Optimizations for Compiling Declarative Models into Boolean Formulas

Darko Marinov[1], Sarfraz Khurshid[2], Suhabe Bugrara[3], Lintao Zhang[4], and Martin Rinard[3]

[1] Department of Computer Science, University of Illinois, Urbana-Champaign, IL 61801, USA
[2] Dept. of Electrical & Computer Engineering, University of Texas, Austin, TX 78712, USA
[3] MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA 02139, USA
[4] Microsoft Research Silicon Valley Lab, Mountain View, CA 94043, USA
marinov@cs.uiuc.edu
khurshid@ece.utexas.edu
{sbugrara, rinard}@csail.mit.edu
lintaoz@microsoft.com

Abstract. Advances in SAT solver technology have enabled many automated analysis and reasoning tools to reduce their input problem to a SAT problem, and then to use an efficient SAT solver to solve the underlying analysis or reasoning problem. The solving time for SAT solvers can vary substantially for semantically identical SAT problems depending on how the problem is expressed. This property motivates the development of new optimization techniques whose goal is to produce more efficiently solvable SAT problems, thereby improving the overall performance of the analysis or reasoning tool.

This paper presents our experience using several mechanical techniques that enable the Alloy Analyzer to generate optimized SAT formulas from first-order logic formulas. These techniques are inspired by similar techniques from the field of optimizing compilers, suggesting the potential presence of underlying connections between optimization problems from two very different domains. Our experimental results show that our techniques can deliver substantial performance improvement results—in some cases, they reduce the solving time by an order of magnitude.

1 Introduction

In recent years, dramatic advances in the capabilities of SAT solvers have made them an attractive target for model checkers and other automated reasoning systems [13, 3, 14, 24]. The standard approach is to automatically generate a SAT problem, invoke the SAT solver to produce a solution, then transform the SAT solution back into a solution for the initial problem.

The efficiency of this approach depends largely on the efficiency of the SAT solver. However, the SAT solving times can vary by significant factors for semantically identical SAT problems depending on the precise formulation of the SAT problem [7]. This suggests that appropriately optimizing the generated SAT problems may significantly improve the overall performance of systems that use SAT solvers.

F. Bacchus and T. Walsh (Eds.): SAT 2005, LNCS 3569, pp. 187–202, 2005.

This paper presents our experience with several techniques that are designed to optimize the SAT problems generated by the Alloy Analyzer [10, 12]—a tool that analyzes declarative specifications by translating them into boolean formulas. Our techniques transform a given Alloy specification into an equivalent Alloy specification that induces faster analysis. These transformations are mechanical and therefore suitable for inclusion in an automatic SAT problem generator. Our experimental results show that our techniques can substantially reduce the solving time for a set of benchmark problems from the standard Alloy distribution and previous case studies. The speed-ups range from a low of 1.04X to a high of 14.52X (it is 14.52 times faster to translate and solve the optimized version than the unoptimized version).

Conceptually, our set of transformations draws heavily on techniques from the field of optimizing compilers [2]. We have developed analogs of standard optimizations such as loop unrolling, loop fission and fusion, loop-invariant code motion, common subexpression elimination, constant propagation, and algebraic simplifications. Like their standard compiler optimization counterparts, the goal of these transformations is to reduce the amount of work that the execution engine (the microprocessor or SAT solver) must perform.

We investigated the effect of applying a range of transformations on a suite of benchmark problems. The initial results showed that the most effective transformations were those that focus on formulas that were universally quantified and expressions that use transitive closure. We have implemented these transformations in our prototype tool. The tool allows the user to select a sequence of optimizations which it then applies fully automatically on the given model.

The ostensible purpose of our transformations is to provide an Alloy solver with improved performance. To this end, our transformations focus on specific Alloy constructs (such as transitive closure and quantified formulas) that are responsible for the vast majority of the SAT solving time. (Note the recurring analogy with traditional compiler optimizations, which often focus on loops because programs tend to spend much of their time in loops.) Despite our focus on Alloy constructs, we expect many of the general patterns we exploit in Alloy problems to show up in other domains, which makes our techniques ripe for incorporation into a range of systems that automatically generate SAT problems.

One intriguing aspect of our system is that SAT solvers are a substantially more complex compilation target than the microprocessors that are the traditional compilation targets. In particular, microprocessors often have an available performance model that the compiler writer can use to guide the optimization decisions. The performance of SAT solvers, in contrast, is much less well understood. There are two heuristics in the field: reducing the number of variables tends to reduce the solving time (presumably because it reduces the search space that the SAT solver must explore) and increasing the number of constraints also tends to reduce the solving time (again because it reduces the search space). These are just heuristics so do not hold always [7]. Interestingly enough, one of our performance-improving transformations (constant subexpression elimination) also increases the number of variables. This fact illustrates the need to explore a variety of transformations, not just transformations that are consistent with the current understanding of the performance of SAT solvers.

