have compared the performance of Titanium to that of the Alloy Analyzer (version 4.2) on three sets of specifications:

- *Publicly available Alloy specifications.* We used several Alloy specifications taken from the work of other researchers that were publicly available [11, 14].
- *Extracted specifications.* We used several specifications extracted automatically by various analysis techniques that leverage Alloy for analysis of real-world software systems [9, 27].
- *Automatically synthesized specifications.* We developed a tool for generating a large number of synthesized Alloy specifications.

The complete list of subject systems and their specifications are available from the project website [4].

5.1 Improvements in Practice

For an initial evaluation, we wanted to assess the kinds of improvement one could expect in practice. We thus used several Alloy specifications that were either publicly available or automatically extracted from real-world software systems, as shown in Table 1.

Decider [2] is an object model of a system to support design space exploration. Its model contains 10 Classes, 11 Associations, and 5 inheritance relationships, all represented as signature extensions in Alloy.

WordPress and Moodle are object models from two open-source applications, obtained by reverse engineering their database schemas. The WordPress model, which is an open source blog system [5], includes 13 classes connected by 10 associations with 8 inheritance relationships. Moodle is a learning management system [3], widely used in colleges and universities. Its model has 12 classes connected by 8 associations and consists of 4 inheritance relationships.

Ecommerce is the object model of an E-commerce system adopted from Lau and Czarnecki [25], that represents a common architecture for open source and commercial E-commerce systems. It has 15 classes connected by 9 associations with 7 inheritance relationships.

The next five specifications, i.e., DBLP, Library, Coach, WebML, GMF Graph, are class diagrams automatically transformed into Alloy specifications by the CD2Alloy apparatus [27]. The selected class diagrams are all publicly available, taken from different sources, and previously published as case studies and research papers in the area of UML analysis.

Finally, the last five rows represent large specifications intended for the assessment of security properties in mobile platforms. Each one represents a bundle of Android apps installed on a mobile device for detecting security vulnerabilities that may arise as a result of inter-application communication, adopted from [1].

For some of the specifications whose change histories were available, we actually used different versions of the specification, which had been manually developed or automatically extracted at different times. For others, we made three to five changes to each specification, such that the updated specifications still had valid model instances. We used a PC with an Intel Core i7 2.4 GHz CPU processor and 8 GB of main memory, and leveraged SAT4J as the SAT solver to keep the extraneous variables constant during all the experiments. We then compared the performance of Titanium to that of the Alloy Analyzer (version 4.2) in all these revised specifications.

The results are provided in Table 1. The table shows the number of relations for each specification, the size of the generated CNF, given as the total number of variables and clauses, the analysis time taken for the model finder, as well as the overhead time taken by Titanium for adjusting the analysis bounds. As shown, for some experiments, the size of CNF variables generated by Titanium is less than those generated by Alloy. This is because relations with exact bound, that essentially represent partial instances, do not need to be translated into a SAT formula, thus reducing the size of the generated CNF.

The results demonstrate the effectiveness of our algorithm, as in every case, and for every update, the analysis time taken by Titanium for computing the instance set of modified specifications is less than that of using the Alloy Analyzer. However, we can also see that the results could vary greatly, because the improvements could depend on several factors, most notably the amount of change, and the size of instance set. This called for further evaluation to determine how such variations affect the efficiency gains, as described next.

5.2 Efficiency vs. Size of Instance Set

Since in Titanium we use the instance sets from the prior run to tighten the bounds for the next run, we expect the efficiency gains to be more pronounced in cases with larger instance sets. In this set of experiments, we attempted to corroborate our intuition and obtain empirical evidence of this relationship. Since we needed access to a large number of Alloy specifications and their revisions, we developed a tool for generating synthesized Alloy models.

