# ClaferIG

## Correcting Clafer models with automatic analysis

Jia Liang
Generative Software Development Lab
University of Waterloo
Canada
jliang@gsd.uwaterloo.ca

## ABSTRACT

ClaferIG[1] is primarily a command line tool for generating instances for the Clafer[2] modeling language. The tool is an important part of the Clafer family. This paper outlines a few big issues with writing Clafer models and how ClaferIG tackles them. The main focus of the paper is debugging models with small portions devoted to related topics. The final sections are dedicated to limitations and possible directions to extend the power of the tool.

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods*

## General Terms

Verification

## Keywords

Clafer, Model finding, UNSAT

## 1. INTRODUCTION

Clafer is a lightweight modeling language designed for conciseness and readability. Clafer models assist communicating requirements across experts of varying domains. The language aims for a low learning curve and should be easily understandable by non-technical personnel. Stakeholders and domain experts can verify that the model captures the correct specification if they find the model semantics easy to grasp.

Writing correct models is difficult despite the simplicity of Clafer. The modeller needs to examine the model from all angles for agreement with the intended specification. Any work or patches on the model can drastically alter the semantics of the model and introduce unintended negative effects. This motivates the creation of tools to assist in writing correct models. Ideally, models should be built incrementally and verified at each stage for correctness, much like programming applications with programming languages.

The Clafer translator is the center of the Clafer project. It compiles a Clafer model into an equivalent Alloy model and performs some analysis on the model during the compilation. The outputted Alloy model can be executed in the Alloy Analyzer to generate *Alloy* instances. A model is satisfiable, or consistent[2], if there exists at least one instance of the model. Verifying satisfiability of a Clafer model is performed in the following sequence: translate the Clafer model to Alloy; execute the Alloy model; if the Alloy analyzer returns at least one instance, then the Clafer model is satisfiable, otherwise the model is unsatisfiable for the given scope. Technically, it is possible to manually translate the generated Alloy instances back to Clafer instances, but this process is done by hand and requires significant overheard per instance. Regardless, with a little bit of scripting, Clafer models can be checked for satisfiability automatically.

## 2. SATISFIABILITY

$$satisfiable(model, scope) = truthvalue \quad (1)$$

A model is a formal specification. A specification where conformance is impossible is useless in practice. Likewise, an unsatisfiable model is indicative of an error in the model. Either the model has an incorrect encoding of the intended properties or the intended properties are contradictory. The satisfiable function is helpful because it can automatically flag the unsatisfiable model as incorrect.

A Clafer modeller can start with a minimal model verified by the satisfiable function and verified again after every change. This incremental process of building a model helps catch errors as early as possible. If the model becomes unsatisfiable after the latest changes, then something amongst the set of recently introduced changes is causing conflict within the model. The source of error is localized to one batch of changes.

Now Clafer has an automatic method of checking for a certain class of errors. Is the satisfiable function satisfactory for writing bug-free Clafer models?

[1]ClaferIG is an open source project found here: https://github.com/gsdlab/claferIG
[2]Clafer is an open source project found here: https://github.com/gsdlab/clafer

## 2.1 Under-constraint

The satisfiable function does not detect the class of under-constraint errors. An under-constraint model is a model that accepts more instances than intended because it is missing some constraint(s) that rule out those extra instances.

Listing 1 is a model of people and their spouses. The constraint enforces the rule: your spouse's spouse is yourself. From a quick glance, the model appears to capture the specification of people and marriages. Upon closer inspection, the model is too permissive and accepts incorrect instances.

**Listing 1: Spouses model**

```
abstract Person
    marriedTo → Person ?
    [this = marriedTo.marriedTo]

Alice : Person
Bob : Person
```

Feeding the spouses model and a small scope to the satisfiable function returns true. From the stance of satisfiability, the model is perfectly legitimate. However, the model suffers from under-constraint and the satisfiable function does not make this obvious. The model accepts an instance where Alice is married to herself and Bob is married to himself. Self-marriage is an unintended property and an error in the model. A constraint that forbids instances with self-marriage is missing.

