Extending Clafer with an SMT Backend

Ed Zulkoski
(ezulkosk)

October 11, 2013

Abstract

We propose to develop a new backend solver for Clafer - a variability modeling language developed in the GSD Lab at the University of Waterloo - using the Z3 SMT solver. Z3 is well respected for its speed and has been shown to outperform other solvers in the domain of partial modeling – one type of modeling within the scope of Clafer. The project will mostly involve a translation from Clafer constraints to Z3 constraints. Furthermore, by using Z3 we should be able to support language features not previously available in Clafer, including constraints over real numbers, and possibly string constraints. The other aspect of this project will involve a comparison of the new Z3 backend with the previously existing Alloy and Choco backends. If feasible, comparison to a bare-bones SAT backend of Clafer would also be useful. Optimistically, we wish to experimentally derive interesting hypotheses on the complexity of Clafer models through these comparisons.

1 Introduction

Clafer is a modeling language with first-class support for feature and meta-modeling [**?**]. In its current state, Clafer is supported by two backend solvers 1) Alloy – a bounded relational model checker, and 2) Choco – a library for constraint satisfaction problems. Clafer input is translated to either of these two backends, which then determine the satisfiability of the Clafer model, and produce instances if the model is satisfiable. Depending on the type of constraints within the Clafer model, one backend may be more suitable than the other. For example, Alloy does not perform well on arithmetic constraints over large integers, however the Choco solver is much more capable on these types of constraints. Some desirable language features are not currently supported by either backend, due to restrictions of the solvers. For example, neither the Alloy nor Choco backends currently support real numbers; although it may be feasible in Choco, it is not currently implemented. The goal of this project is to therefore develop and explore a new backend for Clafer using Z3 - an SMT solver.

Z3 is well known to be a fast solver, motivating our desire to create a new backend using it. In an experiment by [5], randomly generated partial models were analyzed by four solvers: Alloy, a relational logic solver; Minizinc, a *constraint satisfaction problems* (CSP) solver; Clasp with GrinGo, an *answer set programming* (ASP) solver; and Z3. Z3 was shown to be more efficient (based on solving time) in general, and also scaled better to harder problems.Z3 has also been shown to perform well relative to other SMT solvers, winning SMTCOMP 2012 competition in the theory categories of `QF_UFLIA`, `QF_UFLRA`, and `QF_BV`, among others[1].

2 Clafer Overview

Clafer is a structural modeling language which is designed for variability modelling. It unifies both feature models and meta-models [**?**].

In Figure 1 we show a model of a mobile phone in Clafer. The phone has an optional feature *PasswordProtection*, its optionality is denoted by the question mark after it. It also has a mandatory feature *Apps* that refers to a set of software applications, this is denoted by the -¿ symbol following Apps. The star

[1]Results can be found at `http://www.smtexec.org/exec/?jobs=1004` .

```
Phone 1 .. 2
  PasswordProtection  ?
  Apps −> SwApps ∗
  [ Sum Apps.memory <= 100 ]

SwApps ∗
  UID : int
  memory : int
```

Figure 1: An ilustrative model of a mobile phone and the apps it can have using clafer.

following *SwApps* means that *Apps* refers to a set of software applications of cardinality between zero and infinity. Constraints are introduced in square brackets, in this case a constraint is specified that the sum of the memory attribute of each app must be lower to 100. Finally an unbounded number of SwApps are introduced by the clafer SwApps followed by a star. Each SwApps contain a uid and a memory attribute, both of which take integer values. We should note that SwApps refers to both the type SwApps as well as defines a set of objects of type SwApps.

ClaferIG is a program that takes a clafer model specification and produces a clafer instance satisfying such a specification if it can find such an instance, it uses either Alloy or Choco as backends. *ClaferIG* given the specification in Figure 1 would produce for example an instance such as the one shown in Figure **??**.

3 Solution Overview

For consistency (and due to large amounts of overlapping terminology), we define the following terms and notations: 1) we call the Clafer input the *specification*; 2) an output that conforms to the specification is a *model*, and; 3) individual occurrences of a clafer in the model are *instances*. We use capitalized *Clafer* to denote the language Clafer itself, and lowercase *clafer* to denote individual components of the specification (e.g. `Phone` in Fig. 1).

The logic of Clafer can essentially be reduced to the following main components:

- Finite sets, and operations over them (e.g. set union). For the translation to Z3, we use a finite list of bounded integer variables to represent a given clafer (bitvectors would be sufficient as well). Abstractly, each integer in the list corresponds to a *potential* instance of that clafer in the outputted model. An instance is part of the model if Z3 returns a value for it not equal to a predefined *sentinel* number for that clafer (more details are further described in the following section). We then need to add constraints over these sets, such that if a satisfying assignment is returned from Z3 for these variables, then it can be mapped back to a Clafer model conforming to the specification. For example, these constraints must ensure that the number of instances of a clafer are within the specified cardinality, and that all bracketed constraints are satisfied.

- Basic arithmetic constraints (e.g. addition, multiplication) and boolean constraints (e.g. *and*, *or*, *not*), which have direct translations to Z3.

