# 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** **1**

**2 Clafer Overview** **2**

**3 Solution Overview** **3**

**4 Detailed Solution Overview** **4**
4.1 Representation of a clafer . . . . . 4
4.2 Bracketed Constraints . . . . . 5

**5 Extensions to Clafer** **6**
5.1 String Constraints . . . . . 6
5.2 Isomorphism Detection . . . . . 6

**6 Related and Previous Work** **6**

**7 Future Work** **7**

**8 Conclusions** **7**

## 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 integer ranges into boolean formulas, however the Choco solver is much more capable in this regard. Still, Choco requires some bounding on integer ranges to facilitate the search process.

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. Many domains require real numbers to be modeled naturally. For example, any models involving probabilities or percentages can be modeled using reals. One applicable domain that has previously been modeled in Clafer is that of banking. The work investigated the family of available Scotiabank mortgage options, which include interest rates and monetary amounts, both of which can be naturally modeled with reals[1]. However, due to restrictions of previous backends, the specification could not be instantiated. Clafer should be capable of analyzing these models.

In short, the restrictions of Clafer are intrinsically tied to the limitations of its backend solver. In order to address some of these restrictions, 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[2]. Further, a recent extension to Z3: Z3-Str [7], allows support for some string constraints (*e.g.* length, substring).

This paper makes the following contributions:

1. We describe how Clafer models can be reduced to constraints in logics supported by Z3.
2. We have developed a tool called *ClaferZ3* that reflects the translation described in this work, and can be found at `https://github.com/gsdlab/ClaferZ3`.
3. Our tool supports constraints over real numbers and strings, which are not supported by previous Clafer backends, and thus expands the scope of models that Clafer can analyze. Furthermore, integer instances are unbounded.
4. We discuss further extensions to Clafer that are possible with an SMT backend. For example, we have prototyped a new approach for eliminating isomorphic models (Section 5.2).

## 2 Clafer Overview

*Clafer* is a structural modeling language which is designed for variability modeling. It unifies both feature models and meta-models [**?**]. Clafer specifications are built of components called clafers.

In Listing 1 we show a specification of two mobile phones in Clafer. `Phone` is an example of a top-level (non-nested) abstract clafer. **Abstract clafers** do not get directly instantiated in the resulting model, however concrete clafers, such as BudgetPhone and SmartPhone, can extend abstract clafers to inherit their sub-clafers. **Sub-clafers** are indicated by indentation,

[1] See `http://gsd.uwaterloo.ca/node/356`.

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

Listing 1: Clafer specification of two types of phones and apps.

```
1  abstract Phone ∗
2    Wifi ?
3    myApps −> App ∗
4    cost : real
5
6  App ∗
7    appCost : real
8
9  BudgetPhone: Phone 2
10   [ no myApps && no Wifi ]
11   [ cost = 49.99 ]
12
13 SmartPhone: Phone 1
14   [ cost = 99.99 + sum(myApps.appCost)]
```

Listing 2: A generated model of a mobile phone and its apps in Clafer.

```
App0
  appCost = 0.99
App1
  appCost = 2.99
BudgetPhone0 : Phone0
  Wifi0
  cost = 49.99
BudgetPhone1 : Phone1
    cost = 49.99
SmartPhone0 : Phone2
  Wifi1
  myApps0 −> App0
  myApps1 −> App1
  cost = 103.97
```

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

```
Phone: [0, 0, 0]
Wifi: [0, 2, 3]
myApps: [2, 2, 3]
myApps_ref: [0, 1, 3]
cost: [0, 1, 2]
cost_ref: [49.99, 49.99, 103.97]
App: [0, 0, 1]
appCost_ref: [0.99, 2.99, 0]
BudgetPhone: [0, 0]
SmartPhone: [0]
```

and cannot exist without their parent. Although our example only has two levels of indentation, other specifications may have more. The * after `Phone` indicates that zero or more `Phone`s may be included in the resulting model. `Phone` has an optional feature `Wifi`, denoted by the question mark after it. `myApps` is a **reference clafer** (denoted by the -> symbol) that refers to a set of `App`s. We emphasize that this is a set: an implicit constraint is that the same phone cannot have two of the same `App`. However, *different* phones may install the same `App`. A `Phone`, has a final attribute of `cost`, which we indicate is of type real number. Note that clafers that are declared primitive (int/string/real) get *desugared* into reference clafers that point to instances of their respective type. On line 6, `App` is an example of a **concrete clafer**.

