Alloy^{Max}: Bringing Maximum Satisfaction to Relational Specifications

Anonymous Author(s)

ABSTRACT

Alloy is a declarative modeling language based on a first-order relational logic. Its constraint-based analysis has enabled a wide range of applications in software engineering, including configuration synthesis, bug finding, test-case generation, and security analysis. Certain types of analysis tasks in these domains involve finding an *optimal* solution. For example, in a network configuration problem, instead of finding any valid configuration, it may be desirable to find one that is most *permissive* (i.e., it permits a maximum number of packets). Due to its dependence on SAT, however, Alloy cannot be used to specify and analyze these types of problems.

We propose Alloy^{Max}, an extension of Alloy with a capability to express and analyze problems with optimal solutions. Alloy^{Max} introduces (1) a small addition of language constructs that can be used to specify a wide range of problems that involve optimality and (2) a new analysis engine that leverages a *Maximum Satisfiability* (*MaxSAT*) solver to generate optimal solutions. To enable this new type of analysis, we show how a specification in a first-order relational logic can be translated into an input format of MaxSAT solvers—namely, a Boolean formula in *weighted conjunctive normal form* (*WCNF*). We demonstrate the applicability and scalability of Alloy^{Max} on a benchmark of problems. To our knowledge, Alloy^{Max} is the first approach to enable analysis with optimality in a relational modeling language, and we believe that Alloy^{Max} has the potential to bring a wide range of new applications to Alloy.

ACM Reference Format:
Anonymous Author(s). 2021. Alloy^{Max}: Bringing Maximum Satisfaction to Relational Specifications. In *Proceedings of The 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Alloy[16] is a declarative modeling language based on a first-order relational logic with transitive closure. Thanks to its expressive power, along with the analysis capability provided by its back-end engine, the Alloy Analyzer, it has been applied to a wide range of problems in software engineering, including protocol verification [5, 47], configuration analysis [27, 34], test-case generation [19], bug finding [9, 17], and security analysis [2, 18, 43].

ESEC/FSE 2021, 23 - 27 August, 2021, Athens, Greece

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The types of analysis that can be performed with a high-level modeling language like Alloy depends on the capability of its underlying engine. The Alloy Analyzer is at its core a *finite model finder*: Given a set of first-order logic (FOL) constraints that correspond to a system specification M and user query S (e.g., $S \equiv$ "find a valid network configuration"), the analyzer translates formula $M \wedge S$ to an equisatisfiable Boolean formula F such that a satisfying instance to F is also a solution to $M \wedge S$. This analysis is then performed by an off-the-shelf Boolean Satisfiability (SAT) solver. In general, there may be multiple satisfying instances F; the Alloy Analyzer also provides a way to *enumerate* these solutions by repeatedly invoking the solver with additional constraints that are intended to block the previously seen instances.

The reliance on the SAT-based analysis means, however, that the Alloy Analyzer cannot be used to automatically perform the following types of analysis:

- **Generation of optimal solutions**: It is sometimes desirable to find not only any solution but one that is *minimal* or *maximal*, depending on the problem being modeled. For instance, in the context of security exploit generation [43], it may be desirable to synthesize the *smallest* possible exploit that can demonstrate a security violation. In a network firewall configuration problem [34], a configuration setting that *maximizes* the number of allowed packets while satisfying a security policy is typically considered more desirable than a less permissive solution. However, since the Alloy Analyzer returns instances in an arbitrary order, finding such an optimal solution can only be done by enumerating the (typically very large) space of all instances.
- **Analysis with soft constraints**: Certain problems are naturally expressed using a combination of both *hard* and *soft* constraints, where a desirable solution should satisfy as many of the soft ones as possible. For example, a meeting scheduling problem may consist of hard constraints (e.g., "all participants must be present") as well as soft ones (e.g., "participants' time preferences"); if it is not possible to satisfy all of the latter, a solution that fulfills as many of these optional preferences is still more desirable than others. Since all of the constraints in Alloy are treated as hard constraints that must always be satisfied, it is currently not possible to analyze such types of problems.
- **Analysis with priorities**: When a problem has multiple, *incomparable* optimal solutions, it may be useful to rank these solutions using some notion of *priority* among constraints. For instance, in the aforementioned scheduling problem, a participant may have a higher preference towards meeting times in the morning rather than in the afternoon. Again, since Alloy treats all of the constraints equally, it is not possible to solve problems where such a notion of priority plays an important role.

In this paper, we propose Alloy^{Max}, an extension of Alloy that overcomes these limitations by enabling an analysis of Alloy specifications with optimal solution generation. In particular, we propose

both (1) a *language extension* to allow the user to specify soft constraints and indicate parts of an Alloy specification that must be optimized and (2) a *new analysis back-end* that leverages a *maximum satisfiability* (*MaxSAT*) solver to perform the analysis. To enable this analysis, we also provide a new translation mechanism from a high-level Alloy$^{\text{Max}}$ specification in FOL to an equivalent MaxSAT problem.

The proposed extension significantly extends the range of analysis that can be performed over Alloy specifications. It is also a strict generalization of Alloy: It introduces no semantic changes to existing Alloy specifications and includes only a small number of syntactic extensions, and thus should be easily adoptable by existing Alloy users.

We demonstrate the added analysis capability of Alloy$^{\text{Max}}$ on a variety of case studies, including exploit generation on microprocessors [43] (similar to the well-known Spectre [20] and Meltdown [24] attacks), graceful degradation in security [13], a wedding seating assignment [32], and a task scheduling problem [22]. We show that not only can our extension be used to perform analyses that were not possible in Alloy before, but the performance of our analysis engine is competitive to the existing Alloy Analyzer.

The contributions of this paper are as follows:

- An extension to Alloy, called Alloy$^{\text{Max}}$, and an accompanying analysis engine that can be used to generate optimal solutions and solve problems with soft, prioritized constraints (Section 2);
- A translation mechanism from a first-order relational logic to a weighted Boolean formula (Section 4);
- An optimization technique that uses high-level information in an input Alloy specification to guide a MaxSAT solver more efficiently towards optimal solutions (Section 5); and
- A collection of case studies demonstrating the new types of analyses in Alloy$^{\text{Max}}$, and a set of benchmark results demonstrating its performance on problems of varying sizes (Section 6).

2 EXAMPLES

As a motivating example, consider a model of a *course scheduling* problem in Alloy (Figure 1). A university offers a set of courses that students can register for each semester; in this model, we assume the school is offering five courses named CS101, Compiler, OS, ML, and SE (line 5-6). Each lecture can take place during morning (AM) or afternoon (PM) between Monday to Friday (lines 1-3). A set of all available lecture slots (lines 8-9) and their relations to the day and time are defined on lines 12-13, and the timetable for the courses (Table 1) is defined on line 15. Each student is associated with a set of *core* courses that are required for their major. For simplicity, we consider the scheduling problem for one particular student, Alice, who is assigned CS101 as a core course.

Each student's registration schedule must satisfy the following requirements:

- Each student must take at least 3 courses (line 25);
- Each student must take all of the *core* courses that they are assigned, and (line 26);
- A student cannot take courses whose lecture times conflict with each other (line 27).

Given this model, we can use a *run* command (line 36) in Alloy to generate a valid course schedule for Alice that satisfies all of the requirements. Figure 2a shows the first generated instance, indicating that Alice could take Compiler, CS101, OS, and SE to satisfy the requirements.

Table 1: The course timetable of the scheduling problem.