Independent variables in our experiments are (a) the size of an Alloy model, represented as the total number of signatures and fields, as both are indeed translated into relations in the underlying relational logic, (b) the number of update operations, and (c) the type of update operations. As dependent variables, we measured the time needed to update the bounds and the time needed to determine the instance set for the updated model. We discarded all synthesized Alloy specifications that do not have any valid model instances, and repeated the entire process until the appropriate number of models were generated. In the following, we describe the approach taken for synthesis of Alloy specifications and their revisions.

Alloy specification synthesis. Our tool for synthesizing Alloy specifications takes as input ranges for the size of signatures, fields, and formulas and generates an Alloy specification as follows. It starts with the top-level signatures, which are signatures that are declared independent of any other signature and do not extend another signature. A top-level signature may also be defined as an abstract signature, meaning that it has no elements except those belonging to its extensions, or defined as a nonempty signature. The generator then iterates over the set of top-level signatures, and adds sub-signatures. Within the same iteration, singleton signatures—that contain a single element—are also added. In the next step, fields are added as multi-relations (in addition to signatures that represent unary relations), whose domains and ranges are given by the already defined signatures.

The last part of the generated module is a set of constraints in a freestanding fact paragraph. The constraints are generated using formula templates that cover operators defined in the Alloy core language [6], including subset and equality operators in comparison formulas, conjunction and universal quantification formulas, and some binary (union (+), intersection (&), join (.) and product (−>)) and unary (transpose (∼) and transitive closure (ˆ)) expressions. Note that the other types of Alloy formulas, such as existential quantification and disjunction, can be derived from the core language the generator supports. As a concrete example of a synthesized specification, Listing 4 shows an Alloy specification automatically generated, given 6, 5, and 4 as the size of signatures, fields and constraint formulas, respectively.

Revised specification synthesis. To produce an update for an Alloy model, our generator takes as input an Alloy model and the number of edits. It supports the following edit operations on the Alloy models:

```
abstract sig A0 {
  f3: some (A3 + A1)
}
sig A1 {
  f0: one A0
}
sig A2 extends A0 { }
one sig A3 {
  f1: some A2,
  f4: one  A4
}
sig A4 { }
one sig A5 extends A2 {
  f2: lone A0
}

fact{
  (all o: A5| some o.(A5 <: f2))
  (all o: A3| # o.(A3 <: f1) <= # o.(A3 <: f4))
  # A5 = # A1
  # A2 = 3
}
pred show{}
run show for 4
```

Listing 4: An automatically generated Alloy specification.

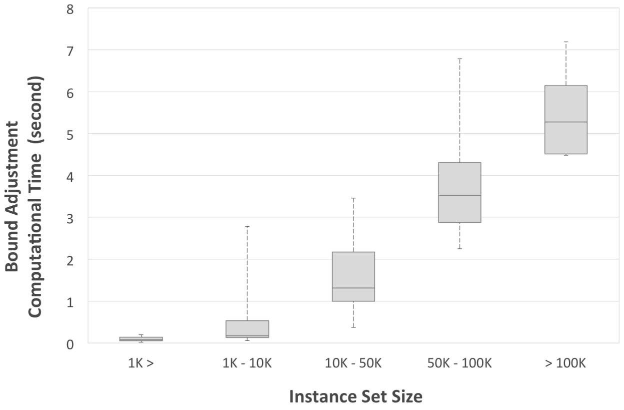
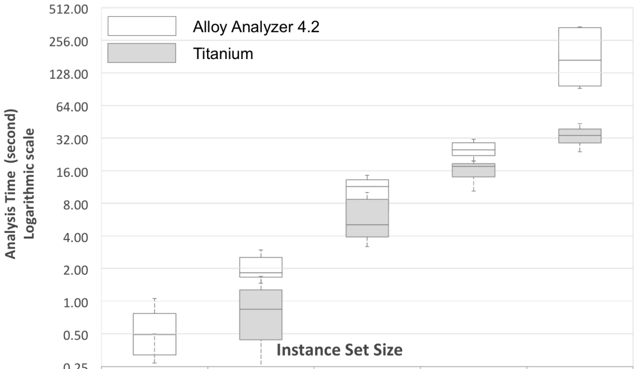


Figure 2: Analysis time for bound adjustment over the size of the original specification instance set.

