Solving Clafer Models with Choco

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

ABSTRACT

The Clafer modelling language relies on Alloy as its backend solver. Although Alloy is a good target language for the Clafer compiler because of the strong similarities between the two languages, it suffers in the presence of large integers. This paper explains the implementation of a new Clafer solver based on the Java constraint programming library called Choco. The solver needs to capture the semantics of the language including the hierarchy structure, type inheritance, references, primitives, and constraints. Performance of the solvers is the main metric for comparison.

1. INTRODUCTION

Clafer (class, feature, reference) is a lightweight modelling language supporting both class modelling and feature modelling[1]. The current implementation compiles Clafer models to Alloy where it is the subject of automatic analysis[3]. The analysis can perform satisfiability checks, instantiate models, and find optimal configurations[4]. Alloy and Clafer have similar semantics, thus Alloy is a good target language for compilation. In fact, Clafer's semantics is heavily inspired by Alloy's semantics[1]. The Clafer language and clafer2alloy compiler were designed and implemented in parallel. New Clafer solvers will have to reproduce Alloy-like semantics in many places.

A Clafer model is a hierarchy of *clafers* and *constraints*. The hierarchy is described by the indentation in the syntax, as shown in Listing 1. `Car`, `Engine`, `SteamEngine`, `GasolineEngine`, `ElectricEngine`, `Gastank`, and `Wheel` are the clafers in the model, whereas the expression `[some Gastank]` is the constraint. Any expression within square brackets is a constraint. The clafers `Engine`, `Gastank`, and `Wheel` all belong to the `Car` clafer hence the three clafers are nested at a higher indentation level than `Car`. In Clafer terminology, `Car` is the parent clafer and the others are child clafers. Likewise, `SteamEngine`, `GasolineEngine`, and `ElectricEngine` are the children of the `Engine` clafer.

Listing 1: Car

```
Car
    xor Engine
        SteamEngine
        GasolineEngine
            [some Gastank]
        ElectricEngine
    Gastank ?
    Wheel 4..8
```

`xor` is a group cardinality keyword that forces a choice on `Engine`. It must choose exactly one child and exclude the other two. The `?` after `Gastank` is a cardinality keyword for optional. The syntax `Gastank ?` is rewritten as `Gastank 0..1` by the compiler, which states that every `Car` has between 0 to 1 `Gastank`. Likewise, it has between 4 to 8 `Wheel`. If the cardinality is not explicitly stated in the syntax, then it defaults to `1..1`.

The constraint `[some Gastank]` asserts a condition within our model, but is only enforced if its parent clafer `GasolineEngine` is present. In plain English, the constraint says that there must at least one `Gastank` in the instance if `GasolineEngine` is in the instance, thus no gasoline-powered cars without tanks. However the reverse is not necessarily true, the model allows for cars with tanks powered by a non-gasoline source.

Listing 2 is an *instantiation/instance/solution* of the model. An instance must follow all the conditions stated by the original model such as group cardinality, cardinality, and constraints. For example, removing `Wheel3` from the instance would violate the lower cardinality of the `Wheel` clafer in the model.

Listing 2: Car instance

```
Car
    Engine
        GasolineEngine
    Gastank
    Wheel0
    Wheel1
    Wheel2
    Wheel3
```

The `Wheel` clafers in the instance are numbered simply to differentiate them. For example, renaming `Wheel0` to `Wheel5`

and `Wheel1` to `Wheel0` does not change the semantics of the instance. The instance in Listing 3 is *isomorphic* to the instance in Listing 2.

Listing 3: Car isomorphic instance

```
Car
    Engine
        GasolineEngine
    Gastank
    Wheel5
    Wheel0
    Wheel2
    Wheel3
```

A model may have many non-isomorphic instances. A model with no instances is *unsatisfiable* or *inconsistent*[2], and indicates a bug in the model. Generating instances is useful for catching over-constraint[3].

Clafer implements integers and type inheritance. In Listing 4, `Feature` is an *abstract* clafer that does not appear in the instantiation, its only purpose is for other clafers to extend. Every `Feature` has a `Cost` and every `Cost` has a *reference* to an integer. `SteamEngine`, `GasolineEngine`, `ElectricEngine`, `Gastank`, and `Wheel` extend `Feature` and thus they all have a `Cost`.

Listing 4: Car features

```
abstract Feature
    Cost -> integer

Car
    xor Engine
        SteamEngine : Feature
        GasolineEngine : Feature
            [some Gastank]
        ElectricEngine : Feature
    Gastank : Feature ?
    Wheel : Feature 4..8
```

Listing 5 is one instance of the new model. Note that the cost of the features are not constrained in the model, hence they can take on any value. The model is a good example of an attributed feature model in Clafer[7]. If the cost of the features are known, then they can be configured or fixed in the model with constraints.

Listing 5: Car features instance

```
Car
    Engine
        SteamEngine
            Cost = 4
    Wheel0
        Cost = 3
    Wheel1
        Cost = 2
    Wheel2
        Cost = 100
    Wheel3
        Cost = -234
```

Listing 6 fixes a cost to every feature in the model. For example, `[this.Cost.ref = 34]` is a constraint that forces the cost to refer to `34`. In general, the expression can be simplified as `[this.Cost = 34]` and the compiler will automatically rewrite it as `[this.Cost.ref = 34]` during type checking. The "." is the join operator in relational logic. Joining `this` with `Cost` will return the `Cost` clafer belonging to "this" clafer. Joining that `Cost` with `ref` will return the integer value. Joins are typically used to navigate down the hierarchy. Clafer supports a special join called the parent join for navigating in the reverse.