	Mon	Tue	Wed	Thu	Fri
AM	CS101	ML/Compiler	CS101	ML/Compiler	
PM	SE	OS	SE	OS	CS101

```
1  abstract sig Day {}
2  one sig Mon, Tue, Wed, Thu, Fri extends Day {}
3  abstract sig Time {} one sig AM, PM extends Time {}
4
5  abstract sig Course { lectures: set Lecture }
6  one sig CS101, Compiler, OS, ML, SE extends Course {}
7
8  abstract sig Lecture { day: one Day, time: one Time }
9  one sig MonAM, MonPM, TueAM, ... extends Lecture {}
10 fact {
11   // The full day/time definition is omitted.
12   day = MonAM -> Mon + MonPM -> Mon + ...
13   time = MonAM -> AM + MonPM -> PM + ...
14   // The full timetable definition is omitted.
15   lectures = SE -> MonPM + SE -> WedPM + ...
16 }
17 abstract sig Student {
18   core: set Course, courses: set Course
19 }
20 one sig Alice extends Student {} {
21   core = CS101
22 }
23 pred validSchedule[courses: Student -> Course] {
24   all stu: Student {
25     #stu.courses > 2
26     stu.core in stu.courses
27     all disj c1, c2: stu.courses | not conflict[c1, c2]
28   }
29 }
30 pred conflict[c1, c2: Course] {
31   some l1, l2: Lecture {
32     l1 in c1.lectures and l2 in c2.lectures
33     l1.day = l2.day and l1.time = l2.time
34   }
35 }
36 run { validSchedule[courses] }
```

Figure 1: An Alloy model for the course scheduling system.

2.1 Solving Optimization Problems

Suppose that we wish to extend this model to encode students' interests in subjects, i.e., every student is interested in a set of courses, and the model should generate a schedule that *maximizes* the number of courses that match a student's interest, *in addition* to satisfying the three basic requirements. The following code snippet shows the new definition for Student and Alice, where Alice is interested in SE and ML.

```
1  abstract sig Student {
2    core: set Course, courses: set Course,
3    interests: set Course
4  }
5  one sig Alice extends Student {} {
```

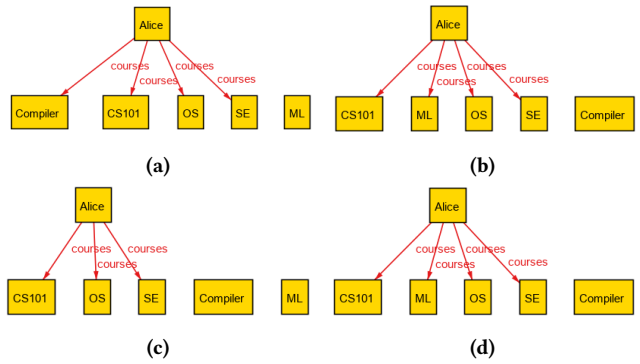

Figure 2: Instances of the course scheduling problem.

```
6    core = CS101
7    interests = ML + SE
8  }
```

Even though this maximization problem can be expressed in Alloy as follows, it contains a higher-order quantification (line 3, over possible relations from *Student* to *Course*), which cannot be solved by the Alloy Analyzer[1]:

```
1  run {
2    validSchedule[courses]
3    no courses': Student -> Course {
4      validSchedule[courses']
5      some stu: Student | #(stu.interests & stu.courses) <
6        #(stu.interests & stu.courses')
7    }
8  }
```

This optimization problem can be solved by our extension, Alloy^{Max}. The following code snippet shows a run command that use our new *maxsome multiplicity* construct:

```
1  run MaxInterests1 {
2   validSchedule[courses]
3   all stu:Student | maxsome stu.interests & stu.courses }
```

The meaning of the constraint on line 3 is that for any student, there exists *some* elements in the intersection of set *stu.interests* and set *stu.courses* that should be maximized, i.e., find an instance that maximizes the number of elements in this intersection.

An alternative way of specifying the optimization problem is by using our new *maxsome quantifier*:

```
1  run MaxInterests2 {
2    validSchedule[courses]
3    all stu: Student | maxsome comm: set Course |
4      comm in stu.interests and comm in stu.courses }
```

Figure 2b shows the instance generated by a MaxSAT solver, which suggests that Alice could take CS101, ML, OS, and SE, where ML and SE are the courses that Alice is interested in.

2.2 Solving Problems with Soft Constraints

We wish to further extend the model to allow students to indicate preferences over lecture times for their courses. For example, suppose that Alice does not want to take courses on Thursday mornings and Friday afternoons. The following code snippet shows a run command with a constraint that encodes Alice's preferences. When the command is executed, the Alloy Analyzer fails to find any valid schedule (i.e., the resulting formula is unsatisfiable) because Alice has to take CS101, which has lectures on Friday afternoons.

```
1  run WithPrefer {
2    validSchedule[courses]
3    all stu: Student | maxsome stu.interests & stu.courses
4    no lec: Alice.courses.lectures |
5      lec.day = Fri and lec.time = PM
6    no lec: Alice.courses.lectures |
7      lec.day = Thu and lec.time = AM }
```

One way to find a satisfying instance is to relax some of the constraints; for instance, Alice could be asked to give up on some of her preferences to satisfy the requirements. However, Alloy does not provide a way to specify such desirable but *optional* constraints.

In Alloy^{Max}, we introduce a notion of *soft constraints*, i.e., a set of constraints that are not required but should be satisfied as much as possible. The following code shows a new type of construct called *soft fact*, which contains a list of soft constraints:

```
1  soft fact {
2    no lec: Alice.courses.lectures |
3      lec.day = Fri and lec.time = PM
4    no lec: Alice.courses.lectures |
5      lec.day = Thu and lec.time = AM
6  }
7  run WithSoftPrefer {
8   validSchedule[courses]
9   all stu:Student | maxsome stu.interests & stu.courses }
```

Given the above run command, Alloy^{Max} will attempt to find an instance that maximizes the number of satisfied soft constraints, in addition to the other hard constraints that encode the three basic requirements. Figure 2c shows an instance where Alice could take CS101, OS, and SE, i.e., Alice gives up her Friday afternoons in order to register for CS101, but her Thursday mornings are freed up.

2.3 Solving with Priorities

However, Alice is not entirely satisfied with the above schedule. Although her Thursdays are free, she has to give up the ML course, which she is interested in. Given the choice between two conflicting goals—keeping Thursday mornings free and studying ML—she would rather choose the latter; i.e., ML has a higher *priority* for her. Currently, in Alloy, there is no way to express and analyze with such a notion of *priorities* among constraints.

In Alloy^{Max}, *priorities* can be explicitly assigned to maximal or soft constraints to indicate the user's preferences on which of the possibly conflicting constraints must be satisfied first. For example, in the following snippet, *maxsome[1]* indicates that maximizing the number of interested courses has a higher priority than satisfying the time preferences (i.e. *soft fact* is assigned default priority of 0).

```
1  soft fact {
2    no lec: Alice.courses.lectures |
3      lec.day = Fri and lec.time = PM
4    no lec: Alice.courses.lectures |
5      lec.day = Thu and lec.time = AM
6  }
7  run WithSoftPreferAndPrior {
8    validSchedule[courses]
9    all stu: Student |
```

[1]Another extension of Alloy, called Alloy* [33], can be used to solve higher-order quantification, but does not support other types of analysis shown in Sections 2.2 and 2.3. We provide a more detailed comparison of Alloy^{Max} and Alloy* in Section 7.

```
10      maxsome[1] stu.interests & stu.courses
11  }
```

Figure 2d shows the instance generated by the solver; i.e., Alice could take CS101, ML, OS, and SE, which maximizes her course interests over her time preferences.

In summary, our extension expands the range of analyses available in Alloy, including (1) generating an instance which maximizes or minimizes a given relation, which can be used for optimization problems (2) defining soft constraints and finding an instance that satisfies as many of these optional but desired constraints, and (3) specifying priorities among different soft or maximal constraints.

3 BACKGROUND ON MAXSAT

A propositional formula in *conjunctive normal form* (CNF) is defined as a conjunction of clauses where a clause is a disjunction of literals such that a literal is either a propositional variable v_i or its negation $\neg v_i$. A clause is satisfied if at least one of its literals is satisfied. Finally, a CNF formula is satisfied if all its clauses are satisfied. Given a CNF formula ϕ, the Satisfiability (SAT) problem corresponds to decide if there is an assignment such that ϕ is satisfied or prove that no such assignment exists.

