Optimizing Alloy for Multi-Objective Software Product Line Configuration

Ed Zulkoski, Chris Kleynhans, Ming-Ho Yee, Derek Rayside, and Krzysztof Czarnecki

University of Waterloo
Waterloo, Ontario, Canada
{ezulkoski, drayside, kczarnec}@gsd.uwaterloo.ca

Abstract. Software product line (SPL) engineering involves the modeling, analysis, and configuration of variability-rich systems. We improve the performance of the multi-objective optimization of SPLs in Alloy by several orders of magnitude with two techniques.

First, we rewrite the model to remove binary relations that map to integers, which enables removing most of the integer atoms from the universe. SPL models often require using large bitwidths, hence the number of integer atoms in the universe can be orders of magnitude more than the other atoms. In our approach, the tuples for these integer-valued relations are computed outside the SAT solver before returning the solution to the user. Second, we add a checkpointing facility to Kodkod, which allows the multi-objective optimization algorithm to reuse previously computed internal SAT solver state, after backtracking.

Together these result in orders of magnitude improvement in using Alloy as a multi-objective optimization tool for software product lines.

Keywords: Product Lines, Multi-objective Optimization, Kodkod, Alloy

1 Introduction

Alloy is used for a wide variety of purposes, from analyzing software designs to checking protocols to generating test inputs and beyond. Recently, there has been some interest in using Alloy for design exploration or product configuration [11, 13]. These specifications often involve constraints on sums of integers (or other arithmetic expressions). For example, there might be a restriction on the total weight of a car, or on the disk footprint of a configured operating system kernel. Sometimes the user wishes to not only compute a viable product configuration, but an optimal one [11], often in the presence of multiple conflicting objectives.

These specifications often require solving with fairly large bitwidths, to support large metric values. In the general case, where the specification involves arbitrary constraints over relations containing integers, Alloy needs to create an atom for every integer in the bitwidth. At higher bitwidths the number of integer atoms dominate the number of other atoms, affecting solving time.

```
 1 one sig Car {
 2   e : Engine,
 3   f : Frame,
 4   w : Int,
 5 }{
 6   w = (e.x).plus[f.x]
 7   w < 9
 8 }
 9
10 abstract sig Part { x : Int }
11 abstract sig Engine extends Part {}
12   one sig Petrol extends Engine {}{ x = 3 }
13   one sig Diesel extends Engine {}{ x = 4 }
14 abstract sig Frame extends Part {}
15   one sig Aluminum extends Frame {}{ x = 5 }
16   one sig Steel extends Frame {}{ x = 6 }
```

Fig. 1. An example design exploration model. The goal is to choose components (engine and frame) for a car design according to some constraints (total weight < 9).

We observe that SPL specifications are not completely arbitrary, but usually associate equality constraints with each integer-valued relation (*e.g.*, lines 12,13,15,16 of Fig. 1). We use these equality constraints to rewrite other parts of the specification that refer to these integer-valued relations. If the rewritten specification meets certain conditions (see Section 2), then most integer atoms can be removed, thus producing much smaller SAT formulas. After solving, we use the equality constraints and the solution to the modified specification to produce a model of the original specification.

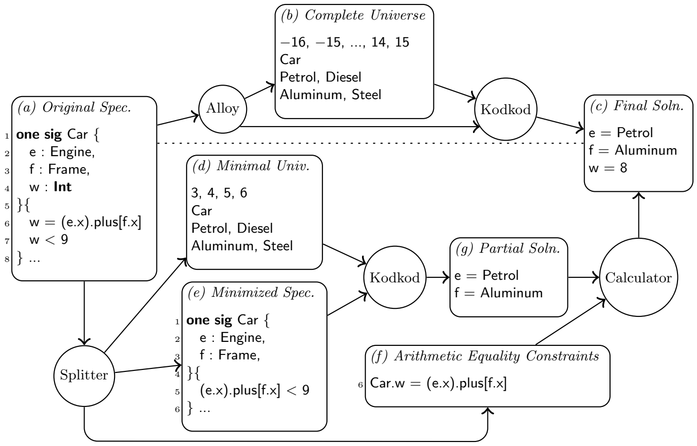
The approach we use for multi-objective optimization, called the *guided improvement algorithm* [13], requires many calls to Kodkod, first by adding constraints to find optimal solutions, and then backtracking. We have enhanced Kodkod to allow the removal of constraints following a stack discipline. (Note that Kodkod 2.0 already supports incremental addition of constraints.)