- create or delete a signature without children;
- change the signature multiplicity[2], i.e., to set, one, lone or some (one that is different from the multiplicity defined in the original specification);
- make an abstract signature non-abstract or vice versa;
- move a sub-signature to a new parent signature;
- add a new field to a signature declaration or delete a field;
- change a multiplicity constraint in a field declaration;
- finally, create a new constraint or delete a constraint.

While the first six types of edits revise relations in the specification, the last one modifies the specification's formulas. The generation and execution of Alloy models and their updates are done using Alloy's APIs.

[2] A signature multiplicity constrains the size of a set corresponding to that signature type.



	1K >	1K - 10K	10K - 50K	50K - 100K	> 100K
75th percentile	0.77	2.54	13.26	29.08	334.90
Median	0.49	1.84	11.46	25.05	168.23
25th percentile	0.32	1.68	6.88	22.14	97.06
75th percentile	0.21	1.26	8.67	18.48	38.55
Median	0.18	0.84	5.07	17.52	33.64
25th percentile	0.06	0.44	3.89	13.92	28.73

Figure 3: Performance comparison: Analysis time taken for Alloy Analyzer and Titanium vs. the size of the instance set.

Experimental Results. We ran the generator with parameters specifying the size of signatures, fields, and formulas varying in the ranges of 5–15, 5–10, and 5–10, respectively. The number of edits to create a revised specification ranged from 1 to 7. For each edit, one of the 7 types of operation was randomly selected and applied. We generated 500 original specifications, and another 500 corresponding revised specifications, for a total of 1,000 specifications.

Figure 2 shows the boxplots of the analysis time for bound adjustment over the varying size of instance sets for the original specifications. According to the diagram, the execution time increases roughly linearly with the size of instance sets, and for a space of size 224,856, it takes just about 7.1 seconds for Titanium to produce an adjusted bound set, showing that the Titanium approach is effective in practice on specifications with large-scale instance sets.

We then compared performance of Titanium with the Alloy Analyzer 4.2. In Figure 3, we show the results of our measurements, comparing the analysis time taken by each of the tools as boxplots with a logarithmic scale. On average, Titanium exhibited a 61% improvement over that of the Alloy Analyzer. For specifications with very small instance sets, the difference in performance of the two techniques is negligible, yet the effects of adjusted bounds are clearly visible when the size of instance sets increases. As illustrated in the diagram, the analysis time by the Alloy Analyzer grows faster than the corresponding time for Titanium. In summary, Titanium is able to analyze the revised specifications in a fraction of time it takes to run Alloy Analyzer, and the difference in analysis time is more pronounced for the larger instance sets, as we expected.

5.3 Efficiency vs. Extent of Change

We then assessed how the efficiency gains in Titanium relate to the extent of change. As a given specification diverges from the original specification, and the shared variables and formulas are reduced, the efficiency gains are expected to gradually diminish. In this set of experiments, we attempted to corroborate this expectation, and to obtain an empirical understanding of this relationship.

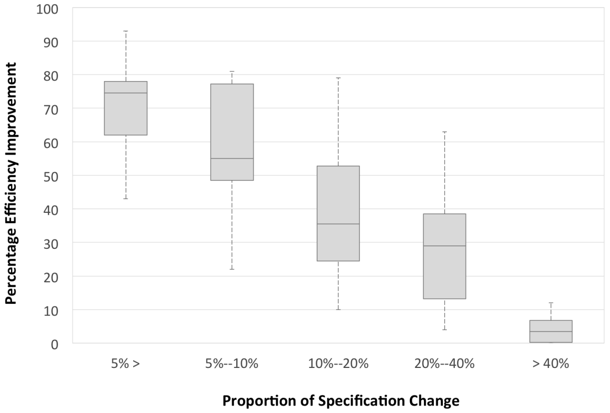


Figure 4: Percentage of average improvement vs. the proportion of change for model specifications of size 20 relations.