On line 9, we introduce the concrete clafer `BudgetPhone`, which inherits all subclafers from `Phone`. The number 2 after it indicates that their must be exactly two `BudgetPhone`s in the resulting model. `BudgetPhone` has two constraints associated with it on lines 10-11 (indicated by square brackets). Line 10 consists of a quantified formula (with quantifier *no*), indicating that a `BudgetPhone` cannot have `App`s. Note that this constraint must be true of *all* `BudgetPhone` instances, due to its level of indentation. Line 11 indicates the price of `BudgetPhone`s, constraining the inherited clafer `cost`. Line 14 restricts the cost of `SmartPhone`s to be its original cost, plus the sum of all installed `Apps`. The dot character in "myApps.appCost" in essence serves as a navigation operator, by performing a join between the set of `App`s associated with this individual `SmartPhone` and the `appCost`s associated with them. A model of the specification is given in Listing 2.

This example only illustrates a fragment of the expressions supported by Clafer. Most notably, set operations such as intersection and union allow richer expressions over instances of clafers. We describe the other components of the language in Section 4

## 3 Solution Overview

We describe the approach of our translation through our example in Section 2. 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 lowercase

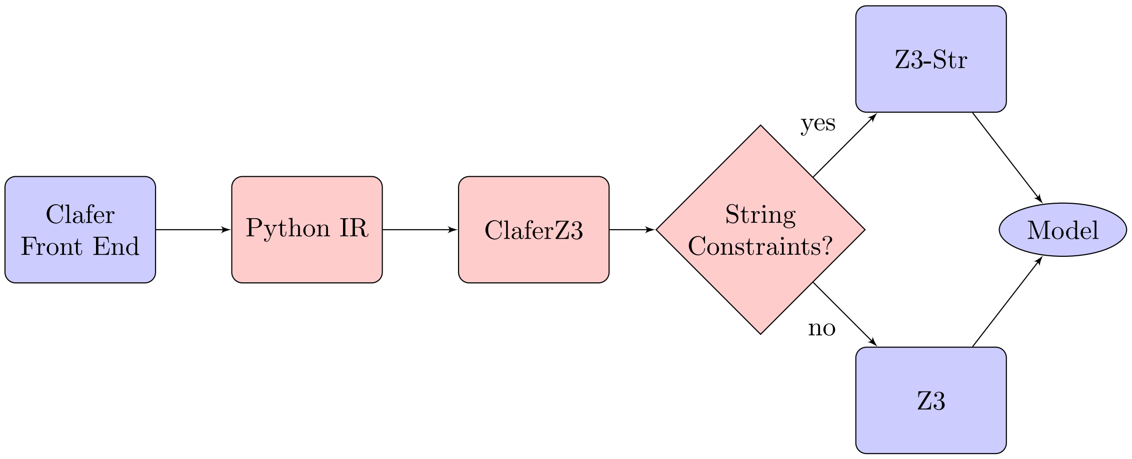

Figure 1: Depicts ClaferZ3 within the Clafer toolchain. Nodes marked red indicate contributions of this work.

*clafer* to denote individual components of the input (*e.g.* `Phone` in Listing 1); 2) we call the Clafer input the *specification* (*e.g.* Listing 1); 3) an output that conforms to the specification is a *model* (e.g. Listing 2); and 4) individual occurrences of a clafer in the model are *instances* (*e.g. Phone0* in Listing 2).

Figure 3 depicts how our translation of Clafer to Z3 fits into the Clafer toolchain. Clafer specifications are first run through the front-end (scanner, parser, type-checker, etc.), and a representation of the input is generated in Python. Using the Z3Py API, ClaferZ3 generates a set of variables and asserts constraints over them. If string constraints are not present, then we can use the standard Z3 implementation to check for satisfiability, else we use Z3-Str. If the solver finds a satisfying assignment over the generated variables and constraints, then the values of these variables can be interpreted as a Clafer model.

