COLLEGES SCIENCES
BRETAGNE POUR L'INGENIEUR
LOIRE ET LE NUMERIQUE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Thèse de doctorat de

L'École Nationale Supérieure
Mines-Télécom Atlantique Bretagne
Pays de la Loire – IMT Atlantique

École Doctorale 648
Sciences pour l'Ingénieur et le Numérique
Spécialité : *Sciences et technologies de l'information et de la communication*

Par
Matthew COYLE

Object Oriented Constraint Programming

Models of UML and OCL semantics using Constraint Programming

Thèse présentée et soutenue à IMT Atlantique Nantes, le « date »
Unité de recherche : « voir README et le site de de votre école doctorale »

Rapporteur·trice·s avant soutenance :

Prénom NOM	Fonction et établissement d'exercice
Prénom NOM	Fonction et établissement d'exercice
Prénom NOM	Fonction et établissement d'exercice

Composition du Jury :

Attention, en cas d'absence d'un·e des membres du Jury le jour de la soutenance, la composition du jury doit être revue pour s'assurer qu'elle est conforme et devra être répercutée sur la couverture de thèse

Président·e :	Prénom NOM	Fonction et établissement d'exercice *(à préciser après la soutenance)*
Examinateur·trice·s :	Prénom NOM	Fonction et établissement d'exercice
	Prénom NOM	Fonction et établissement d'exercice
	Prénom NOM	Fonction et établissement d'exercice
	Prénom NOM	Fonction et établissement d'exercice
Dir. de thèse :	Samir LOUDNI	Fonction et établissement d'exercice
Co-dir. de thèse :	Massimo TISI	Fonction et établissement d'exercice *(si pertinent)*
Co-dir. de thèse :	Théo LE CALVAR	Fonction et établissement d'exercice *(si pertinent)*

Invité·e·(s) :

Prénom NOM	Fonction et établissement d'exercice

ACKNOWLEDGEMENT

Je tiens à remercier
I would like to thank. my parents..
J'adresse également toute ma reconnaissance à ....
....

TABLE OF CONTENTS

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 [?], EMF2CSP [?] and Alloy [?]. 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 [?] 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 [?] 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.

In the context of Model-Driven Engineering (MDE), models represent structured data, and the model of the data structure is known as a metamodel. The Unified Modeling Language (UML) [2] provides visual languages, such as class and object diagrams, to define both models and meta-models. The Object Constraint Language (OCL) [3] complements UML by enabling the specification of constraints over models, based on the underlying

[1]`https://www.omg.org/spec/OCL/2.4/`
[2]`https://www.omg.org/spec/UML/2.4`
[3]`https://www.omg.org/spec/OCL/2.4`

metamodel concepts. The Eclipse Modeling Framework (EMF) [4] supports UML and OCL, enabling validation of models against their meta-models and associated constraints. It also includes model transformation tools such as ATL [?, ?], an OCL-based language that expresses mappings between meta-models. ATLc [?] extends ATL by introducing model space exploration capabilities to facilitate transformation specification. It leverages constraint solvers to generate and visualize model instances, which users can then adjust or repair using solver feedback. The primary use of ATLc is to create a Graphical User Interface for a model, allowing the user to easily edit the model. This generally breaks some of the user defined OCL constraints, and our work hopes to provide a way to repair the models around the user's choices. The core problem is: given a metamodel, a partial model and model constraints as input, the objective is to find model instances that satisfy the metamodel and model constraints. ATLc does so by interpreting part of their OCL expressions upon an instance as a constraint satisfaction problem (CSP), which can be solved by engines like Cassowary (for linear programming) or Choco (for constraint programming). However, ATLc is currently limited to single-valued model attributes, using integers or reals. Our work seeks to generalize this approach to support collection-valued properties: attributes and relations.

Among existing tools, Alloy [?] stands out as a tool offering a dedicated language for defining meta-models and constraints. Alloy is often used for specification testing–such as verifying security protocols or code–by searching for models that satisfy given constraints. It can also be used for checking specifications by searching for valid instances or counterexamples. Alloy has also been applied to model transformation and model repair [?], with some approaches translating UML/OCL into Alloy specifications [?,?]. The core difference with our approach lies in the underlying solving technique: Alloy is based on SAT solving, while we use Constraint Programming (CP). Choosing between SAT and CP for model search tasks is not straightforward, and through our experimentation, we aim to shed some light on that choice in the context of model search. Related work leveraging CP, global constraints and similar models also exists [?], however UML/OCL coverage doesn't include the general case of collection properties discussed in this paper, and required for the experimentation.