**Listing 2: Corrected spouses model**

```
abstract Person
    marriedTo → Person ?
    [this = marriedTo.marriedTo]
    [this ≠ marriedTo]

Alice : Person
Bob : Person
```

An under-constraint model is satisfiable because it accepts some intended and unintended instances. The satisfiable function is impartial towards correct models and under-constraint models, therefore detecting this class of errors is impossible with this function alone.

## 2.2 Over-constraint satisfiable

Over-constraint is the reverse of under-constraint, the model rejects some intended instance(s). An over-constraint but satisfiable model accepts a non-empty, proper subset of intended instances.

Listing 3 is a model of all natural numbers. However, the model is incorrect. It captures all intended instances except for the case where $A = 1$. The constraint is the source of the bug, it should be $[A \geq 1]$.

**Listing 3: Natural numbers**

```
A : integer

[A > 1]
```

The model is still satisfiable, and the satisfiable function returns true. Just like under-constraint, the function does not help detect this of class of errors.

## 2.3 Insufficient scope

Ideally, if the satisfiable function returns false, then the model is unsatisfiable. However, due to undecidability of first-order logic, the satisfiable function for Clafer requires a scope as a compromise. The scope draws a finite boundary around the search space of instances, reducing a possibly infinite space to a finite one. When Clafer's satisfiable function returns false, it claims that the model is unsatisfiable *for the given scope*. It makes no commitment about the unsatisfiability of the model in general.

The satisfiable function returning false does not guarantee an error in the model because an insufficient scope will produce a false positive. Listing 4 is a model where the satisfiable function returns false when the scope $\leq 4$. The model is correct however, and increasing the scope will show that it can be satisfied.

**Listing 4: Five or more**

```
A 5..*
```

When checking for unsatisfiability, the modeller needs to rerun the function with increasing scopes until he or she feels confident that the problem is indeed an unsatisfiable model and not a limitation of scope. There is a trade-off when upping the scope: increasing the scope increases the confidence in the function[3]; increasing the scope drastically increases the computation required and deteriorates the performance.

Manually setting the scope is a nuisance for models with many objects. A lazy user will increase the global scope to the scope required by the most demanding object, needlessly multiplying the instance search space. Hand-selecting individual scopes for the objects is more dignified but laborious.

## 2.4 Over-constraint unsatisfiable

The class of over-constraint and unsatisfiable models are models that do not accept any instances given any finite scope. This is the original class of errors the satisfiability function tries to detect.

Listing 5 is the model of all numbers $A$ that are both odd and even. Unknowingly, the set of even and odd are disjoint so the model is unsatisfiable. The intention of the modeller was wrong, he or she is attempting to model a contradiction. The problem should be abandoned or restated.

**Listing 5: Even and odd**

```
A : integer
B : integer
C : integer

[ A = 2 * B ]
[ A = 2 * C + 1 ]
[ B > 0 ]
[ A > 0 ]
[ C > 0 ]
```

Not all over-constraint models are from wrongful intent. It is possible that the intent was right but the execution was wrong. Listing 6 is a model of legal driving age. In Canada, the legal driving age is 16. The model enforces the driving age by constraining all eligible drivers to be 16 years old or older.

**Listing 6: Drivers**

```
abstract Person
    CanDrive ?
    Age : integer
    [ CanDrive ⟹ Age ≥ 16 ]

Alice : Person
    [ CanDrive ]
    [ Age = 15 ]
```

The driving age rule is specific to the Canadian jurisdiction and it does not apply to Alice, a native of New Zealand, where she can legally drive at the age of 15. The addition of Alice breaks the model because it tries to enforce the Canadian legal driving age on everyone. The constraint needs to be rewritten to be less restrictive and apply only to Canadians.

An unsatisfiable model needs to be corrected. The satisfiable function helps detect unsatisfiability but does explain *why* it is. Debugging a large model with many constraints can be difficult without any hints.