It is worth emphasizing that the optimizations we propose translate Alloy models into equivalent Alloy models that enable faster analysis—we do not present how to optimize SAT solvers in general. Our approach is inspired by compiler optimizations and can be extended to various other SAT-based techniques, such as bounded model checking [4, 24]. In fact, our approach is not limited to SAT. Any technique that translates a more complicated logic to a simpler logic for reasoning purposes has the potential to benefit from our compiler analogy. As the use of decision procedures becomes more and more popular, we hope this can get more attention from the research community.

This paper makes the following contributions:

- **Optimization Concept:** It introduces the concept of mechanically transforming problems to improve the solving time of the resulting automatically generated SAT problems.
- **Transformations:** It presents a precisely defined set of transformations that optimize the SAT problems that the Alloy Analyzer generates.
- **Implementation:** It presents our implementation, which allows the user to select a sequence of transformations that are then applied fully automatically.
- **Experimental Results:** It presents experimental results that characterize the effectiveness of our optimized SAT problem generator. The results show that our techniques can substantially decrease the solving time for the generated SAT problem.

2 Example

This section illustrates some performance gains that compiler optimizations can provide for SAT-based constraint solving. We present a simple example that models in Alloy a singly-linked acyclic list. We formalize three different constraints that specify acyclicity. We then use the Alloy Analyzer to check that these formulations are equivalent. We re-write the model by applying our optimizations and illustrate how they enable faster analysis. We describe essentials of Alloy as we introduce them. Section 3.1 describes Alloy in more detail.

The following Alloy code declares a list:

```
sig List {
  header: option Entry }

sig Entry {
  next: option Entry }
```

Each list has a `header` entry, and each entry has a `next` entry. The keyword `sig` introduces *signatures*, i.e., basic sets/types. *Fields* in signature declarations introduce relations. The field `header`, for example, introduces the relation `header: List -> Entry`. Further, the keyword `option` constrains this relation to be a partial function.

The following three Alloy *functions* use various Alloy constructs to state the acyclicity constraint:

```
fun Acyclic1(l: List) {
  all e: l.header.*next | e !in e.^next }

fun Acyclic2(l: List) {
  no l.header || (some e: l.header.*next | no e.next) }

fun Acyclic3(l: List) {
  no e: l.header.*next | e -> e in ^next }
```

The *dot operator* (‘.’) represents relational join. Alloy allows intuitive *path expressions* that use transitive closure (‘^’) and reflexive transitive closure (‘*’). For example, `l.header.*next` denotes the set of all entries reachable along the `next` field from the `header` entry of the list `l`.

`Acyclic1` uses universal quantification (‘`all`’), negation (‘`!`’), and set membership (‘`in`’) to state that it is not possible to start a traversal from any list entry and follow one or more fields to get back to the same entry. `Acyclic2` uses existential quantification (‘`some`’) and states that the list is either empty or contains an entry that is reachable from the header and has no next entry. `Acyclic3` uses cross product (‘`->`’) to state that the transitive closure of `next` does not contain any self-loops.

To check equivalence of these definitions, we use the following *assertion*:

```
assert Equiv {
  all l: List {
    Acyclic1(l) => Acyclic2(l)
    Acyclic2(l) => Acyclic3(l)
    Acyclic3(l) => Acyclic1(l) } }
check Equiv for 6
```

The `check` command instructs the Alloy Analyzer to try to generate a counterexample to the given assertion, i.e., a list `l` that does not satisfy the (implicit conjunction of) implication formulas. The analyzer performs the analysis within the given *scope*—a bound on the universe of discourse. In this example, a scope of 6 states that the number of elements in each basic set (known also as “atoms in the signature”) should be at most 6, i.e., at most 6 `Entry` and `Object` elements.

The analyzer takes 199.91 seconds to check `Equiv` and reports that no counterexample exists for this assertion. We next illustrate how compiler optimizations could improve the analyzer’s performance.

Our first optimization is inspired by common subexpression elimination (CSE) [2]. Note that the transitive closure operator (‘^’) appears in the formula body of `Acyclic1` and also of `Acyclic2`. A naive translation of our Alloy assertion to a boolean formula would translate each of these expressions independently. However, they represent the same value and we can apply CSE.

The CSE transformation *adds* a new state component in our model: it introduces a new field, `nextPlus`, in the declaration of `Entry` and constrains it to be the transitive closure of `next`.

```
sig Entry {
  next: option Entry,
  nextPlus: set Entry }

fact { nextPlus = ^next }
```

Each *fact* specifies a constraint that all solutions to the model must satisfy. We next replace each occurrence of `^next` in the functions with `nextPlus`. The analyzer now takes 138.03 seconds to check `Equiv` and (as before) reports no counterexample. It is worth pointing out that by applying an optimization that adds a new state component and is therefore seemingly counter-intuitive (since it increases the size of the underlying state space), we have achieved a reduction in the solving time.