We generated a specification with a fixed size of 20 relations, and automatically revised the specification by randomly applying the edit operators described in the prior section on 1 to 10 of its relations, resulting in a revised specification with 5% to 50% change. We then measured the time taken by both Alloy Analyzer and Titanium in analyzing the revised specification. We repeated this experiment for 100 times. In this way, we were able to determine whether more changes decrease the efficiency gains in analysis achieved through bound adjustments.

The boxplots in Figure 4 show the efficiency gains of using Titanium over Alloy Analyzer. On average, for specifications of size 20 relations, one can expect to obtain more than 50% reduction in analysis time (compared to that of the Alloy Analyzer) for up to 10% change in the specification. After that, the efficiency achieved through bound adjustments decrease, and for changes above 40% of specification, the improvements are reduced to less than 5%. Thus, the extent to which edits can negatively affect Titanium's efficiency gains depends on the proportion of the original specification that has changed.

6. RELATED WORK

Much work is related to this research. Here, we provide a discussion of the related efforts in light of our research.

Alloy extensions. The widespread use of Alloy has driven a number of extensions to the language and its underlying automated analyzer [29, 30, 32, 43]. Certain techniques have been developed for exploring model instances from Alloy's relational logic constraints [16, 26, 32, 37, 41]. Among others, Aluminum [32] extends the Alloy Analyzer to generate minimal model instances. It relies on a procedure in which tuples are iteratively removed from the tuple set of found model instances until a minimal instance is reached. Macedo et al. [26] studied the space of possible scenario explorations in the context of relational logic. This work, similar to Aluminum [32], mainly focuses on the order in which model instances are explored, rather than facilitating the exploration of the solution space for evolving models. Torlak and Jackson introduced a heuristic, polynomial-time algorithm to substantially exclude symmetric instances of relational models [42], given that such model instances present no additional information. Identifying isomorphisms of relational models has no known polynomial-time solution. Montaghami and Rayside [29] extended Alloy to explicitly support specification of partial models. However, this research effort does not consider the analysis of evolving specifications. Indeed, it is commonly acknowledged that development of efficient techniques for the analysis of Alloy specifications is a much needed area of research [43]. However, to the best of our knowledge, no prior research has attempted to optimize the performance of analysis time for evolving Alloy specifications.

Incremental analysis. The other relevant thrust of research has focused on incremental solving of constraints specified in the first-order logic [20, 44–46]. Among others, Uzuncaova and Khurshid partitioned a model of constraints into a base and derived slices, where solutions to the base model can be extended to generate a solution for the entire model [45]. Titanium is fundamentally different in that the problem addressed by Uzuncaova and Khurshid is in the context of a fixed specification and the evolution of specification is not considered. Moreover, the two approaches use declarative slicing for totally different purposes: In [45], declarative slicing is used to prioritize constraints (to first analyze constraints with higher priorities). However, Titanium uses declarative slicing to identify a base set of relations, the bounds of which can potentially be tightened in the analysis of evolving specification. Ranger [38] uses a divide and conquer method relying on a linear ordering of the solution space to enable parallel analysis of specifications written in first-order logic. While the linear ordering allows for partitioning of the solution space into ranges, there is no clear way in which it can be extended with incremental analysis capabilities, essential for analysis of evolving systems.

Effectively reducing the exploration space has been used in a variety of forms to optimize bounded analysis techniques [7, 18, 19, 35]. Galeotti et al. [18, 19] presented a technique, called TACO, that targets efficient analysis of JML-specifications for linked data structures, through translating them into the Alloy language. TACO eliminates the values that violate constraints introduced by class invariants via adjusting only the upper bounds for the translated Alloy fields. Titanium, however, is (1) a general solution, independent of any particular domain, (2) capable of adjusting both upper and lower bounds, and (3) aimed at efficient analysis of any evolving Alloy specification. To the best of our knowledge, Titanium is the first, general solution that supports analysis of evolving Alloy specifications.