The Maximum Satisfiability (MaxSAT) is an optimization version of the SAT problem. Given a CNF formula ϕ, the goal is to find an assignment that maximizes the number of satisfied clauses in ϕ. In partial MaxSAT, clauses in ϕ are split in hard ϕ_h and soft ϕ_s. Given a formula $\phi = (\phi_h, \phi_s)$, the goal is to find an assignment that satisfies all hard clauses in ϕ_h while minimizing the number of unsatisfied soft clauses in ϕ_s. This problem can be further generalized to weighted MaxSAT, where each soft clause has a positive weight and the goal becomes to minimize the sum of the weights of unsatisfied soft clauses. MaxSAT algorithms have seen a remarkable improvement in the last decade [4] and can be used to solve problems in domains such as planning [48], data analysis [28], security [10], and bioinformatics [11]. In this paper, we leverage existing MaxSAT technology and extend Alloy to handle optimization problems.

4 SYNTAX AND SEMANTICS

This section describes the syntax and semantics of Alloy$^{\text{Max}}$. It also presents the formal translation rules from an Alloy$^{\text{Max}}$ specification to a MaxSAT problem.

4.1 Abstract Syntax

Figure 3 shows the abstract syntax of Alloy$^{\text{Max}}$. An Alloy$^{\text{Max}}$ problem is a tuple $P = \langle \mathcal{A}, D, F, F_S \rangle$ where $\mathcal{A}$ is the set of atoms in the universe, D is a set of relation declarations, F is an Alloy formula which defines the *hard* constraints that must be satisfied, F_S is a *soft* Alloy formula which defines the *soft* constraints that may or may not be satisfied. A (*hard*) Alloy formula may contain keywords such as *maxsome* for generating optimal solutions.

Optimization operators. Alloy$^{\text{Max}}$ extends the existing *some* multiplicity operator with *maxsome* and *minsome* for maximization and minimization, respectively. Informally, *some r* is true when relation r is not empty; but with *maxsome* (*minsome*), the solver returns a model with a non-empty r that has a maximal (minimal) number of tuples.

Alloy$^{\text{Max}}$ also extends the *no* multiplicity operator with *softno*. Informally, *no r* is true when r is empty. For *softno*, the solver tries to find a model with an empty r, but if no such satisfying instance exists, the solver instead returns a model where r has the minimal number of tuples. The difference between *minsome* and *softno* is that *minsome* requires the relation r to have at least one tuple.

In addition, the existential quantifier (i.e., *some*) is also extended with *maxsome* and *minsome*. In Alloy, *some* $e : r \mid F$ is true when there exists a tuple in relation r which makes formula F true. In Alloy$^{\text{Max}}$, when *maxsome* (*minsome*) is used in place of *some*, the solver finds a model where r contains at least one tuple and tries to maximize (minimize) the number of tuples in r that makes F true.

Soft constraints. Alloy$^{\text{Max}}$ introduces the *soft* fact keyword to specify a soft constraint, i.e., *soft fact F* where F is an Alloy formula but becomes *soft*. When multiple *soft fact* are present in a given specification, the overall soft constraint F_S is the conjunction of those soft Alloy formulas.

Priority. A priority $p \in \mathbb{N}_0$ may be specified along with every one of the new operators in Alloy$^{\text{Max}}$. When it is left unspecified, a default lowest priority 0 is assigned to the associated Alloy formula.

4.2 Translation

Overview. Figure 4 formally defines the translation rules for Alloy$^{\text{Max}}$, as an extension to the existing translation process in Alloy [15, 42]. Due to limited space, we focus on the parts of the translation process that are most relevant for Alloy$^{\text{Max}}$.

At high-level, a relation in Alloy is translated into a matrix of Boolean variables (each of which is true if and only if the tuple represented by this particular variable is in the relation), and a relational expression (e.g. dot join) is represented by operations over one or more matrices. In Alloy, a FOL problem with bounds is translated into a Boolean formula, which is eventually converted into a CNF. In Alloy$^{\text{Max}}$, the idea is to instead convert it into *weighted* conjunctive normal form (WCNF), where (1) hard Boolean formulas are assigned the special weight of $+\infty$, to ensure that they are satisfied in every instance and (2) soft formulas are assigned different weights (depending on the user-specified priorities), to guide the solver towards an instance that maximizes the total sum of weights.

Translation Steps. First, an Alloy$^{\text{Max}}$ problem is translated into a Boolean formula with priorities, denoted by bool^p. A prioritized Boolean formula F^k is the Boolean formula F with an optional priority, k, which is used in a later part of the translation to assign weights to the corresponding WCNF clauses. When k is omitted, it represents a hard formula that must be satisfied.

In Figure 4, function M_S is used to translate soft constraints, i.e., constraints that are defined as part of *soft fact* in Alloy$^{\text{Max}}$. Given a soft constraint *soft fact F* with priority k, when F is a conjunction of Alloy formulas, M_S translates each conjuncts into a soft constraint also with priority k (line 17); otherwise, M_S translates the contained Alloy formula F by calling M and assigns k as its priority (line 18).

Function M is used to translated Alloy formulas. Keywords in vanilla Alloy follow the existing translation rules. For each of the new keywords in Alloy$^{\text{Max}}$ (e.g., *maxsome*, *minsome*), the translation rule is similar to the existing rule, except additional prioritized Boolean formulas are introduced to instruct the solver to search for a maximal or minimal solution. For instance, on line 21,

problem := univDecl relDecl* formula softFormula
softFormula := **soft** formula | **soft[0..]** formula | compositeSoft
compositeSoft := softFormula **and** softFormula

univDecl := {atom[,atom]*}
relDecl := rel :$_{\text{arity}}$
varDecl := var : expr
constant := {tuple*}
tuple := ⟨atom[,atom]*⟩

arity := 1 | 2 | 3 | ...
atom := identifier
rel := identifier

var := identifier

expr := rel | var | unary | binary | comprehension
unary := unop expr
unop := ~ | ^
binary := expr binop expr
binop := + | & | - | . | ->
comprehension := {varDecl | formula}

formula := elementary | composite | quantified
elementary := expr **in** expr | mult expr
mult := **some** | **maxsome** | **maxsome[0..]** | **minsome** | **minsome[0..]** | **no** | **softno** | **softno[0..]** | **one**
composite := **not** formula | formula logop formula
logop := **and** | **or**
quantified := quantifier varDecl | formula
quantifier := **all** | **some** | **maxsome** | **maxsome[0..]** | **minsome** | **minsome[0..]**

Figure 3: Abstract syntax of Alloy^{Max}. Bolded text are existing keywords in Alloy; text in blue are new keywords in Alloy^{Max}.

