Aluminum: Principled Scenario Exploration through Minimality

Tim Nelson[1], Salman Saghafi[1], Daniel J. Dougherty[1], Kathi Fisler[1], Shriram Krishnamurthi[2]
[1]Department of Computer Science
WPI
Worcester, MA 01609, USA
[2]Computer Science Department
Brown University
Providence, RI 02912, USA
Corresponding Email: tn@cs.wpi.edu

***Abstract*—Scenario-finding tools such as Alloy are widely used to understand the consequences of specifications, with applications to software modeling, security analysis, and verification. This paper focuses on the exploration of scenarios: which scenarios are presented first, and how to traverse them in a well-defined way.**

We present Aluminum, a modification of Alloy that presents only *minimal* scenarios: those that contain no more than is necessary. Aluminum lets users explore the scenario space by adding to scenarios and backtracking. It also provides the ability to find what can consistently be used to extend each scenario.

We describe the semantic basis of Aluminum in terms of minimal models of first-order logic formulas. We show how this theory can be implemented atop existing SAT-solvers and quantify both the benefits of minimality and its small computational overhead. Finally, we offer some qualitative observations about scenario exploration in Aluminum.

I. Introduction

A. The Uses and Benefits of Scenarios

In many software engineering situations, authors of specifications in a high-level, declarative language (such as first-order logic) benefit from *scenarios*, which are instances of the specification. For instance, a user might specify a filesystem, against which a scenario-generating tool would generate directory structures populated with files (using invented names and contents). When a size-bound on scenarios is given, or can be computed [1], users can even obtain an exhaustive set of scenarios: that is, the number of scenarios is finite and they can all be generated. Tools such as Alloy [2] and Margrave [3], [4] support scenario-finding as their primary activity.

The concreteness of scenarios makes scenario-finding a popular technique in software engineering. Software modeling helps system designers understand the consequences of their specifications, determine missing constraints, and explore alternatives. Specification languages like UML benefit from scenario-finding to help bridge concrete and abstract representations. Scenarios are also useful for presenting counter-examples to verification tasks. This is especially widely used when studying security policies, where the concreteness helps envision attack configurations [3], [4], [5], [6]. Network modeling systems similarly use scenarios to visualize designs [7], [8]. Indeed, tools like Alloy use scenarios for both exploration and counter-examples, and also let users visualize scenarios in a variety of ways, both textual and graphical.

In short, scenarios are useful for many reasons:

- They are concrete, making it easy for users to grasp the output and map it to reality.
- They do not require logical expertise to grasp. Thus, a logic-aware modeler working with a domain expert innocent of the joys of logic can present the output scenarios to the domain expert, who should be in a position to understand them.
- Despite the above, they are rigorous, in that we can ascribe a precise semantics to them. This makes it possible to employ them in formal settings.

Because of this unique combination of characteristics, scenarios are attractive software engineering tools.

B. Scenarios, Formally

Because the term "scenario" has many informal meanings, it helps to pin down our terminology. A *specification* is a first-order logic description written by a user, e.g., in Alloy syntax. This will include an (Alloy) command to be run: the result of running a command is a set of *models.* Here "model" has its traditional meaning from logic: an assignment of values to variables that makes a formula true. A model can be either *propositional* or *relational*, the latter being the structures appropriate to first-order logic; we need to refer to both, because our specifications are first-order but the underlying SAT-solver produces propositional models. Which one we mean will usually be clear from context, but where necessary we will disambiguate. Finally, a *scenario* is a relational model that is shown to the user. It may thus have embellishments for compelling visual presentation, such as atom names drawn from the specification. Nevertheless, because its semantic content is just a relational model, we will feel free to use "scenario" and "model" interchangeably whenever it is clear that we are in a non-propositional context.

C. Principles of Scenario Exploration

Almost any interesting specification will have many scenarios; in fact, most first-order logic specifications will have an infinite number of them. But even when scenarios are constrained to be finite (for example by the imposition of a size-bound), so that there are only finitely many distinct scenarios, making scenario-finding effective requires focusing on what scenarios to present, in what order, and how to help users navigate them. There is some work on this: for instance, Alloy tries to exclude isomorphic scenarios [9], on the grounds that these present no additional information. Beyond that, however, Alloy lets the underlying SAT-solver dictate the order of presentation, which is effectively unordered, and lets users go from one to the next, again with no semantics associated with the order of presentation.

Alloy does, however, hope to present scenarios in a sensible order. As Jackson's book [2, page 7] explains (*instance* is Alloy's name for a relational model):

> [T]he tool's selection of instances is arbitrary, and depending on the preferences you've set, may even change from run to run. In practice, though, the first instance generated does tend to be a small one. This is useful, because the small instances are often pathological, and thus more likely to expose subtle problems.

In other words, Alloy *wishes* to present the smallest scenarios first. It is therefore natural to ask whether it is possible to force it to do so. There are two difficulties we might encounter: semantic (can this be computed?), and performance (can it be done efficiently?), each of which can be an obstacle.

D. Contributions

In this paper, we present a theory for scenario exploration, which has been implemented in a modified version of Alloy called **Aluminum**. Aluminum has the following features:

- It *presents minimal* scenarios. Thus, when confronting a scenario given by Aluminum, a user can be confident that every tuple[1] in the scenario is necessary for that scenario to satisfy the specification's constraints. When a user chooses to view another scenario, Aluminum ensures this too is minimal. By browsing the initial set of scenarios, the user can quickly obtain a sense of the scope of scenarios engendered by the specification.
- Aluminum allows the user to *augment* a scenario with a tuple. Here again Aluminum computes a minimal scenario, which includes any other tuples that may necessarily follow from the augmentation.
- For a given scenario and specification, Aluminum *computes the set of tuples consistent with that scenario*—that is, consistent with the specification— but not currently realized. These suggest natural ways to augment the scenario, and hence continue the exploration.

[1]We use *tuple* to represent an atomic truth (or falsity) instead of the standard logical *fact* in order to avoid potential confusion with Alloy's **fact** keyword.

```
abstract sig Subject {}
sig Student extends Subject {}
sig Professor extends Subject {}
sig Class {
      TAs: set Student,
      instructor: one Professor
}
sig Assignment {
      forClass: one Class,
      submittedBy: some Student
}
pred PolicyAllowsGrading(s: Subject,
                         a: Assignment) {
      s in a.forClass.TAs or
      s in a.forClass.instructor
}
pred WhoCanGradeAssignments() {
      some s : Subject | some a: Assignment |
         PolicyAllowsGrading[s, a]
}
run WhoCanGradeAssignments for 3
```

Fig. 1. A simple gradebook specification

These features, respectively, comprise the core operations of Aluminum: GenerateMin, Augment, and ConsistentTuples. The precise specifications of these operations, comprising the basic semantics of Aluminum as a tool, are given in Theorem 1.