[4]`https://projects.eclipse.org/projects/modeling.emf.emf`

Model Driven Engineering

Artificial Intelligence

Problem to solve

Chapter 1

Context: Model Driven Engineering & Constraint Programming

1.1 Model Driven Engineering

1.1.1 Models

- models represent systems under study
- models allow us to easily manipulate and learn properties of the system under study
- table of definitions (cite)
- EMF provides serialisation XMI
- **figure:** graph or informal object diagram
- **figure:** Table of data

1.1.2 Metamodels

- metamodels model models
- conformsTo relation
- metamodel conformsTo self
- **figure:** m0,m1,m2,m3 levels
- domain specific languages

- UML proposes metamodels: Class Diagram, Object Diagram
- with a class diagram you can describe any UML metamodel, which is cool
- EMF provides tools to model metamodels

1.1.3 Model Queries & Constraints

- language to describe queries on models
- upon those queries we can apply constraints
- OCL provides a language for this (cite)

1.1.4 Model Transformations

- fundamental operation on models
- specified between two metamodels
- allows for collaboration between different experts
- **figure:** model transformation
- MT languages are commonly based on OCL queries
- a property of a model resulting from a transformation, generally holds the result of an OCL query on the source model
- EMF provides ATL, QVT, Viatra,.. (cite)

1.1.5 Model Verification

- given a set of models we can verify them against a metamodel and model constraints
- general use of model constraints
- EMF provides EMF2CSP(cite) for validation

1.1.6 Model Search

- kind of AI model transformation
- Domain Space Exploration when part of a model transformation
- given a metamodel and model constraints what are the possible models

1.1.7 Class Diagrams

B. Class diagrams identify concepts and their properties. In a family tree for instance, the core concept is Person, with attributes such as age and references such as parent (or its inverse, child) to express relationships between people.

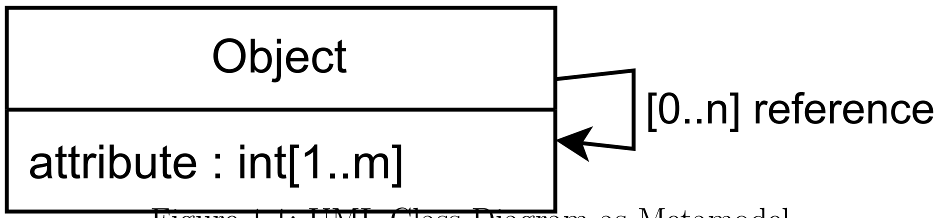

Figure 1.1: UML Class Diagram as Metamodel

Figure 1.1 present a generic metamodel. It describes a class named `Object`, which has two properties: `attribute`: a collection of integers, with at least one and at most m elements, `reference`: a collection of up to n references to other `Object` instances. These illustrate the two main types of properties in object-oriented modeling: Attributes, which store intrinsic data values (e.g., numbers or strings), References, which define relationships between objects in the model.

UML allows properties to be collections, and distinguishes four standard collection types, based on two dimensions: order and uniqueness.

- `Sequence`: ordered, allows duplicates – e,g., [2,3,1,1],
- `Bag`: unordered, allows duplicates – e.g., [1,1,2,3],
- `Set`: unordered, unique elements only – e.g., [1,2,3],
- `OrderedSet`: ordered, unique elements – e.g., [2,3,1].

An important note is that ordered doesn't pertain to the values. In [2,3,1,1]: 2 is the first value, and 1 is the last value. The intended collection type can be indicated in the class diagram using annotations such as `ordered`, `unique`, or `seq` (for sequences).

1.1.8 Object Diagrams

C. Object Diagrams describe instances of the classes defined in a class diagram. For example, Figure 1.2 shows an instance conforming to the class diagram in Figure 1.1. It includes three objects, each identified by a unique ID (e.g., `o1`, `o2`, `o3`). For instance, object `o1` has as attribute a collection of 3 integers and is connected to other objects (e.g., `o2` and `o3`).

