Extensions to Clafer Using an SMT Backend

Ed Zulkoski

University of Waterloo, Waterloo, Ontario, Canada
ezulkosk@gsd.uwaterloo.ca
(ezulkosk, 20456819)

Abstract

We present a new backend for Clafer - a variability modeling language - using the Z3 Satisfiability Modulo Theory (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. We show that by retaining the notion of a *scope*, Clafer can generally be expressed within the logic of `QF_UFNRA` (quantifier-free non-linear real arithmetic), however all Clafer models encountered so far can be expressed in `QF_LIA` (quantifier-free linear integer arithmetic). We are capable of supporting language features not previously available in Clafer, including constraints over real numbers, and string constraints, leveraging Z3-Str: a theory extension to Z3. We conclude with a discussion of other extensions we have made, as well as new directions for future work that would not be possible with other Clafer backends.

Contents

1 Introduction

Clafer is a modeling language with first-class support for feature and meta-modeling [1]. In earlier work, Clafer has been translated to two backend solvers 1) Alloy – a bounded relational model checker, and 2) Choco – a library for constraint satisfaction problems. A Clafer specification can be translated to either of these two backends, which produce models of the specification if it is satisfiable, or produce an unsatisfiable core otherwise. Depending on the type of constraints within the Clafer specification, one backend may be more suitable than the other. For example, Alloy does not perform well on arithmetic constraints over large integers, as it deals with them by flattening bounded integers into boolean formulas, however the Choco solver is much more capable on these types of constraints. Furthermore, some desirable language features such as constraints over real numbers and strings are not currently supported by either backend, due to restrictions of the solvers.

We have developed a new translation from Clafer to Z3 [4] – a state-of-the-art SMT solver. Z3 is well known to be a fast solver, motivating our desire to create a new Clafer backend with it. In an experiment by [6], 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 the SMTCOMP 2012 competition in the theory categories of `QF_UFLIA`, `QF_UFLRA`, and `QF_BV`, among others[1]. Further, a recent extension to Z3: Z3-Str [7], allows support for some string constraints (*e.g.* length, substring).

2 Clafer Overview

Clafer is a structural modeling language which is designed for variability modeling. It unifies both feature models and meta-models [**?**]. In Listing 1 we show a specification of a mobile phone in Clafer. `Phone` refers to a top-level clafer of cardinality between 1 and 2 (*e.g.* either 1 or 2 of such phones must exist). `Phone` has an optional feature `PasswordProtection`; its optionality is denoted by the question mark after it. It also has a feature `Apps` that refers to a set of software applications; this is denoted by the -> symbol following `Apps`.

The star following `SwApps` means that `Apps` refers to a set of software applications of cardinality greater than or equal to zero. Constraints are introduced in square brackets. The constraint denoted by [sum Apps.ref.memory $\leq$ 100], specifies that the sum of the `memory` attributes of all `SwApps` for a given `Phone` instance must be lower than 100. Note that this constraint must be true of *all* `Phone` instances, due to its level of indentation. The dot character in "Apps.ref" in essence serves as a navigation operator, by performing a join between the set `Apps` and the automatically created set `Apps_ref`. This operator is applied again to obtain a set of `memory` values. These values are then added together by the sum operator, and it is enforced that such sum must be less than 100. Finally an unbounded number of `SwApps` are introduced by the clafer `SwApps` followed by a star. Each SwApps contains 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 specification as input and produces a clafer model satisfying such an specification, if it exists. It uses either Alloy or Choco as backends. Given the specification in Listing 1, *ClaferIG* will produce for example a model such as the one shown in Listing 2. This generated model has single mobile phone named *Phone0*, has the optional attribute PasswordProtection present ("PasswordProtection0") and has two apps: *SwApps0* and *SwApps1*, that have memory attribute of 15 and 26 respectively. Hence the mobile phone model respects the constraint that the sum of the memory attributes of its *Apps* is less than 100.

In essence the goal of this project is, among others, to replace *ClaferIG*.

3 Solution Overview

For consistency (and due to large amounts of overlapping terminology), we define the following terms and notations: 1) we use capitalized *Clafer* to denote the language Clafer itself, and

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

Listing 1: A specification of a mobile phone and its apps in clafer.

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

SwApps ∗
  UID : int
  memory : int
```

Listing 2: A generated model of a mobile phone and its apps in clafer from the specification in Listing 1.

```
Phone0
  PasswordProtection0
  Apps0 −> SwApps0
  Apps1 −> SwApps1

SwApps0
  UID0=5
  memory1=15

SwApps1
  UID1=6
  memory1=26
```

Listing 3: The integer variables associated with each clafer in Z3 and corresponding to Listing 2.

