Towards Enforcing Structural OCL Constraints using Constraint Programming

Matthew Coyle, Théo Le Calvar, Samir Loudni, Massimo Tisi
name.family-name@imt-atlantique.fr
DAPI, IMT-Atlantique, LS2N – CNRS, 44307
Nantes, France

Abstract

Model constraints (e.g. in the Object Constraint Language – OCL) are commonly defined in Model-Driven Engineering (MDE) and used to validate models. Several problems require a way to automatically enforce such constraints, e.g. to complete models or repair them. Because of its combinatorial nature, the problem of enforcing constraints can be computationally hard even for small models. Part of the constraint programming community focuses on developing efficient algorithms for specific but common constraint problems. Such algorithms are made available in constraint solvers in the form of global constraints. Leveraging such global constraints is necessary to efficiently enforce constraints on models.

OCL is the most common constraint language in MDE. However, mapping OCL constraints to global constraints is not trivial, especially for constraints that predicate on the graph structure of the model. In this paper we propose a methodology in two steps. We use annotations of OCL expressions to select model properties to solve for. Then we translate the annotated OCL expressions to a composable Constraint Satisfaction Problem (CSP) that uses global constraints on integer variables to model OCL structural constraints. We illustrate the method on a use case based on reconfigurable manufacturing systems, and provide an assessment of its performance.

CCS Concepts: • **Software and its engineering** → **Search-based software engineering**; **Model-driven software engineering**; **Unified Modeling Language (UML)**.

Keywords: Object Constraint Language, Constraint Programming, Global Constraints

ACM Reference Format:
Matthew Coyle, Théo Le Calvar, Samir Loudni, Massimo Tisi. 2024. Towards Enforcing Structural OCL Constraints using Constraint Programming. In . ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The Object Constraint Language (OCL)[1] is a popular language in Model-Driven Engineering (MDE) to define constraints on models and metamodels. OCL invariants are commonly used to express and validate model correctness. For instance, logical solvers have been leveraged to validate UML models against OCL constraints, used by tools like Viatra [1], EMF2CSP [2] and Alloy [4]. However several problems in MDE require a way to automatically enforce constraints on models that do not satisfy them, e.g. to complete such models or repair them. Because of its combinatorial nature, the problem of enforcing constraints can be computationally hard even for small models.

Constraint Programming (CP) aims to efficiently prune a solution space by providing tailored algorithms. Such algorithms are made available in constraint solvers like Choco [8] in the form of global constraints. Leveraging such global constraints would potentially increase the performance of constraint enforcement on models. However, mapping OCL constraints to global constraints is not trivial. Previous work [6] has started to bridge from arithmetic OCL constraints to arithmetic CP models, but it exclusively focused on constraints over attributes.

In this paper we focus instead on structural constraints, i.e. OCL constraints that predicate on the links between model elements. In detail, we present a method in two steps: 1) we provide an in-language solution for users to denote CP variables in OCL constraints; 2) we describe a general CP pattern for enforcing annotated structural OCL constraints, i.e. constraints predicating on navigation chains. To evaluate the effectiveness of the method, we discuss the size of the Constraint Satisfaction Problems (CSPs) it produces, and the resolution time in some examples.

The paper is structured as follows. In Section 2 we present a problem on a UML model as our running case. In Section 3 we describe how we annotate CP variables in OCL. In Section 4 we show how we model the annotated UML and OCL

[1] https://www.omg.org/spec/OCL/2.4/

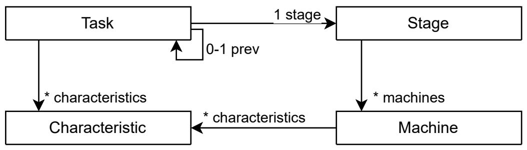

Figure 1. Class Diagram for RMS Task constraints

```
context Task inv SameCharacteristicConstraint:
    self.stage.machines.forall(m | m.characteristics
        ->includesAll(self.characteristics))

context Task inv PrecedenceConstraint:
    self.stage.stageNum >= self.prev.stage.stageNum
```

Listing 1. RMS Task constraints in OCL.

problem as a CSP. In Section 5 we determine the size and performance of the resulting CP model. In Section 6 we discuss limitations and future work. In Section 7 we look at related work. Finally, we conclude in Section 8.

2 Background and Running Case

For illustrative purposes, throughout the paper we exemplify the method on a well-known example, about Reconfigurable Manufacturing Systems (RMS). Notice however that the way to enforce constraints presented in this paper can be applied to any class diagram with OCL constraints.

2.1 Reconfigurable Manufacturing Systems

An RMS [5] is an industrial solution to the problem of varying product demand. In the most common version of the problem (from [10]), a factory is organised into subsequent stages of identical machines. To change the productivity of the factory, machines are added and removed from stages. Manufacturing tasks are allocated to stages, and are generally at least partially ordered. RMSs provide a number of problems to solve, such as: matching tasks and machines with stages, optimising those matches to achieve new productivity goals, as well as allocating the products to a machine of the stage it's going through, or planning machine maintenance.

2.1.1 A Class Diagram for RMS. Figure 1 uses a class diagram to describe the concepts of an RMS, and how they relate (inspired by [10]). In this figure, we focus on the graph structure of the model (classes and references among them), omitting attributes. We use the class diagram flavor from the Eclipse Modeling Framework (EMF)[2], that connects classes by unidirectional *references*, instead of bidirectional *associations*.

The two main components of a reconfigurable manufacturing system are stages, and machines which are organised into stages. A `Machine`'s property is its relation to a set of `Characteristics`. Objects of type `Task` are partially ordered, as expressed by the `prev` reference. Tasks have two other properties: a reference to a `Stage` (allocating the task to that stage), and a reference to characteristics (i.e., the machine characteristics needed to perform the task). Similarly to the example in [10], tasks and machines can be linked to any number of characteristics.