Together, these two enhancements to the Alloy toolchain result in several orders of magnitude improvement for performing multi-objective optimization of SPLs. We experienced an average of over 200X speedup on our experiments.

We focus on multi-objective optimization (MOO) on software product lines (SPLs) for our experiments. The goal of SPL engineering is to facilitate the modeling and analysis of variability-rich systems [3, 12]. These systems are typically represented as *feature models*: concise tree-like structures, whose *products* are valid configurations of the system [7]. Features may additionally contain *attributes*, indicating the effect of a feature on the overall *quality* of a product.

A natural analysis on attributed feature models is to identify *optimal* products with respect to the set of quality attributes. There may be many products that are considered optimal, particularly when *conflicting* objectives exist (*e.g.*, low cost *vs.* high performance). In such a case we say a product is *Pareto optimal* if increasing its value in some objective decreases its value in another. The goal of MOO is to discover all Pareto optimal solutions.

In this paper we work with a version of Alloy extended [11] with partial instances [10] and the *guided improvement algorithm* (GIA), an exact algorithm for MOO (see [13] for a full description). ClaferMOO [11] – an extension to Clafer [1] for MOO of attributed feature models – has been built using the GIA. We use a set of ClaferMOO specifications to evaluate our tool in Section 4.



```
one sig Car {
  e : Engine,
  f : Frame,
  w : Int
}{
  w = (e.x).plus[f.x]
  w < 9
} ...
```

```
one sig Car {
  e : Engine,
  f : Frame,
}{
  (e.x).plus[f.x] < 9
} ...
```

Fig. 2. Contrast of a standard Alloy run (above the dotted line) and our approach (below). The example model here is abridged from Fig. 1.

2 Eliding Integer Relations and Atoms

Fig. 2 contrasts the standard Alloy approach (above the dotted-line), and our approach to eliding integer relations and atoms through substitutions (below the dotted-line). Normally, Alloy generates the complete universe (Fig. 2b) and Kodkod specification, after which Kodkod produces our final solution (Fig. 2c).

First we divide the integer relations into dependent and independent (denoted *Splitter*). Integer-valued relations are identified as *independent* if they are bound to constants through equality constraints. *Dependent* integer relations are defined by an expression involving independent relations (*e.g.*, w in Fig. 1). Standard substitution techniques are used to remove dependent relations (Fig. 2e), however the equality constraints are retained (Fig. 2f). Integer atoms that are not explicitly named as constants in the specification may also be elided (Fig. 2d, see conditions below). A solution to the modified specification elicits values for independent relations (Fig. 2g). Dependent integer relations are computed from the retained equality constraints and the partial solution (Fig. 2c).

Conditions for when substitutions can be performed and relations can be elided:

1. The candidates for *dependent* integer-valued relations are functional (*i.e.*, *one*-multiplicity in Alloy) binary relations that map atoms to integers.
2. If the dependent relations depend on each other (and not just the independent relations), then they must do so according to some partial order.
3. The equality constraints must occur in top-level conjuncts, inside a single universal (*all*) quantifier. Alloy's *appended facts* meet this criteria.
4. The equality constraints must name just the dependent relation on one side or the other. (This constraints could be relaxed in future.)

5. If there exists multiple constraints on the same dependent variable, *e.g.*, $w = expr_1$ and $w = expr_2$, we remove both but add the constraint $expr_1 = expr_2$.