P : problem → bool^p (1)
R : relDecl → univDecl → matrix (2)
M : formula → env → bool^p (3)
M_S : softFormula → env → bool^p (4)
X : expr → env → matrix (5)
env: (quantVar ∪ relVar) → matrix (6)

$\overrightarrow{x}, \langle i_1, \ldots, i_k \rangle$: vectors (7)
$[m]$: matrix → $\{\langle int \rangle\}$, set of indices of matrix m (8)
$|m|$: matrix → dim, dimension of matrix m (9)

$\mathcal{M}$: dim → (⟨int⟩ → bool^p) → matrix, constructor (10)

$\mathcal{M}(s^d, f) = \{m \mid |m| = s^d \wedge \forall \overrightarrow{x} \in \{0, \ldots, s-1\}^d, m[\overrightarrow{x}] = f(\overrightarrow{x})\}$ (11)

$\mathcal{M}$: dim → ⟨int⟩ → matrix, constructor (12)
$\mathcal{M}(s^d, \overrightarrow{x}) = \mathcal{M}(s^d, \lambda \overrightarrow{y}.$ if $\overrightarrow{y} = \overrightarrow{x}$ then $true$ else $false)$ (13)

$P[\mathcal{A}\ d_1, \ldots, d_n\ F\ F_S] =$ let $e = \bigcup_{i=1}^{n}(r_i \mapsto R[d_i]\mathcal{A})$ in $M[F]e \wedge M_S[F_S]e$ (14)
$R[r :_k]\mathcal{A} = \mathcal{M}(|\mathcal{A}|^k, \lambda \overrightarrow{x}$. freshVar()) (15)

$M_S[F_S$ and $G_S]e = M_S[F_S]e \wedge M_S[G_S]e$ (16)
$M_S[$soft[k] F and $G]e = M_S[$soft[k] $F]e \wedge M_S[$soft[k] $G]e$ (17)
$M_S[$soft[k] $F]e = (M[F]e)^k$, where F is not a conjunction (18)

$M[p$ in $q]e = \bigwedge(\neg X[p]e \vee X[q]e)$ (19)
$M[$some $p]e = \bigvee(X[p]e)$ (20)
$M[$maxsome[k] $p]e = M[$some $p]e \wedge \bigwedge(X[p]e)^k$ (21)
$M[$minsome[k] $p]e = M[$some $p]e \wedge \bigwedge(\neg X[p]e)^k$ (22)
$M[$one $p]e = ExactlyOne(X[p]e)$ (23)
$M[$no $p]e = \bigwedge(\neg X[p]e)$ (24)
$M[$softno[k] $p]e = \bigwedge(\neg X[p]e)^k$ (25)
$M[$not $F]e = \neg M[F]e$ (26)
$M[F$ and $G]e = M[F]e \wedge M[G]e$ (27)
$M[F$ or $G]e = M[F]e \vee M[G]e$ (28)
$M[$all $v : p \mid F]e =$ let $(m = X[p]e)$ in $\bigwedge_{\overrightarrow{x} \in [m]}(\neg m[\overrightarrow{x}] \vee M[F](e : v \mapsto \mathcal{M}(|m|, \overrightarrow{x})))$ (29)
$M[$some $v : p \mid F]e =$ let $(m = X[p]e)$ in $\bigvee_{\overrightarrow{x} \in [m]}(m[\overrightarrow{x}] \wedge M[F](e : v \mapsto \mathcal{M}(|m|, \overrightarrow{x})))$ (30)
$M[$maxsome[k] $v : p \mid F]e = M[$some $v : p \mid F]e \wedge \bigwedge(X[p]e)^k$ (31)
$M[$minsome[k] $v : p \mid F]e = M[$some $v : p \mid F]e \wedge \bigwedge(\neg X[p]e)^k$ (32)
$X[p + q]e = X[p]e \vee X[q]e$ (33)
$X[p \,\&\, q]e = X[p]e \wedge X[q]e$ (34)
$X[p - q]e = X[p]e \wedge \neg X[q]e$ (35)
$X[p \cdot q]e = X[p]e \cdot X[q]e$ (36)
$X[p$ -> $q]e = X[p]e \times X[q]e$ (37)
$X[\sim p]e = (X[p]e)^T$ (38)
$X[\hat{} p]e = IterativeSquare(X[p]e)$ (39)
$X[\{v : p \mid F\}]e =$ let $(m = X[p]e)$ in $\mathcal{M}(|m|, \lambda \overrightarrow{x}$. $m[x] \wedge M[F](e : v \mapsto \mathcal{M}(|m|, \overrightarrow{x})))$ (40)

Figure 4: Translation rules. bool^p is a Boolean formula with priority where $p \in \mathbb{N}_0$. When p is omitted, the formula represents a hard formula that must be satisfied. In other words, it has an infinite priority, i.e. $F = F^{+\infty}$.

maxsome[k] p is translated by (1) stating that p must not be empty, as it is done for *some* in vanilla Alloy (line 20), and (2) including a prioritized Boolean formula for the presence of each tuple in p; given the corresponding WCNF, a MaxSAT solver attempts to find a solution that maximizes the number of tuples in p.

Each prioritized Boolean formula is transformed into a CNF formula with a priority for each clause, by using the Tseitin transformation [44]. Finally, a CNF formula with priority is translated into a WCNF formula, to be solved by a MaxSAT solver. To guarantee that a soft clause with a higher priority is always satisfied first before the clauses with lower priorities, its weight is assigned to

be greater than the sum of weights of all the clauses with smaller priorities. Formally, let C_i represent all the clauses with priority i. Then, for their weight W_i is assigned as:

$$W_i = 1 + \sum_{j=0}^{i-1} W_j * |C_j|$$

where $|C_j|$ is the number of clauses of C_j and $W_0 = 1$. This kind of optimization is called lexicographic optimization and using this weight distribution is a known way of converting it to MaxSAT [29].

Example. We use an example to illustrate one of the translation rules (in particular, rule for *maxsome* on line 21 in Figure 4). Consider two relations $p : A \times B$ and $q : B \times C$, where $A = \{A1, A2\}$, $B = \{B1\}$, and $C = \{C1\}$. Suppose that we wish to find a model that satisfies the following formula:

$$maxsome\ p.q$$

Relation p is represented by a set of Boolean variables $\{p_{00}, p_{10}\}$ (where, for example, p_{00} is true if and only if $(A1, B1)$ is a tuple in p); similarly, q is represented by $\{q_{00}\}$. Then, the formula is translated into a prioritized Boolean formula as follows:

$$((p_{00} \wedge q_{00}) \vee (p_{10} \wedge q_{00})) \ \wedge (p_{00} \wedge q_{00})^0 \wedge (p_{10} \wedge q_{00})^0$$

where the first top-level conjunct says that $p.q$ must contain at least one tuple (equivalent to *some* $p.q$); the next two conjuncts are formulas with the lowest priority of 0 stating that $p.q$ should contain as many of the two possible tuples as possible.

To translate the above formula into CNF, the Tseitin transformation introduces fresh variables that represent sub-formulas:

$$a \leftrightarrow p_{00} \wedge q_{00}\ ,\ b \leftrightarrow p_{10} \wedge q_{00}$$

where $\{a, b\}$ can also be seen as the variables representing the tuples in relation $p.q$. Then, after the transformation, the following prioritized CNF is produced:

$$\begin{aligned}&(a \vee b) \wedge a^0 \wedge b^0 \wedge \\ &(\neg a \vee p_{00}) \wedge (\neg a \vee q_{00}) \wedge (a \vee \neg p_{00} \vee \neg q_{00}) \wedge \\ &(\neg b \vee p_{10}) \wedge (\neg b \vee q_{00}) \wedge (b \vee \neg p_{10} \vee \neg q_{00})\end{aligned}$$

Finally, we replace the priorities with weights. Since we only have one priority in the formula, a^0 and b^0 are assigned weight 1.

4.3 Semantics Comparison

A *model* or an *instance* of an Alloy formula is an assignment of tuples to relations that makes the formula true. An Alloy problem may have multiple models. We call the set of all models the *solution space* of a problem. Then, every Alloy$^{\text{Max}}$ problem has a corresponding Alloy problem with the same solution space. In particular:

- The *maxsome* and *minsome* operators have the same impact on the overall solution space as *some* does. A model is a solution to a problem using *maxsome* or *minsome* if and only if it is also a solution to the same problem using *some*. The only difference in Alloy$^{\text{Max}}$ is that the solutions are returned in the decreasing order of optimality (e.g., for *maxsome*, start with a maximal solution and explore the space in the order of decreasing optimality).
- For the *softno* operator, given *softno* r, it is true when r is either empty or non-empty. Thus, it is semantically equal to *true*, but the returned instances also follow the decreasing order of optimality (i.e, starting from the minimal r).
- Given *soft fact* F, it is also semantically equal to *true*, i.e., it is true when F is either satisfied or not. But again, the returned instances start with one where F is satisfied if possible.