Optimization of other techniques relying on constraint solving. Related to our research are the applications of constraint solving techniques to software engineering problems. Of particular relevance is symbolic execution of software, which is a means of analyzing a program to determine what inputs cause each part of a program to execute. A software program is first abstractly interpreted to identify a set of symbolic values for inputs and conditional expressions, which when solved with the aide of a solver, produce concrete values to exercise different branches of the program. Similar to Alloy Analyzer, due to their reliance on SAT solving engines, symbolic execution tools face scalability problems. Substantial recent research has focused on improving the performance of symbolic evaluation techniques [23, 34, 47, 48].

Some of these approaches [23, 47] follow the general strategy of storing and reusing previously solved constraints, which result in less calls to the solver, thereby improving the performance. But most closely related to our research is regression symbolic execution [34, 36, 48], where one attempts to co-analyze two program versions which are, often, very similar. Here, the differences between two versions of a program are first identified, and the new run of the symbolic execution on the revised program is then only guided through the regions of the program that have changed. Similar to all of these approaches, Titanium aims to improve the performance of analysis for Alloy specifications. However, in addition to targeting a different type of analysis (i.e., formal specifications rather than programs), it employs a different technique that uses the previously calculated instances to tighten the bounds on shared relational variables.

7. CONCLUSION

Alloy has found applications in a variety of software engineering problems, from automated synthesis and exploration of design alternatives [13, 14, 22] to analysis of programs [8, 9, 31, 39] and generation of tests [24, 28]. The development of solutions for automatically extracting Alloy specifications from software artifacts has made Alloy practical for use even after the deployment of software, and possibly at runtime [8, 10, 31]. Such applications of Alloy, however, are challenged by the time it takes for an analysis to run, especially given that the analysis may need to be repeated frequently.

We presented an approach and an accompanying tool, dubbed Titanium, that significantly reduces the time it takes to analyze evolving Alloy specifications. While the approach is particularly suitable in settings where a specification is kept in sync with the changing software system, it could also be as effective in settings where a specification is incrementally developed, often involving repeated analysis of the specification to assess its semantics. Titanium is able to achieve a significant speed-up by tightening the analysis bounds without sacrificing soundness and completeness. It first identifies the shared relational variables between two versions of a given specification. It then uses the instances produced for the original specification to determine a tighter bound for the revised specification, thereby reducing the state space, enabling the SAT solver to find the model instances for the revised specification at a fraction of time needed for Alloy Analyzer. Our experimental results using both real Alloy specifications constructed in the prior work, as well as synthesized Alloy specifications, corroborate the significant performance gains achieved through Titanium.

While the results obtained so far are quite promising, we believe further improvements are possible. Specifically, in spite of the adjustments made to the analysis bounds, the solver still needs to solve for the shared constraints. A promising avenue of future research is a memoization-based approach, were the constraints solved in a prior analysis of the model are stored, and retrieved as encountered in the subsequent analyses. Such an approach would not eliminate the need for adjusting the bounds for relational variables, as some of those variables may be used in the derived specification.

We have made Titanium, as well as Alloy specifications and the model synthesizer used in conducting our experiments, publicly available for use by other researchers [4].

Acknowledgment

The authors are grateful to Daniel Jackson for the discussions regarding the initial idea of this work. We also thank Tim Nelson for his help on Alloy's symmetry breaking.

8. REFERENCES

[1] Alloy models from the covert project. http://www.sdalab.com/projects/covert.

[2] Alloy models from the trademaker project. http://www.jazz.cs.virginia.edu:8080/Trademaker/data/models.zip.

[3] Moodle. http://docs.moodle.org/dev/Grades#Database_structures.