# 3. CLAFERIG

Due to the problems listed above, the satisfiable function is severely limited in its usefulness, but the promise of an automatic verification tool is appealing. ClaferIG extends the function in different directions with more powerful algorithms to deal with issues caused by the classes of errors discussed earlier. The goal is to create a well-rounded tool that detects and explains the source of errors. Any serious Clafer modeller should consider having ClaferIG in their tool belt.

The current back end is Alloy. Alloy's back end is Kodkod. Kodkod's back end is a SAT solver implementation. This relationship is important and the following sections will comment on the implementation hierarchy when relevant.

## 3.1 Model finding

The problem with under-constraint models is too many instances. The satisfiable function only cares for existence of instances, disregarding their shape, size, and number. Instead of returning true or false, the satisfiable function can return the number of instances that satisfy the model so the modeller can see if there are too many. However, this would require the modeller to know how many instances to expect before making the comparison. For large models, this approach is not feasible due to symmetry breaking[5][6], and isomorphism[4].

Instead, the instances should be the output. The function that takes a model and scope and returns all the instances within the scope is called the model finding function. The

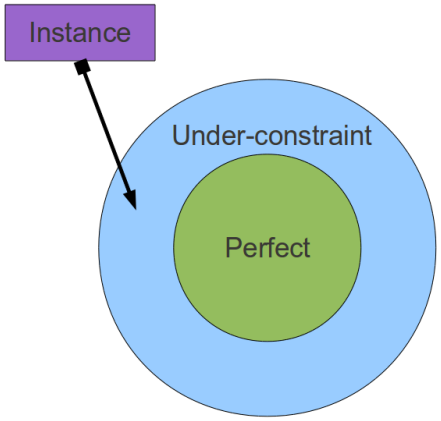



**Figure 1: Proving under-constraint**

modeller can scrutinize the instances for unexpected abnormalities.

$$modelfind(model, scope) = \{instances\} \quad (2)$$

ClaferIG implements model finding for the Clafer language. The implementation heavily relies on Alloy's model finding algorithm. Listing 7 shows one of the instances of running Listing 1 with ClaferIG.

**Listing 7: ClaferIG instance for Listing 1**

```
Alice
  marriedTo1 = Alice
Bob
  marriedTo2 = Bob
```

The problem with the model is visible. Self-marriage is not intended but the instance proves that the model does not adequately restrict it. Once the missing constraint is added, this instance is removed. Hence detecting under-constraint simply requires finding one unintended instance in the set of instances returned by the model finding algorithm. For large models, the algorithm will return many instances and the modeller will need to manually inspect as many as possible.

In the context of debugging models, model finding is useful for detecting the class of under-constraint errors. Aside from debugging, there are other reasons for wanting model finding for Clafer.

### *3.1.1 Education*

Clafer aims to be easy to learn, and the best way to learn is to play with language. The syntax is easy to learn because the Clafer translator will report any malformed syntax. The syntax is easy to grasp through experimentation with the Clafer translator to see what passes and what fails.

Understanding the semantics is quite tricky. Without ClaferIG, a novice modeller does not see any *Clafer* instances. The

novice must understand the Alloy output to play around with the Clafer semantics.

With ClaferIG, the learning is more hands-on. A novice can toy around with a Clafer model to see how changes in the model affect the set of generated instances. This is a more approachable method of discovering how the semantics of Clafer work.

Model finding is not only useful for beginners learning the language. Clafer experts can use the generated instances to help visualize the model. Instances created by ClaferIG are auto-generated "documentation" that are useful for explaining, recalling, and discussing the intent of the model within a group of modellers, some who are possibly unfamiliar with the work.

### 3.1.2 *Product line*

Clafer modelling unifies feature and class modelling and can naturally capture variability of a product line in a single model[1]. Listing 8 is a model of a computer product with one feature of interest: performance. The performance feature is split into two mutually exclusive subfeatures: fast and slow. The architecture and physical components of the computer are detailed under `Computer`. A computer is "fast" if it has more than 4GB of Ram and clock speed exceeds 4GHz. Slow is the negation of fast because it is the only alternative subfeature within the `xor` group.