Listing 6: Car product line

```
abstract Feature
    Cost -> integer

Car
    xor Engine
        SteamEngine : Feature
            [this.Cost.ref = 34]
        GasolineEngine : Feature
            [some Gastank]
            [this.Cost.ref = 25]
        ElectricEngine : Feature
            [this.Cost.ref = 32]
    Gastank : Feature ?
        [this.Cost.ref = 20]
    Wheel : Feature 4..8
        [this.Cost.ref = 9]

TotalCost -> integer
[TotalCost = sum Feature.Cost.ref]

<<min TotalCost>>
```

Let's take a closer look at the constraints in the previous model. Constraint expressions are based on relational logic and (almost) every expression evaluates to a set. `this` expression evaluates to a singleton set containing the current clafer. `this.Cost.ref` will evaluate to a singleton set containing the integer value of the cost of this. The equality constraint `this.Cost.ref = 34` states that the expression `this.Cost.ref` equals the singleton set containing `34`[1]. In fact, even clafers are sets. The `Wheel` clafer for example is a set of size 4 containing `Wheel0`, `Wheel1`, `Wheel2`, and `Wheel3` in Listing 5. This will have large ramifications later in the paper.

Listing 6 is an example of a car manufacturing product line. We can run the model to generate all possible instances of the model. Each instance corresponds to a specific configuration of a product, and the `TotalCost` stores the final cost of the car. A keen buyer wants to minimize spending, so his objective is to minimize the total cost of the car. This objective is denoted as ≪`min TotalCost`≫ in the model. Under this minimization objective, the analyzer must find the instance with the lowest total cost. Clafer supports multiple objectives, where the solution will be the set of Pareto-optimal instances[7].

[1]Perhaps it would make more sense mathematically for the syntax to say `this.Cost.ref = {34}`

Clafer supports first-order expressions, and first-order logic is famously undecidable. To cope with undecidability, everything in Clafer is bounded. Integers have a lower and upper bound. Clafers are bounded by scopes. Scope dictates the maximum number of times a Clafer can appear in the solution. For example, if $Scope(Wheel) = 3$, then `Wheel` can appear at most 3 times in the instance, hence the instance in Listing 5 is not allowed. Scopes are automatically computed using a heuristic based on integer programming.

2. SOLVER

The current implementation only supports Alloy as the backend solver for analyzing Clafer models. The jump from Clafer to Alloy is not too large. A clafer in the Clafer model corresponds to a signature in the Alloy model. Logic in Alloy is also relational, thus the semantics between the two languages are very alike. Alloy has its own backend called Kodkod, a bounded constraint solver for relation logic. Kodkod in turn compiles the problem down to a boolean satisfiability problem and relies on SAT solvers such as Minisat or SAT4J to solve the model. One problem with this stack of solvers is the handling of integers. A Clafer model after a series of steps is ultimately translated down to a boolean satisfiability problem, so all the semantics of arithmetic need to be encoded as well. The approach is called "bit-blasting"[8] and the main drawback is that it does not scale well for large integers in both time and space. One work around is to scale down all integers by a constant factor and rounded, but precision is lost in the process.

The specification of Clafer does not forbid other solvers aside from Alloy. Having a choice of solvers is healthy for the Clafer universe since certain solvers have certain relative strengths. A Clafer model can target a solver that best suits its characteristics. Since large integers are a big weakness with the current solver[2], a new solver should cover this weakness.

This paper details a Clafer solver using Choco[3], a Java library for constraint programming[5]. A few features of the library is well suited for the implementation.

Sets: The library supports variables over sets of integers. In Clafer, almost everything is a set, so set support is vital to a good encoding.

Integers: Integer variables can have large bounds without draining performance in many situations.

Objective:[4] Choco can solve optimization problems by either minimizing or maximizing an objective variable, hence useful for finding optimal attributed feature models. Unfortunately, it does not support optimizing multiple variables natively, although this can be remedied by a manual implementation of the guided improvement algorithm.

Custom constraints and operators: Choco can handle custom constraints and operators. Some of the Clafer semantics might not translate efficiently to the current set of supported constraints and operators, so custom ones may be needed for performance purposes.

3. IMPLEMENTATION

The paper is example driven, where at each step, it will show an example model plus the details of the encoding. Given a model, the solver must enumerate all valid instances up to the scope or claim unsatisfiability. For the remainder of the paper, the scopes are given as part of the input in the comments[5]. Some examples will set the scope higher than necessary purely for demonstration. To keep the details concise, the following notation is employed.

$$\begin{aligned}
\mathbb{Z} &= \{..., -2, -1, 0, 1, 2, ...\} \\
\mathbb{Z}_{m,n} &= \{m, m+1, m+2, ..., n\} \\
\mathbb{P} &= \{0, 1, 2, ...\} \\
\mathbb{P}_n &= \{0, 1, 2, ..., n\} \\
\mathbb{P}_{m,n} &= \mathbb{Z}_{m,n} \quad \text{where } m, n \geq 0
\end{aligned}$$

Names of set variables will be written in **bold** and names of integer variables will be written in *italics*. Here are some examples using the notation.

Define a set variable "mySet" that is a subset of $\{0, 1, 2, 3\}$.

$$\mathbf{mySet} \subseteq \mathbb{P}_3$$

Assign the set to an exact value.

$$\mathbf{mySet} = \{1, 3\}$$

Two vertical bars retrieve the size of the set.

$$|\{1, 3\}| = 2$$

Define an integer variable "myInteger" that is an element of $\{0, 1, 2, 3\}$.

$$myInteger \in \mathbb{P}_3$$

Assign the set to an exact value.

$$myInteger = 2$$

Define an array of 5 integer variables that are elements of $\{0, 1, 2, 3\}$.

$$myIntegerArray \in \mathbb{P}_3^5$$

[2]My rule of thumb is that a bit-width of 11 (ie. range of [-1024, 1023]) is large for Alloy but still manageable for simple models. A bit-width of 14 (ie. range of [-8192, 8192]) is when it becomes a challenge. Even near-trivial models will take a few minutes to solve. Larger models will run out of memory.