The above optimization factors out the transitive closure operation. We can, in fact, represent the closure directly by its definition; for our example, in the scope of 6, we can *unroll* the closure with respect to this scope and replace the above fact with:

```
fact { nextPlus = next + next.next + next.next.next + next.next.next.next +
                  next.next.next.next.next + next.next.next.next.next.next }
```

where '+' denotes set union in Alloy.

For a given scope, we can also unroll some quantified formulas, besides unrolling the definition of the transitive closure. In particular, we can unroll the formulas, such as `all e: l.header.*next | e !in e.^next`, where the quantified variable ranges over a path expression. (Section 4.1 presents the details of unrolling.) We next apply the loop unrolling transformation, which automatically unrolls all three definitions of acyclicity with respect to scope 6, and check the assertion. The analyzer now takes only 15.84 seconds to check `Equiv` and (as before) reports no counterexample. It is worth pointing out that as a result of this series of optimizations, the Alloy-to-CNF compilation time has gone up from 0.75 seconds to 1.13 seconds; however, in the same time, the SAT-solving time has gone down from 199.16 seconds to just 14.71 seconds.

In summary, even this simple example shows a good potential for optimization: by applying common-subexpression elimination and loop unrolling, we obtain a reduction of more than 92% in the total time to check the formula, i.e., a 12X speedup!

3 Background

This section gives a brief overview of Alloy and SAT technology; following the compiler analogy, Alloy is our input language, and SAT is the target language. We also discuss some of the optimizations that already exist for generating boolean formulas that are likely to induce efficient solving.

3.1 Alloy

Alloy [10] is a first-order declarative language based on sets and relations. The Alloy Analyzer [12] is a tool for automatically analyzing models written in Alloy. The analyzer translates Alloy models into boolean formulas and uses off-the-shelf SAT technology to solve the formulas. Following the compiler analogy, the analyzer consists of the following: a front-end that parses Alloy models into an intermediate representation (IR), a set of optimizations on this IR, and a back-end that translates IR into boolean formulas.

Each Alloy model consists of data (i.e., several sets and relations), several facts (i.e., formulas that put constraints on the data) and an assertion (i.e., a formula to check on the data). These formulas can be structured using functions (i.e., parameterized formulas that can be invoked elsewhere), which the analyzer inlines into the facts and the assertion. Additionally, each analysis specifies a scope (i.e., a bound on the size of basic sets within which to check the formulas). The analyzer translates a conjunction of all facts and the negation of the assertion into a boolean formula such that the boolean formula has a solution iff there are some sets and relations that satisfy all the fact and the negation of the assertion (thus providing a counterexample for the assertion).

Alloy is a relational language; every expression in Alloy denotes a relation (or a set in the case of a relation of arity one). Even the scalars are represented as singleton sets. More details of the Alloy language are available elsewhere [10].

3.2 SAT

Given a propositional formula over a set of boolean variables, the boolean Satisfiability Problem (SAT) asks whether there exists a variable assignment that makes the formula evaluate to true. SAT is a classical NP-Complete problem; therefore, it is unlikely that there is a polynomial algorithm for solving the SAT problem. However, due to its practical importance in areas such as theorem proving, formal verification, and AI planning, much research effort has been put into developing efficient algorithms for solving SAT problems. Although in the worst case these algorithms require exponential time, in practice current state-of-the-art SAT solvers can often determine the satisfiability of boolean formulas with tens of thousands of variables in a reasonable amount of time [27].

Modern SAT solvers determine the satisfiability of a formula by systematically searching the entire boolean space of the formula. They typically require the input formula to be in the Conjunctive Normal Form (CNF), i.e., a conjunction of clauses, where each clause is a disjunction of literals. A literal is either a positive or negative occurrence of a boolean variable. Some recent SAT solvers can operate without the CNF requirement [8], but the Alloy Analyzer translates all formulas into CNF.

Applications that use SAT as the reasoning engine often look like compilers. They take the domain-specific description of the problem as input and translate it into a boolean formula or a sequence of formulas. The formulas are then given to a SAT solver to determine their satisfiability, and the results are fed back to the application to extract meaningful information for the user to understand. For some applications such as circuit verification [17], translation to SAT is straightforward. On the other hand, for applications such as checking the Alloy models, the translation is not trivial [12]. In these applications, different translations of a high-level description may greatly influence the time it takes to solve the output boolean formula.

3.3 Existing Optimizations

There are many heuristics and optimizations developed for making SAT solving efficient in real-world applications. For example, there are several SAT pre-processors [16, 23] that can take a SAT instance and transform it into another form that the SAT solver can easier solve. These transformations happen at the propositional formula level, i.e. after the actual translation has been done. These low-level formulas make it harder or impossible to detect optimizations that can be applied at the higher-level, before the translation. Some researchers tried to influence the output with different translations. For example, Velev [25, 26] tried different encodings of circuit structures for efficient microprocessor verification. Seshia et al. [20] evaluated different encodings for deciding separation logic and proposed a hybrid approach.