**Listing 8: Computer**

```
1 Computer
2     GbRam : integer
3     Ghz : integer
4
5 xor Performance
6     Fast
7     Slow
8
9 [ Fast ⇔ GbRam ≥ 4 && Ghz ≥ 4]
```

Creating a custom product is as simple as defining the set of features the product must contain. Desired feature(s) are appended as additional constraints to the model. For example, adding the constraint `[Fast]` (semantically equivalent to the constraint `[some Fast]`) will require the computer to be "fast". Running the example in ClaferIG will generate all possible configurations of computers with the "fast" feature. In practice, a product line will have an assortment of features to choose from. ClaferIG will find all possible combinations of configurations that satisfy the set of requested features.

ClaferIG can return many possible answers but some configurations are preferable when creating a new product. A high-end commercial computer and a state-of-the-art supercomputer both implement the "fast" feature but a customer might be more inclined to purchase one over the other due to finances. There is current work on Clafer for maximizing and minimizing objectives when searching for instances[3]. When used in conjunction with ClaferIG, it will be possible to configure a product based on features while optimized against a set of criteria like cost.

[3]The work is based on Moolloy, an extension of Alloy: http://sdg.csail.mit.edu/moolloy/

## 3.2 UNSAT Core

An unsatisfiable model is too restrictive because a subset of constraints are imposing conditions that are impossible to meet. If the model is mostly correct then only small a subset of the constraints are problematic. When ClaferIG detects unsatisfiability, it will return this subset of unsatisfiable constraints called the UNSAT core. The implementation is based on the Alloy's UNSAT core algorithm.

**Listing 9: ClaferIG UNSAT core of drivers**

```
1 No more instances found. Try increasing
      scope to get more instances.
2 The following set of constraints cannot
      be satisfied in the current scope.
3   1) some this . CanDrive
4   2) (this . Age) = 15
```

Listing 9 shows the output of ClaferIG after running the model in Listing 6[4], and it detected that the model cannot be satisfied. Two constraints cannot be simultaneously true: the constraint claiming that Alice can drive; the constraint stating that Alice is 15 years old.

ClaferIG's output matches the intuitive explanation of the error, Alice cannot be 15 and a driver. The message pinpoints why the model is over-constraint and directs the attention to the critical sites. The modeller will either modify the national driving rules or revoke Alice's driving privilege to correct the model.

A trivial UNSAT core is the subset containing every constraint but a useful core is more limited. A smaller core is more precise because it narrows down the actual faulty constraints. In theory, the minimal UNSAT core is ideal but minimizing the core is an expensive operation. The back end of ClaferIG implements three settings, in order of fastest to slowest: initial UNSAT core, medium minimization, guaranteed local minimum. ClaferIG sets the fastest setting by default and allows the user to switch between settings. The slowest setting is many times more expensive than the fastest and the gains are often small.

## 3.3 Fix

An unsatisfiable model is incorrect and needs to be fixed. ClaferIG understands the problem, a subset of constraints cannot be satisfied. It will try to "fix" the model by removing constraints inside the UNSAT core until the modified model is satisfiable.

Listing 6 has two constraints in the UNSAT core. ClaferIG will attempt to fix the model by removing one of the two constraints. Listing 10 shows the actual output. The constraint stating "Alice is an eligible driver" is removed and new model is now consistent.

[4]Due to a temporary limitations with the parsing method of the Clafer translator, the reported faulty constraints do not match the exact syntax in the user-defined model. ClaferIG shows the internal AST representation of the constraint until the limitation is resolved.

**Listing 10: ClaferIG fix of drivers**

```
Altering the following constraints
    produced a counterexample.
  1) removed some this . CanDrive
Alice
   Age = 15
```

After fixing the model, ClaferIG shows a hypothetical instance if the modeller follows the suggested fix. In Listing 10, removing Alice's driving eligibility generates an instance where Alice is 15 years old and cannot drive. The instance is dubbed the *counterexample*, although misleading since the term has a different meaning in Alloy[3]. A new term is needed to avoid confusion.