In sum, Alloy$^{\text{Max}}$ problem $P = \langle \mathcal{A}, D, F, F_S \rangle$, is semantically equivalent to Alloy problem $P' = \langle \mathcal{A}, D, F' \rangle$ where (1) the soft constraint F_S is replaced by *true*, (2) the *maxsome* and *minsome* operators are replaced by *some*, and (3) the *softno* formula is replaced by *true*.

5 OPTIMIZATION

By encoding problems in MaxSAT, Alloy$^{\text{Max}}$ can potentially take advantage of advances in MaxSAT technology to more efficiently solve the underlying optimization problem. However, when encoding the problem to a low-level WCNF, the high-level structure of the problem may be lost and become unavailable for MaxSAT algorithms to take advantage. Prior approaches try to recover this information by clustering soft clauses into partitions and using them during solving [30, 37]. However, these approaches are heuristic-based and sometimes fail to capture meaningful relationships between soft clauses. In this paper, we propose to extract the partition information directly from the Alloy$^{\text{Max}}$ syntax and give this information to existing partition-based MaxSAT algorithms.

5.1 Partition-based MaxSAT solving

We briefly present an overview of unsatisfiability-based MaxSAT algorithms that use partition strategies [30, 37]. Unsatisfiability-based algorithms work by finding a sequence of unsatisfiable subformulas that correspond to increasing a lower bound on the number of falsified soft clauses until the formula becomes satisfiable and an optimal solution is found. These algorithms can be extended with partition strategies by taking as input the MaxSAT formula ϕ and partitioning this formula into formulas $\phi_1, \ldots, \phi_n$ with disjoint set of soft clauses. Different approaches have been proposed to partition a MaxSAT formula but they are mostly based on having a graph representation of the formula and then using graph partitioning algorithms to create the partitions [30, 37].

The MaxSAT solver then takes as input these partitions and starts by solving ϕ_1. If the formula is unsatisfiable, then the lower bound on the number of falsified soft clauses is increased and the formula is refined to allow one additional soft clause to be falsified. This loop is repeated until the formula becomes satisfiable. In this case, the algorithm found a solution to the formula that may or not be optimal. If no more partitions are left, then this solution is guaranteed to be optimal. Otherwise, the next partition is added to the solver and this process is repeated until an optimal solution is found. Partitioning has the benefit of potentially finding smaller unsatisfiable subformulas at each iteration of the algorithm, which often leads to a sequence of unsatisfiable formulas that is easier to solve than without partitions. For further details on MaxSAT algorithms, we refer the interested reader to the literature [4].

5.2 Extracting Partitions from Alloy$^{\text{Max}}$

Our approach, which we call *spec-based partitioning*, changes this existing process by using partitions that can be extracted directly from Alloy$^{\text{Max}}$ as the translation to MaxSAT is performed, rather

than trying to guess "good" partitions at the WCNF level. In particular, partitions can be generated as follows.

- When adding a hard clause to the WCNF, it is assigned group 1. All the hard clauses in the WCNF belong to the same group.
- When translating an Alloy formula with optimization operators, i.e., *maxsome*, *minsome*, and *softno*, Alloy^{Max} creates a new group g. Then, all the soft clauses associated with this optimization operator are assigned group g.
- When translating a soft constraint, i.e., *soft fact* F, Alloy^{Max} creates a new group g' for it. Thus, each soft constraint is in its own group.

A common pattern that we leverage for creating partitions in Alloy^{Max} is to use *all* keyword. For example, consider set P and Q, and a relation $r : P \times Q$. For formula *all* $p : P \mid$ *maxsome* $p.r$, the *all* ($\forall$) operator will be expanded generating a *maxsome* $p.r$ formula for each $p \in P$. Therefore, each *maxsome* formula will be in a separate group, and we can easily partition the soft clauses by the tuples in set P.

In Section 6.3, we compare the graph-based partitioning strategy used in MaxSAT algorithms with Alloy^{Max} partitioning and show the benefit of extracting partitions from a high-level language.

6 EVALUATION

We evaluate Alloy^{Max} based on the following research questions:

RQ1: How useful is Alloy^{Max} *for modeling different types of problems?* For usefulness, we focus on *applicability* and *scalability*.

RQ2: Is spec-based partitioning more efficient than no-partitioning and graph-based auto-partitioning?

RQ3: Is Alloy^{Max} *more efficient than Alloy* for generating optimal solutions?* Alloy* [33] is an extension that enables an analysis with higher-order quantifiers, so it could, in principle, be used to solve problems that generates an optimal solution as the one in Section 2.1. This question aims at comparing which approach is more efficient in this particular type of problem.

To answer these research questions, we apply Alloy^{Max} to five case studies from different domains. Section 6.2 gives brief introductions to each case study and presents the optimization goals.

6.1 Implementation

We implemented Alloy^{Max} based on Alloy 5.1.0, which, in turn, relies on Kodkod, a general-purpose relational model finder [42]. We extended Alloy to support the new syntax, and also modified Kodkod to support the new translation process. We use Open-WBO [31] as our backend MaxSAT with the same algorithm for both partitioning and non-partitioning. For spec-based partitioning, we extended the WCNF format to allow specifying group index at the beginning of each clause. We modified Open-WBO to read this new file format and changed the partition strategy to use the one specified in the file. All the experiments were run on a Linux machine with a 4 core 3.6GHZ CPU and 20 GB memory. Every problem was run 5 times (each with a 30 minutes timeout), from which the average was computed and reported in Table 2. We make the benchmark and tool available for reproduction[2], and plan to turn our data into archived open data in case of acceptance.

[2]https://figshare.com/s/8a21692ea54ff475651f

6.2 Case Studies

6.2.1 Course Scheduling. Recall the course scheduling model described in Section 2. In this benchmark, we randomly generate problems with C courses and S students; a course should have either *two* or *three* lectures among a week; each student should take at most R core courses and is interested in at most I courses.

The optimization goal is to maximize the number of interested courses that a student is registered for. The problem names in Table 2 follow the pattern *course_*{C}*_*{S}*_*{R}*_*{I}. The size of the Alloy^{Max} specifications ranges from 252 to 552 LOC.

6.2.2 CheckMate. CheckMate [43] is an automated tool based on Alloy for synthesizing proof-of-concept exploit code for hardware security attacks similar to Meltdown [24] and Spectre [20]. It uses a "micro- architecturally happens-before" graph [25], denoted as μhb, to model and analyze the execution process of micro-operations. In a CPU, an instruction goes through a sequence of stages to be executed (e.g., Fetch, Exec., and Commit). Then, a μhb graph represents the execution process of a series of program instructions.

An attack pattern (e.g., Meltdown or Spectre) is represented as a sub-graph pattern in μhb. CheckMate models the micro-architecture of a CPU, the events of instructions, and the attack pattern in Alloy, and uses the Alloy Analyzer to generate a μhb graph that contains the attack pattern. Moreover, CheckMate aims to synthesize security litmus tests, i.e., most *compact* programs that can demonstrate a security attack. However, since this optimization goal cannot be solved by Alloy, the authors of CheckMate developed an additional Python program to enumerate all the possible instances and find the litmus tests.

With Alloy^{Max}, this optimization task can be formulated as minimizing the relation that represents the μhb graph. In particular, this goal is modeled by using *softno*, and *all* is used for leveraging partitioning (as shown below), and the returned instance contains the smallest μhb graph which represents a security litmus test.

```
run {
  ... // CheckMate constraints
   all n: Node | softno n.uhb }
```

The original CheckMate model is around 1,100 LOC in Alloy; to specify the optimization goal, it only involved adding the above *softno* formula to the specification.