[4] Titanium. https://seal.ics.uci.edu/projects/titanium.

[5] WordPress. http://codex.wordpress.org/Database_Description/3.3.

[6] *D. Jackson, Software Abstractions, 2nd ed. MIT Press, 2012.* MIT Press, 2012.

[7] P. Abad, N. Aguirre, V. Bengolea, D. Ciolek, M. F. Frias, J. Galeotti, T. Maibaum, M. Moscato, N. Rosner, and I. Vissani. Improving Test Generation under Rich Contracts by Tight Bounds and Incremental SAT Solving. In *Verification and Validation 2013 IEEE Sixth International Conference on Software Testing*, pages 21–30, Mar. 2013.

[8] H. Bagheri, E. Kang, S. Malek, and D. Jackson. Detection of design flaws in the android permission protocol through bounded verification. In N. BjÃÿrner and F. de Boer, editors, *FM 2015: Formal Methods*, volume 9109 of *Lecture Notes in Computer Science*, pages 73–89. Springer International Publishing, 2015.

[9] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek. COVERT: Compositional Analysis of Android Inter-App Permission Leakage. *IEEE Transactions on Software Engineering*, 41(9):866–886, Sept. 2015.

[10] H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek. Practical, formal synthesis and automatic enforcement of security policies for android. In *Proceedings of the 46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 514–525, 2016.

[11] H. Bagheri and K. Sullivan. Monarch: Model-based development of software architectures. In *Proceedings of the 13th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 376–390, 2010.

[12] H. Bagheri and K. Sullivan. Pol: Specification-Driven Synthesis of Architectural Code Frameworks for Platform-Based Applications. *ACM SIGPLAN Notices*, 48(3):93–102, Mar. 2013.

[13] H. Bagheri and K. Sullivan. Model-driven synthesis of formally precise, stylized software architectures. *Formal Aspects of Computing*, 28(3):441–467, Mar. 2016.

[14] H. Bagheri, C. Tang, and K. Sullivan. Trademaker: Automated dynamic analysis of synthesized tradespaces. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 106–116, New York, NY, USA, 2014. ACM.

[15] J. Boyland, A. Sloane, X. Li, D. Shannon, J. Walker, S. Khurshid, and D. Marinov. Analyzing the Uses of a Software Modeling Tool. *Electronic Notes in Theoretical Computer Science*, 164(2):3–18, Oct. 2006.

[16] A. Cunha, N. Macedo, and T. Guimaraes. Target oriented relational model finding. In *Proc. of International Conference on Fundamental Approaches to Software Engineering*, FASE'14, pages 17–31, 2014.

[17] J. Edwards, D. Jackson, E. Torlak, and V. Yeung. Faster constraint solving with subtypes. In *Proc. of International Symposium on Software Testing and Analysis*, ISSTA'04, 2004.

[18] J. P. Galeotti, N. Rosner, C. G. Lopez Pombo, and M. F. Frias. Analysis of invariants for efficient bounded verification. In *Proceeding of International Symposium on Software Testing and Analysis*, ISSTA'10, pages 25–36, 2010.

[19] J. P. Galeotti, N. Rosner, C. G. L. Pombo, and M. F. Frias. Taco: Efficient sat-based bounded verification using symmetry breaking and tight bounds. *IEEE Trans. Software Eng.*, 39(9):1283–1307, 2013.

[20] S. Ganov, S. Khurshid, and D. E. Perry. Annotations for alloy: Automated incremental analysis using domain specific solvers. In *Proc. of ICFEM*, pages 414–429, 2012.

[21] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[22] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 62–73, New York, NY, USA, 2001. ACM.

[23] X. Jia, C. Ghezzi, and S. Ying. Enhancing reuse of constraint solutions to improve symbolic execution. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 177–187, New York, NY, USA, 2015. ACM.

[24] S. Khurshid and D. Marinov. Testera: Specification-based testing of java programs using sat. *Automated Software Engineering*, 11:2004, 2004.