The process of selecting the constraint from the UNSAT core to remove is arbitrary. When fixing a model, some constraints should be favoured by the constraint selector, rather than the current simple approach. For example, cardinality constraints should be prioritized last. This requires further investigation.

## 3.4 Scope

The trade-off imposed by the scope is a big challenge, a small scope limits the power of ClaferIG but a large scope grinds it to a halt. Solving Clafer models is an NP-complete task and the slightest increase in scope has a noticeable effect on computation time. The scope sweet spot is the minimum scope that satisfies the model. Running model finding with the minimum scope is the fastest possible execution that returns an instance. If the modeller would like more instances, then he or she can slowly increase the scope past the minimum.

Hand computing the minimum scope is too primitive and tedious, ClaferIG should automatically compute the scope of the model. If the model is not satisfiable, then set the scope to 0.

$$minimumScope(model) = \begin{array}{l} minimum(scope) \\ \text{where} \\ satisfiable(model, scope) \end{array} \quad (3)$$

Implementing the `minimumScope` function is difficult. Finding a counterexample for Fermat's last theorem specialized for a given degree is reducible to a minimum scope problem. Applying Listing 11 to the hypothetical `minimumScope` will either find the smallest counterexample of Fermat's last theorem for the given degree, or internally prove Fermat's last theorem for the specified degree and set the scope to 0.

**Listing 11: Disprove Fermat's last theorem for degree 3**

```
A 1..*
B 1..*
C 1..*

[ (#A × #A × #A) +
  (#B × #B × #B) =
  (#C × #C × #C) ]
```

A Clafer model consists of two types of elements, clafers and constraints. Clafers contain accessible scope information such as cardinalities and hierarchies but constraints make analyzing scope difficult. ClaferIG's scope analysis algorithm ignores constraints and only traverses the clafer regions of the model[5]. This subproblem is tractable and the result approximates the ideal `minimumScope` function. The subsections below outline the algorithm and the problems encountered during implementation.

### 3.4.1 Parents and children

The analysis of the parent clafer must precede the analysis of its children. In Listing 12, any valid instance must contain at least 3 vehicles and at least 4 wheels per vehicle. The algorithm ignores the upper cardinalities of the clafers, only the lower cardinalities affect the *minimum* scope. After analyzing the `Vehicle` and then `Wheel` clafers in that order, the algorithm concludes that minimum scope of the model is $\{Vehicle \to 3, Wheel \to 12\}$.

**Listing 12: Automotive system**

```
Vehicle 3..4
    Wheel 4..*
```

### 3.4.2 Supertypes and subtypes

The analysis of the subtypes must precede the analysis of its supertype. In Listing 13, the algorithm needs to analyze `CarA` and `CarB` before `Vehicle`. The set of subtypes form a partition of the super type. The minimum scope must contain at least 1 `CarA` and 2 `CarB`. Therefore any valid instance will contain at least 3 vehicles and therefore at least 12 wheels.

**Listing 13: Automotive system**

```
abstract Vehicle
    Wheel 4..*

CarA : Vehicle
CarB : Vehicle 2..3
```

### 3.4.3 References

The analysis of the references must precede the analysis of its referee. In Listing 14, `CarLot` cannot be satisfied unless there exists at least 3 vehicles and therefore at least 12 wheels.

**Listing 14: Automotive system**

```
Vehicle *
    Wheel 4..*

CarLot → Vehicle 3..*
```

The combination of references and subtyping is a special situation. The algorithm needs to take the maximum between the subtype and reference analysis. In Listing 15, the `CarLot` references must refer to at least 3 vehicles. However, the subtype scope analysis already designated 2 `CarA`

[5]The scope analysis algorithm now resides in the Clafer translator source. It is useful outside of ClaferIG as well.

and 2 `CarB`, hence there are enough vehicles in the model to satisfy the `CarLot` reference requirement. The minimum scope is $\{CarLot \rightarrow 3, CarA \rightarrow 2, CarB \rightarrow 2, Vehicle \rightarrow max(2+2,3) = 4, Wheel \rightarrow 16\}$.