[2]https://eclipse.dev/modeling/emf/

2.1.2 OCL Constraints for RMS. The class diagram shown in Figure 1 cannot encode all constraints that are required for an instance to be a correct RMS instance. Additional constraints can be specified using OCL.

Listing 1 shows the two constraints we use as running example in this paper. These are the structural constraints for tasks, part of a more detailed constraint model for RMS, with budget and productivity constraints.

OCL provides a way to query a model. For a given object, or collection of objects, we can query properties using ".", in expressions following an *objects.property* pattern. Their properties can be references or attributes. In our running example, tasks have a reference to a stage, named `stage`, representing the stage a task is assigned to. In the OCL in Listing 1 line 2, from the context of a `Task`, we query its `stage` by the expression `self.stage`. The sub-expression `self` resolves to an individual object of type `Task`. The whole expression resolves to another object, of type `Stage`, associated with the task. We can see this as a navigation starting at a `Task` node, and traversing the reference to the associated `Stage` node. We can chain navigations: e.g. to find the machines of the stage of a given task in Listing 1 line 2 we use `self.stage.machines`.

`SameCharacteristicConstraint` ensures that *the machines of the stage performing a task* have all the required characteristics. From the given task (`self`) we navigate to the stage where it is performed, and find all the machines of that stage (`self.stage.machines`). For each machine `m` we impose that the set of `characteristics` it supports (`m.characteristics`) includes all the characteristics required by the given task (`self.characteristics`).

`PrecedenceConstraint` ensures that tasks with precedence are performed after their predecessors, i.e. they are assigned to the same stage or a later one.[3]

We will consider these constraints in the following three scenarios.

- **S1:** machines have already been assigned to stages, and the assignment of tasks to stages must be found;
- **S2:** tasks have already been assigned to stages, and the assignment of machines to stages needs to be found;
- **S3:** both tasks and machines must be assigned to stages.

[3]The constraint uses the `stageNum` constant integer attribute that indicates the position of the stage in the manufacturing line, omitted in Figure 1.

2.2 Constraint Programming

CP is a powerful paradigm to model and solve combinatorial problems. In CP, a model, also referred as a CSP, is stated by means of *variables* that range over their *domain* of possible values, and *constraints* on these variables. A constraint restricts the space of possible variable values. For example, if x and y are variables which both range over the domain $\{1, 2, 3\}$, the constraint $x + y \leq 4$ forbids the instantiations $(x, y) = (3, 2), (2, 3), (3, 3)$. While there are many types of constraints, the *global constraints* are perhaps the most significant - being the most well-studied - and have the ability to encode in a compact way combinatorial substructures. A global constraint is defined as a constraint that captures a relation between a non-fixed number of variables.

The $allDifferent$ constraint is probably the best-known, most studied global constraint in constraint programming. It states that all variables listed in the constraint must be pairwise different. For instance the Sudoku problem can be naturally modeled with $allDifferent$: fill a $n \times n$ grid with digits so that each column, each row, and each block contains all of the digits that must be different.

In this paper we will specifically make use of the *element* constraint. Let y be an integer variable, z a variable with finite domain, and $vars$ an array of variables, i.e., $vars = [x_1, x_2, \ldots, x_n]$. The element constraint $element(y, vars, z)$ states that z is equal to the y-th variable in $vars$, or $z = vars[y]$. The element constraint can be applied to model many practical problems, especially when we want to model variable subscripts.

2.2.1 Propagation.

Propagation for a constraint is the action of updating the domains of the variables bound by that constraint. When solving, propagations will generally run when the domain of one of the variables bound by the constraint is updated.

For instance, let $y = \{0, 1\}, x_0 = \{0\}, x_1 = \{2, 5\}, z = \{-10..10\}$ be the domains of the variables given to the element constraint. The element constraint's propagator can update the domain of z to $\{0, 2, 5\}$. The meaning of this propagation is, the possible values for z are a subset of the union of possibilities for x_y, here the union of x_0 and x_1. If during another constraint's propagation, or during search, 0 is removed from the domain of z, such that $z = \{2, 5\}$, the element constraint can update the domain of y to just $\{1\}$. Here, because the domains of x_0 and z are disjoint, then z can not be equal to x_0, hence the element constraint propagation can remove 0 from the domain of y. Finally, if the element constraint is given the following variable instances: $y = 0, x_0 = 0, z = 2$, propagation for the constraint would tell us it is not satisfiable, and serve as a counter proof in model validation.

Propagation is one of the fundamental pillars of constraint programming, along with modeling and search. Global constraints spanning a large number of variables allows one to leverage propagation to the fullest. The application of propagation to the problem of OCL is our fundamental difference to much of the related work. To apply it to OCL we need a systematic way to model OCL expressions using gobal constraints, and particularly to model OCL query expressions.

```
-- Scenario S1
 context Task inv SameCharacteristicConstraint:
    self.var('stage').machines
        ->forall(m | m.characteristics
            ->includesAll(self.characteristics))

-- Scenario S2
 context Task inv SameCharacteristicConstraint:
    self.stage.var('machines')
        ->forall(m | m.characteristics
            ->includesAll(self.characteristics))

-- Scenario S3
 context Task inv SameCharacteristicConstraint:
    self.var('stage').var('machines')
        ->forall(m | m.characteristics
            ->includesAll(self.characteristics))
```

Listing 2. Denoting variables in `SameCharacteristicConstraint` from Listing 1 using `.var()` in accordance with the three scenarios.