Translation has also been studied in the context of Alloy, and the analyzer includes several optimizing translations. Symmetry breaking [21] conjoins new boolean constrains with the input formula to direct the SAT solver's search on non-isomorphic instances. Efficient encoding of (partial) functions [24] replaces the general translation for Alloy relations with a specialized, tighter translation that targets partial functions. Type-based reduction of the number of variables [6] introduces subtyping in Alloy and assigns individual scopes to the subtypes, which partition their supertype, to reduce the

overall search space. Narain [19] provides three guidelines for *manual* rewriting of Alloy models to obtain a more efficient analysis but does not consider automation of these rewritings.

A key difference of our work from the previous work is that our optimizations are automatic and at the level of Alloy: they transform an Alloy model to an equivalent Alloy model (without requiring a new type system) that is likely to induce faster solving. This allows our optimizations to be employed in conjunction with existing optimizations in a seamless fashion, thereby increasing the overall performance gain.

4 Optimizations

This section presents some of the optimizations that are applicable to Alloy models. These optimizations are either inspired by or lifted directly from the optimizing compiler literature. We present the following optimizations:

- loop-invariant hoisting (LIH)
- loop fusion (LFU)
- loop unrolling (LUR)
- common subexpression elimination (CSE)
- algebraic transformations (AT)
- partial evaluation (PE)

4.1 Loop Optimizations

Quantifiers (and transitive closure) in Alloy serve a similar purpose as loops in imperative programs. We therefore apply several standard loop optimizations to Alloy formulas that use quantifiers. Each quantified formula introduces a quantified variable (analogous to a loop-index variable) that ranges over some set (analogous to the set of values for the loop index) and has a body formula (analogous to the body of the loop).

Loop-Invariant Hoisting (LIH). This optimization first identifies those subformulas in a body of a given quantified formula that do not depend on the quantified variables and then moves these subformulas out of the quantifier scope. For example, in the Alloy formula `all n: Node | (some Node || n !in n.^next)`, the subformula `some Node` is independent of the variable `n`. Moving the subformula outside of the quantification gives the formula `some Node || all n: Node | n !in n.^next`. This optimization does not give a substantial speedup if the hoisted subformula is much simpler than the rest of the body.

Loop Fusion (LFU). Just as the standard loop fission and fusion optimizations split or merge loops (to make them parallelizable or eliminate the overhead of checking branches), we can split or merge several Alloy quantifiers that range over the same set. For example, the Alloy formulas `(all n: N | F1(n)) && (all n: N | F2(n))` and `(all n: N | (F1(n) && F2(n)))` are semantically equivalent. It turns out,

however, that there can be a significant difference in the solving time for these formulas. We have thus implemented loop fusion in the Alloy Analyzer.

Loop Unrolling (LUR). This optimization targets quantified Alloy formulas where the quantified variables range over some path expression. For instance, the example section shows the formula `all e: l.header.*next | e !in e.^next`. We illustrate how such formulas can be completely unrolled for all elements from `l.header.*next`.

Consider the general formula `all` x`:` E `|` F, where E is an expression of some basic type/set α, and F is the quantified formula that depends on the variable x. An operational reading of this formula would first evaluate E to some set $\{a_1, \ldots, a_k\} \subseteq \alpha$ and then check F for each a_i substituted for x, i.e., $F[a_i/x]$. However, Alloy formulas need to be translated into declarative SAT, so the translation does not follow the operational view. Instead, the analyzer must translate this formula under the assumption that E can evaluate to any subset of α, i.e., in the general form $\bigwedge_{a \in \alpha} a \in E \Rightarrow F[a/x]$. The analyzer does not use the general form directly, but applies several optimizations [9].

Consider the special case where the analyzer can statically enumerate the elements $\{a_1, \ldots, a_k\}$ that E evaluates to in some scope. It can then translate `all` x`:` E `|` F into the Alloy formula F`(`a_1`) && ... &&` F`(`a_k`)`. Universally quantified formulas translate into a conjunction; existentially quantified formulas analogously translate into a disjunction: `some` x`:` E `|` F translates into F`(`a_1`) || ... ||` F`(`a_k`)`.

A common pattern in Alloy formulas is to quantify over path expressions. For such expressions, we cannot enumerate the elements in the set, but we can represent them using the equality on reflexive transitive closure, expressed in Alloy for the example `next` relation: `*next = iden[Node] + next + next.next + ... +` $\texttt{next}^{s-1}$, where s is the scope for `Node`. (A similar equality also holds for transitive closure: `^next = next + next.next + ... +` $\texttt{next}^{s}$.)