**Listing 15: Automotive system**

```
abstract Vehicle *
    Wheel 4..*

CarA : Vehicle 2..*
CarB : Vehicle 2..*

CarLot→ Vehicle 3..*
```

### 3.4.4 Dependency

DEFINITION 1. *Clafer A depends on clafer B if the scope analysis of B must precede the scope analysis of A.*

For the minimum scope algorithm to compute the correct result, it needs evaluate the clafers in an order that respects the rules of dependency. The 3 rules of dependencies discussed previously are:

**Parent dependency** A child depends on its parent.

**Subtype dependency** A supertype depends on its subtypes.

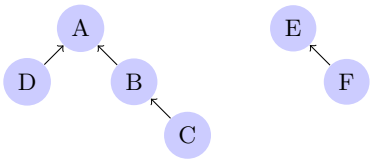
**Reference dependency** A referee depends on its referers.

A dependency graph is a directed graph where the nodes are clafers and the edges are the dependencies. If the graph only contains parent dependencies, then the graph is a collection of trees where the root clafers are the root nodes. Listing 16 is a model without references and subtyping and its dependency graph consists of two trees. The order of solving a dependency graph of trees is simple, solve the top level and then the next level, and so on.

**Listing 16: Parents**

```
A
    B
        C
    D

E
    F
```




Parent dependencies create dependencies in the form of trees and the order of evaluation is simple. Subtype and reference dependencies create edges between trees and the dependency graph loses the simple tree forms. In the worst case scenario, it is possible to form cycles in the dependency graph. The order of solving with all 3 dependencies is not trivial.

The evaluation order of scope must respect the dependency graph. A common strategy for evaluating a dependency graph is to evaluate on the topological order. Unfortunately, topological sort requires an acyclic dependency graph. To overcome this limitation, first compute all strongly connected components of the graph, and find the topological order of the components. Solve each component in this order. If each component is a singleton, then the dependency graph contains no cycles and the scope evaluation proceeds without hurdles. If a component is larger than a singleton, then the algorithm employs an heuristic to solve the cycle of clafers. The current heuristic is to assume the scope of each clafer in the component to be 1 and solve the component's clafers in an arbitrary order. The frequency of cyclic scope dependencies in practice is unknown.

## 4. IMPLEMENTATION

ClaferIG requires two tools, the Clafer translator and the Alloy analyzer API. Alloy is currently the only supported back end for the Clafer language and plays a large role in the semantics and functionality of Clafer.

When the ClaferIG executes, it passes the Clafer model to the Clafer translator to retrieve 3 important pieces of information: the semantically equivalent Alloy model, the internal intermediate representation (IR) of the Clafer model, and a mapping of constraints. The `minimumScope` of the model is approximated with the algorithm discussed earlier. The Alloy model is fed to the Alloy API to search for solutions. If Alloy instances are found, ClaferIG will transform every Alloy instance back into a Clafer instance. If the Alloy model is unsatisfiable, then ClaferIG will transform the Alloy UNSAT core into a Clafer UNSAT core. The constraint mappings encode the types of the constraints in the Alloy model and the relationship between the Alloy constraints and their corresponding Clafer constraints. The IR is traversed to reconstruct the unsatisfiable constraints into textual form.

ClaferIG needs to dynamically remove constraints to generate a fix for an unsatisfiable model. The mapping between Alloy constraints and Clafer constraints is not one-to-one so care is needed when choosing constraints to remove. For example, the Clafer translator adds additional constraints in the Alloy model to enforce structural rules between parent and child clafers: a child belongs to *one* parent. Removing these constraints will violate the semantics of Clafer. Constraint information is maintained in the constraint mappings created by the Clafer translator and ClaferIG consults the mappings to avoid catastrophic behaviour.