Exploration with minimal scenarios results in a different form of traversal of the space of scenarios than what Alloy currently provides. Putting the user in control of exploration is perhaps the chief merit of Aluminum's approach (Section II). We present the theory (Section III), and explain how we implement it atop Alloy (Section IV), including a brief discussion on the impact of symmetry-breaking on minimality. We show that users incur minimal performance penalty (Section V), and finally (Section VI) examine the user experience that ensues from the minimality-driven approach.

II. A Worked Example

Figure 1 provides the Alloy specification for a simple gradebook. There are two kinds of users (called `Subjects` in the specification): Student and Professor. A class has one Professor and a set of Students designated as TAs. An assignment is submitted by a set of students for a specific class. The gradebook specifies a policy on who may grade assignments: specifically, professors and TAs may grade assignments in their associated classes.

A. Scenario Selection in Alloy vs. Aluminum

The specification's `run` command asks for scenarios of who can grade assignments. Both Alloy and Aluminum present one scenario at a time; users may request another scenario by clicking the Next button. Aluminum produces a three initial scenarios in response, shown in Figure 2. In contrast, Figure 3 shows the first three scenarios that Alloy produces (out of over 10,000). The order in which each of Alloy

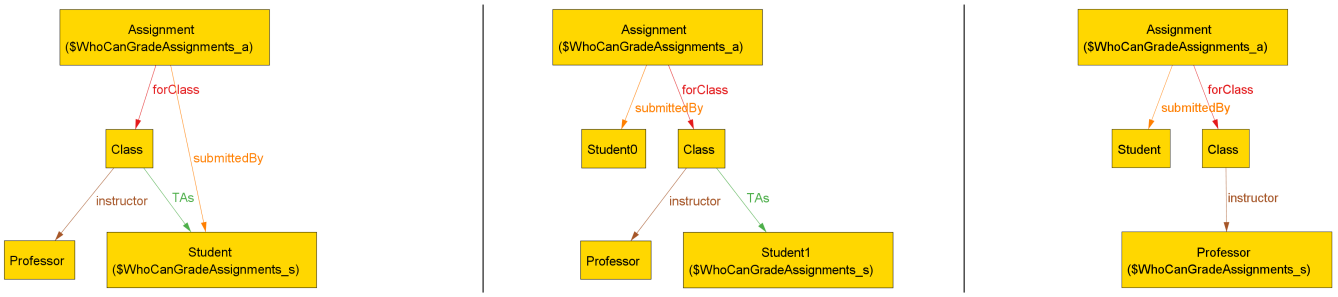

Fig. 2. Gradebook scenarios from Aluminum

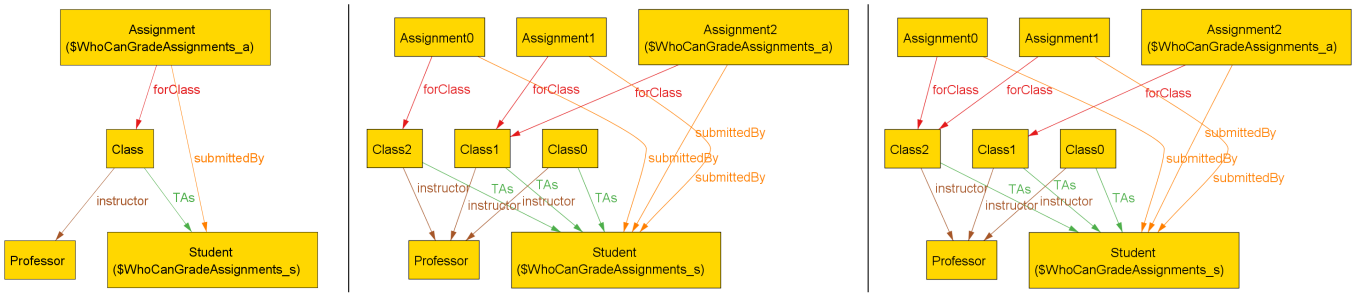

Fig. 3. Gradebook scenarios from Alloy

and Aluminum produces scenarios is non-deterministic; the scenarios in Figure 3 are representative of what we got in several independent runs. (Space limits restrict the size of these images, but their general shape will tell the story.)

Aluminum's scenarios illustrate three conditions under which one can grade an assignment: (1) the subject is a TA and also the submitter of the assignment, (2) the subject is a TA for the class but not the submitter of the assignment, (3) the subject is a Professor for the class. Each of Alloy's first three scenarios illustrate the first condition, but with additional elements that are not necessary to satisfy the specification (such as two additional classes). Some of Alloy's scenarios (not shown) include additional tuples, such as a second student submitter of some assignment. While these extra elements and tuples *can* exist in a satisfying scenario, they are not *necessary*. Aluminum, in contrast, weeds out all unnecessary tuples, focusing instead on the essence of the scenario.

For the particular scenarios that Alloy generated on this example, one might posit that the minimality problem is about the domain bounds used in the analysis: had we run the gradebook specification with tighter bounds (1 Class, 1 Professor, 1 Assignment, and 2 Students), Alloy would produce fewer and "tighter" scenarios. Minimality is, however, about more than just setting good domain bounds, for two reasons. First, the bounds only control the number of elements, not the number of tuples; even with tight bounds, non-minimal scenarios can contain unnecessary tuples. Second, by setting bounds too tight, a user can fail to learn about potentially dangerous scenarios, thus gaining inappropriate confidence in their specification. Thus, discovering good bounds is often a process of trial-and-error; minimal scenarios eliminate unnecessary elements and tuples automatically, without placing that burden on the user.

B. *Partitioning the Scenario Space*

Each of Aluminum and Alloy embodies a choice of which scenarios to present. Since both tools build upon externally-developed SAT-solvers, their choices are constrained to principles that can be encoded in the propositional formulas on which SAT-solvers operate. Within this constraint, Alloy attempts to partition the scenario space into equivalence classes based on isomorphism and presents one scenario per class (as we discuss in more detail in Sections IV and V)—but chooses the scenario indiscriminately. Aluminum instead organizes scenarios into a partial order based on the tuples they contain and presents the scenarios that are lowest in that ordering.