The relations in the path expressions are often (partial) functions, e.g., when modeling fields of object-oriented programs [24]. If `next` is a partial function, `n.next` denotes a set with at most one element when `n` denotes a set with at most one element. The Alloy formula `some n` holds iff the set denoted by `n` is not empty. In the case of `n.next`, `some n.next` holds iff `n` denotes a singleton set, which in Alloy represents a scalar. We can therefore translate `all n: l.header.*next | F(n)` into the following conjunction:

```
some l.header => F(l.header)
some l.header.next => F(l.header.next)
...
some l.header.next^(s-1) => F(l.header.next^(s-1))
```

where the guard `some n` ensures that we preserve the semantics of Alloy in this translation. In the general case, `next` could be an arbitrary relation, and the translation would need to use `one n` (instead of `some n`) to specify that `n` has exactly one element. However, the translations of `some n` and `one n` into boolean formulas differ significantly, and the translation of `some n` is much more efficient than `one n`.

Similarly, we can translate `some n: l.header.*next | F(n)` into the disjunction of `(some l.header.next`^{i} `&& F(l.header.next`^{i}`))` for $0 \leq i \leq s-1$.

To illustrate an application of loop unrolling, let us reconsider the list declaration from Section 2 and checking the equivalence of formulas for acyclicity. We can apply loop-unrolling to rewrite `Acyclic1` as:

```
fun Acyclic1(l: List) { // all n: l.header.*next | n !in n.^next
  some l.header => l.header !in l.header.^next
  some l.header.next => l.header.next !in l.header.next.^next
  some l.header.next.next => l.header.next.next !in l.header.next.next.^next
  some l.header.next.next.next =>
    l.header.next.next.next !in l.header.next.next.next.^next
  some l.header.next.next.next.next =>
    l.header.next.next.next.next !in l.header.next.next.next.next.^next
  some l.header.next.next.next.next.next =>
    l.header.next.next.next.next.next !in l.header.next.next.next.next.next.^next }
```

Our implementation of LUR automatically determines whether it is legal to apply LUR to each loop and applies LUR whenever legal.

4.2 Non-loop Optimizations

We next present the optimizations that do not target quantified Alloy expressions.

Common Subexpression Elimination (CSE). This is a standard compiler optimization that replaces the evaluation of N identical expressions with (1) one evaluation whose result is stored and (2) $N - 1$ reads of the stored result. This optimization effectively trades the space that stores the result for the time to recompute the expressions. Alloy has no operational computation, but it is still desirable to save and reuse results. Others have observed this effect, and the back-end of the Alloy Analyzer actually implements a sophisticated optimization that detects and exploits sharing of subformulas during the translation of quantified Alloy formulas into boolean formulas [22]. However, our results (Section 5) show that a substantial amount of sharing can be detected even in the front-end, at the level of Alloy, before the translation to SAT.[1]

Algebraic Transformations (AT). Alloy expressions offer numerous opportunities for applying algebraic transformations, i.e., using the equational rules of the relational algebra that underlies the Alloy semantics to replace selected expressions with equivalent rewritten expressions. For example, one such rule is that the transitive reflexive closure is idempotent: `**next = *next` for all relations `next`. Our anecdotal experience indicates that Alloy users sometimes write (typically by making a typographic mistake) such expressions that can be significantly optimized. For example, replacing `**next` with `*next` in the formula `**next = iden[Node] + ^next` reduces the checking time by 2X (in scope 7).

Algebraic transformations also apply to the models discussed in Section 5. Moreover, even the transformations that produce expressions of similar complexity, and thus do not look profitable by themselves, can enable the profitable application of other optimizations. For example, one of the algebraic rules is that transpose and reflexive transitive closure commute for example, `*~next = ~*next`. Our implementation applies this rule as a rewrite from left to right to enable CSE to detect more common subexpressions that contain transitive closures.

Partial Evaluation (PE). Partial evaluation is a standard optimization technique for evaluating expressions at compile-time. It generalizes constant folding (which evaluates

[1] We have recently found that we can obtain some of the CSE speed-up by adding to the Alloy model new fields that trigger the existing sharing, without performing the whole CSE.

only basic operations on constant arguments). We illustrate partial evaluation for Alloy models on the following function adapted from [11]:

```
fun modifies(pre, post: State, mods: set Ref) {
  (all r: Ref - mods | F2) &&
  (all r: mods | F1) &&
  F3 }
```

This function is a part of the model that specifies state transitions. Specifically, `modifies` expresses the general constraint that a transition from the `pre` state to the `post` state can modify only references in the set `mods`. Each specific transition instantiates the generic function with the appropriate value for `mods`. Transitions that do not modify any reference thus instantiate `mods` with the empty set: `modifies(s, s', none[Ref])`, where `none[Ref]` represents the empty set.

This constant argument offers the opportunity to partially evaluate `modifies`. We can clone `modifies` and specialize the copy for the empty set: `Ref - mods` evaluates to `Ref`, and the whole second quantification evaluates to `true`:

```
fun modifiesEmpty(pre, post: State) {
  (all r: Ref | F2) &&
  F3 }
```

We then replace the calls involving the empty set with `modifiesEmpty(s, s')`. Standard partial evaluation usually involves cloning of functions. However, the translation of Alloy already inlines all functions[2] so we can instead specialize the inlined formulas.