Kodkod, Alloy's back end, requires a SAT solver for solving models. Kodkod supports an array of SAT solver implementations, each with different advantages and disadvantages. ClaferIG will always select the MiniSatP SAT solver for 2 reasons, the implementation is fast and the solver implements proof logging required for the UNSAT core operation. The MiniSatP binaries for a few popular architectures are bundled inside the Alloy distribution. There are some

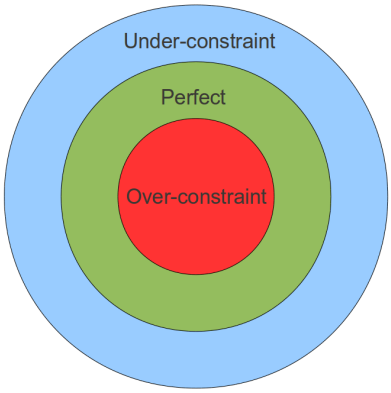



**Figure 2: Over-constraint but satisfiable**

portability issues distributing native binaries in this manner. SAT4J, Alloy Analyzer's default solver, is more reliable but it does not offer proof logging. UNSAT core is an important functionality of the project and any plausible alternative for MiniSatP must provide proof logging support.

ClaferIG is written in Haskell and provides the command line interface where the user interacts with their model. The Alloy API is written in Java and is inaccessible to ClaferIG. To overcome the language barrier, the project implements a second component nicknamed AlloyIG that exposes relevant Alloy API functionality through interprocess communication. The main ClaferIG process communicates with AlloyIG via a simple protocol. Both components are maintained in the same open-source project.

ClaferIG invokes the Clafer translator with system calls. In the future, the project will incorporate the Clafer translator as a library instead to ease the communication and improve robustness.

# 5. LIMITATION

The biggest limitation, from the perspective of debugging models, is detecting over-constraint but satisfiable models. Finding a single unintended instance is enough to recognize under-constraint. Detecting over-constraint but satisfiable models requires examining every instance returned by the model finding algorithm and then realizing that an instance is missing from the set for the current scope. This detection scheme is impractical for models with many instances.

ClaferIG provides little assistance for the subclass of over-constraint but satisfiable models. Neither model finding or UNSAT core are adequate at solving this problem.

The Alloy language provides the `condition` syntax for detecting this class of errors[3]. Each `condition` statement guards against one specific case of possible over-constraint. A large model may require many conditions for decent coverage. The quality of the coverage depends on the intuition and foresight of the modeller. There is no Clafer equivalent in the current state of the language. More research is needed in the future for tackling this issue.

# 6. FUTURE WORK

ClaferIG is still in its infant stage and it needs time to mature. Some features require further development. Here are some challenging areas of research worthwhile of pursuit.

## 6.1 Smart fixes

Fixing a broken model is a delicate procedure. ClaferIG's only option for fixing a model is to remove constraints. There are other options for repairing a model that should be considered.

**Listing 17: Three's company**

```
1 Crowd
2     Member → Person 3..*
3
4 abstract Person
5
6 Jack : Person
7 Janet : Person
```

Listing 17 is an unsatisfiable model, a crowd requires 3 or more people but there are only 2 people around. ClaferIG will try to relax the model by removing the lower cardinality constraint of `Member`. The definition changes from `Member → Person 3..*` to the less restrictive `Member → Person 0..*`. After the change, there exists an instance with a crowd of 0, hence the model is "fixed".

The new definition of crowd does not match the real world concept. A group of 0 people is a very pathetic crowd. Intuitively, the shortage of people is the cause of unsatisfiability in Listing 17, the definition of `Crowd` is perfectly fine. A less naive algorithm will fix the model adding a third person `Chrissy : Person`, thus fulfilling the requirements imposed by `Crowd`. Improving the current fix algorithm is an area of that needs attention.

## 6.2 Tightening scope

The ideal `minimumScope` function is not achievable. The goal is to narrow the gap between the implemented scope analysis algorithm and the hypothetical, ideal function.

The current algorithm should be extended to take constraints into consideration. Constraints that begin with a quantifier affect the minimum scope in a way that is amenable to analysis. Listing 18 requires at least 6 `AudioSystem` and at least 3 `CDPlayer` in the instance. The analysis is more complex when the quantified expression is complicated. These types of constraints are common when specializing abstract clafers and modelling variability and they should not be ignored. Other common constraints may affect the scope, and requires further research.