Clearly the difficulty lies in creating constraints over the finite sets, such that the integer variables can be mapped to a model conformant to the input. We discuss some details of this next.

4 Detailed Solution Overview

We illustrate some components of our approach through the example Clafer specification and instance in Figure 1. Note that there are many aspects of Clafer not present in this model, such as abstract clafers and inheritance; we reserve a description of these for the final report.

4.1 Representation of a clafer

As previously stated, a clafer is represented as a finite list of integers. Consider the `Phone` clafer in Figure 1. Since our outputted Clafer model will have at most two `Phone` clafers, we can represent it with a list of two integer variables, for example $[phone0, phone1]$.

For star-cardinalities (as in the reference clafer `Apps`), we place a finite *scope* s on the clafer, indicating that only s instances can occur in the model. For the sake of example, let us assume that the scope of all unbounded clafers is 4. Then we can represent the clafer `Apps` with the list $[Apps0, Apps1, Apps2, Apps3]$.

Semantically, the values of these integer variables represent *parent pointers*, indicating where the clafer should be placed in the outputted model. For example, if the variable $Apps0$ is set to 0 by the solver, then it should fall directly beneath $Phone0$ in the hierarchy of the outputted model. Likewise, if $Apps0 = 1$, then $Apps0$ would be placed under $Phone1$.

A clafer instance is *excluded* from the model if its integer variable is set to a pre-defined *sentinel* value, which equals the total number of instance variables of the clafer's parent. For example, since the `Apps` clafer is directly underneath `Phone`, and `Phone` has two instances variables, any `Apps` instance set to 2 will not be included in the model. For *top-level clafers* that do not have a parent, the instance is included if its corresponding variable is set to 0, and not included if set to 1.

5 Related and Previous Work

The work of Michel et. al. [3] discusses how configuration problems can be encoded in SMT using TVL [2] – a text based feature modeling language similar to Clafer. TVL supports many of the components of feature modeling, including hierarchy, group cardinality, attributes, enums, and cross-tree constraints. However, several aspects of Clafer make the translation to an SMT solver much more difficult than that of TVL in [3]. Most notably, Clafer supports many set constraints that make translation much more difficult.

A previous project [4] within the GSD lab was capable of translating a small subset of Clafer to Z3, in order to support attributed feature models for multi-objective optimization. The primary components of Clafer necessary for this domain include hierarchical constraints, fixed-size cardinalities (e.g. [0..1]), and basic arithmetic constraints. This project subsumes that translation by allowing arbitrary Clafer constraints.

6 Planned Contributions

The main deliverables for this project will be the implementation of the translation from Clafer to Z3, as well as hypotheses as to *why* certain solvers outperform other solvers, and which characteristics of Clafer models make solving difficult. The expected components of this project are:

1. **A translation of all base-components of Clafer to Z3.** This includes hierarchical constraints, set constraints, arithmetic, and cross-tree constraints.

2. **Additional Clafer functionality not supported by other backends.** Two important constructs not currently supported by Clafer are real numbers and strings. We would like to support these with our project to expand the scope of models that can be evaluated with Clafer.

3. **An evaluation against other backend implementations.** This would be primarily beneficial for Clafer users, as it would be difficult to make a fair comparison of how the underlying solvers (Z3, Alloy, Choco) perform on Clafer specifications. This is primarily due to the large differences in implementations; certain backends have different optimizations and general structure, and would certainly bias results.

4. **Hypotheses regarding which Clafer components make solving difficult.** First, it will be necessary to somehow measure the complexity of a given Clafer model. One suggested approach would be to translate Clafer models to SAT formulas, and, using known metrics for the complexity of SAT formulas, approximate the complexity of the original Clafer model. However, no current translation from Clafer to SAT exists (aside from through Alloy), so this will require investigation. Then, we will need to determine which Clafer components add the most to the overall complexity of the model. It

will be beneficial to generate random Clafer models to conduct this experiment, however this in itself is a difficult task, as Clafer has many components.

This experiment can have multiple implications. First (and most obviously), we can gain insight as to what makes Clafer models hard. However, since Clafer is general enough to support both feature and class modeling, we may gain insights on these domains as well.

References

[1] K. Bk, Z. Diskin, M. Antkiewicz, and K. Czarnecki. Clafer : Unifying Class and Feature Modeling.

[2] A. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming*, 76(12):1130–1143, Dec. 2011.

[3] R. Michel, A. Hubaux, V. Ganesh, and P. Heymans. An SMT-based Approach to Automated Configuration. *SMT Workshop 2012 10th . . .*, pages 107–117, 2012.

[4] R. Olaechea. *Comparison of Exact and Approximate Multi-Objective Optimization for Software Product Lines.* PhD thesis, 2013.

[5] P. Saadatpanah, M. Famelis, J. Gorzny, N. Robinson, M. Chechik, and R. Salay. Comparing the effectiveness of reasoning formalisms for partial models. *Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation - MoDeVVa '12*, 1(c):41–46, 2012.