3 Denoting CP Variables in OCL Expressions

In this section we describe the first step of the methodology we propose, i.e. a method to select what parts of UML and OCL to model in CP. In a second step, described in Section 4, the resulting annotated OCL will be translated to a CP model.

Since OCL was not originally designed for enforcing constraints, it does not include primitives to drive the search for a solution that satisfies the constraints, as typical CP languages do. For instance, it does not include a way to define which properties of the model have to be considered as constants or variables, while trying to enforce the constraint. Distinguishing variables from constants has a double importance, both for correctly modeling the CP problem, and for reducing its search space to a limited number of variables.

Note that the distinction of variables from constants can not be performed automatically in general, as it depends on the user intent. For instance, in our use-case scenarios, we want to enforce the reference between `Task` and `Stage` to conform to `SameCharacteristicConstraint` of Listing 1. To do so we annotate the references for which information is missing, but for uses such as model repair, annotations can direct where to look for fixes in the model. For instance given a factory configuration which breaks `SameCharacteristicConstraint`, we could choose between fixes reassigning tasks, or reassigning machines, or both to stages.

To allow users to explicitly denote properties (attributes or references) in an OCL expression as variables (variable attributes or variable references), we propose the `var()` operator with the following syntax:

`source.var('property')`

where `source` identifies the objects resulting from the prior sub-expression, `property` is the name of one of the attributes or references of the objects. In Listing 2 we apply the operator to `SameCharacteristicConstraint` in Listing 1 for each one of our three scenarios, defining the properties that we consider as variables for that scenario. All properties that are not included in a `var()` operation call are considered constant.

Notice that our in-language solution does not extend the syntax of the OCL language, but we add an operation to the OCL library: `var(propertyName: string) : OclAny`. When the OCL constraint is simply checked over a given model (and not enforced), the `var` operation simply returns the value of the named property (as a reflective navigation).[4] Whereas, if one wants to enforce OCL, `var` is used as a hint to build the corresponding CSP.

We can add extra parameters to `var` to drive CP modeling, e.g. for bounding the domain for a property, or choosing a specific CSP encoding among the ones presented in the next section. In future work we plan to add other parameters to guide model repair, by describing how much we can change properties in order to fix the model. Note that, alternatively, users can also annotate the variable references in the metamodel, instead of the constraints. In this case, we can always statically translate such variable annotations on metamodels into the variable annotations on constraints discussed here.

3.1 Annotation in the OCL Abstract Syntax Tree

The annotated OCL is parsed in the form of an AST. Given an instance model to solve for, each object will have their own instance of the AST, where `self` resolves to said object. Figure 2 shows the AST of `SameCharacteristicConstraint` from Listing 2 Scenario S3. We show `var` annotations as dotted rectangles.

Figure 2 illustrates a key function of `var` annotations: they define the scope of the CP problem, i.e. a frontier between what can be simply evaluated by a standard OCL evaluator, and what needs to be translated and solved by CP. In Figure 2, the scope defined by each `var` annotation is indicated by a dotted rounded rectangle. The `var` annotation requires everything *inside* the corresponding scope to be translated to CP.

For instance, since the reference between `Task` and `Stage` is annotated (`self.var('stage')`), the result of the `stage` `NavigationOrAttributeCallExp` needs to be found by the

[4]Look at getRefValue from ATL/OCL for a similar reflective operation https://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language#OclAny_operations

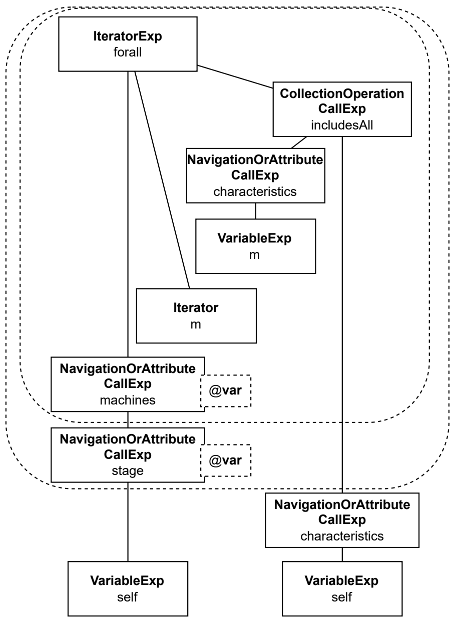

Figure 2. AST of `SameCharacteristicConstraint` from Listing 2 Scenario S1, S2 & S3.

solver. All nodes in the scope of an annotated node will be in the CSP, as what they resolve to depends on the solution the solver is searching for. Conversely, nodes that are not in the scope of any `var` annotation do not need to be translated to CP, making the CP problem smaller.

The processing of the AST in Figure 2 (corresponding to Scenario S3) starts from the bottom: `self` is directly evaluated by standard OCL, as is `self.characteristics`. However we don't know the result of `self.stage`, which implies we don't know the result of `self.stage.machines`. Above, we iterate on the unknown machines and for all of them: ask what their characteristics are, and if they include the characteristics of the task. All these questions must also be answered by the solver, which means any node of the tree within the dotted box must be resolved by the solver.

In Scenario S2, `SameCharacteristicConstraint` from Listing 2 has the same AST as in Figure 2, but only the `machines` node is annotated as `var`. Hence, in this case the CP scope is smaller, since `self.stage` can be directly evaluated by OCL.