[3]Choco project's homepage: http://www.emn.fr/z-info/choco-solver/

[4]Moolloy implements the guided improvement algorithm for Alloy.

[5]Clafer comments are prefixed by two slashes "//" like in Java or C.

This paper will go over how the Clafer structure is implemented along with the important expressions. It will not cover all expressions, there are simply too many for the scope of this paper.

3.1 Hierarchy

The hierarchy structure is pervasive in Clafer models and it needs to be captured in the encoding to Choco. Listing 7 is a small model with `Car` as the parent and `Wheel` as the child.

Listing 7: Simple model

```
// Scope(Car) = 3, Scope(Wheel) = 9
Car
    Wheel 4..8
```

Remember, clafers are sets. The solver needs two set variables for the two clafers.

$$\mathbf{Car} \subseteq P_2$$
$$\mathbf{Wheel} \subseteq P_8$$

Since the scope of `Car` is 3, the solution can have up to 3 cars.[6] $\texttt{Car}_\texttt{i}$ is in the solution if and only if $i \in \mathbf{Car}$. For example, if $\mathbf{Car} = \{2\}$ then the solution contains `Car2`.

Each `Car` is the parent to a set of `Wheel` and this is encoded with an additional set per possible `Car`.

$$\mathbf{Car_0Wheel}, \mathbf{Car_1Wheel}, \mathbf{Car_2Wheel} \subseteq P_8$$
$$\mathbf{Wheel} = \bigcup_{i=0}^{2} \mathbf{Car_iWheel}$$
$$(\bigcap_{i=0}^{2} \mathbf{Car_iWheel}) = \emptyset$$

$\mathbf{Car_iWheel}$ denote the children of Car_i. For example, if $\mathbf{Car_1Wheel} = \{2, 4, 5, 7\}$, then `Wheel2`, `Wheel4`, `Wheel5`, and `Wheel7` must lie directly under `Car1` in the solution. The two last constraints state that **Wheel** is partitioned by $\mathbf{Car_iWheel}$ because every `Wheel` must belong to exactly one `Car`. However, these child sets are constrained by the cardinality.

$$\bigwedge_{i=0}^{2} (i \in \mathbf{Car} \implies 4 \leq |\mathbf{Car_iWheel}| \leq 8)$$
$$\bigwedge_{i=0}^{2} (i \notin \mathbf{Car} \implies |\mathbf{Car_iWheel}| = 0)$$

[6] Intuitively, every solution has exactly one car so the scope given is higher than necessary. Choco will come to the same conclusion during solving.

3.1.1 Root

Top level clafers are special in the hierarchy because they are the only clafers to not have parents. A Clafer model can have many top level clafers and they are all treated differently from non-top level clafers. To minimize the number of special cases, the compiler adds a new clafer called `Root` as the new highest level clafer. Listing 8 is how the model is represented internally in the compiler.

Listing 8: Simple desugared model

```
// Scope(Root) = 1
// Scope(Car) = 3, Scope(Wheel) = 9
Root 1..1
    Car 1..1
        Wheel 4..8
```

Since `Car` is now a child, the rules in the previous section apply to it.

$$\mathbf{Root} \subseteq P_0$$
$$\mathbf{Car} \subseteq P_2$$
$$\mathbf{Root_0Car} \subseteq P_3$$
$$\mathbf{Car} = \bigcup_{i=0}^{0} \mathbf{Root_iCar}$$
$$(\bigcap_{i=0}^{0} \mathbf{Root_iCar}) = \emptyset$$
$$\bigwedge_{i=0}^{0} (i \in \mathbf{Root} \implies 1 \leq |\mathbf{Root_iCar}| \leq 1)$$
$$\bigwedge_{i=0}^{0} (i \notin \mathbf{Root} \implies |\mathbf{Root_iCar}| = 0)$$

Add one constraint for the special top level clafer `Root`.

$$\mathbf{Root} = \{0\}$$

The original simple model is now encoded in Choco. The following is an example a solution Choco will return.

$$\mathbf{Root} = \{0\}$$
$$\mathbf{Root_0Car} = \{0\}$$
$$\mathbf{Car} = \{0\}$$
$$\mathbf{Car_0Wheel} = \{0, 1, 2, 3\}$$
$$\mathbf{Car_1Wheel} = \emptyset$$
$$\mathbf{Car_2Wheel} = \emptyset$$
$$\mathbf{Car_3Wheel} = \emptyset$$
$$\mathbf{Wheel} = \{0, 1, 2, 3\}$$

The solution maps to the instance in Listing 9.

Listing 9: Simple desugared instance

```
Root0
    Car0
        Wheel0
        Wheel1
        Wheel2
        Wheel3
```

3.2 Joining with children

Suppose the model allowed more than one car and there exists a hypothetical expression named "FastCar" that evaluates to $\{1, 2\}$, hence `Car1` and `Car2` are fast cars. How can we retrieve the wheels belonging to the fast cars, ie. compute the join expression $FastCar.Wheel$? Mathematically, the join is computed like the following.

$$\bigcup_{i \in \mathbf{FastCar}} \mathbf{Car_i Wheel}$$

The issue is that Choco does not support iterating over sets in such a manner. The straightforward solution is to test for each element individually. Note that $\mathbf{FastCar} \subseteq \mathbf{Car} \subseteq \mathbb{P}_2$.

$$\bigcup_{i=0}^{2} (if\ i \in \mathbf{FastCar}\ then\ \mathbf{Car_i Wheel}\ else\ \emptyset)$$

The above equation[7] is more or less how joining with children is implemented. It tests all possible `Car` clafers, and checks to see if they are part of $FastCar$. Many other Clafer expressions also work with sets and often times can be implemented with the same strategy of testing all the possible values in the set.