Although this optimization in theory bears great potential, we found that it is rarely applicable in Alloy models, so we did not implement it. Our experiment with manual application of PE showed a speed-up of 1.16X for the analysis of the model Views [11].

5 Experiments

This section presents the performance results for several Alloy models taken from the Alloy distribution (`http://alloy.mit.edu/`) and from previous case studies that used Alloy models. We evaluate the effectiveness of the optimizations by using our implemented Alloy transformation system to automatically apply the optimizations to these models. We report the total time that the Alloy Analyzer takes to compile and solve the original and optimized models.

We have implemented our optimizations by changing the Java source code of the Alloy Analyzer version 2.0. (The most recent version is 3.0; we have modified 2.0 because 3.0 was unstable when we had started implementing the optimizations.)

We performed all experiments on a Linux machine with a 1.8 GHz Pentium 4 processor using Sun's Java 2 SDK 1.3.1 JVM. We used our modified Alloy Analyzer and the mChaff [18] SAT solver.

[2] Alloy does not support recursive functions, because it needs to generate SAT formulas. Hence, non-termination of inlining or partial evaluation is not an issue.

Table 1. Benchmark models

model	description	solving (#solutions)	optimizations	#ncnb
List	from the example section	satisfiability (0)	CSE, LUR	21
Types	soundness of Java type system	satisfiability (0)	CSE	106
INS	dynamic networks	satisfiability (0)	AT, CSE, LUR	117
ProtonId	patient identity for proton machine	satisfiability (0)	AT, LFU, CSE	164
Life	game of life	satisfiability (1)	AT, LFU, CSE	88
NetConf	network configuration	satisfiability (1)	LUR	192
BinTree	binary search trees	enumeration (1430)	LFU, CSE	62
RedBlack	red-black trees	enumeration (35)	LFU, CSE, LUR	115

5.1 Benchmarks

Table 1 lists the benchmark models that we use to evaluate our optimizations:

- List models singly linked-lists as described in Section 2. It checks the equivalence of the three definitions of acyclicity.
- Types [6] models a part of the Java type system and checks its soundness by checking a subject reduction theorem: statement executions preserve the correspondence between run-time types and the compile-time types.
- INS [14] models the key data structures and algorithms of the Intentional Naming System [1], a resource discovery and service location architecture for mobile networks. It checks partial correctness of these algorithms.
- ProtonId [6] models the tracking of patients in a (proton) radiotherapy facility. It checks that patients are correctly identified.
- Life [22] is a model of Conway's game of life. It simulates multi-step executions of the game on a variable size grid.
- NetConf [19] is a complex Alloy model for network configuration management. It builds network configurations that satisfy given management policies.
- BinTree models simple binary search trees [5, 15].
- RedBlack models red-black trees [5, 15], which implement balanced binary search trees. The last two models enumerate all appropriate trees within a given size.

For each model, we list whether we are just checking satisfiability (i.e., looking for one solution or showing that none exists) or enumerating all possible solutions (which is, for instance, useful in testing [15]). We also list the optimizations that are applied to get from an original model to an optimized version. We finally show the number of non-comment non-blank lines in each original model. This is a simple illustration of the model size and not a measure of the model complexity. (It may be a measure of the complexity of *developing* the model but not *solving* the model.)

5.2 Results

Table 2 summarizes the results that we obtain by applying the optimizations to the different models. For each model, we present corresponding sets of data for both the original (unoptimized) version and the optimized version. We report the number of

Table 2. Comparison of analyses for models with and without optimizations

	unoptimized				optimized				
model	indep. vars	total vars	clauses	time	indep. vars	total vars	clauses	time	speed-up
List	150	3686	19113	199.92	204	4068	20075	15.85	12.61
Types	1888	82713	381816	4.30	2088	77672	357334	3.49	1.23
INS	1295	75725	728022	1208.47	1511	76176	727564	83.21	14.52
ProtonId	1370	37566	237175	3.97	1514	37938	238227	3.83	1.04
Life	351	38312	188423	51.71	927	35844	186088	27.61	1.87
NetConf	770	44848	415018	10.36	770	44700	414574	8.19	1.26
BinTree	146	4504	20064	39.93	274	4056	19024	28.39	1.41
RedBlack	263	7662	34472	136.22	361	7459	34230	27.30	4.99

independent variables in the translated boolean formula, the total number of variables and clauses in the CNF formula, and the time (in seconds) it takes to check (i.e., translate and solve) the formula. We also present the speed-up resulting from the optimizations.