3.2 Refactoring OCL Around Annotations

Given that everything above an annotated node of the AST is within the scope of the CSP, it's interesting to find strategies to reduce the scope as much as possible, as it results in a smaller CSP to solve. The annotated expressions of Listing 2, all have their annotations low in the tree Figure 2. Ideally, all the annotations should be at the top of the tree. The semantics of the expression gives clues to refactor them, the expression requires that: *All the machines connected to a task (via a stage), each individually match the task's characteristics* this is the same as requiring that: *The set of machines that match the task, includes the set of machines connected to the task.*

In Listing 3 we can see the result of this rewrite for all three scenarios. The beginning of the expressions are now constant queries, and search for all the suitable machines (or stages), here isolated as `sel`:

```
let sel = Class->AllInstances().select(...) in
```

At the end of the expression we state that selection must include the result of the variable query over the machines and/or stage of the task:

```
sel->includesAll(...)
```

In Figure 3 we can see the AST resulting from the parsing of the expression of Listing 3 Scenarios 2 and 3. The AST is significantly different to the previous one, but most importantly, the number of nodes within the scope of the solver is greatly reduced, to just navigation and the top level constraint. The CSP now only models the inclusion.

We applied this strategy manually with knowledge of the context, but it is generalisable. In the case of any constant sub-expression applied to a variable query, it is possible to determine candidates, or candidate sets, for that sub-expression and enforce the result of the variable query to be among them. For example, for: `self.var(ref).attrib<3` we can find candidates which satisfy the constant sub-expression `.attrib<3`. This adds more computation ahead of building the CSP, but also allows us to leverage the OCL engine in cases where it's more efficient such as this one.

```
-- Scenario S1
 context Task inv SameCharacteristicConstraint:
    inv: Stage.AllInstances()
        ->select(s|
            s.machines.forall(c | c.characteristics
                ->includesAll(self.characteristics))
        ->includesAll(self.var(stage)))

-- Scenario S2
 context Task inv SameCharacteristicConstraint:
    inv: Machine.AllInstances()
        ->select(m| m.characteristics
            ->includesAll(self.characteristics)
        ->includesAll(self.stage.var(machines)))

-- Scenario S3
 context Task inv SameCharacteristicConstraint:
    inv: Machine.AllInstances()
        ->select(m| m.characteristics
            ->includesAll(self.characteristics)
        ->includesAll(self.var(stage).var(machines)))
```

Listing 3. Annotated `SameCharacteristicConstraint` from Listing 2 refactored around the annotations.

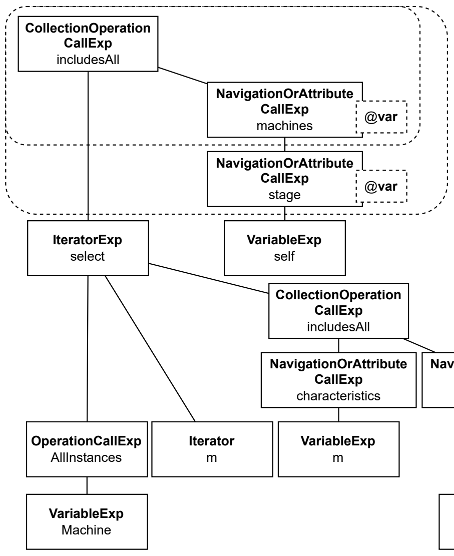
CollectionOperation CallExp includesAll
NavigationOrAttribute CallExp machines
@var
NavigationOrAttribute CallExp stage
@var
IteratorExp select
VariableExp self
CollectionOperation CallExp includesAll
NavigationOrAttribute CallExp characteristics
NavigationOrAttribute CallExp characteristics
OperationCallExp AllInstances
Iterator m
VariableExp m
VariableExp Machine
VariableExp self

Figure 3. AST of SameCharacterticConstraint from Listing 3 Scenario S2 & S3

4 Modeling Annotated OCL Constraints using Constraint Programming

To enforce OCL constraints on existing model instances using CP, we must encode parts of the model instances as CSP variables and constraints (Section 4.1), and then encode the OCL using additional CSPs on the variables of the model instances (Section 4.2).

4.1 References in CP

There are two main families of CSP, those using booleans, and those using integers. Both of these can be used to model our problem. In this paper we are focusing on structural constraints, which implies modeling references, and chains of references in OCL queries. Simply put: using boolean variables we can ask whether two objects are connected, using integers we can ask how many times two objects are connected, but more interestingly: to which object a given object is connected to. This last encoding uses variables as pointers, and is the one we will be presenting here.
Reference: exists between two Classes; can be seen as the lines in a class diagram such as Figure 1. An example from our use-case, is the references between `Task` and `Stage`. Here our references are only navigable one way. For an object diagram or an instance, reference instances are called **links**. References in an EMF instance, are instantiated as objects of the type EReference.

Variable Reference: these imply adding variables and possibly constraints to the UML CSP. Non-annotated references will be called constant. From annotation of the OCL we infer which references are variable. From the expression `self.var(stage).var(machines)`, we can determine that for `Task` the reference `stage` is variable, and for `Stage` the reference `machines` is variable. This is the bridge between the model instance and the CSP, when we find a structure in the CSP, we will update these references.
Pointer Variable: integer variable answering the question *which object is a given object connected to?* The number of instances of the target class gives the upper bound of the variable's domain. The lower bound being 0, in our model meaning *null pointer*. This happens when an object has less links than the maximum allowed by the metamodel or the annotation.
AdjList Variable: models a variable reference instance in the UML CSP. Essentially an adjacency list, it is a list of pointer variables associated with an EReference. The number of pointer variables in an `AdjList` is defined by the cardinality of the reference, or can be informed by the annotations. This choice is one of the main dimensions of the complexity of the resulting problem. These will be our primary *problem variables*, as opposed to the intermediate variables the OCL expressions may require.
UML CSP: what we call the CSP of the annotated instance properties. It includes the above models of variable references, and any constraints the metamodel (Figure 1) may define upon them, such as containment, uniqueness, coherence between opposite references, etc.. For our running example, the UML CSP only holds the problem variables, no additional constraints are applied because of the metamodel. Regardless we're calling it a CSP because it is in the general case.