Figure 4 illustrates this fundamental difference between Alloy and Aluminum. The black dots in (A) are scenarios of a specification. The blobs group them into equivalence classes. Part (B) shows how Alloy might present this space: the red dots are representatives of each class; Alloy presents one representative each, and the other members of the class (hollow dots) are not presented. However, there is no ordering to how these representatives are presented, as we have already seen: a fairly complex scenario could precede a very simple one. Part (C) shows what Aluminum does (we describe part (D) in Section II-C.) It groups scenarios and picks minimal representatives to show. The Next button shows only minimal scenarios, but Aluminum's augmentation command (described in Section II-C) enables users to find scenarios (the gray dots) with selected tuples added.

As part (C) suggests, augmentation might lead the user to scenarios (the gray circles) that are isomorphic to ones that have been seen previously. We believe showing these

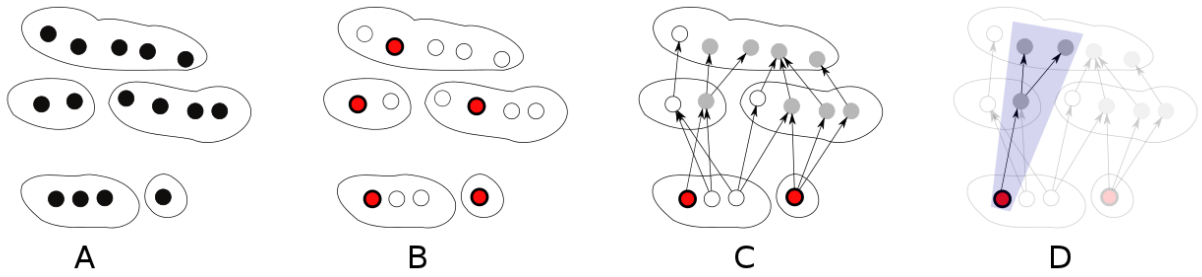

Fig. 4. How Aluminum and Alloy organize the space of scenarios. Each circle is a scenario, and blobs represent equivalence classes. Red circles are those obtained using the Next button. Hollow circles are those never shown to the user. Arrows represent augmentation operations, and gray circles are scenarios presented after exploration. (A) shows a set of scenarios grouped by isomorphism. (B) shows Alloy's unordered presentation of representatives. (C) shows Aluminum's output, presenting minimal scenarios and augmentation. (D) shows a cone of scenarios.

scenarios is more sensible than refusing to show a user-constructed scenario just because symmetry-breaking in the original generation procedure would have suppressed it. Nevertheless, unreachable isomorphic scenarios are still excluded, since the user cannot get to them through any operations; this therefore shrinks the exploration space.

C. Exploration via Augmentation

Aluminum views minimal scenarios as a *starting point* for understanding specifications, but not as sufficient. Alloy users commonly sanity-check specifications with a simple query that says "show me satisfying scenarios". The following query does this for the gradebook specification:

```
run {some Class} for 3
```

Aluminum produces only one minimal scenario for this query; it contains a single Class and a single Professor who is the instructor for that Class. This is a sufficient scenario because the gradebook specification does not require Students, TAs, or Assignments in a Class. Sanity-checking, however, requires illustrating some of the optional components of a specification. These optional components are not, however, independent: any gradebook scenario that contains an Assignment, for example, must also contain a Student who submitted the Assignment. A systematic technique for exploring the space of scenarios should help the user understand these dependencies.

Aluminum provides two operations that support systematic exploration of the scenario space. Consistent Tuples suggests avenues for exploration by producing a list of all additional tuples that are consistent with the current scenario (and hence can be added to the scenario). Augment adds a user-selected tuple to the current scenario, producing a list of minimal scenarios that include both everything in the current scenario and the new tuple. Returning to our gradebook example, for the scenario showing just a single Class and Professor, Aluminum indicates that the following additions are consistent:

```
Student[NEW(Subject$0)]
Professor[NEW(Subject$0)]
Class[NEW(Class$0)]
Assignment[NEW(Assignment$0)]
Class.TAs[NEW(Class$0), NEW(Subject$0)]
Class.TAs[Class, NEW(Subject$0)]
Class.instructor[NEW(Class$0), NEW(Subject$0)]
Class.instructor[NEW(Class$0), Professor]
Assignment.forClass[NEW(Assignment$0), NEW(Class$0)]
Assignment.forClass[NEW(Assignment$0), Class]
Assignment.submittedBy[NEW(Assignment$0),
                       NEW(Subject$0)]
```

Each description string contains a relation name and, enclosed in square brackets, a list of atoms. If an atom already exists in the current scenario, it appears by the name given it in the visualizer (e.g., `Class` or `Professor`). If the atom is not present in the current scenario, its descriptor is `NEW`, along with its internal name (`Class$0`, `Subject$0`, etc.).

Visually, it helps to imagine the *cone* of scenarios that extend, and are consistent with, a given scenario. This is shown in part (D) of Figure 4. Aluminum's consistent tuple computation shows all the tuples that can inhabit this cone, thereby mapping out the cone's landscape. These tuples may not all, however, necessarily co-habit: a tuple appears in this list if it exists in *some* satisfying scenario that extends the current one, but the presence of some can exclude others, due to specification constraints or domain size bounds.

Suppose a user augments the scenario with the following:

```
Assignment.forClass[NEW(Assignment$0), Class]
```

There is only one extended scenario, and it includes a Student as well as an Assignment. Since Aluminum produces only minimal scenarios (even under augmentation), this tells the user that every addition to this model which adds an Assignment requires adding a Student—a form of deductive reasoning that Alloy does not provide. Asking Aluminum for the consistent tuples for this new scenario suggests some potentially interesting situations:

```
Class.TAs[Class, Student]
Assignment.submittedBy[Assignment, NEW(Subject$0)]
...
```

Specifically, (1) the Student who authored the Assignment could also be a TA in the class, and (2) more than one Student is allowed to submit the same Assignment. Each of these may suggest additional queries to a user. (Aluminum also implements a Backtrack button so users can undo augmentation and continue exploration from a previous scenario.)