3.2.1 Working with large bounds

The strategy in the previous example worked because the domain of the elements in $FastCar$ is small. Every possible element is in $\mathbb{P}_2$, so it is cheap to test for each one because $|\mathbb{P}_2|$ is small. Elements with large domains, such as integers, do not work well with this strategy. Remember, one of the goals of the solver is to support large integers. If the integers have a bound of [-1000000,1000000], then testing each possible integer in this range is not feasible. This is not a problem for joins since joins cannot be from integers. However, we still need a strategy for dealing with sets of integers, or sets of other large domain elements.

The following strategy requires some static analysis from the compiler. Every set has a size. If the compiler can prove bounds on the size of a set, then it is possible to retrieve the elements as integer variables to work on. For example, if $|FastCar| = 2$:

$$elements \in \mathbb{P}_2^2$$

$$allDifferent(elements) \wedge (\bigwedge_{i=0}^{1} elements[i] \in FastCar)$$

$$FastCar.Wheel = \bigcup_{i=0}^{1} Car_{elements[i]} Wheel$$

Sometimes the exact size of an expression cannot be determined at compile time. In general, the compiler can determine the lower bound and upper bound of the size of an expression because Clafers and primitives are bounded by scopes. For example, suppose the analysis can determine that $|FastCar|$ is between 1 and 2. Then the implementation checks with a series of implications and conjunctions whether the size is indeed 1 or 2 and does the appropriate operations given the size.

This second approach only makes sense if the difference between provable lower and upper bounds of $|FastCar|$ is small. In the case for sets of Clafer integers, the difference is significantly smaller than the domain of integers.

3.3 Parent

Joins are normally used to traverse down from parents to children. Clafer allows traversing in the reverse direction. For example, the expression `Wheel.parent` will take the set of wheels, then navigate to their parent clafers. To implement parent joins, parent "pointers" are stored in an array of integer variables. The examples in this section describe how to implement parent pointers between `Wheel` and `Car`, but the rules also apply to `Car` and `Root`.

$$WheelParent \in \mathbb{P}_2^9$$

The $WheelParent$ array stores the id of the wheel's parents. $j = WheelParent[i]$ means that $\mathtt{Wheel_i}$ is under $\mathtt{Car_j}$. How does Choco keep the parent pointers consistent with the rest of the model? With the "inverseSet" constraint. The definition below is taken from the official documentation.[8]

$inverseSet(\langle x_0, .., x_n \rangle, \langle \mathbf{y_0}, .., \mathbf{y_m} \rangle)$ states that x_i has value j if and only if $\mathbf{y}_j$ contains value i:

$$x_i = j \iff i \in \mathbf{y_j}, \quad \forall i = 0...n, j = 0...m$$

Note that by the definition, the sets $\mathbf{y_1}, .., \mathbf{y_m}$ must fully partition the set $\mathbb{P}_n$ The constraint is applied like the following.

$$\mathbf{Car_{unused}Wheel} \subseteq \mathbb{P}_2$$

$$inverseSet(WheelParent, \langle \mathbf{Car_0 Wheel}, \mathbf{Car_1 Wheel}, \mathbf{Car_2 Wheel}, \mathbf{Car_{unused} Wheel} \rangle)$$

[7] The if-then-else is an expression like in functional languages, or the ternary operator in Java.

[8] There are a few mistakes in the official documentation's definition. The definition here corrects those mistakes.

The list of sets in the second argument must fully partition $\mathbb{P}_8$, hence $\mathbf{Car_{unused}Wheel}$ is required to hold all the wheels that do not appear in $\mathbf{Car_0Wheel}$, $\mathbf{Car_1Wheel}$, nor $\mathbf{Car_2Wheel}$. For example, in Listing 9, $WheelParent = \langle 0,0,0,0,3,3,3,3,3\rangle$ where 3 means that it is not used in the solution. Also note the constraint $(\bigcap_{i=0}^{2} \mathbf{Car_iWheel}) = \emptyset$ introduced early on is not needed anymore since the inverseSet constraint enforces a partition and hence are disjoint.

Mathematically, joining on parents looks like the following equation, although the implementation is more complicated. Suppose the expression is $MyWheels.parent$.

$$MyWheels.parent = \{WheelParent[i] \mid i \in MyWheels\}$$

3.3.1 Isomorphism

Aside from joining with parents, the parent pointers have other uses, the most important one is to break symmetry. Intuitively, Listing 8 has exactly 5 solutions: 1 Car with either 4, 5, 6, 7, or 8 wheels. With the current translation, Choco returns 1143 solutions for that model. Here is an example of one solution from the 1143.

Listing 10: Simple isomorphic instance

```
Root0
    Car0
        Wheel0
        Wheel2
        Wheel3
        Wheel4
```

Choco is returning isomorphic instances. Even a simple model like Listing 8 has too many isomorphic solutions. In this case, every solution has on average over 200 solutions isomorphic to it. Reducing isomorphic instances is important for at least two reasons. Firstly, it hurts the performance of the solver. Secondly, isomorphic solutions are useless to a human modeller. It is a waste of time to read the same solution more than once.