Conditions for when integer atoms can be elided from the universe:
1. All dependent integer-valued relations must be elided.
2. There can be no quantification over the integers ($e.g.$, $\{all \text{ x } : Int \mid p(x)\}$).

3 Checkpointing

The GIA works through repeated calls to the solver, and then backtracking to find other Pareto optimal points. When backtracking, constraints must be removed in order to find new points. Checkpointing allows us to revert to a previously saved state of the solver, without discarding all of the work that the solver has performed. Removal of constraints can be achieved by checkpointing before every constraint addition and reverting at a later time to remove that constraint. In the case of the GIA, it is not necessary that we be able to remove arbitrary constraints from the problem. It suffices to checkpoint after finding each starting point before we begin the drive to the Pareto front. This allows us to return to a solver state that only contains the problem constraints and the exclusion constraints specified by the previously found Pareto optimal points. We can then begin our search for a new starting point by adding the exclusion constraints from the last Pareto optimal point. To test the performance benefits of adding checkpointing support to Kodkod on the guided improvement algorithm, we have added the required support to version 2.2.0 of the MiniSat solver. This implementation simply creates a copy of the entire MiniSat solver object and stores it on a checkpoint stack. While simple, it is sufficient to show the benefits of checkpointing as a concept.

4 Evaluation

Our evaluation was over a set of 93 variants of nine MOO-specifications of SPLs,[1] compiled for work in [11], and originally described in [4, 14–16]. Each variant specification modifies the original by adding additional objectives and/or adjusting attribute values. The number of objectives ranges from one to seven.

Table 1 summarizes the speedup produced by just checkpointing, just the reductions (which include both formula changes and universe reductions), and their combination. Both techniques always result in some speedup in all experiments. Their combination results in an average speedup of over 200X, ranging from 20X to almost 1500X. Fig. 3 gives a graphical view of the data underlying the summary in Table 1. The x axis contains an entry for each of the 93 multi-objective product line specifications, ordered by their baseline solving time.

[1] The product line specifications can be found at:`https://github.com/TeamAmalgam/test-models/tree/f348271b005ee7d4929f73846e6ad8c4a19e0bd4/spl`.

Table 1. Summary of speedups obtained on a 3.4GHz quad-core Intel i7, 16 GB RAM, 64-bit Ubuntu 12.04, Java SE 64-bit 1.7.0.12. GIA 'magnifying glass' turned off.

	Min	Max	Mean	Median	Std. Dev.
Baseline	2473 ms	3,515,676 ms	145,523 ms	15,800 ms	449758
Checkpointing	1.32 X	19.23 X	2.53 X	2.57 X	1.75
Reductions	13.25 X	1014.93 X	161.70 X	193.52 X	104.09
Combined	22.81 X	1458.82 X	221.02 X	276.18 X	134.23

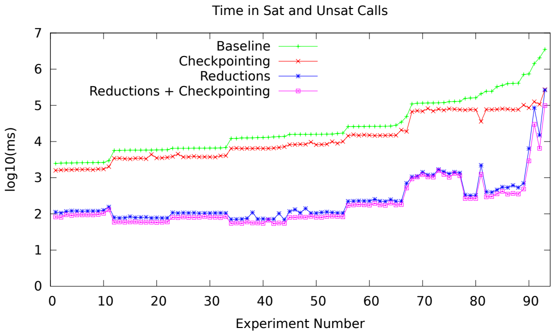


Fig. 3. Solving times of 93 multi-object product line models. Each point on the x axis represents a different model. The models are ordered by their baseline solving time.

5 Related Work

A variety of researchers have used equality constraints to rewrite formulas for improved performance. The idea is perhaps as old as the Knuth-Bendix completion algorithm [8]. In recent years the idea has been used in a number of SMT solvers [2, 5, 17, 6] and bounded model checkers [9]. For example, STP [6] is intended to solve constraints generated from the static analysis of software that makes use of arrays. STP uses rewriting to reduce these constraints into a form suitable for a SAT solver. In addition, solvers such as Z3 [17] support checkpointing as well.