Aluminum's combination of minimal scenarios and exploration helps users understand the implications of their specifications in a lightweight manner. Alloy, in contrast, does not offer an exploration mode (much less one based on minimality). If a user wants to see classes with assignments, she must create a new **run** command that adds the assignment constraint. The resulting scenarios may include unnecessary truths; determining the status of each is left to the user to sort out, typically by continuing to refine the query. Furthermore, each change is followed by a new execution, which may start the user out from a completely different initial scenario. By supporting interactive exploration, we feel that Aluminum (a) reduces context switching, (b) reduces "exploration clutter" in the specification, and (c) helps users stay with the same example, which can be lost when executing afresh.

D. Tuple Provenance

Because of the semantics of Aluminum, when confronted with a particular scenario, a user can understand the provenance of each tuple in it: it is present because it is either (a) part of a minimal scenario, or (b) chosen by a user for augmentation, or (c) the consequence of a user augmentation. While we have not modified Alloy to present this information explicitly, this could easily become part of the user interface.

III. Foundations

a) Models for first-order languages: A *(relational) model* for a language L is a map $\mathcal{I}$ binding each relational variable R of L to an actual set-theoretic relation $\mathcal{I}(R)$. If F is sentence of L and $\mathcal{I}$ is an model making F true, we sometimes say that "$\mathcal{I}$ satisfies F" or "$\mathcal{I}$ witnesses F", and write $\mathcal{I} \models F$. If T is a set of first-order sentences we say that $\mathcal{I}$ is a model for T, or $\mathcal{I}$ *satisfies,* or *witnesses,* T if $\mathcal{I}$ satisfies each sentence of T, and we write $\mathcal{I} \models T$.

As Jackson notes [2], one can view Alloy's specification language as first-order logic or relational algebra. A *tuple* over a model $\mathcal{I}$ is given by a n-ary relation R and a sequence $[a_1, \ldots, a_n]$ of elements of $\mathcal{I}$ such that $[a_1, \ldots, a_n]$ is in $\mathcal{I}(R)$.

If $\mathcal{I}_1$ and $\mathcal{I}_2$ are models, we define $\mathcal{I}_1 \leq \mathcal{I}_2$ to mean that for each relation R, $\mathcal{I}_1(R) \subseteq \mathcal{I}_2(R)$; we write $\mathcal{I}_1 < \mathcal{I}_2$ if for at least one R the inclusion is strict. The *cone* of a model $\mathcal{I}$ is $\{\mathcal{I}' \mid \mathcal{I} \leq \mathcal{I}'\}$: intuitively, this is the set of extensions of $\mathcal{I}$; such an extension can add elements and tuples, but never lose information. This relation is a partial order on the set $\mathrm{Mod}(T)$ of models for any T; we say that a model $\mathcal{I}$ is *minimal* for T if (i) $\mathcal{I} \models T$ and (ii) there is no $\mathcal{I}' \models T$ with $\mathcal{I}' < \mathcal{I}$. If T has any finite models then it has at least one minimal model; in general, of course, T may have several minimal models.

We will often consider models "up to isomorphism": models $\mathcal{I}$ and $\mathcal{I}'$ are isomorphic if they can be made identical by renaming of elements. We write $\mathcal{I}_1 \preceq \mathcal{I}_2$ to mean that, for some $\mathcal{I}_1'$ and $\mathcal{I}_2'$ isomorphic to $\mathcal{I}_1$ and $\mathcal{I}_2$ respectively, $\mathcal{I}_1' \leq \mathcal{I}_2'$.

b) Models for propositional languages: A *(propositional) model* for a given set of atoms for propositional logic is a function M from atoms to $\{0, 1\}$, (*qua* $\{false, true\}$).

If $\mathcal{M}_1$ and $\mathcal{M}_2$ are propositional models, we define $\mathcal{M}_1 \leq \mathcal{M}_2$ to mean that, for each propositional atom p, $\mathcal{M}_1(p) \leq \mathcal{M}_2(p)$; write $\mathcal{M}_1 < \mathcal{M}_2$ if for at least one p, $\mathcal{M}_1(p) < \mathcal{M}_2(p)$. Analogously with relational models, the relation $\leq$ is a partial order on the set of models for any set B of propositional formulas and determines the obvious notion of "cone" over a propositional model. A propositional model $\mathcal{M}$ is minimal for a set B of propositional formulas if it satisfies B and B has no satisfying model $\mathcal{M}' < \mathcal{M}$.

Finite relational models can be encoded as propositional models using standard techniques. Indeed, Alloy's engine, Kodkod [9], translates users' relational specifications to the propositional world, and the results back again for output.

c) Propositional encoding of specifications: Alloy converts a "specification" into a constraint represented as a relational algebra expression encoding the axioms and declarations of an Alloy module together with the predicate for which we seek a (relational) model. Kodkod translates such a constraint into (following the language of [9]) a *Kodkod problem,* which is a triple consisting of a universe of elements, a set of lower- and upper-bounds for each relation symbol in the language, and a relational formula.

Any Kodkod problem P gives rise to a formula $B(P)$ of propositional logic. So given an Alloy specification S, we eventually arrive at a propositional logic formula $B(P_S)$. Moreover – if we remove the secondary variables that Kodkod introduces when translating to conjunctive normal-form – Kodkod also gives us a one-to-one correspondence between relational models of P_S and propositional models of $B(P_S)$. The orderings defined above for relational and propositional models are preserved under this mapping.

Theorem 1 states our correctness claims about Aluminum. Space limitations prevent us from including proofs, but they follow naturally from the algorithms in Section IV.

Theorem 1.

1) *(Completeness of Generation) Let S be an Alloy specification. Procedure* GenerateMin *generates a complete set of minimal models for S up to isomorphism. That is, for any model $\mathcal{I}$ witnessing S,* GenerateMin *will produce some minimal $\mathcal{I}_0$ such that $\mathcal{I}_0 \preceq \mathcal{I}$.*
2) *(Correctness of Augmentation) Let $\mathcal{I}$ be an model witnessing specification S and let F be a tuple over $\mathcal{I}$. Procedure* Augment *either returns a model $\mathcal{I}'$ such that (i) $\mathcal{I} \preceq \mathcal{I}'$, (ii) $\mathcal{I}'$ satisfies S and F, and (iii) is minimal with respect to these properties, or detects failure if there is no such model.*
3) *(Completeness of Exploration) Let $\mathcal{I}$ and $\mathcal{I}^+$ each witness specification S, with $\mathcal{I} < \mathcal{I}^+$. There is a finite sequence of* Augment *steps leading from $\mathcal{I}$ to $\mathcal{I}^+$.*
4) *(Completeness of Consistency-Checking) Let $\mathcal{I}$ be an model witnessing specification S. Procedure* ConsistentTuples *returns the set of those tuples F such that there is at least one model $\mathcal{I}'$ with $\mathcal{I} \preceq \mathcal{I}'$ with $\mathcal{I}'$ witnessing S and F.*