The solution is to add a constraint to force the parent pointer arrays to be sorted. For example, the parent pointer for `Wheel` in Listing 10 is $WheelParent = \langle 0,3,0,0,0,3,3,3,3\rangle$, so the new constraint disallows this case. In fact, for any instance where `Car0` has 4 `Wheel`, the first 4 values in the parent pointer array $WheelParent$ must be 0. Listing 9 is valid since $WheelParent = \langle 0,0,0,0,3,3,3,3,3\rangle$ in this case.

$$isSorted(WheelParent)$$
$$isSorted(CarParent)$$

These constraints force an ordering on how parents choose children. The lowest index parent must choose the children with the lowest indices. The next lowest index parent must choose the children with the next lowest indices. With the new constraints, Choco only returns 5 solutions as expected, eliminating the other 1138 isomorphic solutions.

3.4 Inheritance

There are several ways to encode abstract clafers into Choco. The best way is to encode them the same as all the other clafers, the uniformity simplifies the implementation of expressions.

Listing 11: Simple feature model

```
// Scope(Root) = 1, Scope(Car) = 3,
// Scope(Gastank) = 2, Scope(Wheel) = 9
// Scope(Feature) = 11, Scope(Cost) = 13
abstract Feature
    Cost
Root
    Car
        Gastank : Feature ?
        Wheel : Feature 4..8
```

In Listing 11, `Feature` is compiled using the previous rules. Abstract clafers are also a top level clafer like the `Root`.

$$\begin{aligned}
\mathbf{Feature} &\subseteq \mathbb{P}_{10}\\
\mathbf{Feature_0Cost} &\subseteq \mathbb{P}_{12}\\
\mathbf{Feature_1Cost} &\subseteq \mathbb{P}_{12}\\
&\dots\\
\mathbf{Feature_{10}Cost} &\subseteq \mathbb{P}_{12}\\
\mathbf{Feature_{unused}Cost} &\subseteq \mathbb{P}_{12}\\
\mathbf{Cost} &\subseteq \mathbb{P}_{12}\\
\mathbf{Cost} &= \bigcup_{i=0}^{12} \mathbf{Feature_iCost}\\
CostParent &\in \mathbb{P}_{12}^{10}\\
inverseSet(CostParent, \langle &\mathbf{Feature_0Cost},\\
&\mathbf{Feature_1Cost},\\
&\dots,\\
&\mathbf{Feature_{10}Cost},\\
&\mathbf{Feature_{unused}Cost}\rangle)\\
isSorted(CostParent)&
\end{aligned}$$

Suppose the expression $FreeFeature$ evaluates to a set of `Feature`, then the expression $FreeFeature.Cost$ is compiled using the rules in section 3.2. However, if $FreeWheel$ is a set of `Wheel`, the the expression $FreeWheel.Cost$ cannot be compiled the same way because `Cost` is a direct child of `Feature`, not `Wheel`.

`Feature` is abstract and cannot be instantiated without a concrete subclafer. The model has exactly two subclafers that partition `Feature`. Every `Feature` is represented twice in the model, once as its concrete type, and another as its abstract type. Suppose the correspondence is as follows.

$$\texttt{Gastank0} = \texttt{Feature0}$$
$$\texttt{Gastank1} = \texttt{Feature1}$$
$$\texttt{Wheel0} = \texttt{Feature2}$$
$$\texttt{Wheel1} = \texttt{Feature3}$$
$$\ldots$$
$$\texttt{Wheel8} = \texttt{Feature10}$$

`Gastank` reserves $\mathbb{P}_{0,1}$ on `Feature`, and `Wheel` reserves $\mathbb{P}_{2,10}$. There cannot be any "holes" in the reservation, ie. `Gastank` cannot reserve $\{1, 3\}$. The assignment is somewhat arbitrary, `Wheel` could occupy the lower indices of `Feature`, instead of `Gastank`. The relationship is reified with constraints.

$$\bigwedge_{i \in \mathbb{P}_{0,1}} i \in \textbf{Gastank} \iff i \in \textbf{Feature}$$
$$\bigwedge_{i \in \mathbb{P}_{0,8}} i \in \textbf{Wheel} \iff i + 2 \in \textbf{Feature}$$

Before performing the join in $FreeWheel.Cost$, the set of `Wheel` needs to be converted to a set of `Feature` by upcasting the subexpression $FreeWheel$. This is accomplished by offsetting the numbers in $FreeWheel$ by 2. For example, suppose $FreeWheel = \{3, 5\}$, ie. `Wheel3` and `Wheel5` are free. The solver converts this to a set called $FreeWheelFeature = \{5, 7\}$, ie. a set containing `Feature5` and `Feature7`. Note that the two expressions, $FreeWheel$ and $FreeWheelFeature$, refer to the same clafers. The join $FreeWheel.Cost$ is then replaced by $FreeWheelFeature.Cost$.

3.5 Reference

A reference can either point to another clafer or a primitive. This section will cover the implementation of referencing to integers, but references to clafers are almost identical in implementation. For the sake of simplicity, assume that integers are bounded between [-99, 99] in this section. In general, the bounds can be much larger.

Listing 12: Simple attributed feature model

```
// Scope(Root) = 1, Scope(Car) = 3,
// Scope(Gastank) = 2, Scope(Wheel) = 9
// Scope(Feature) = 11, Scope(Cost) = 13
abstract Feature
    Cost -> integer
Root
    Car
        Gastank : Feature ?
        Wheel : Feature 4..8
```

In Listing 12, every `Cost` points to an integer value. Since the scope of `Cost` is 13, the solver needs to reserve 13 integers.

$$CostRef \in \mathbb{Z}^{13}_{-99,99}$$

The intuition is that $CostRef[0]$ stores the reference for `Cost0` and so on. Any `Cost` that does not exist in the solution must have their reference "zeroed" to avoid problems with generating solutions. For example, if `Cost9` does not exist in the solution and $CostRef[9]$ is not zeroed, then the solver can add 1 to $CostRef[9]$ and claim it as a new solution.

$$\bigwedge_{i=0}^{12} i \notin \textbf{Cost} \implies CostRef[i] = 0$$

Suppose $MyCosts$ is some expression that evaluates to a subset of `Car`. Mathematically, the join $MyCosts.ref$ looks like the following equation, which has an uncanny resemblance to joining on parents.

$$MyCosts.ref = \{CostRef[i] \mid i \in MyCosts\}$$

3.6 Constraints

Constraints in Clafer are nested and only apply if the parent exists.