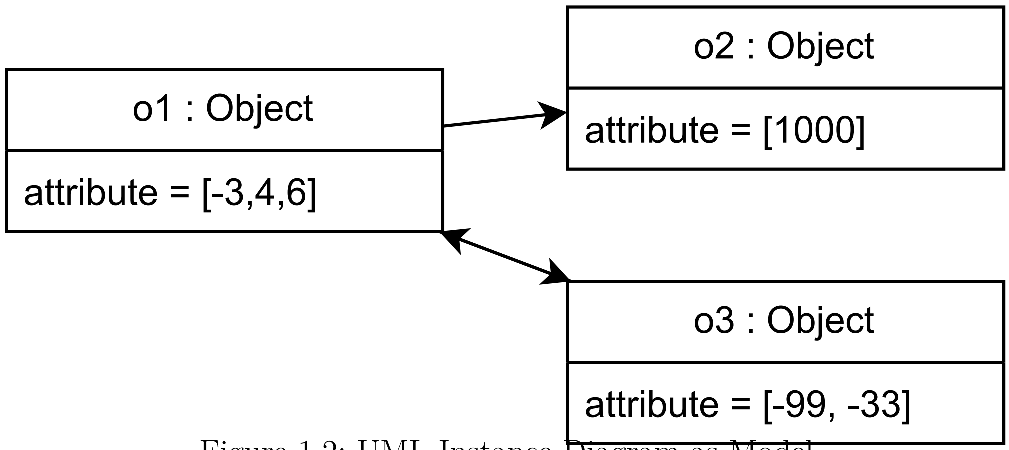

Figure 1.2: UML Instance Diagram as Model

1.1.9 Object Constraint Language

D. The Object Constraint Language (OCL) is a declarative language used to specify additional rules and constraints on UML models that cannot be expressed using diagrams alone. It enables the formalization of conditions that instances of the model must satisfy, serving as a powerful complement to class and object diagrams. For example, in the context of a family tree, a constraint such as "a child must be younger than their parents" cannot be represented directly in a class diagram. However, it can be expressed in OCL as follows:

```
context Person inv:
  self.parents.age.forall(a| a > self.age)
```

This constraint states that for every Person instance, all of their parents must be older. The `context` keyword specifies the class to which the constraint applies, and `inv` stands for invariant, i.e., a condition that must always hold true. This invariant states that for every Person, the age of each parent must be greater than the person's age. OCL Supports navigation expressions (e.g., `self.parents.age`) and collection operations (e.g., `forall`, `exists`, `size`) that apply to attributes and references. The expression `self` refers to the current object, `self.attribute` returns its attribute values, and `self.reference` retrieve related objects. Chained queries like `self.reference.attribute` retrieve the attributes of referenced objects.

OCL also supports a rich set of operations on primitive types and collections. Examples include: Boolean expressions (`forall`, `exists`, `not`, `and`, `or`), Arithmetic and comparison (`+`, `-`, `>`, `<`), and Collection operations (`sum`, `size`, `includes`, `asSet`, `asSequence`, etc). Each collection type comes with its own operations and can be explicitly cast using operations like `asSet()`.

Given an instance such as the one shown in Figure 1.2, OCL is typically used to verify whether it satisfies the specified constraints. In this work, however, we aim to use OCL as a means to guide model search, thereby enabling the completion or correction of partial or inconsistent data. To this end, we propose an approach that reformulates OCL specifications as constraint satisfaction problems (CSPs). This paper focuses on how OCL's collection typing, defined in the Class Diagram, and type casting operations can be modeled using global constraints over bounded domains.

1.2 Constraint Programming

- a CP model = CSP (avoid confusion with MDE Model)
- CSP + Propagation + Search
- CSP = V,D,C
- global constraints: modeling and propagation
- problem variables, intermediate variables and search
- reification: constraints as variables
- notable global constraint for this paper: element, gcc, stable keysort, regular

Constraint Programming [?] is a powerful paradigm that offers a generic and modular approach to modeling and solving combinatorial problems. A CP model consists of a set of variables $X = \{x_1, \ldots, x_n\}$, a set of domains $\mathcal{D}$ mapping each variable $x_i \in X$

to a finite set of possible values $dom(x_i)$, and a set of constraints $\mathcal{C}$ on X, where each constraint c defines a set of values that a subset of variables $X(c)$ can take. Domains can be either bounded, defined as an interval $\{lb..ub\}$, or enumerated, explicitly listing all possible values (e.g., $1, 10, 100, 1000$). This distinction impacts the choice of constraints: for instance, the global cardinality constraint is more effective with enumerated domains. An assignment on a set $Y \subseteq X$ of variables is a mapping from variables in Y to values in their domains. A solution is an assignment on X satisfying all constraints.

1.2.1 CP Solvers

CP solvers use backtracking search to explore the search space of partial assignments. The main concept used to speed up the search is constraint propagation by *filtering algorithms*. At each assignment, constraint filtering algorithms prune the search space by enforcing local consistency properties like *domain consistency* (a.k.a., *Generalized Arc Consistency* (GAC)). A constraint c on $X(c)$ is domain consistent, if and only if, for every $x_i \in X(c)$ and every $v \in dom(x_i)$, there is an assignment satisfying c such that $(x_i = v)$.

1.2.2 Global Constraints

Global constraints provide shorthand to often-used combinatorial substructures. More precisely, a global constraint is a constraint that captures a relationship between several variables [?, ?], for which an efficient filtering algorithm is proposed to prune the search tree. In other words, the “global" qualification of the constraint is due to the efficiency of its filtering algorithm, and its capacity to filter any value that is not globally consistent relative to the constraint in question. Global constraints are thus a key component to solving complex problems efficiently with CP. Some notable examples of global constraints used in this paper are:

- Element is useful when "selecting a variable from a list" is part of the problem. Let $X = [x_1, \ldots, x_n]$ be an array of integer variables, $z \in \{1, \ldots, n\}$ be an integer variable representing the index, and y be an integer variable representing the selected value. *element*(y, X, z) [?, ?] holds iff $y = x_z$ and $1 \leq z \leq n$, this means that variable y is constrained to take the value of the z-th element of array X.