In our work the efficiency gains come more from reducing the size of the universe than from the rewriting *per se*: the rewriting is a transformation that enables the universe size reduction. Smaller universes correspond to smaller SAT formulas with faster solving times.

6 Conclusions

For Alloy specifications that characterize multi-objective product lines, rewriting based on equality constraints facilitates the elision of both integer-valued relations and integer atoms. This elision results in an average speedup of over 150X. Further, adding checkpointing to the underlying SAT solver results in a 2X speedup. The combined speedup is over 200X.

References

1. Bąk, K., Czarnecki, K., Wąsowski, A.: Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled. In: Malloy, B., Staab, S., van den Brand, M. (eds.) Proc. 3rd SLE. LNCS, vol. 6563. Springer-Verlag (2010)
2. Brummayer, R.: Efficient SMT solving for bit vectors and the extensional theory of arrays. Ph.D. thesis, JKU Linz (2010)
3. Clements, P.C., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2001)
4. Esfahani, N., Malek, S.: Guided Exploration of the Architectural Solution Space in the Face of Uncertainty. Tech. rep., George Mason U., Dept. of C.S. (March 2011)
5. Franzen, A.: Efficient solving of the satisfiability modulo bit-vectors problem and some extensions to SMT. Ph.D. thesis, Univ. of Trento (2010)
6. Ganesh, V., Dill, D.L.: A Decision Procedure for Bit-Vectors and Arrays. In: Proc. CAV. LNCS, vol. 4590, pp. 524–536. Springer-Verlag (Jul 2007)
7. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) feasibility study. Tech. rep., SEI-CMU (1990)
8. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebra. In: Proc. Conf. on Computational Problems in Abstract Algebra. Pergamon Press (1970)
9. Merz, F., Falke, S., Sinz, C.: LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In: Joshi, R., Müller, P., Podelski, A. (eds.) Proc. VSTTE. LNCS, vol. 7152. Springer-Verlag (2012)
10. Montaghami, Vajihollah., Rayside, D.: Extending Alloy with partial instances. In: Derrick, J., Fitzgerald, J.A., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) Proc. 3rd ABZ. LNCS, vol. 7316. Springer-Verlag (Jun 2012)
11. Olaechea, R., Stewart, S., Czarnecki, K., Rayside, D.: Modelling and Optimization of Quality Attributes in Variability-Rich Software. In: NFPinDSML Workshop at MODELS Conference (2012)
12. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag (2005)
13. Rayside, D., Estler, H.-Christian., Jackson, D.: A Guided Improvement Algorithm for Exact, General Purpose, Many-Objective Combinatorial Optimization. Tech. Rep. MIT-CSAIL-TR-2009-033, MIT CSAIL (2009)
14. Siegmund, N., Kolesnikov, S., Kastner, C., Appel, S., Batory, D., Rosenmuller, M., Saake, G.: Predicting performance via automated feature-interaction detection. In: Murphy, G., Pezze, M. (eds.) Proc. 34th ICSE. Zurich, Switzerland (2012)
15. Siegmund, N., Rosenmuller, M., Kastner, C., Giarrusso, P.G., Apel, S., Kolesnikov, S.S.: Scalable prediction of non-functional properties in software product lines. In: Schaefer, I., John, I., Schmid, K. (eds.) SPLC Workshops. ACM (2011)
16. Siegmund, N., Rosenmuller, M., Kuhlemann, M., Kastner, C., Apel, S., Saake, G.: SPL Conqueror: Toward optimization of non-functional properties in software product lines. Software Quality Journal 1(3), 1–31 (June 2011)
17. Wintersteiger, C., Hamadi, Y., de Moura, L.: Efficiently solving quantified bit-vector formulas. Formal Methods in System Design 42(1), 3–23 (2013)