```
Phone: [0, 1]
PasswordProtection: [0, 2]
Apps: [0, 0, 2, 2]
Apps_ref: [0, 1, 4, 4]
SwApps: [0, 0, 1, 1]
UID: [0, 1, 4, 4]
UID_ref: [5, 6, 0, 0]
memory: [0, 1, 4, 4]
memory_ref: [15, 26, 0, 0]
```

lowercase *clafer* to denote individual components of the specification (*e.g.* `Phone` in Listing 1); 2) we call the Clafer input the *specification*; 3) an output that conforms to the specification (e.g. Listing 2) is a *model*, and; 4) individual occurrences of a clafer in the model are *instances*.

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.

Most of the difficulty lies in creating constraints over the finite sets in such a way that the integer variables can be mapped to a model conformant to the specification. 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 Listings 1 and 2, respectively. 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.

Listing 4: A more complex mobile phone specification in clafer.

```
SwApps *
  memory : int

IPhoneApps -> SwApps *
  [ this.ref.memory < 15 ]
AndroidApps -> SwApps *

[ sum AndroidApps.ref.memory <= 100 ]
```

4.1 Representation of a clafer

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

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. In Fig. 1c, since *Apps2* and *Apps3* both equal 2, they do not appear in the model in Fig. 1b. 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. For simplicity in the remainder of the paper, given a clafer `x`, an instance x_i is excluded if $x_i = x_{sentinel}$ ($Apps2 = Apps_{sentinel}$ in our example).

Reference clafers (e.g. `Apps`) require an additional integer variable associated with each instance, which corresponds to where the reference points. For a reference clafer instance x_i, we label its reference x_i_ref. For example, in Listing 2, $Apps0$ points to $SwApps0$, indicating that $Apps0\_ref$ in the Z3 output equaled 0. Each reference variable is bounded by the number of instances of the referenced clafer. A reference exists *iff* its corresponding clafer exists (e.g. $Apps0 \neq Apps_{sentinel} \Leftrightarrow Apps0\_ref \neq SwApps_{sentinel}$).

Integer clafers (*e.g.* memory) can be treated similarly to reference clafers, however their references are unbounded. Also, if an integer clafer is not present in the model, we set its reference to 0 to facilitate other set operations, such as summation.

As a larger example, once again consider Listing 2, along with the corresponding output of Z3 in Listing 3. For space limitations, each line of Listing 3 corresponds to all instances of the specified clafer; the line "Apps: [0,0,2,2]" is shorthand for $Apps0 = 0, Apps1 = 0, Apps2 = 2, Apps3 = 2$. Since $Apps0 = Apps1 = 0$, they are both beneath $Phone0$, however $Apps2$ and $Apps3$ are not present in the model (since $Apps_{sentinel} = 2$). Since $Apps0\_ref = 0$, $Apps0$ must reference $SwApps0$. Furthermore, since $Apps2$ and $Apps3$ are not present in the model, $Apps2\_ref = Apps3\_ref = SwApps_{sentinel}$.

4.2 Bracketed Constraints

We must also support Clafer's bracketed constraints (as in [*sum Apps.ref.memory <= 100*] from Listing 1), which we only discuss at a very high-level. To generate this constraint, we must first compute each of the joins in *Apps.ref.memory*. which is intuitively the set of `memory` clafers that are beneath any `SwApp` referenced by an `Apps` instance. We must then ensure that the sum of this set of `memory` instances is less than 100. Note that this is indeed the case in our example, since $memory0 + memory1 = 41 \leq 100$.

Although this example is not too difficult to convert to Z3 constraints, more complicated expressions and language features make set constraints challenging. Consider the specification in Listing 4. Both `IphoneApps` and `AndroidApps` reference `SwApps`, however when considering the summation on the last line, we must only consider instances of `SwApps` referenced by instances of `AndroidApps`. As another example, two previously unmentioned Clafer keywords are `this` and `parent`, which essentially allows one to consider each instance of a clafer individually. For example, the constraint: [$this.ref.memory = 15$] in Listing 4 essentially states that the `memory` field of any `SwApp` referenced by an `IphoneApp` must be equal to 15. This requires us to generate constraints corresponding to *all* individual instances of `IphoneApp`.

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 [5] 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, etc.

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 un-

derlying 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, K. Czarnecki, and A. Wsowski. Feature and meta-models in Clafer: mixed, specialized, and coupled. *Software Language Engineering*, 2011.

[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] L. D. Moura and N. Bjø rner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and ...*, 2008.

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

[6] 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.

[7] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: a z3-based string solver for web application analysis. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, page 114, 2013.