6.2.3 Graceful Degradation. In network security, a simple way of mitigating an on-going attack is to shut down the entire system. In practice, however, this is unacceptable because the system will lose all of its functions. *Graceful degradation* [13, 46] is an approach for dynamically re-configuring parts of a system under an attack to maximize the remaining uncompromised functions. Typically, this is achieved by (1) shutting down only those components that have been compromised, and (2) replacing these components with backups (if available) to maintain the affected functions. This reconfiguration process also involves removing network connections to the compromised components and adding new ones to the backups.

Alloy can be used to model and analyze the problem of generating a valid network reconfiguration that maintains all system functions. However, depending on the extent of an attack, it may be impossible to restore some of those functions. To determine which of them can be restored, the Alloy user could manually comment out a selection

of constraints that correspond to the functions, but the process to find an optimal solution (i.e., one that restores as many functions as possible) would be tedious.

With Alloy$^{\text{Max}}$, we use *soft fact* to model the functions as soft constraints (line 4-6). Given these, the solver will try to maximize the functions and give up on those unsatisfiable ones. Moreover, in practice, every change to the architecture could have a cost, and the total cost (the number of architectural modifications) should also be minimized. Thus, we use *maxsome* to maximize the overlapping connections between the degraded architecture and the initial one (line 9), and use *softno* to minimize the new connections added to the degraded architecture (line 10). Furthermore, system function is considered more important than cost. Thus, soft constraints representing the functions are assigned a higher priority. The following snippet shows our optimization goals.

```
1  sig Component {
2    init: set Component, degraded: set Component
3  }
4  soft[1] fact {
5    NonCriticalFunction and CriticalFunction
6  }
7  run {
8    validArchitecture[degraded]
9    maxsome degraded & init
10   softno degraded - init
11 }
```

In our benchmark, each problem has C components including backup ones; the attacker's capability (in terms of the degree that the system might be compromised) is set to K, and; each problem has A initially compromised components. The problem names follow the pattern *degrade*_{C}_{K}_{A}. The size of the specifications ranges from 240 to 326 LOC.

6.2.4 Wedding Table Seating. Consider a wedding table seating problem [32], where (1) each table seats a minimum and a maximum number of guests; (2) each person should be assigned to only one table; (3) each person is associated with a set of tags indicating their interests. The hard constraint is to generate a seating assignment for each person, and the optimization goal is to minimize the number of different tags between all people sitting at the same table.

Alloy can be used to model this problem and generate an arbitrary assignment. The following snippet shows relevant constraints.

```
1 sig Tag {}
2 sig Person { tags: set Tag }
3 sig Table { seat: set Person } {
4   #seat =< MAX_NUM_PERSON
5   #seat >= MIN_NUM_PERSON
6 }
7 fact { all p: Person | one seat.p }
8 run {}
```

However, with this model, we cannot find the optimal solution that minimizes the number of different tags at a table. With Alloy$^{\text{Max}}$, this optimization goal can be modeled by using *softno*. In particular, to take advantage of the partitioning optimization, the goal is specified as, for each table, it should have no or minimized tags. The following expresses this optimization goal.

```
1 run TableBased { all t: Table | softno t.seat.tags }
```

For the benchmark, we randomly generated problems with G tags, T tables, and P people, where each table has minimum L and maximum H people, and each person has maximum M tags. The problem names follow the pattern *seat*_{G}_{P}_{M}_{T}_{L}_{H}. The size of the specifications ranges from 50 to 55 LOC.

6.2.5 Single Machine Scheduling (SMS). In many real-time systems, it is often impossible to schedule tasks to make them meet all their time requirements. In practice, one must schedule the tasks in such a way to can maximize the number of on-time tasks [22]. Consider a real-time system with a set of n tasks $\{\tau_1, \ldots, \tau_n\}$ to be executed on a uni-processor machine, i.e., only one task can be executed at a time. Each task is a triple $\langle r_i, p_i, d_i \rangle$ where r_i is the release time (i.e., the earliest start time), p_i the processing time, and d_i the deadline.

Tasks can be split into fragments so that one task can be interrupted in order to execute fragments of other tasks. However, the execution order of the fragments of one task should not be changed. Formally, a task τ_i has k_i fragments and each fragment f_i^j where $j \in \{1, \ldots, k_i\}$ has process time p_i^j, and we have $\sum_{j=1}^{k_i} p_i^j = p_i$. Finally, a task may depend on other tasks. Thus, if a task τ_i depends on task τ_j, τ_i must be executed only after τ_j has been completed.

Alloy can be used to model this problem and find an arbitrary subset of tasks that can complete within their deadlines. However, the maximality is not guaranteed. With Alloy$^{\text{Max}}$, we use *maxsome* to find the optimal schedule which maximizes the number of on-time tasks. The following snippet shows the optimization goal.

```
1 sig Task {} sig Completed in Task {}
2 run {
3   CompleteOntime[Completed]
4   maxsome Completed
5 }
```

In our benchmark, we randomly generate problems with N tasks. The release time of each task is within a time frame R; each task has a maximum processing time P, and a maximum number of fragments F. A slack factor S is used to compute the deadline for each task where $d_i = r_i + random(1, S) * p_i$. Finally, a task depends on a maximum number of D tasks. The problem names follow the pattern *sms*_{N}_{R}_{P}_{F}_{S}_{D}. The size of the specifications ranges from 192 to 259 LOC.

6.3 Results

6.3.1 RQ1: How useful is Alloy$^{\text{Max}}$ *for modeling different types of problems?* We successfully model the aforementioned problems with Alloy$^{\text{Max}}$ (or migrate from existing ones). It only requires appending the optimization formulas to the original run commands in plain Alloy without any reconstruction.

Table 2 shows the problems used in our benchmark. The problem name reflects the scale of each problem. Specifically, the size of the Alloy$^{\text{Max}}$ specifications for these problems ranges from 50 LOC to 1,109 LOC. The number of variables in the generated WCNF ranges from 2,236 to 24,583,678, and the number of clauses ranges from 6,921 to 43,729,378. Although translating from a high-level language like Alloy$^{\text{Max}}$ may not produce the most succinct MaxSAT formulas, it can solve all these problems within a 30 minutes timeout.

6.3.2 RQ2: Is spec-based partitioning more efficient than no- partitioning and graph-based auto-partitioning? The results in Table 2 show that spec-based partitioning outperforms both no- and auto-partitioning in Course, CheckMate, and Seating problems. On

Table 2: Results for comparing MaxSAT non-partitioning vs. MaxSAT partitioning vs. Alloy*. The numbers in the table are the solving time of the MaxSAT solver or the solver for Alloy* (i.e., excluding the translation time). There is no significant difference in translation time among these methods. * indicates auto-partition fails because the problem is too large and it falls back to no-partition. ** indicates only one partition exists in the problem and it falls back to no-partition.

Problem		No-Part. Solve (s)	Auto-Part. Solve (s)	Spec-Part. Solve (s)	Alloy* Solve (s)
Course	course_30_40_3_6	0.93	0.19	0.17	112.08
	course_40_50_3_6	3.90	0.42	0.31	437.72
	course_50_60_3_6	11.84	0.78	0.62	860.49
	course_60_70_3_6	31.38	1.14	0.90	1607.65
	course_70_80_3_6	74.83	1.89	1.47	TO
	course_80_90_3_6	175.40	2.96	2.09	TO
	course_90_100_3_6	277.93	4.72	3.09	TO
CheckMate	Flush+Reload	123.01	133.60*	94.68	N/A
	Meltdown	353.74	368.13*	272.35	N/A
	Spectre	634.55	660.15*	520.91	N/A
Seating	seat_7_28_7_6_3_7	3.06	7.19	1.76	26.47
	seat_7_30_7_6_3_7	0.70	1.23	0.65	52.27
	seat_8_30_8_7_3_7	133.55	54.78	35.36	239.21
	seat_8_32_8_7_3_7	513.73	619.33	349.20	TO
SMS	sms_34_20_6_3_3_2	20.06	31.00	20.05**	479.84
	sms_38_20_6_3_3_2	19.13	35.12	19.47**	863.60
	sms_42_20_6_3_3_2	21.71	52.12	20.63**	1464.60
	sms_46_20_6_3_3_2	186.98	263.31	184.93**	TO
	sms_50_20_6_3_3_2	179.86	173.91	186.74**	TO
	sms_52_20_6_3_3_2	81.22	198.73	85.21**	TO
Degradation	degrade_10_2_1	0.26	0.77	0.28	N/A
	degrade_26_2_1	25.55	43.08*	27.76	N/A
	degrade_20_3_1	659.46	673.30*	TO	N/A
	degrade_26_2_2	1442.41	1453.32*	898.7	N/A