For our use case, the references from `Task` to `Stage`, and from `Stage` to `Machines` can be variable. This implies for objects such as tasks, and specifically their reference to a stage, associating them to an `AdjList` identifying the stage with a single pointer variable. Equally for stage objects, we associate their reference object to an `AdjList`, but with multiple pointer variables. These all can be organised into tables, where for each object and a property, we match that property with CSP variables. The row numbers of these tables are the actual domains of pointer variables.

4.2 OCL Expressions in CP

With annotated references, navigation and querying become part of the CSP. Given an OCL query expression such as `self.var(stage).machines`, if the CSP must determine which `Stage` is associated to a `Task`, it will also determine the set of `Machine` associated to the `Task` through the `Stage`.
OCL CSP: Nodes within the scope of the solver are associated with a CSP, all of which combined make up the OCL CSP. The models for the OCL expressions are built on top of the UML CSP, by reusing the problem variables modeling the instance properties. The OCL node CSPs will generally also add their own intermediate variables to pass information upwards. Constants of the model will also sometimes appear in the OCL CSP as integer variables, but their domain of possible values is *the* value found in the model.
Variable Expression: if an OCL expression has an annotation, it is referred to as variable. If it has none we call it constant. Expressions can be decomposed into sub-expressions, and variable expressions can be decomposed into variable and constant sub-expressions. A particular type of expression is the query, which in this paper are the primary sub-expressions of structural constraints.
Variable Query: variable queries are similarly any annotated query expression, but can be divided into two main parts:

1. *variable property access*: `src.var(prop)`
2. *variable navigation*: `.var(src).prop`.

Variable property access is sourced from constant (non annotated) queries, e.g. `self.prev.var(stage)`. Variable navigation is sourced from a variable query. For example in: `self.var(stage).var(machines).characteristics` machines and characteristics are reached through variable navigations, the first being from `Stage` to `Machines`, the second from `Machines` to `Characteristics` .

4.2.1 `NavigationOrAttributeCallExp` on `EReference`.

When annotated, there are now two contexts in which this OCL object can be parsed, here, the simple case when the source is an expression with no var annotation, a constant query, such as: `self.var(prop)` or `self.prev.var(stage)` from our use case. All this requires is a map between the instance's `EReferences` and their associated `AdjList` Variables in the case of navigation. Or between the EAttributes and regular integer variables in the case of attribute access.

4.2.2 `NavigationOrAttributeCallExp` on `AdjList`.

The second context is when the source is an `AdjList`. In our use case the same characteristic constraint gives this case, when stage is annotated: `self.var(stage).machines` or `self.var(stage).var(machines)`.

In CSP 1 we model the core problem of resolving a variable navigation. Given a variable query with a variable source `.var(stage)`, what are the values of the referred properties `.machines`? These properties can themselves be variable or known references, navigating to objects such as here, allowing us to chain the problem.

One way to picture this CSP is as a function $navCSP$: $(AdjList, property) \rightarrow IntVar[]$, to which we give a list of pointers for it to return the desired properties as a list of integer variables. If the properties are constants from the model, it still returns integer variables. This is very similar to the element constraint, and why the latter serves us to model the former.

The incoming pointers of the AdjList ($src.pointer_i$ in CSP 1), are either problem variables associated to an EReference, or result from a prior navCSP. We use these pointers to identify the object from which we want to copy the property.

To make the $Table$ we collect information from the instance model and the UML CSP, either the problem AdjList variables associated with the annotated references, or the model's data instantiated as integer variables with a single possible value. This information is organised into *object to property* tables. For example in the cases of reference, the table associates each object to their AdjLink variable, a row of pointer variables. Variable reference models including *null pointers*, will also need the 0-th row of the table to have *dummy variables* of which the value is 0, or *null pointer*.

This table is flattened, and corresponds to the $vars$ in the element constraint definition. Because the table is flattened, we do some pointer arithmetic to go from the id of the object (or the row of the table), to the positions of the object's properties in the flat table. In summary, $Obj_{ID} *$ *number of properties* + *property number*. The result of this constraint, $pointer_{ij}$ is the integer variable used as the index of the element constraint (y in definition)

These properties are *copied* to intermediate variables modeling the current node of the query: $src.pointer_i.property_j$ (z in definition). To copy a problem variable to an intermediate variable, the element constraint is used. Hence, for every intermediate variable modeling this node of the query, there is an element constraint, and pointer arithmetic. Together we call them *query atoms*. In Section 5.1 we will count the number of query atoms to evaluate the query model.

4.3 Translation

The general translation strategy traverses the AST of the OCL constraint, and translates each node type of the AST in a CSP scope, such that combined they model the expression. Here we apply it to the running case in Scenario S3.

From the static analysis, we can determine what to model from the instance the OCL is being applied to. In Scenario S3 this means mapping EReference to AdjList, for the references stage and machines.