Listing 13: Nested constraint

```
\\ Scope(Car) = 3
abstract Feature
    Cost -> integer
Car : Feature
    [this.Cost.ref = 10000]
```

In Listing 13, the constraint $[this.Cost.ref = 10000]$ only holds for the cars that exists in the instance. The constraint is compiled like the following.

$$\bigwedge_{i=0}^{2} i \in \textbf{Car} \implies compileExpression(\{i\}.Cost.ref = 10000)$$

Note that each instance of "this" in the syntax is replaced with the singleton set $\{i\}$. The function *compileExpression* compiles a Clafer expression into the corresponding Choco expression.

3.7 Tools

The ideas here are implemented in the Clafer compiler written in Haskell.[9]. This new project implements a new backend that targets Choco. Given a Clafer model as input, the backend will output a Javascript file that builds a Choco

[9]The Clafer compiler is an open source project available here: https://github.com/gsdlab/clafer. The project is in the "choco" branch, it has not been merged into the stable branch yet.

model when executed. It performs the translation and optimizations detailed in this paper.

The second part of the project is the Choco solver that runs this Javascript output file[10]. The solver has two parts. The first part implements a Javascript library file that implements joins, expressions, and constraints which the Javascript output of the Clafer compiler can take advantage of. The second part is written in Java that initializes Choco, and prepares a Javascript interpreter before feeding it the library file and the compiler's output file. After the Javascript interpreter terminates, the Choco model is ready and solved. One disadvantage of using a Javascript interpreter is the overhead that it adds to benchmarks, around half a second on my computer.

4. OPTIMIZATION

The most important optimization is the symmetry breaking rule in section 3.3.1. Other optimizations are detailed in this section.

4.1 Fixed size set

One of the major inefficiencies in the encoding is working with set variables in during the translation of Clafer constraints. Most of the useful operators in Choco only work with integer variables. Many expressions are implemented by transferring elements from a set variable to integer variables, performing some work on the integer variables, then transferring the result back to a new set variable.

The issue is that Clafer expressions are almost always sets, hence the prolific use of set variables. However, if the compiler can prove that an expression is of a fixed size n, then it can use an array of n integer variables to represent the set instead.

Listing 14: Fixed size set model

```
Person
    Age -> integer
        [this.ref = 3]
```

In Listing 14, the constraint is equating two sets: the set $this.ref$ equals the set $\{3\}$. Normally, the encoding will create sets for all 3 expressions: $this$, $this.ref$, and 3. An optimized encoding will take advantage of the fact that the expressions are of fixed size, determined at compile time.

$$[AgeRef[this]] = [3]$$

The above constraint is significantly more efficient implementation of the constraint in Listing 14. In fact, it avoids the use of set variables all together.

[10]The Choco solver is an open source project available here: https://github.com/gsdlab/chocosolver. The name is perhaps misleading. It does not solve general Choco models, it only solves Clafer Choco models outputted by the Clafer compiler.

4.2 Reference uniqueness

References in Clafer have a condition not mentioned yet in this paper. References are unique under a parent.[11]

Listing 15: Reference model

```
// Scope(Person) = 2
// Scope(FavouriteNumber) = 8
Person 1..*
    FavouriteNumber -> integer 1..*
```

In Listing 15, each person cannot have an integer listed twice under his or her favourite numbers. Two different people can have the same favourite number though. For example, the instance in Listing 16 is permitted. However, changing `FavouriteNumber2 = 6` to `FavouriteNumber2 = 3` is not allowed because `FavouriteNumber0 = 3`.

Listing 16: Reference model instance

```
Person0
    FavouriteNumber0 = 3
    FavouriteNumber1 = 4
    FavouriteNumber2 = 6
Person1
    FavouriteNumber3 = 3
    FavouriteNumber4 = 4
    FavouriteNumber5 = 6
```

The Alloy encoding will make the constraint explicit in the model, using expressions within the language. The input model looks like Listing 15, but internally, the models looks more like Listing 17 during the translation to Alloy.

Listing 17: Internal reference model

```
// Scope(Person) = 2
// Scope(FavouriteNumber) = 8
Person 1..*
    FavouriteNumber -> integer 1..*
    [all disj x;y : this.c2_FavouriteNumber
        | x.ref != y.ref]
```

The constraint takes advantage of the `all` quantifier in the language. In the Choco encoding, there is a more efficient way of implementing the uniqueness constraint using the parent pointer arrays. For example, the parent pointer array for `FavouriteNumber` in the instance in Listing 16 is $FavouriteNumberParent = [0,0,0,1,1,1,2]$. For any two `FavouriteNumber` under the same parent that is not unused, their references must be different.

$$\bigwedge_{i=0}^{1}\bigwedge_{j=0}^{7}\bigwedge_{k=j+1}^{7}(FNP[j] = FNP[k] \implies FNR[j] \neq FNR[k])$$

The abbreviations are $FNP \equiv FavouriteNumberParent$ and $FNR \equiv FavouriteNumberRef$. The above constraint enforces the uniqueness constraint as described.

[11]Clafer also supports references without this constraint.

4.3 Smaller domains

Listing 18 is identical to an earlier example except the scope of `Wheel` has been increased.