**Listing 18: Automotive audio system**

```
1 abstract AudioSystem
2     CDPlayer ?
3
4 LuxuryCar 3..*
5     audio : AudioSystem
6         [ some CDPlayer ]
7
8 CheapCar 3..*
```

```
9       audio : AudioSystem
```

## 6.3 Isomorphism

Two instances are isomorphic if they are semantically equivalent, but may have a different syntax due to structural reordering. Listing 19 and Listing 20 are examples of an isomorphic pair.

**Listing 19: Isomorphic instance 1**

```
1 Party
2       Person1
3           Leader
4       Person2
```

**Listing 20: Isomorphic instance 2**

```
1 Party
2       Person1
3       Person2
4           Leader
```

The model finding algorithm should filter out extra isomorphic instances. It is a waste of time for the modeller to examine two isomorphic instances since they are the identical answer, just rephrased differently. Kodkod implements symmetry breaking, an optimization that reduces isomorphic instances as a side effect[5]. In general, Kodkod and ClaferIG may return many isomorphic solutions despite the optimizations.

The Clafer instance isomorphism problem is similar to the graph isomorphism problem with two exceptions: objects in the Clafer instance are named; Clafer exhibits parent-child structure between non-reference objects. However, Listing 21 shows how to embed a graph into Clafer such that detecting isomorphism in the graph is equivalent to detecting isomorphism in the corresponding Clafer instance. With this reduction, Clafer isomorphism is at least as hard as graph isomorphism in terms of complexity class.

**Listing 21: Graph reduction**

```
1 Node *
2       Edge -> Node *
```

The best known algorithm for checking graph isomorphism runs in exponential time[5]. An algorithm for Clafer isomorphism will run in exponential time as well, barring a huge breakthrough.

$$isomorphic?(instanceA, instanceB) = truthvalue \quad (4)$$

One possible implementation of the `isomorphic?` function is to modify a graph isomorphism algorithm to take the name of nodes into consideration. The function would be applied between all possible combinations of instances to filter out all isomorphic duplicates. This approach is prohibitively slow.

Another approach is to transform every instance into canonical form, where every isomorphic instance has the same form. Filtering out isomorphic duplicates can be done with string comparison in a hash set. References are a challenge for canonizing instances. A possible approach is to ignore the edges created by references, and treat the instances as trees. Canonize the trees by sorting nodes by size, breaking ties with string comparison on node names. This algorithm approximates the canonical form by ignoring references, and will work poorly on instances where references are minimal. On the positive side, it is a very fast algorithm. Perhaps it can be extended with some clever insight to better approximate instances with references.

This area needs further investigation. The desire to remove isomorphic duplicates is matched by performance concerns. A compromise is possible if an approximation algorithm can remove the majority of isomorphic duplicates without costing too much performance.

# 7. CONCLUSION

ClaferIG plays an important role in the Clafer ecosystem, and a necessary step to expand the usefulness of the language. The paper focuses on its use for debugging common modelling errors. Model finding targets under-constraint errors by searching for unintended instances. UNSAT core is useful for debugging over-constraint and unsatisfiable models. ClaferIG will attempt to fix the model based on information from the UNSAT core. The `minimumScope` approximation algorithm works well with the hierarchical nature of Clafer, and extendible to some cases of constraints. The class of over-constraint but unsatisfiable models are an issue for the tool. The paper proposes a few challenging directions to extend the power ClaferIG.

# 8. REFERENCES


[1] K. Bak. Clafer: a unified language for class and feature modeling. Technical report, Generative Software Development Lab, April 2010.

[2] K. Bak. Optimized translation of clafer models to alloy. Technical report, Generative Software Development Lab, July 2011.

[3] D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, April 2002.

[4] E. Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications*. PhD thesis, Massachusetts Institute of Technology, February 2009.

[5] E. Torlak and D. Jackson. The design of a relational engine. Technical report, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, September 2006.

[6] E. Torlak and D. Jackson. Kodkod: a relational model finder. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'07, pages 632–647, Berlin, Heidelberg, 2007. Springer-Verlag.