UML CSP Variables:
$Table = object_o.property_j \mid \forall o \in [0..O], \forall j \in [0..N'[$
Source AdjList Variable:
$src.pointer_i \mid \forall i \in [0..N[$
Intermediate Variables:
$\forall i \in [0..N[, \forall j \in [0..N'[$:
$src.pointer_i.property_j \in \mathbf{N}$
$pointer_{ij} \in \mathbf{N}^+$
Constraints: $\forall i \in [0..N[, \forall j \in [0..N'[$:
$$\begin{cases} pointer_{ij} = src.pointer_i * N' + j \\ element(pointer_{ij},\ Table,\ src.pointer_i.property_j) \end{cases}$$

CSP 1. NavCSP: NavigationOrAttributeCallExp semantics modeled in CSP in the context of integer variables modeling pointers

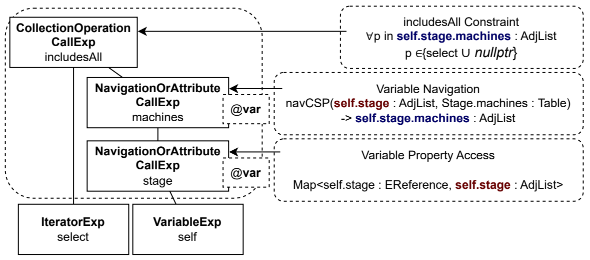
CollectionOperation CallExp
includesAll
NavigationOrAttribute CallExp
machines
@var
NavigationOrAttribute CallExp
stage
@var
IteratorExp
select
VariableExp
self
includesAll Constraint
∀p in self.stage.machines : AdjList
p ∈{select ∪ nullptr}
Variable Navigation
navCSP(self.stage : AdjList, Stage.machines : Table)
-> self.stage.machines : AdjList
Variable Property Access
Map<self.stage : EReference, self.stage : AdjList>

Figure 4. Compilation to OCL CSP of the AST from Figure 3

To build the OCL CSP, for each Task we start from the bottom of its tree, upon reaching an annotated node, we get the UML CSP variables modeling that property, the variable property access in Figure 4 described in Section 4.2.1. In this case the accessed property is a reference, allowing us to navigate further. Navigating from a variable reference modeled with pointers means using CSP 1 described in Section 4.2.2. We can see it applied in Figure 4 modeling the variable navigation node. Finally at the very top of this tree, we have includesAll, which we model here using the member constraint, which constrains a variable to be within a certain domain.

5 Evaluation

In order to evaluate the performance of the method we propose the following metrics: counts of variables and constraints and solving times. The number of problem variables generated depends on the number of objects and the number of pointer variables in the AdjList variables of the UML CSP. The number of intermediate variables will depend on the number of problem variables and their use in OCL CSPs such as CSP 1.

5.1 NavCSP

Navigating a model adds a great deal of complexity. The pointer navigation CSP 1 is the greatest factor in that complexity. It takes effect in variable query expressions such as: src.var(ref).prop where we want to find a property based on variables in the scope of the solver. Whether the property is variable or not, or is an attribute or a reference, the same navCSP applies. In the case the property is a reference, we can chain the CSP, which greatly increases complexity.

5.1.1 OCL Query Dimensions. To evaluate the navigation provided by CSP 1, we will look at the size of the CSP modeling the following OCL expression:

```
let query = self.ref.ref...ref in
```

Such that all the references are variable, and `self.ref` is a reflexive reference modeled with N pointer variables. The depth of the navigation, is noted d, with $d = 0$ as the case of variable property access, `query = self.ref`. Adding further navigations increments d, for example $d = 2$ corresponds to `self.ref.ref.ref`.

5.1.2 OCL Query Size. In Figure 5 we can see the number of *query atoms*, meaning equally: the intermediate pointer variables, element constraints or pointer arithmetic required to model this query, which is found using the formula:

$$f(N, 0) = 0$$

$$f(N, d) = f(N, d-1) + N^{1+d}$$

1) No matter the size of the `AdjList`, the first annotated reference implies no intermediate pointers, as we simply find the problem variables associated to `self`.
2) If we are to navigate deeper, we make an additional hyper-table of intermediate variables, indexed by the prior lower dimension table of pointers. To examine the formula, let's look at the case of $d = 1$, or `self.ref.ref`:

$$f(N, 1) = 0 + N^2$$

We have N pointers coming in from `self.ref`, and they each point to N pointers. Resulting in a table of intermediate pointer variables. If we navigate deeper, let $d = 2$:

$$f(N, 2) = 0 + N^2 + N^3$$

For every pointer in the previous table N^2, we associate N more pointers. Giving us now a *hyper-table*, cubed. If we navigate deeper, the 3D hyper-table will similarly index a 4D hyper-table.

The graph in Figure 5 starts at 1 on the x,y axes or $f(1, 1)$, which gives 1 on the z axis (log scale). For a single navigation from a single pointer variable (AdjList of size 1), we have a single query atom. For a single navigation from an `AdjList` of size 10 or $f(10, 1)$, we have 100 query atoms. For `AdjList` variables of size 1, navigating with a query depth of 10 or $f(1, 10)$, results in 10 query atoms.

On the left background, we can see the curve resulting from increasing `AdjList` size. While on the right, we can see the curve resulting from increasing navigation depth. We can see from this that increasing the navigation seems to increase the size of the problem logarithmically, while increasing the number of pointers for a reference is exponential.

The complete navigation model has twice as many constraints $2f(N, d)$, as we need both an element and some pointer arithmetic for each intermediate variable. Our implementation of the pointer arithmetic implies an additional intermediate variable, giving a total of $2f(N, d)$ intermediate variables.