Listing 18: Simple model again

```
// Scope(Car) = 3, Scope(Wheel) = 18
Car
    Wheel 4..8
```

The encoding produces set variables for the parent-child `Car-Wheel`.

$$\mathbf{Car_0Wheel}, \mathbf{Car_1Wheel}, \mathbf{Car_2Wheel} \subseteq P_{17}$$

One of the consequences of symmetry breaking in section 3.3.1 is that `Car0` has `Wheel` of lowest indices, than `Car1` is next. The symmetry breaking between `Root` and `Car` implies that a $\texttt{Car}_\texttt{i}$ of a higher index cannot exist unless all the $\texttt{Car}_\texttt{j}$ with a lower index exists. For example, `Car1` cannot exist if `Car0` does not exist (again, this is a consequence of having *CarParent* being sorted). With this in mind, it is possible to reduce the domains of the set variables declared above.

$$\mathbf{Car_0Wheel} \subseteq P_7$$

`Car0` can have at most 8 `Wheel` under it due to cardinality, and it must pick the ones with the lowest 8 indices, ie. it cannot have `Wheel8` as a child. If `Car1` exists than so does `Car0`. $\mathbf{Car_1Wheel}$ must have higher indices than $\mathbf{Car_0Wheel}$. In the best case, `Car0` has 4 children: `Wheel0`, `Wheel1`, `Wheel2`, and `Wheel3`. The first wheel (if any) under `Car1` must be `Wheel4`. In the worst case, `Car0` has all the wheels up to `Wheel7`. In which case, the highest possible wheel under `Car1` is `Wheel15`.

$$\mathbf{Car_1Wheel} \subseteq P_{4,15}$$

Similar analysis can reduce the domain of $\mathbf{Car_2Wheel}$. Note that the upper limit of the domain is bounded by the scope of `Wheel`.

$$\mathbf{Car_2Wheel} \subseteq P_{8,17}$$

Therefore the domains have been decreased in size without changing the problem. Solvers work more efficiently when the domains are as small as possible.

5. RESULT

The Clafer structure is why the modelling language stands out against other modelling languages. Any backend solver for the language needs to efficiently handle the structure imposed by a Clafer model. The first test will test the solver's ability to solve structure.

Listing 19: Zoo

```
abstract Animal
    Head
        Eye 2
        Ear 2
        Mouth
    Age -> integer
    Torso
    Leg 4
        Feet
Cat : Animal 4
    Whiskers 6
Rhino : Animal 3
    Horn
Elephant : Animal 2
    Trunk
```

Listing 19 is a contrived example containing hierarchy, references, cardinality, and inheritance. It contains no constraints in the Clafer model. Intuitively, the model is simple because it is easy to solve manually by hand. The model is tested against the two solvers, the existing Alloy solver and the new Choco solver. The tests are performed on my personal 6-year-old duo-core laptop. Alloy tests are executed on Alloy 4.2 with Minisat as the backend SAT solver. The Alloy analyzer is given 2 gigabytes of memory.

Solver	Time to compute first solution
Alloy	15 min
Choco	2 sec

The Alloy solver performs rather poorly. However, the results are entirely different if the "flatten inheritance" optimization flag is enabled in the Clafer compiler. The flattened model solves in less than 2 seconds. Inheritance flattening is an optimization that removes abstract clafers entirely by copying their children directly under the subclafers. It is a destructive optimization because flattening the inheritance technically alters the semantics of the model. Also, the optimization is only applicable if certain conditions are met.

The next few tests are Clafer model adaptations of attributed feature models from Scalable Prediction of Non-functional Properties in Software Product Lines. The models were adapted by Rafael Olachea as part of his ClaferMoo project[12]. The tests have two criteria, how long did it take to compute the first solution, and how long did it take to optimum solution. All models have a single objective.

The Alloy "first" tests measure the time it took to compute the first solution. These tests are executed with the official Alloy 4.2 release. The Alloy "optimum" tests measure the time it took to compute the optimum solution: either the solution with the lowest footprint or the solution with the highest performance. These tests are executed with 3 Alloy extensions: Moolloy, partial instances, and sparse integers. The Alloy optimum tests are compiled with ClaferMoo rather than the standard Clafer compiler. ClaferMoo

[12]The project plus models are available at GitHub: https://github.com/gsdlab/claferMooStandalone

is a special compiler for attributed Clafer feature models, because it is a much better compiler for this subset of problems. Tests that ran out of memory will be denoted with "OOM" in the table. These feature models use large integers.

Linked list feature model

Solver	Time to compute solution	
	first	optimum
Alloy	OOM	18 sec
Choco	1.3 sec	4.7 sec

Apache web server feature model

Solver	Time to compute solution	
	first	optimum
Alloy	36 sec	0.4 sec
Choco	1.1 sec	3.6 sec

BerkeleyDB feature model

Solver	Time to compute solution	
	first	optimum
Alloy	OOM	1.7 sec
Choco	0.9 sec	2.8 sec

The Alloy optimum column performs extremely well. The translation of the models for these tests assume the input models are attributed feature models and take advantage of that fact. Thus it is optimized for these tests. The Choco solver does not make any assumptions and the attributed feature models are compiled like any other model.

The results show that Alloy 4.2 struggles with large integers. Extensions to Alloy cover its shortcomings, for feature models anyways. The Choco solver is very competitive, even against the Alloy solving strategy tailored for these feature models. Non-feature models with large integers will not perform well in Clafer with the Alloy solver.