Algorithm 2 (Minimize)**.**

Input: a model M and formula P with $M \models P$
Output: M', a minimal model for P

```
repeat
    M' ← M
    M ← Reduce(M)
until M = M' // Cannot minimize any more
```

Algorithm 3 (Reduce)**.**

Input: model M and formula B, with $M \models B$
Output: $M' < M$ with $M' \models B$, if one exists; otherwise M

```
C ← ⋀{¬p | p is false in M}
D ← ⋁{¬p | p is true in M}
if there is a model N with N ⊨ B ∧ D ∧ C then
    return N
else
    return M          // M is minimal
```

IV. Implementation

Aluminum modifies both the Alloy Analyzer's user interface and underlying constraint solver, Kodkod [9]. Aluminum's user interface is based on that of Alloy: the user submits an Alloy specification, Kodkod translates the associated constraint to a propositional formula, the SAT-solver is iterated to produce propositional models, and these are eventually translated to scenarios by Kodkod and rendered by Alloy.

The differences between Alloy and Aluminum lie in the nature of the iterator to produce the initial suite of scenarios, and the facility for user-controlled exploration of the space of scenarios. This section outlines the algorithms underlying the functionality specific to Aluminum.

A. GenerateMin

The sequence of models produced by Aluminum as a result of the initial execution of the specification is produced by procedure GenerateMin. Given a Kodkod problem P, Aluminum initializes an iterator by invoking a SAT-solver (SAT4J [10], in our implementation) to return a model M of P; M and P are then passed to the minimizer.

Minimization consists of repeated calls to Algorithm Reduce (Algorithm 3), until there is no change: see Algorithm 2. Minimization is not a deterministic process, because a given model can lie in the cone of several different minimal models (as Figure 4 (C) and (D) show). This non-determinism manifests itself in practice in Aluminum, because the output of Reduce depends on choices made by the SAT-solver.

After each (minimal) model $\mathcal{M}$ is generated, we of course want to ensure that $\mathcal{M}$ is excluded from future generation. It is straightforward to filter the output to ensure this, as Alloy does. In our setting the problem is somewhat more subtle, since the process of minimization will of course return identical results from quite different models. But in fact this is an opportunity for a significant optimization. After each output model $\mathcal{M}$ is computed, Aluminum adds to the current problem the disjunction of the negations of the atoms true in $\mathcal{M}$. This ensures that no subsequent model from the SAT-solver will be in the cone of $\mathcal{M}$. This is a sound pruning of the model space, since $\mathcal{M}$ represents each model in its cone.

Algorithm 4 (Consistent Tuples)**.**

Input: model M and formula B, with $M \models B$
Output: the atoms false in M consistent with M and B

```
C ← ⋀{p | p is true in M}
D ← ⋁{p | p is false in M}
R ← ∅
repeat
    if there is a model N with N ⊨ B ∧ D ∧ C then
        F ← {p | N(p) = 1 and M(p) = 0}
        R ← R ∪ F
        D ← D − F
until no change in R
return R
```

Suppressing Isomorphic Models. We have seen how to suppress the generation of identical models, but eliminating *isomorphic models* is much harder. Kodkod tries to avoid generating isomorphs by adding "symmetry-breaking" formulas [11], [12] to the input. This is necessarily heuristic, since no polynomial-time algorithm is known for detecting isomorphisms of relational models.

Unfortunately, symmetry-breaking does not interact well with minimization. For intuition, consider part (C) of Figure 4, in which two non-isomorphic models each lie in the cone of the rightmost minimal model in the equivalence class with three entries. When these models are minimized they may return the same minimal representative, depending on choices made by the SAT-solver.

It is even the case that if Aluminum were to use the same mechanism as Kodkod for symmetry-breaking it could incorrectly conclude that a non-minimal model is minimal. Aluminum thus uses a slightly different heuristic for symmetry-breaking. Space considerations preclude a detailed discussion of the issue here, but one consequence is that Alloy sometimes eliminates more isomorphic models than Aluminum.

B. Augment

A key observation is that augmenting a model of a specification by a tuple is merely an instance of the core problem of minimal-model generation: the result of augmentation is precisely an iterator over minimal models of the specification given by the original specification, along with the tuples of the given model plus the new tuple.

One detail is important for performance. Letting B be the conjunctive normal form of the original specification and A_p the Boolean variable representing the augmenting tuple A, the input for model-generation is $B \cup \{p \mid M(p) = 1\} \cup \{A_p\}$. The point here is that since the augmentation is performed entirely at the propositional level, Aluminum avoids the cost of re-translating from relational to propositional logic.

C. ConsistentTuples

To aid the user in deciding how to explore the space of scenarios by augmentation, Aluminum can determine in advance which new tuples can be consistently added to a given scenario in the context of a given specification. If M is a model of B, we say that atom d is *consistent* with M and B if there is a model of B and d extending M. Aluminum's procedure for computing consistent tuples (modulo translation to and from the propositional level) is given in Algorithm 4.

If two unused atoms occur in the same position in otherwise identical consistent tuples, Aluminum presents only one such. For instance, suppose that following two augmentations are consistent:

```
Student[NEW(Subject$0)]
Student[NEW(Subject$1)]
```

That is, there are two unused Subject atoms, and either of them may be instantiated into Student. In this case, Aluminum will only present the first. Since minimal models often have many unused atoms, this filtering can substantially reduce the number of consistent tuples shown.

V. Numeric Evaluation

We now compare Aluminum to Alloy numerically. We first study how the resulting scenarios compare mathematically to those produced by Alloy, and then explore how long it takes to compute these minimal scenarios.

We conduct our experiments over the following specifications, with a short name that we use to refer to them in the rest of the paper. (In the tables, where a file contains more than one command, we list in parentheses the ordinal of the command used in our experiments.) The following specifications are taken from the Alloy distribution: Addressbook 3a (Addr), Birthday (Bday), Filesystem (File), Genealogy (Gene), Grandpa (Gpa), Hanoi (Hanoi), Iolus (Iolus), Javatypes (Java), and Stable Mutex Ring (Mutex). In addition, we use three independent specifications: (1) Gradebook (Grade), which is defined in Figure 1, and enhanced with two more commands:

```
run WhoCanGradeAssignments for 3 but
1 Assignment, 1 Class, 1 Professor, 3 Student
```

and

```
run {some Class} for 3
```

(2) Continue (Cont), the specification of a conference paper manager, from our prior work. (3) The authentication protocol of Akhawe, et al.'s work [6] (Auth), a large effort that tries to faithfully model a significant portion of the Web stack.