The models for List, Types, INS, and ProtonId all check an assertion; the scopes were set to 6, 12, 6, and 6, respectively. The analyzer does not find any counterexample. For Life and NetConf, the analyzer generates a solution (a simulation of the game and an appropriate network configuration, respectively). We present the time to find the first solution in both the unoptimized and the optimized versions. We used a scope of 12 for Life and a varying scope for different network elements in NetConf (from 4 security tunnels to 6 routers to 12 interfaces). For BinTree and RedBlack, we present the times the analyzer takes to enumerate all corresponding trees with 8 and 7 nodes, respectively. For Types, INS, and Life (all of which are available as models in the Alloy distribution), we ran the analyzer using the maximum scopes for which those models were previously analyzed (as stated in the distribution files). For other models (which are not yet a part of the standard Alloy distribution), we chose sizes that are representative from previous case studies [6, 15, 19]. The performance results indicate the benefits of applying optimizations, both for determining the satisfiability of boolean formulas and for enumerating all solutions.

In all cases the optimizations produce a performance improvement. The speed-up varies from 1.04X to 14.52X. We point out that the optimizations do not necessarily decrease the number of variables (whether independent or total) or increase the number of clauses. In fact, for INS (for example) the optimizations both increase the number of (independent and total) variables and decrease the number of clauses. Nevertheless, the benchmark still exhibits a more than 10X speed-up.

Note that we obtain the maximum speed-up on one of the most complex models (INS). INS models recursive algorithms that were implemented in Java [14], which requires non-trivial fixed-point computations to be expressed in Alloy. Several researchers, including the first two authors of this paper, analyzed this model in various projects [14, 6, 22]. During these projects, the model was optimized (by both manual rewriting and some automatically applied SAT-generation optimizations) to reduce the solving time. Based on our previous experience with this model, it came as a great surprise to us that it was possible to obtain such a substantial speed-up. We hypothesize that more complex models would indeed benefit the most from systematic compiler optimizations, in

part because the complexity of the model inhibits the manual application of large-scale transformations such as loop unrolling.

We next discuss how different optimizations contribute to the speed-up. For Types and NetConf, the entire speed-up comes from a single optimization—CSE for Types and LUR for NetConf. For List, the overall speed-up is split between CSE and LUR as 33.6% for CSE and 66.4%, for LUR. For INS, the overall speed-up is split between AT, CSE, and LUR as 2.8%, 98.4%, and -1.2%, respectively. In other words, applying all optimizations (AT, CSE, LUR) to INS does not give the best speed-up; namely, applying just AT and CSE, without LUR, gives 17.34X speed-up over the unoptimized version! This problem is well-known for the compiler optimizations: the "optimizations" are heuristics that improve the code in most cases but can also degrade the code in some cases. Choosing the best set of optimizations to apply (and the best order in which to apply them) is an open research problem in compiler community. We plan to investigate this problem in the context of Alloy.

Our experiments include the change in the compilation time that results from the automatic application of the optimizations. Specifically, the time that we present is the sum of execution times of three parts: (1) the compilation time that includes the time required to parse the Alloy model and apply the optimizations, (2) the translation time required to translate the (optimized) model into a boolean formula, and (3) the SAT solving time. For the original models, the compilation time ranges from 0.39 sec (for List) to 2.55 sec (for RedBlack), and the translation time ranges from 0.35 sec (for List) to 1.52 sec (for Types). With all our optimizations, the compilation time increases by at most 0.24 sec (for List), and the translation time increases by at most 0.37 sec (for NetConf).[3] Some optimizations even decrease the total compilation time as also observed in other compilation projects [22]. Our results indicate that the compilation and translation times in the experiments were negligible in comparison with the SAT solving time, so even a small increase in the compilation and translation times can be easily outweighed by a decrease in the SAT solving time.

5.3 Discussion

The experiments indicate that the most beneficial optimizations are common subexpression elimination and loop unrolling. The most commonly factored out expressions were those that were based on transitive closure. Indeed, loop unrollings also involved formulas that used (quantified expressions with) transitive closure. Alloy specifications often use transitive closure and quantifiers. These constructs tend to be heavily used since they are the source of much of Alloy's expressive power (as compared to first-order logic without transitive closure or relational queries without quantification) [10]. These Alloy constructs are also analogous to loops in code. Solving formulas with these constructs, however, is expensive. Our experience indicates that, in much the same way that the optimizations in a regular compiler focus on the most expensive parts of the

[3] Due to our unfamiliarity with the details of the analyzer, our modified version does not translate the internal representation of the model after optimizing it. Instead, our version parses the model, optimizes it, then pretty-prints it and parses it again. Our times do not include pretty-printing and reparsing, since an actual implementation would not have these two steps.

code (i.e., the inner loops), the Alloy compilation should focus on the most expensive constructs (i.e., quantified formulas and formulas involving transitive closure).