6. RELATED WORK

Alloy relies on Kodkod as its solver. Kodkod can set lower bounds to its relation, essentially providing a partial solution as part of the input[9]. The official Alloy implementation does not expose this functionality to Alloy models. Recent work by Montaghami and Rayside expose lower bounds as "partial instances" and greatly improves the performance of solving feature models[6]. To take advantage of partial instances, the Clafer compiler would need to partially solve the model. The current Clafer compiler does not support partial instances, only the ClaferMoo project, which assumes its inputs are attributed feature models.

As for integers, Alloy and Kodkod rely on bit-blasting. Any model requiring many large integers will be a problem. An extension for sparse integers has been developed. Essentially, the extension will only account for an explicit subset of integers. The idea is that the modeller can specify only the integers that can appear in the model. The extension is an attempt to extend Alloy's effectiveness in the presence of large integers. The current Clafer compiler does not support sparse integers, only the ClaferMoo project. Sparse integers can work very well for attributed feature models. The reason is that all the integer references are assigned as part of the model. For attributed feature models, sparse integers cover Alloy's shortcomings, and perhaps the Choco solver is redundant for these models. Sparse integers does not always apply in general.

Listing 20: Unassigned integer

```
Computer : Feature
    Age -> integer
        [this.ref >= 0]
...rest of the model...
```

Suppose Listing 20 is part of a Clafer model. At the very best, the Clafer compiler can infer that $Age.ref \in \mathbb{P}$. Sparse integers will need to cover all the positive integers since it is unknown at compile time the value of $Computer.Age.ref$.

7. FUTURE WORK

One of the future goals is to build a suite of realistic Clafer models to properly benchmark the solvers. The only set of realistic models we have in our collection are the attributed feature models, hence the experiment largely focuses on this special subset of Clafer models. Without a proper set of models, it is hard to evaluate how the Choco solver will perform outside of feature models.

7.1 Real support

A problem with Choco is that sets are only supported over integers. Real number expressions in Clafer are always sets. Section 4.1 explains a strategy to avoid set variables, so it is possible to implement fixed size sets of real numbers as an array of real variables. In the future, the Choco solver should support reals, but limited to fixed size set expressions. This restriction is quite severe.

Listing 21: Races

```
BikeRace 2
    End -> real
BoatRace 2
    Start -> real
[BikeRace.End.ref = BoatRace.Start.ref]
```

Listing 21 states that the boat races start immediately after the bike races. The model, although simple, would not compile because $|BikeRace.End.ref|$ is unknown at compile time. The set has size of 1 if both end times are the same. It has size of 2 if the end times are different.

7.2 Custom constraints

Some expressions have less than optimal encodings in Choco. In the future, the implementation should take advantage of custom constraints and operators to improve the performance of these expressions. The following is a non-exhaustive list of possible custom constraints/operators.

Join on an array of integers: The signature and definition of the joia (join on integer array) operator would look like this:

$$joia : SetVariable \times [IntegerVariable] \rightarrow SetVariable$$
$$joia(s, is) = \{is[i] \mid i \in s\}$$

The new operator can be used to directly implement both join on parents and join references.

Offset: The signature and definition of the offset operator would look like this:

$$offset : SetVariable \times IntegerVariable \rightarrow SetVariable$$
$$offset(s, o) = \{i + o \mid i \in s\}$$

The operator is used for upcasting.

Sum set: One consequence of using sets for almost all expressions is that arithmetic has an unintuitive definition. When performing addition, both operands are sets of integers. Addition adds the sums of its operands.

Listing 22: Addition

```
Version : int *
[Version.ref + 2 > 0]
```

The problem is that $Version.ref$ can evaluate to a non-singleton set. $Version.ref+2$ is essentially rewritten as $sumset(Version.ref) + sumset(\{2\})$. The current sumset implementation can be more efficient as a custom operator.

Integer array to set: The signature and operator of iats (integer array to set) would look like this:

$$iats : [IntegerVariable] \rightarrow SetVariable$$
$$iats([x_1, x_2, ..., x_n]) = \{x_1, x_2, ..., x_n\}$$
$$precondition : allDifferent(x_1, x_2, ..., x_n)$$

Optimized expressions use fixed size integer arrays as discussed in section 4.1. If an expression is a variabled size set, but its subexpressions are fixed size set optimized, then this operator is needed to bridge the mismatch.

8. CONCLUSION

This paper illustrates the important concepts of the new Choco backend for our Clafer project. For attributed feature models, the new implementation is at least as competitive as ClaferMoo, a special Clafer compiler dedicated for optimizing attributed feature models. The new solver looks promising as a substitute or replacement of the Alloy solver. Implementing support for real numbers will make the solver even more desirable, as Alloy does not provide support for reals. Future work with custom constraints and operators may lead to a more efficient solver.

9. 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] D. Jackson, H.-C. Estler, and D. Rayside. The guided improvement algorithm for exact, general-purpose, many-objective combinatorial optimization. Technical Report MIT-CSAIL-TR-2009-033, MIT Computer Science and Artificial Intelligence Laboratory, July 2009.

[5] N. Jussien, G. Rochart, and X. Lorca. Choco: an open source java constraint programming library. 2008.

[6] V. Montaghami and D. Rayside. Extending alloy with partial instances. In *Proceedings of the Third international conference on Abstract State Machines, Alloy, B, VDM, and Z*, ABZ'12, pages 122–135, Berlin, Heidelberg, 2012. Springer-Verlag.

[7] R. Olaechea, S. Stewart, K. Czarnecki, and D. Rayside. Modeling and multi-objective optimization of quality attributes in variability-rich software. In *International Workshop on Non- functional System Properties in Domain Specific Modeling Languages (NFPinDSML'12)*, Innsbruck, Austria, 10/2012 2012.

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

[9] E. Torlak and D. Jackson. The design of a relational engine. Technical report, In Foundations of Software Engineering, 2006.