A. Scenario Comparison

We consider a set of satisfiable specifications for which we can tractably enumerate *all* scenarios. This lets us perform an exhaustive comparison of the scenarios generated by Alloy and Aluminum. The results are shown in Figure 5.

The first (data) column shows how many scenarios Alloy generates in all. This number represents one more than the number of times the user can press the Next button. Because the Alloy user interface suppresses some duplicate scenarios, this is a smaller number than the number of scenarios produced by Kodkod; we present this smaller number. The second column shows the corresponding number of minimal scenarios presented by Aluminum (the red dots in Figure 4 (C)).

The third column shows how many scenarios it takes before Alloy has presented at least one scenario in the cone of every minimal model from Aluminum. Because a given scenario can be in the cone of more than one minimal scenario, the number of scenarios needed for cone coverage may in fact be fewer than the number of minimal models. An important subtlety is that an Alloy scenario may not fall in a new cone, but may be isomorphic to one that does (i.e., Alloy may have produced a hollow circle in Figure 4). We use Kodkod's symmetry-breaking algorithm to try to identify such situations and "credit" Alloy accordingly.

The fourth column shifts focus to minimal scenarios. If a user is interested in only minimal scenarios, how many scenarios must they examine before they have encountered all the minimal ones (up to isomorphism-detection)? This column lists the earliest scenario (as an ordinal in Alloy's output, starting at 1) when Alloy achieves this.

This number does not, however, give a sense of the distribution of the minimal scenarios. Perhaps almost all are bunched near the beginning, with only one extreme outlier. To address this, we sum the ordinals of the scenarios that are minimal. That is, suppose Aluminum produces two minimal models. If the second and fifth of Alloy's models are their equivalents, then we would report a result of $2 + 5 = 7$. The fifth and sixth columns present this information for Alloy and Aluminum, respectively. The sixth column is technically redundant, because its value must necessarily be $1 + \ldots + n$ where n is the number of Aluminum models; we present it only to ease comparison.

To ensure understanding, let us examine two rows in detail:

Addr: Aluminum finds two minimal models. The very first Alloy model is in both their cones, so the cone coverage value is lower than the number of minimal models. Such a model clearly cannot itself be minimal; indeed, since the minimal scenario coverage requires 5 models and the Alloy ordinal sum is 8, the 3rd and 5th Alloy models must have been (equivalent to) Aluminum's minimal ones.

Grade (3): Aluminum finds just one minimal model. Thus, any model found by Alloy (including the first one) must be in its cone, so the cone coverage value is 1. However, it takes Alloy another 104 models before the minimal one is found.

We can now see Aluminum's impact on covering the space of scenarios. Even on small examples such as Grade (2), there is a noticeable benefit from Aluminum's more refined strategy. This impact grows enormously on larger models such as Java and Grade (1). We repeatedly see a pattern where Alloy gets "stuck" exploring a strict subset of cones, producing numerous models that fail to push the user to a truly distinct space of scenarios. Even on not very large specifications (recall that Grade is presented in this paper in its entirety), it often takes hundreds of Nexts before Alloy will present a scenario from

Spec.	Models (Alloy)	Models (Aluminum)	Cone Coverage	Min. Scenario Coverage	Ordinal Sum (Alloy)	Ordinal Sum (Aluminum)
Addr	2,647	2	1	5	8	3
Bday (2)	27	1	1	3	3	1
Bday (3)	11	1	1	1	1	1
Gene	64	64	64	64	2,080	2,080
Gpa	2	2	2	2	3	3
Grade (1)	10,837	3	289	10,801	11,304	6
Grade (2)	49	3	2	12	33	6
Grade (3)	3,964	1	1	105	105	1
Hanoi (1)	1	1	1	1	1	1
Hanoi (2)	1	1	1	1	1	1
Java	1,566	3	374	1,558	4,573	6

Fig. 5. Alloy's coverage of minimal models and their cones.

Spec.	Aluminum		Alloy		d
	Avg	σ	Avg	σ	
Bday (1)	9	8	5	3	1.57
Cont (6)	1	<1	1	<1	-0.37
Cont (8)	5	6	3	<1	5.88
File (2)	5	5	6	4	-0.28
Iolus	5,795	239	5,000	177	4.49
Mutex (3)	18,767	52	8,781	60	165.91

Fig. 6. Relative times (ms) to render an unsatisfiable result.

a cone it has not shown earlier. The real danger here is that the user will have stopped exploring long before then, and will therefore fail to observe an important and potentially dangerous configuration. In contrast, Aluminum presents these at the very beginning, helping the user quickly get to the essence of the scenario space.

The Gene specification presents an interesting outlier. The specification is so tightly constrained that Alloy can produce *nothing but minimal models*! Indeed (and equivalently), it is impossible to augment any of these models with additional tuples, as we will see in Figure 8.

One important caveat is that these experiments were conducted with one particular SAT-solver, using Alloy's default parameters. Different SAT-solvers and parameters will likely have different outcomes, and studying this variation is a worthwhile task. Nevertheless, the above numbers accurately represent the experience of a user employing Alloy (version **4.2**) in a state of nature.

B. Scenario Generation

Having examined the effectiveness of Aluminum, we now evaluate its running time (the space difference is negligible). All experiments were run on an OS X 10.7.4 / 2.26GHz Core 2 Duo / 4Gb RAM machine, using SAT4J version `2.3.2.v20120709`. We handled outliers using one-sided Winsorization [13] at the 90% level. The times we report are obtained from Kodkod, which provides wall-clock times in *milliseconds* (ms). All experiments were run with symmetry-breaking turned on. Numbers are presented with rounding, but statistical computations use actual data, so that values in those columns do not follow precisely from the other data shown.