SMS, spec-based partitioning shows little improvement over no-partitioning because these problems do not have a natural, problem-specific way of partitioning; as a result, the solver falls back to no-partitioning. On the other hand, spec-based partitioning has negative impact on the first three Degradation problems. Moreover, we have the following observations and insights:

- When partitioning is applicable, spec-based partitioning outperforms auto-partitioning, because (1) the former does not suffer from the overhead introduced by the heuristic for guessing partitions, which can be significant for large problems, and the heuristic may fail and fall back to no-partitioning (e.g., in CheckMate and Degradation), and (2) auto-partitioning might fail to guess effective partitions (e.g., Seating problems where spec-based improves the performance but auto does not).
- When partitioning is not applicable (i.e., only one partition exists), spec-based partitioning falls back to no-partitioning. On the other hand, forcing auto-partitioning in this situation may have negative impact on performance (e.g., SMS).
- We hypothesize that partitioning works well if the optimal solution of a sub-problem (hard clauses plus some partitions) is part of the global optimal solution. The Course problems have this property, since there are no registration limits on the number of students for each course, and the registration for each student is completely independent from one another.
- We observed in the Degradation problems that if a partition added to the iterative process does not help refine the solution towards the optimal one, the overhead of the iteration may exceed the time of solving the problem at once. In other words, partitioning works well when every sub-problem contributes towards the optimal solution.

6.3.3 RQ3: Is Alloy^{Max} *more efficient than Alloy* for generating optimal solutions?* As shown in Table 2, both with or without partitioning, Alloy^{Max} outperforms Alloy* on optimization problems that can be specified using higher-order quantifiers. Alloy* is a general purpose higher-order solver (can be used to solve problems that Alloy^{Max} is not applicable to), and so its analysis method—based on counterexample-guided inductive synthesis (CEGIS) [40]—is not specifically targeted for solving optimization problems. On the other hand, since MaxSAT is designed for solving optimization, it is expected that Alloy^{Max} outperforms Alloy* on such problems.

We did not apply Alloy* to CheckMate because it would require significantly reconstructing the specification to express a higher-order formula that quantifies over all relevant relations to be searched (e.g., relations representing various properties of an attack graph to be minimized). Due to the complexity of this problem, there are 40 such relations, and so this would result in a formula with a quantifier depth of 40—not an Alloy expression that a user can reasonably be expected to write. In contrast, Alloy^{Max} introduces only one line of an optimization formula without changing

the existing specification; this also shows the benefit of the built-in constructs in Alloy^{Max} for specifying optimization tasks.

Alloy* is not applicable to degradation problems, which contain soft constraints that cannot be specified in or solved by Alloy*.

6.3.4 Threats to Validity. Our selected case studies might not be representative enough to demonstrate the applicability of Alloy^{Max}. We believe that our benchmark contains a diverse set of examples from realistic domains (including exploit generation, network reconfiguration, and task scheduling) and of varying sizes (from relatively simple models like wedding seating to complex ones like CheckMate). However, we plan to continually expand our benchmark to explore other potential applications of Alloy^{Max} and further refine or modify the language as needed for new use cases.

In addition, the size of the problems in our benchmark might not be large enough to show how well Alloy^{Max} scales to other Alloy problems in practice. Given that CheckMate is one of the most complex Alloy specifications available (based on our past experience with Alloy) and also has been used to generate practical attacks on real systems [43], we believe that our experiments show that Alloy^{Max} scales well to realistic problems. In addition, since its scalability depends on the efficiency of the underlying MaxSAT solver, Alloy^{Max} will likely benefit from further advances in MaxSAT technology, which is an active area of research [3].

7 RELATED WORK

Extensions to Alloy. Alloy* [33] is an Alloy extension that enables analysis of Alloy specifications with higher-order quantifiers. Although Alloy* is designed for a different purpose than Alloy^{Max}—mainly, enabling synthesis problems in Alloy—it could be used to encode certain types of optimization problems by using higher-order quantifiers, as shown in Section 2.1. However, as shown in Section 6.3, Alloy^{Max} outperforms Alloy* on these problems, in part because the former leverages MaxSAT solvers, which are designed for finding optimal solutions. In addition, Alloy^{Max} provides additional analyses involving soft constraints and priorities.

Aluminum [35] is an extension of Alloy that supports generation of *minimal instances*. Aluminum and Alloy^{Max} are designed for different purposes and differ on the notion of minimality. In particular, the former works by automatically removing as many tuples as possible such that the resulting instance remains a valid instance of the given Alloy specification, and allows the user to influence the order in which Alloy generates its next instances by augmenting the current instance with an additional tuple.

Cunha et. al [8] propose an approach called *target-oriented model finding*, where the goal is to allow the user to provide a partial specification of a *target* instance, and use a solver to generate an instance that is as *close* to the target as possible. Their work also leverages an extension of Kodkod with a MaxSAT solver as the underlying engine, although its overall goal is different from Alloy^{Max}.

Other types of solvers. There is a large body of work on *constrained optimization problems* (COPs)—a generalization of constraint satisfaction problems (CSPs) with an additional notion of an *objective function* to be optimized [38]. The state-of-the-art tools in this domain (e.g., Gurobi [12], CPLEX [14]) employ solving techniques that are different from MaxSAT, such as Branch and Cut algorithms [38]. Since the Alloy Analyzer already relies on SAT, we found MaxSAT to be a natural choice for bringing optimization to Alloy. Using a COP solver as an alternative backend to Alloy^{Max} is an interesting open problem, although it would involve a significantly different translation than the SAT-based one in Kodkod (which Alloy^{Max} builds on).

MiniZinc [36] is a constraint modeling language that is capable of solving various types of CSPs, including optimization. Built on top of solvers such Gurobi or CPLEX, MiniZinc is intended to be a high-level language for specifying CSPs and provides convenient built-in such as functions, data types, and global constraints. MiniZinc provides certain features (e.g., floating numbers) that are missing from Alloy^{Max}, while the relational core of Alloy is better suited for specifying complex structures that arise in software.

Answer set programming (ASP) [23] is a line of work on extending logic programming with capabilities for expressing and solving search problems, including those involving optimization [1]. Although both are declarative in nature, ASP and Alloy^{Max} support distinct styles of modeling: The logic programming paradigm underlying ASP is typically used for data and knowledge representation [7], while Alloy was designed for modeling software designs.

There are recent works on extending satisfiability modulo theories (SMT) solvers with optimization capabilities, such as OptiMathSAT [39], Symba [21], and Z3 [6]. These solvers could serve as an alternative backend for Alloy^{Max}. One benefit of this approach is that they operate over unbounded domains and for formulas involving decidable theories (e.g., integers), can provide a stronger analysis guarantee than a SAT-based backend. On the other hand, dedicated MaxSAT algorithms have shown to be more efficient than the SMT-based methods [26]. A more systematic investigation of these alternative backends for Alloy^{Max} remains a future work.

8 CONCLUSION AND LIMITATIONS

We have presented Alloy^{Max}, an extension of the Alloy modeling language that leverages MaxSAT to solve optimization problems. We have demonstrated that despite introducing only a small number of new language constructs, Alloy^{Max} can be used to specify and solve a diverse set of problems that were previously not possible in Alloy. By enabling these new types of analyses, we believe that Alloy^{Max} has the potential to bring a wide range of new applications to Alloy, and its analysis capability will continue to grow as it benefits from rapid progress in the MaxSAT technology.