[25] S. Q. Lau. *Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates*. Master's thesis, University of Waterloo, Canada, 2006.

[26] N. Macedo, A. Cunha, , and T. Guimaraes. Exploring scenario exploration. In *Proc. of International Conference on Fundamental Approaches to Software Engineering*, FASE'15, pages 301–315, 2015.

[27] S. Maoz, J. Ringert, and B. Rumpe. Cd2alloy: Class diagrams analysis using alloy revisited. In *Proceedings of the 14th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 592–607, 2011.

[28] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek. Reducing Combinatorics in GUI Testing of Android Applications. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 559–570. ACM, 2016.

[29] V. Montaghami and D. Rayside. Extending alloy with partial instances. In *Proc. of ABZ*, pages 122–135, 2012.

[30] V. Montaghami and D. Rayside. Staged evaluation of partial instances in a relational model finder. In *Proc. of ABZ*, pages 318–323, 2014.

[31] J. P. Near and D. Jackson. Derailer: Interactive security analysis for web applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 587–598, New York, NY, USA, 2014. ACM.

[32] T. Nelson, S. Saghafi, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Aluminum: principled scenario

exploration through minimality. In *Proc. of International Conference on Software Engineering*, ICSE'13, pages 232–241, 2013.

[33] J. Nijjar and T. Bultan. Bounded verification of ruby on rails data models. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 67–77, New York, NY, USA, 2011. ACM.

[34] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 504–515, New York, NY, USA, 2011. ACM.

[35] P. Ponzio, N. Rosner, N. Aguirre, and M. F. Frias. Efficient tight field bounds computation based on shape predicates. In *Proc. of International Symposium on Formal Methods*, FM'14, pages 531–546, 2014.

[36] D. Ramos and D. Engler. Practical, low-effort equivalence verification of real code. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 669–685. Springer Berlin Heidelberg, 2011.

[37] N. Rosner, J. H. Siddiqui, N. Aguirre, S. Khurshid, and M. F. Frias. Ranger: Parallel analysis of alloy models by range partitioning. In *Proc. of ASE*, pages 147–157, 2013.

[38] N. Rosner, J. H. Siddiqui, N. Aguirre, S. Khurshid, and M. F. Frias. Ranger: Parallel analysis of alloy models by range partitioning. In *Proceeding of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 147–157, 2013.

[39] M. Taghdiri. Inferring specifications to detect errors in code. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ASE '04, pages 144–153, Washington, DC, USA, 2004. IEEE Computer Society.

[40] T. Thüm, D. Batory, and C. Kastner. Reasoning about edits to feature models. In *Proc. of International Conference on Software Engineering*, ICSE'09, 2009.

[41] E. Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications*. PhD thesis, MIT, Feb. 2009.

[42] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'07, pages 632–647, Berlin, Heidelberg, 2007. Springer-Verlag.

[43] E. Torlak, M. Taghdiri, G. Dennis, and J. P. Near. Applications and extensions of alloy: past, present and future. *Mathematical Structures in Computer Science*, 23(4):915–933, 2013.

[44] E. Uzuncaova and S. Khurshid. Kato: A program slicing tool for declarative specifications. In *Proc. of International Conference on Software Engineering*, ICSE'07, pages 767–770, 2007.

[45] E. Uzuncaova and S. Khurshid. Constraint prioritization for efficient analysis of declarative models. In *Proc. of International Symposium on Formal Methods*, FM'08, 2008.

[46] E. Uzuncaova, S. Khurshid, and D. S. Batory. Incremental test generation for software product lines. *IEEE Trans. Software Eng.*, 36(3):309–322, 2010.

[47] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 58:1–58:11, New York, NY, USA, 2012. ACM.

[48] G. Yang, C. S. Păsăreanu, and S. Khurshid. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 144–154, New York, NY, USA, 2012. ACM.