Spec.	Aluminum			Alloy			d
	Avg	σ	N	Avg	σ	N	
Addr	13	12	2	8	6		0.89
Auth	800	32		110	36		19.35
Bday (2)	9	9	1	5	6		0.59
Bday (3)	8	6	1	7	6		0.17
Cont (3)	4	2	3	1	<1	9	4.86
File (1)	16	13		7	4		2.39
Gene	10	5		6	5		0.85
Gpa	9	4	2	6	5	2	0.62
Grade (1)	6	6	3	3	3		1.10
Grade (2)	3	4	3	3	4		0.01
Grade (3)	8	6	1	4	4		0.96
Java	5	4	3	2	2		1.11
Hanoi (1)	2,509	11	1	1,274	1,239	1	1.00
Hanoi (2)	14	3	1	10	2	1	2.14

Fig. 7. Relative times (ms) per scenario (minimal, in Aluminum).

Every process described below was run thirty (30) times to obtain stable measurements.

To measure effect strength, we use Cohen's d [13]. Concretely, we subtract Alloy's mean from that of Aluminum, and divide by the standard deviation for Alloy. We use Alloy's in the denominator because that system is our baseline.

Because Aluminum slightly modifies Kodkod to better support symmetry-breaking, we begin by measuring the time to translate specifications into SAT problems. Across all these specifications, Aluminum's translation time falls between 81% and 113% that of Alloy, i.e., our modification has no effective impact. The absolute translation times range from 5ms (for Gradebook) to 55,945ms (for Auth). (We use commas as separators and our decimal mark is a point. Thus 55,945ms = 55.945s = almost 56 seconds.)

Though our focus is on the overhead of minimization, in the interests of thoroughness we also examine unsatisfiable queries. Figure 6 shows how long each tool spends in SAT-solving (ignoring translation into SAT, and then presentation) to report that there are no scenarios. The d values show that in some cases, the time to determine unsatisfiability is much worse in Aluminum. (It is very important for the reader to remember that the d value is measuring the effect size, *not* the ratio of average times! Thus, a d of almost 166

still results in only a 2.1x increase in time.) The effect is because of the way Aluminum and Alloy handle symmetry-breaking: Aluminum splits the formula produced by Kodkod into two parts, one representing the specification and query and the other capturing symmetry-breaking, whereas Alloy keeps the formulas conjoined. The conjoined formula offers greater opportunities for optimization, which the SAT-solver exploits. Nevertheless, we note that even in some of the large effects, the *absolute* time difference is relatively small.

For satisfiable queries, we calculate the time to compute the first ten scenarios (equivalent to clicking Next nine times), which we feel is a representative upper-bound on user behavior. When the tool could not find ten scenarios, the N column shows how many were found (and the average is computed against this N instead).

When queries are satisfiable, Aluminum's performs well compared to Alloy. First, the overall running time is small for both tools, so even large effect sizes have small wall-clock impact. Indeed, in the most extreme case, Aluminum takes only about 1.2 seconds longer, for a total time of 2.5 seconds—surely no user can read and understand a scenario in less time than that, so Aluminum could easily pre-compute the next scenario. Second, in many cases Aluminum offers many fewer scenarios than Alloy, helping users much more quickly understand the space of models. Finally, the time taken by Kodkod to create the SAT problem can be vastly greater than that to actually solve it, which suggests that a more expensive SAT-solving step will have virtually no perceptible negative impact on the user experience.

We observe two outliers in the data. First, the time for minimization for Auth is very significant. For this specification, we found that the number of extraneous tuples eliminated during minimization is 78 on average. This shows a direct trade-off: 0.7 seconds in computing time for a possibly great impact on user comprehension. Second, the standard deviation for Alloy on Hanoi (1) looks enormous relative to the mean. This is because Kodkod is producing a second duplicate model (which in fact is discarded by the Alloy user interface) very quickly. This results in datapoints that take a long time for the first solution but close to zero for the second.

We also examine the time taken by Aluminum's exploration features: how long it takes to compute the consistent tuples, and how long it takes to augment a scenario. Since the complete space of models is enormous, we restrict attention to the first set of scenarios produced by Aluminum (i.e., the red circles in the bottom row of Figure 4 (C)).

Figure 8 shows the consistent tuples times. We compute up to ten scenarios (five, in the case of Auth) and for each of these we determine the number of consistent tuples. The first column indicates the number of consistent tuples found, averaged over the number of minimal scenarios (i.e., consistent tuples per model). The zero-values are not errors: they arise because the specifications are sufficiently constrained that there is no room for augmentation beyond the initial scenarios. The next two columns show how long it took to compute the number of consistent tuples. These numbers are clearly very modest. Auth is the exception: it produces models with, on average, over 1,300 consistent tuples, more than can be explored by hand. Implementing strategies for helping the user in such situations is an interesting topic for future work.

Spec.	# Cons. Tupls	Cons. Tup. Time Avg	σ	Aug. Time
Addr	57	45	25	4
Auth	1,335	106,456	17,268	194
Bday (2)	20	19	24	6
Bday (3)	8	9	10	2
Cont (3)	2	5	<1	3
File (1)	24	29	16	3
Gene	0	2	<1	N/A
Gpa	0	1	<1	N/A
Grade (1)	25	13	10	1
Grade (2)	6	2	1	1
Grade (3)	36	14	7	1
Java	25	17	11	2
Hanoi (1)	0	4	<1	N/A
Hanoi (2)	0	1	1	N/A

Fig. 8. Times (ms) to compute consistent tuples and to augment scenarios.

The last column shows how long, on average, it takes to augment a scenario with a consistent tuple (backtracking between each augmentation). The "N/A"s correspond to specifications that have no more consistent tuples, and hence cannot be augmented. This too takes very little time.

VI. Qualitative Impressions

We have run Aluminum on many representative uses of Alloy beyond those reported in Section V. Here, we report on qualitative impressions. Instead of dwelling on successes, we focus on those aspects of Aluminum that are interesting, potentially controversial, and certainly deserving of future work (such as rigorous user studies).

a) Verification versus Realization: There is an (informal) distinction between two modes of exploring a specification: *verification*, where the user has a property to check (and whose failure yields a counter-example), versus *realization*, in which a user wants to see which scenarios are compatible with a specification. Minimality and orderly growth appear especially useful for the former (where the pithiest counterexample is often the most useful) but perhaps less so for the latter, since the scenarios presented are less likely to surprise the user. Adding some of the other features discussed below would ameliorate this weakness.

b) Negative Information: Aluminum's consistent tuple enumeration does not list what *cannot* become true. This negative information can be vital in comprehending scenarios, and should ideally be incorporated into the exploration process.

c) Random Exploration: Using exploration, an Aluminum user can *eventually* reach any scenario that Alloy would have generated (Theorem 1). But our experience suggests that there is an important benefit to Alloy's more random generation of scenarios: richer scenarios raise situations that a user didn't think to ask about, and can thus be insightful and thought-provoking. Perhaps Aluminum could add a Surprise

Me! button that presents a random, non-minimal model to encourage adventitious discovery (or Alloy could be enhanced with Aluminum's minimization and exploration facilities).

d) Transition Systems and Framing Conditions: Framing conditions describe what should not change between states of a system. We found Aluminum good at highlighting some missing frame conditions. For instance, in an address book (as in [2, chapter 2]), if the `add` predicate fails to mention that all existing entries must be retained, the book will contain only the entry just added. Thus, two consecutive `add` operations will starkly illustrate the absence of a frame condition. This absence may be masked by the non-minimal models of Alloy.