We devote the remainder of this section to describing the variables generated to express our Clafer specification in Z3, and how their valuations are interpreted back into a Z3 model. The actual constraints over the variables are discussed in Section 4. We hope that by discussing the end result first, the constraint that need to be generated are more intuitive.

### 3.1 Interpreting Z3 output as a Clafer Model

## 4 Detailed Solution Overview

This section describes the Z3 constraints that are generated to represent a Clafer specification in Z3. We begin with our representation of a clafer in Section 4.1. The remaining sections constrain this representation to conform to all restrictions of the specification (*e.g.* cardinality constraints, bracketed constraints, etc.).

### 4.1 Representation of a clafer

We borrow two key ideas from a previous translation of Clafer to Choco [**?**]: 1) clafers are represented as a finite number of integers; and 2) clafers that extend abstract clafers have a direct mapping to their supers (we discuss this point later).

We reprensent a clafer 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

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 ]
```

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 Extensions to Clafer

A notable difference of using Z3, as opposed to previous backends of Clafer, is its ability to handle constraints on real numbers[3]. Z3 handles real numbers naturally, so incorporating constraints over them into our translation required minimal effort. We therefore do not discuss further details.

We discuss further extensions allowing string constraints, such as length, concatenation, and substring. We also propose a new algorithm to prevent the generation of models that are isomorphic to previously generated models.

### 5.1 String Constraints

asdf

### 5.2 Isomorphism Detection

## 6 Related and Previous Work

DISCUSS THE TVL PAPER AN SMT APPROACH TO AUTO...
DISCUSS LEODEMOURA SET THEORY

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

[3]Due to complications with the Clafer frontend grammar, we have only been able to prototype these constraints with small hand-generated models. This also applies to string constraints. These restrictions will be addressed in the near future.

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.

This work most closely resembles that of [**?**]. Their work translates Clafer to Choco, a CSP language. We have borrowed components from that work, particularly in our representation of clafers and inherited clafers. We are able to extend Clafer with constraints over strings and real numbers, which neither the Choco nor Alloy backend currently support. Z3 also allows us to handle arbitrarily larger integers, which neither of the other backends support. In addition, in the future we intend to extend this project in order to eliminate the need for scopes on unbounded clafers. We discuss this in the following Section.

## 7 Future Work

This project utilized only a small subset of the available features of Z3. For example, Z3 has support for uninterpreted functions and data types. We would like to modify our translation to take advantage of these features. With this approach, we may be able to eliminate the need for scopes in our implementation. The work in [**?**], which translates components of Alloy to SMT, may be a useful starting point for this extension.

One particular reason we did not take this approach yet is due to translation efficiency. As an example, in an early version of this project we used an uninterpreted function to represent cardinality constraints. Given a clafer, the function would restrict its instances to conform to the cardinality constraints. By blasting this function into simple arithmetic constraints (which we can do since we have a bounded number of instances), our translation times were significantly decreased[4], even for small models. We emphasize that here we are referring to *translation time*; Z3 took approximately the same time to solve models whether or not functions were used.

A proper evaluation is necessary to compare the three Clafer backends that are now available. Since each backend was developed independently and contain unique optimizations, this comparison cannot properly evaluate the effectiveness of Alloy/Choco/Z3 as a backend. However, it would be useful to users of Clafer who wish to choose the ideal backend for their application.

Finally, we intend to further evaluate our extensions for string and real number constraints. Although we currently have models with these constraints, they cannot be properly compiled by Clafer, due to lack of support in the Clafer frontend (see footnote 3). These issues will be addressed in the near future.

## 8 Conclusions

We have developed a translation from Clafer to Z3. We extend capabilities of Clafer with string constraints using Z3-Str, as well as constraints over real numbers.

## References

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

[4]Although time differences were not recorded, some relatively small specifications required 2-3 additional seconds.

[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.