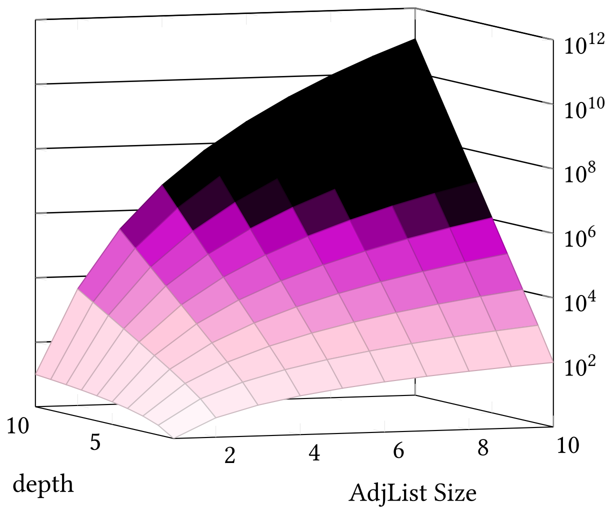

Figure 5. Number of query atoms in relation to `AdjList` size and navigation depth

The total number of propagations required to find all counter proofs, or validate a model also aligns with the number of constraints found here $2f(N, d)$, validation would correspond to all the problem variables having only one possible value. While it is a large number it's still fast to run all these propagators once, and running out of memory space for the model became a more limiting factor than time in our tests.

Going beyond validation, and searching for a model fix, or completing a model such as in our use-case, means increasing the domains of the problem variables and by consequence the intermediate variables, and in the case of model completion having the full range of possibilities for all these variables.

5.1.3 Subset Sum Problem. [5] by applying the following constraints to the query from a single object (among up to 120), we can model a variation on the subset sum problem:

```
query->sum(attribute) = Target
and query->isUnique(attribute)
```

Where `self.attribute` of an object is a constant integer attribute between 10 and 29. All of them together forming a multiset, from which we'll find a subset with the right sum. Initial testing with this problem gives fast non-trivial solutions, up to a few minutes, for queries with up to around 10^4 intermediate variables. When no subset sums equal the target, such as finding a subset summing to 1, or when solving for trivial targets such as 0, the process takes less than a few minutes up to 10^6 intermediate variables. Bigger problems reached our memory limit. These results color Figure 5, the lightest area being quickly solvable, the darker area being quickly verifiable and the black area being too big to model.

[5] https://github.com/ArtemisLemon/navCSP_SubsetSum

5.2 RMS use-case

In our running example, our instance will have 4 stages (S) and 43 task (T), directly taken from [10]. We infer 24 machines (M) from their constraint model. In our annotation of `machines` we will choose 24 as the maximum cardinality of the reference, and thus have 24 pointer variables in the associated `AdjList`. Notice that the combinatorial complexity of the problem is quite challenging, since there are around 2.10^{40} possible graphs satisfying these assumptions. Generating all graphs and checking the OCL constraints would be unfeasible. The full code for this problem instance is available online.[6]

var()	variables	domain	constraints	time
stage	43	S	(43)	0.1s
	0	M		
machines	0	S	(96)	0.1s
	96	M		
stage machines	43	S	2064 (+1032)	0.3s
	96+1032	M		

Table 1. Size and resolution times of the use-case CSPs

In Table 1 we find the metrics for the three RMS scenarios. Additionally to the same characteristic constraint, we also enforce the precedence constraint in the scenarios where the reference `Task.stage` is annotated. The first column identifies the annotations, and the scenarii. The second column gives is the variable counts for each domain. The variable count includes all variables (problem and intermediate) involved in the expression. While the intermediate variable are unique to the model of this expression, the problem variables (tied to the instance objects) are shared by any other expression. Domains, informed by the third column, are identified by their upper bound, as the lower bound generally 0 for pointer variables. For the last row, we have both 96 problem variables of domain M, and because of the navigation from the 43 problem variables of domain S, each requiring their own copies of the variables of domain M, there are 1032 (43*N) intermediate variables.

In the constraints column, we count constraints of the OCL CSP, mainly element constraints and pointer arithmetic. In between brackets is the counter of *member* constraints. As *member* only propagates once, by default our solver doesn't include them in the constraint count. As it is the only constraint in two of these cases, we have included them.

Finally for times, we can see this problem is trivial for the solver. Most of the time taken is to build the model.

6 Limitations and Future Work

Alternative CP Models. Here we present a navigation model based on pointers, modeled by lists of integer variables. In OCL the result of accessing a reference and navigation is of type collection, and there are 4 concrete collection types, at the crossroads of two qualities: orderedness and uniqness. This pointer variable model provides a representation for ordered collections with non-unique elements, or the OCL collection type `Sequence`. Applying `AllDifferent` can model the collection type `OrderedSet`. The other OCL collection types, namely `Set` and `Bag`, can be modeled differently and more efficiently. Notably references and navigation as `Set` can make use of *set variables* which can answer the question *which objects are connected to the given object?*, the global constraint $union(y : set, vars : set[], z : set)$, mirroring the element constraint, and trivially providing efficient variable navigation.

Navigation in OCL often resolves to Sequences of pointers, hence a requirement to model them to maximize OCL coverage. But if modeling with constraint enforcing in mind, it is interesting to ask if orderedness is important to your navigations, as it is a costly quality. Choosing between these encodings for references could be informed by the annotations.

Further Evaluation. Initial evaluations unsurprisingly revealed navigation is a complex issue, however it also revealed its complexity is complex to measure. We give the size of the navCSP in terms of intermediate variables and constraints, but the domain size of the variables, the number of objects and types, are among the another variables. And while we give some aspect of the size, it doesn't tell the whole story of how hard a problem is to solve. Models with low `AdjList` size and deep navigation are very difficult problems, even if the overall model is small. While models with larger `AdjList` variables and shallow navigation depths produce large problems that can sometimes be solved faster. Graph density is also an interesting dimension. Additionally the complete evaluation for this method requires comparisons to Alloy [4] as it applies SAT techniques [9] for model finding.

Automation of the Refactoring. As hinted as in Section 3.2, some OCL refactoring seems to be systematically applicable. However it does require more computation ahead of building the CSP. This method needs to be formalised and the pre-computation tested for efficiency gains. For our use case, the pre-computation around `forall` was trivial, but allowed for instant solving times. The general case however allows for much more complex pre-computation.