However, frame conditions express not only "lower bounds"—what must remain—but also "upper bounds": what must not be added. In the absence of upper-bounds, a scenario-finder is free to add extra domain elements or relational tuples. Alloy will frequently present just such models, alerting the user to the lack of appropriate framing. In contrast, Aluminum is guaranteed to excise such superfluity! These superfluous entries are actually still present in the consistent tuples, but we believe it is too difficult for users to find them there.

VII. Related Work

Logic programming languages produce single, *least*, models as a consequence of their semantics. Because user specifications are not limited to the Horn-clause fragment, Aluminum can offer no such guarantee. The more general notion of minimal model in this paper has already been used in specifying the semantics of disjunctive logic programming [14] and of database updates [15] and in non-monotonic reasoning, especially circumscription [16].

The development of algorithms for the *generation* of relational models is an active area of research. The two most prominent methods are "MACE-style" [17], which reduce the problem to be solved into propositional logic and employ a SAT-solver, and "SEM-style" [18], which work directly in first-order logic. The goals of these works are mostly orthogonal to ours, since their concern is usually not the *exploration* of the space of all models of a theory. Generation of minimal models specifically usually relies on dedicated techniques, often based on tableaux (e.g., [19]) or hyperresolution (e.g., [20]). Since we are more concerned with exploration as an enhancement of established software design methodology, we made a choice to work with the SAT-solver technology bundled with a tool (Alloy) designed for presenting models.

Koshimura et al. [21] uses minimality of propositional models to optimally solve job-scheduling problems. Our minimization algorithm is essentially identical to theirs. However, their algorithms do not make use of symmetry-breaking, nor do they address augmentation or consistent-tuple generation.

Janota [22] offers an algorithm to compute all minimal models of a formula once one is computed, and shows how to compute a minimal model by modifying a SAT-solver satisfying a certain technical property. Our technique works with unmodified solvers; it would still be interesting to compare the performance of Janota's method with ours. Another instance of changing the solver would be to use a specialized solver like Max-SAT. We opted to use an ordinary SAT-solver (SAT4J) because using Max-SAT would have involved significant changes that might have introduced confounding factors that would complicate a side-by-side evaluation.

The goals of the Cryptographic Protocol Shapes Analyzer [5] are closely aligned with ours. In analyzing protocols, it too generates minimal models. However, its application domain and especially algorithms are quite different from ours.

Acknowledgments

We are grateful to Joshua Guttman, Daniel Jackson, and Emina Torlak for valuable discussions about this work. Joe Beck and Janice Gobert gave us guidance on data presentation. This work was partially supported by the NSF.

References

[1] T. Nelson, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "Toward a more complete Alloy," in *International Conference on Abstract State Machines, Alloy, B, and Z*, 2012.
[2] D. Jackson, *Software Abstractions*, 2nd ed. MIT Press, 2012.
[3] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "The Margrave Tool for Firewall Analysis," in *USENIX Large Installation System Administration Conference*, 2010.
[4] K. Fisler, S. Krishnamurthi, L. Meyerovich, and M. Tschantz, "Verification and change impact analysis of access-control policies," in *International Conference on Software Engineering*, 2005.
[5] S. F. Doghmi, J. D. Guttman, and F. J. Thayer, "Searching for shapes in cryptographic protocols," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2007.
[6] D. Akhawe, A. Barth, P. Lam, J. Mitchell, and D. Song, "Towards a formal foundation of web security," in *IEEE Computer Security Foundations Symposium*, 2010.
[7] R. Becker, S. Eick, and A. Wilks, "Visualizing network data," *IEEE Transactions on Visualization and Computer Graphics*, 1995.
[8] T. Tran, E. Al-Shaer, and R. Boutaba, "PolicyVis: firewall security policy visualization and inspection," in *USENIX Large Installation System Administration Conference*, 2007.
[9] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2007.
[10] D. Le Berre and A. Parrain, "The Sat4j library, release 2.2," *Journal on Satisfiability, Boolean Modeling and Computation*, 2010.
[11] I. Shlyakhter, "Generating effective symmetry-breaking predicates for search problems," *Discrete Applied Mathematics*, 2007.
[12] J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy, "Symmetry-breaking predicates for search problems," in *Principles of Knowledge Representation and Reasoning*, 1996.
[13] R. Wilcox, *Introduction to Robust Estimation and Hypothesis Testing*. Academic Press, 2012.
[14] J. Lobo, J. Minker, and A. Rajasekar, *Foundations of Disjunctive Logic Programming*. The MIT Press, 1992.
[15] R. Fagin, J. Ullman, and M. Vardi, "On the semantics of updates in databases," in *Symposium on Principles of Database Systems*, 1983.
[16] A. Robinson, *Handbook of Automated Reasoning*. Elsevier, 2001, vol. 2.
[17] W. McCune, "MACE 2.0 reference manual and guide," *CoRR*, 2001, cs.LO/0106042.
[18] J. Zhang and H. Zhang, "SEM: a system for enumerating models," in *International Joint Conference On Artificial Intelligence*, 1995.
[19] I. Niemelä, "A tableau calculus for minimal model reasoning," in *Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, 1996.
[20] F. Bry and A. Yahya, "Positive unit hyperresolution tableaux and their application to minimal model generation," *Journal of Automated Reasoning*, 2000.
[21] M. Koshimura, H. Nabeshima, H. Fujita, and R. Hasegawa, "Minimal model generation with respect to an atom set," in *International Workshop on First-Order Theorem Proving*, 2009.
[22] M. Janota, "SAT solving in interactive configuration," Ph.D. dissertation, University College Dublin, November 2010.