The translation rules shown in Section 4 support a richer semantics than the current syntax of Alloy^{Max} allows. For example, the rules support defining *soft* predicates (e.g., *soft* pred P[x: S]) and reuse them in other formulas. We plan to explore this richer semantics through further syntactic extensions (while also considering the possible risk of complicating the syntax) and investigate additional classes of problems that Alloy^{Max} could be applied to.

The current version of Alloy^{Max} supports only priorities among constraints. However, MaxSAT solvers generally use weights, which support more fine-grained optimizations. Adding weights to Alloy requires more subtle investigation on its semantic impact, but it could support a larger set of problems, such as modeling and maximizing utility functions (e.g., those used in self-adaptive systems planners [41] and automated data mismatch repair [45]).

REFERENCES

[1] Answer set optimization. In G. Gottlob and T. Walsh, editors, *International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.

[2] D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song. Towards a formal foundation of web security. In *CSF*, pages 290–304, 2010.

[3] F. Bacchus, J. Berg, M. Järvisalo, and R. Martins. Maxsat evaluation 2020: Solver and benchmark descriptions. Technical report, University of Helsinki, Department of Computer Science, 2020.

[4] F. Bacchus, M. Järvisalo, and R. Martins. Maximum Satisfiability. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, chapter 24, pages 929 – 991. IOS Press, 2021.

[5] H. Bagheri, E. Kang, S. Malek, and D. Jackson. Detection of design flaws in the android permission protocol through bounded verification. In *International Symposium on Formal Methods (FM)*, pages 73–89, 2015.

[6] N. Bjørner, A. Phan, and L. Fleckenstein. *νz* - an optimizing SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 194–199, 2015.

[7] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Surveys in computer science. Springer, 1990.

[8] A. Cunha, N. Macedo, and T. Guimarães. Target oriented relational model finding. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 17–31, 2014.

[9] G. Dennis, F. S. Chang, and D. Jackson. Modular verification of code with SAT. In *ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 109–120, 2006.

[10] Y. Feng, O. Bastani, R. Martins, I. Dillig, and S. Anand. Automated synthesis of semantic malware signatures using maximum satisfiability. In *Proceedings of the Annual Network and Distributed System Security Symposium*. The Internet Society, 2017.

[11] A. Graça, J. Marques-Silva, and I. Lynce. Haplotype inference using propositional satisfiability. In *Mathematical Approaches to Polymer Sequence Analysis and Related Problems*, pages 127–147. Springer, 2011.

[12] Gurobi Optimization. Gurobi optimizer reference manual, 2021.

[13] M. P. Herlihy and J. M. Wing. Specifying graceful degradation. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):93–104, 1991.

[14] International Business Machines Corporation (IBM). V12. 1: User's manual for CPLEX, 2009.

[15] D. Jackson. Automating first-order relational logic. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-First Century Applications*, SIGSOFT '00/FSE-8, page 130–139, New York, NY, USA, 2000. Association for Computing Machinery.

[16] D. Jackson. *Software Abstractions: Logic, language, and analysis*. MIT Press, 2006.

[17] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 14–25, 2000.

[18] E. Kang, A. Milicevic, and D. Jackson. Multi-representational security analysis. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 181–192, 2016.

[19] S. Khurshid and D. Marinov. Testera: Specification-based testing of java programs using SAT. *Autom. Softw. Eng.*, 11(4):403–434, 2004.

[20] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[21] Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik. Symbolic optimization with SMT solvers. In S. Jagannathan and P. Sewell, editors, *Symposium on Principles of Programming Languages (POPL)*, pages 607–618, 2014.

[22] X. Liao, H. Zhang, M. Koshimura, R. Huang, and W. Yu. Maximum satisfiability formulation for optimal scheduling in overloaded real-time systems. In A. C. Nayak and A. Sharma, editors, *PRICAI 2019: Trends in Artificial Intelligence*, pages 618–631, Cham, 2019. Springer International Publishing.

[23] V. Lifschitz. *Answer Set Programming*. Springer, 2019.

[24] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[25] D. Lustig, M. Pellauer, and M. Martonosi. Pipecheck: Specifying and verifying microarchitectural enforcement of memory consistency models. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 635–646. IEEE, 2014.

[26] I. Lynce, V. M. Manquinho, and R. Martins. Parallel maximum satisfiability. In Y. Hamadi and L. Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 61–99. Springer, 2018.

[27] F. A. Maldonado-Lopez, J. Chavarriaga, and Y. Donoso. Detecting network policy conflicts using alloy. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ)*, pages 314–317, 2014.

[28] D. Maliotov and K. S. Meel. MLIC: A MaxSAT-based framework for learning interpretable classification rules. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 312–327. Springer, 2018.

[29] J. Marques-Silva, J. Argelich, A. Graça, and I. Lynce. Boolean lexicographic optimization: algorithms & applications. *Annals of Mathematics and Artificial Intelligence*, 62(3-4):317–343, 2011.

[30] R. Martins, V. Manquinho, and I. Lynce. Community-based partitioning for MaxSAT solving. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 182–191. Springer, 2013.

[31] R. Martins, V. Manquinho, and I. Lynce. Open-WBO: A modular MaxSAT solver. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 438–445. Springer, 2014.

[32] R. Martins and J. Sherry. Lisbon Wedding: Seating arrangements using MaxSAT. *MaxSAT Evaluation 2017 : Solver and Benchmark Descriptions*, B-2017-2:25, 2017.

[33] A. Milicevic, J. P. Near, E. Kang, and D. Jackson. Alloy*: A general-purpose higher-order relational constraint solver. In *International Conference on Software Engineering, (ICSE)*, pages 609–619, 2015.

[34] S. Narain. Network configuration management via model finding. In *USENIX Conference on Systems Administration (LISA)*, pages 155–168, 2005.

[35] T. Nelson, S. Saghafi, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Aluminum: principled scenario exploration through minimality. In *International Conference on Software Engineering (ICSE)*, pages 232–241, 2013.

[36] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming CP 2007*, pages 529–543, 2007.

[37] M. Neves, R. Martins, M. Janota, I. Lynce, and V. M. Manquinho. Exploiting resolution-based representations for MaxSAT solving. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 272–286. Springer, 2015.

[38] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[39] R. Sebastiani and P. Trentin. Optimathsat: A tool for optimization modulo theories. *J. Autom. Reason.*, 64(3):423–460, 2020.

[40] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In J. P. Shen and M. Martonosi, editors, *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415, 2006.

[41] C. Stevens and H. Bagheri. Reducing run-time adaptation space via analysis of possible utility bounds. In *International Conference on Software Engineering (ICSE)*, pages 1522–1534, 2020.

[42] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[43] C. Trippel, D. Lustig, and M. Martonosi. CheckMate: Automated synthesis of hardware exploits and security litmus tests. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2018-Octob:947–960, 2018.

[44] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[45] P. Velasco-Elizondo, V. Dwivedi, D. Garlan, B. Schmerl, and J. M. Fernandes. Resolving data mismatches in end-user compositions. In Y. Dittrich, M. Burnett, A. Mørch, and D. Redmiles, editors, *End-User Development*, pages 120–136, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[46] L. Wang, S. Jajodia, A. Singhal, P. Cheng, and S. Noel. k-zero day safety: A network security metric for measuring the risk of unknown vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 11(1):30–44, 2013.

[47] P. Zave. Reasoning about identifier spaces: How to make chord correct. *IEEE Trans. Software Eng.*, 43(12):1144–1156, 2017.

[48] L. Zhang and F. Bacchus. MAXSAT heuristics for cost optimal planning. In J. Hoffmann and B. Selman, editors, *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI Press, 2012.