Translating Full OCL. In this paper, we focus on the translation for the OCL `NavigationOrAttributeCallExp` node. We also hint at a translation of `includesAll` using the member constraint, but all collection operations can be similarly modeled in CP, with varying efficiency. Finding the models using global constraints for OCL words isn't trivial, and finding efficient ones may require testing and even the development of new propagation algorithms. Having a translation scheme for each OCL word, will allow us to implement a

[6] https://github.com/ArtemisLemon/navCSP_RMSTaskConstraints

compiler, and integrate it into a model transformation framework such as ATLc [6], giving us a framework to implement model repair and domain space exploration.

Application to Design Space Exploration and Integration with ATLc. ATLc is an extension to the ATL transformation language, based on OCL, which couples constraint solvers [6] with incremental model transformations [7]. The greater objective of this work is to add to the ATLc compiler, and use it's GUI framework to generate interfaces, allowing users to interact with the search for solutions to structural constraints, in a form of human-in-the-loop solving.

7 Related Work

The most similar work to our overall objective is by Le Calvar et al. [6], which provides a tool for domain space exploration by means of model transformations coupled with CP solvers. This work provides the compilation of arithmetic OCL constraints on attributes to a variety of constraint solvers. This work also provides a framework for modeling CSPs on Java GUIs and also discusses weakening constraints, and solver collaboration. However this compiler doesn't provide the compilation of structural constraints; they provide *variable property access* but not *variable navigation*. Because of this they don't require an annotation system, as they can assume that the top level of the query is the variable.

Alloy [4] also provides a modeling language similar to UML and OCL, based on the Z specification language with it's own query language. Allowing the declaration of problems over it's OCL-inspired query language makes it particularly relevant to what we present in this paper. This also points at a first difference, the choice of source constraint language: Alloy provides it's own language, where we try to cover a well established standard. To analyze models the Alloy toolkit uses KodKod [9] to translate the problems towards SAT solvers for verification and model finding. This is similar to our work, but we translate the problem to global constraint models. Both SAT and CSP can similarly model the problem, but they each provide different methods to find a solution. These different methods provide *off-the-shelf* solvers such as Choco, which also provides a framework to develop propagators. Choosing between SAT and CSP to model and solve a problem isn't simple, but the ability to design propagators and direct search makes it an interesting avenue to pursue. With our work, we lay the foundations to explore the alternative.

Another example, which bares strong resemblance to our work is that of [1, 3]. Constraints on target models are encoded as patterns of elements. This work gives rise to CSP(m) for Viatra, allowing visual pattern mappings to define transformation rules, with anti-patterns similar to constraints to enforce. CSP(m) is a solver based on backtracking algorithms designed for this specific purpose, which also solves the whole transformation. This differs from our solution which reuses *off the shelf* tools to model the problem, and global constraints on integers which are implemented by many of these solvers. Solving for the whole transformation is another significant difference, as ATLc splits the effort between tools better suited to their respective tasks. Alleviating the constraint solver, and leveraging the efficiency of the transformation engine [7].

8 Conclusion

We have shown how to model the core of OCL queries and navigations using CP, laying the foundation of modeling OCL with global constraints. We did so using models encoding orderedness and non-uniqueness of OCL Sequences, the most complex version of the problem. The models are also designed to be assembled by an OCL compiler. To identify variables during compilation and guide CP modeling, we have proposed the `var()` operator as an annotation of the OCL constraint model. We have also raised the question of how to better model OCL with enforcing in mind, which hints at a systematic method applied during static analysis. We have also outlined other branches of future work, such as: further modeling OCL using global constraints, implementation atop of ATLc, and evaluation and comparison with other methods.

References

[1] Bergmann, G., Dávid, I., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z., and Varró, D. Viatra 3: A reactive model transformation platform. In *Theory and Practice of Model Transformations*, Lecture notes in computer science. Springer International Publishing, Cham, 2015, pp. 101–110.

[2] Gonzalez, C. A., Buttner, F., Clariso, R., and Cabot, J. EMFtoCSP: A tool for the lightweight verification of EMF models. In *2012 First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA)* (June 2012), IEEE.

[3] Horváth, Á., and Varró, D. CSP(M): Constraint satisfaction problem over models. In *Model Driven Engineering Languages and Systems*, Lecture notes in computer science. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 107–121.

[4] Jackson, D. Alloy. *ACM Trans. Softw. Eng. Methodol. 11*, 2 (Apr. 2002), 256–290.

[5] Koren, Y., Heisel, U., Jovane, F., Moriwaki, T., Pritschow, G., Ulsoy, G., and Van Brussel, H. Reconfigurable manufacturing systems. *CIRP Ann. Manuf. Technol. 48*, 2 (1999), 527–540.

[6] Le Calvar, T., Chhel, F., Jouault, F., and Saubion, F. Coupling solvers with model transformations to generate explorable model sets. *Softw. Syst. Model. 20*, 5 (Oct. 2021), 1633–1652.

[7] Le Calvar, T., Jouault, F., Chhel, F., and Clavreul, M. Efficient ATL incremental transformations. *J. Object Technol. 18*, 3 (2019), 2:1.

[8] Prud'homme, C., and Fages, J.-G. Choco-solver: A java library for constraint programming. *J. Open Source Softw. 7*, 78 (Oct. 2022), 4708.

[9] Torlak, E., and Jackson, D. Kodkod: A relational model finder. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 632–647.

[10] Wang, W., and Koren, Y. Scalability planning for reconfigurable manufacturing systems. *J. Manuf. Syst. 31*, 2 (Apr. 2012), 83–91.