It is worth pointing out that we use the analyzer to check the partial correctness of our optimizations. It is conceivable that our optimizations, because of a compiler bug, could incorrectly create optimized models that are not equivalent to the original models. To check the (partial) correctness, we employ two techniques: (1) we check the equivalence of the original and optimized formulas (using the analyzer) and (2) we enumerate all solutions to the original formula and all solutions to the optimized formulas and compare the number of solutions. Even though the second technique does not, in general, check equivalence of the formulas, it increases our confidence in the correctness of the translation and tends to complete more quickly than the first technique. We therefore use it when the first technique times out. Note that determining the equivalence of boolean formulas, which are our compilation target, is decidable. In regular compilers, on the other hand, determining the equivalence of the unoptimized and optimized code is undecidable in general.

6 Conclusions

In many situations, the performance of an analysis or reasoning tool is the critical factor that determines its utility to the user or the size of the problem that it can meaningfully address. Reduction to SAT is an increasingly popular solution technique. Previous results, as well as ours, show that the overall performance of the system depends not only on the inherent capabilities of the underlying SAT solver, but also on how the problem is expressed: semantically identical SAT problems have widely varying solving times.

We have presented a set of mechanical transformations that, for Alloy model checking problems, can substantially reduce the SAT solving time. As currently formulated, these transformations apply specifically to Alloy models. However, they all have direct analogs in the field of traditional compiler optimizations, and we anticipate that other researchers should be able to apply similar optimizations to their systems. The overall result should be a substantial improvement in the performance of systems that use SAT solvers to solve important analysis and reasoning problems.

Acknowledgements

We would like to thank Ilya Shlyakhter for discussions on Alloy Analyzer and Alexandru Salcianu for comments on a previous version of this paper. This work was funded in part by NSF ITR award #0086154 and in part by NSF ITR—SoD award #0438967.

References

1. William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, December 1999.
2. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

3. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. 36thConference on Design Automation (DAC)*, New Orleans, LA, June 1999.
4. Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
5. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.
6. Jonathan Edwards, Daniel Jackson, Emina Torlak, and Vincent Yeung. Faster constraint solving with subtypes. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, July 2004.
7. Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld. Automatic SAT-compilation of planning problems. In *IJCAI-97, Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1169–1176, Nagoya, Japan, August 1997.
8. Malay K. Ganai, Lintao Zhang, Pranav Ashar, Aarti Gupta, and Sharad Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In *Proc. 39thConference on Design Automation (DAC)*, pages 747–750, June 2002.
9. Daniel Jackson. Automating first-order relational logic. In *Proc. 8th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, San Diego, CA, November 2000.
10. Daniel Jackson. Micromodels of software: Modelling and analysis with Alloy, 2001. `http://sdg.lcs.mit.edu/alloy/book.pdf`.
11. Daniel Jackson and Alan Fekete. Lightweight analysis of object interactions. In *Proc. Fourth International Symposium on Theoretical Aspects of Computer Software*, Sendai, Japan, October 2001.
12. Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. ALCOA: The Alloy constraint analyzer. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 2000.
13. Henry Kautz and Bart Selman. Planning as satisfiability. In *Proc. European Conference on Artificial Intelligence (ECAI)*, Vienna, Austria, August 1992.
14. Sarfraz Khurshid and Daniel Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE)*, Grenoble, France, Sep 2000.
15. Sarfraz Khurshid, Darko Marinov, Ilya Shlyakhter, and Daniel Jackson. A case for efficient solution enumeration. In *Proc. Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, Santa Margherita Ligure, Italy, May 2003.
16. Ines Lynce and Joao P. Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. In *Proc. the IEEE International Conference on Tools with Artificial Intelligence (ICTAI'03)*, November 2003.
17. Joao P. Marques-Silva and T. Glass. Combinational equivalence checking using satisfiability and recursive learning. In *Proc. the IEEE/ACM Design, Automation and Testing in Europe (DATE)*, pages 145–149, March 2003.
18. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 39th Design Automation Conference (DAC)*, June 2001.
19. Sanjai Narain. Network configuration management via model finding. Internal report, Telcordia Research, Piscataway, NJ, Sep 2004.
20. Sanjit A. Seshia, Shuvendu K. Lahiri, and Randal E. Bryant. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions". In *Proc. 40thConference on Design Automation (DAC)*, pages 425–430, June 2003.

21. Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. In *Proc. Workshop on Theory and Applications of Satisfiability Testing*, June 2001.
22. Ilya Shlyakhter, Manu Sridharan, Robert Seater, and Daniel Jackson. Exploiting subformula sharing in automatic analysis of quantified formulas. In *Proc. 6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, May 2003.
23. Sathiamoorthy Subbarayan and Dhiraj K Pradhan. NiVER: Non increasing variable elimination resolution for preprocessing SAT instances. In *Proc. 7th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, May 2004.
24. Mandana Vaziri. *Finding Bugs Using a Constraint Solver*. PhD thesis, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2003.
25. M. N. Velev. Efficient translation of boolean formulas to CNF in formal verification of microprocessors. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 310–315, January 2004.
26. M. N. Velev. Encoding global unobservability for efficient translation to SAT. In *Proc. 7th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 197–204, May 2004.
27. Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In *Proc. 8thConference on Automated Deduction (CADE)*, July 2002.