- Regular expression constraints are very expressive when describing sequences of variables, and offers powerful filtering. Let $X = [x_1, \ldots, x_n]$ be an array of integer variables and A be a finite automaton. *regular*(X, A) [?] enforces that the sequence of values in X must form a valid word in the language recognized by the automaton A.

- The *stable_keysort*(X, Y, z) [?,?] defined over two matrices of integer variables holds iff (1) there exists a permutation π s.t. each row y_k of Y is equal to the row $x_{\pi(k)}$ of X ($k \in \{1, \ldots, i\}$); (2) the sequence of rows in Y, truncated to the first z columns, is lexicographically non-decreasing; (3) if two rows in X have equal key values for the first z columns, then their relative order in Y must match their original order in X. Table **??** illustrates with an instance that satisfied this constraint.

- Cumulative is generally used for scheduling tasks defined by their start time, duration and resource usage: $< s_i, d_i, r_i >$. It requires that at any instant t of the schedule, the summation of the amount of resource r of the tasks that overlap t, does not exceed the upper limit C. The values of t range from: a the earliest possible start time s_i, to b the latest possible end time $s_j + d_j$. Let $S = [s_1, \ldots, s_n]$ be the start times of n tasks, $D = [d_1, \ldots, d_n]$ their durations, $R = [r_1, \ldots, r_n]$ their resource demands, and C the total capacity of the resource (a constant). *cumulative*(S, D, R, C) [?, ?] holds iff $\forall t \in [a, b], \sum_{i|s_i \leq t < s_i + d_i} r_i \leq R$ [?]. where $a = min(s_0, .., s_n)$ and $b = max(s_0 + d_0, .., s_n + d_n)$,

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.

Chapter 2

Contribution : OCL Variable Declaration `VarOperationExpression`

2.1 Introduction

I. Originally, ATLc would assume that the last `AttributeOrNavigationCallExp` would identify the variable of the expression

2.2 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 ??, 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 ??. 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.

```
-- Scenario S1
(*@\label{lst:ocl:var:char:s}@*) context Task inv
    SameCharacteristicConstraint:
     self.var('stage').machines
          ->forall(m | m.characteristics
               ->includesAll(self.characteristics))
-- Scenario S2
(*@\label{lst:ocl:var:char:m}@*) context Task inv
    SameCharacteristicConstraint:
     self.stage.var('machines')
          ->forall(m | m.characteristics
               ->includesAll(self.characteristics))
-- Scenario S3
(*@\label{lst:ocl:var:char:sm}@*) context Task inv
    SameCharacteristicConstraint:
     self.var('stage').var('machines')
          ->forall(m | m.characteristics
               ->includesAll(self.characteristics))
```

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

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 1 we apply the operator to `SameCharacteristicConstraint` in **??** 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).[1] Whereas, if one wants to enforce OCL, `var` is used as a hint to build the corresponding CSP.

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

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.

2.2.1 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 2.1 shows the AST of `SameCharacteristicConstraint` from 1 Scenario S3. We show `var` annotations as dotted rectangles.

Figure 2.1 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 2.1, 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 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 2.1 (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 1 has the same AST as in Figure 2.1, but only the `machines` node is annotated as `var`. Hence, in this case the CP

```
-- Scenario S1
(*@\label{lst:ocl:var:derive:s}@*) context Task inv
    SameCharacteristicConstraint:
     inv: Stage.AllInstances()
          ->select(s|
               s.machines.forall(c | c.characteristics
                    ->includesAll(self.characteristics))
          ->includesAll(self.var(stage)))
-- Scenario S2
(*@\label{lst:ocl:var:derive:m}@*) context Task inv
    SameCharacteristicConstraint:
     inv: Machine.AllInstances()
          ->select(m| m.characteristics
               ->includesAll(self.characteristics)
          ->includesAll(self.stage.var(machines)))
-- Scenario S3
(*@\label{lst:ocl:var:derive:sm}@*) context Task inv
    SameCharacteristicConstraint:
     inv: Machine.AllInstances()
          ->select(m| m.characteristics
               ->includesAll(self.characteristics)
          ->includesAll(self.var(stage).var(machines)))
```

Listing 2: Annotated `SameCharacteristicConstraint` from Listing 1 refactored around the annotations.

scope is smaller, since `self.stage` can be directly evaluated by OCL.

2.2.2 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 1, all have their annotations low in the tree Figure 2.1. 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 2 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 2.2 we can see the AST resulting from the parsing of the expression of 2 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.

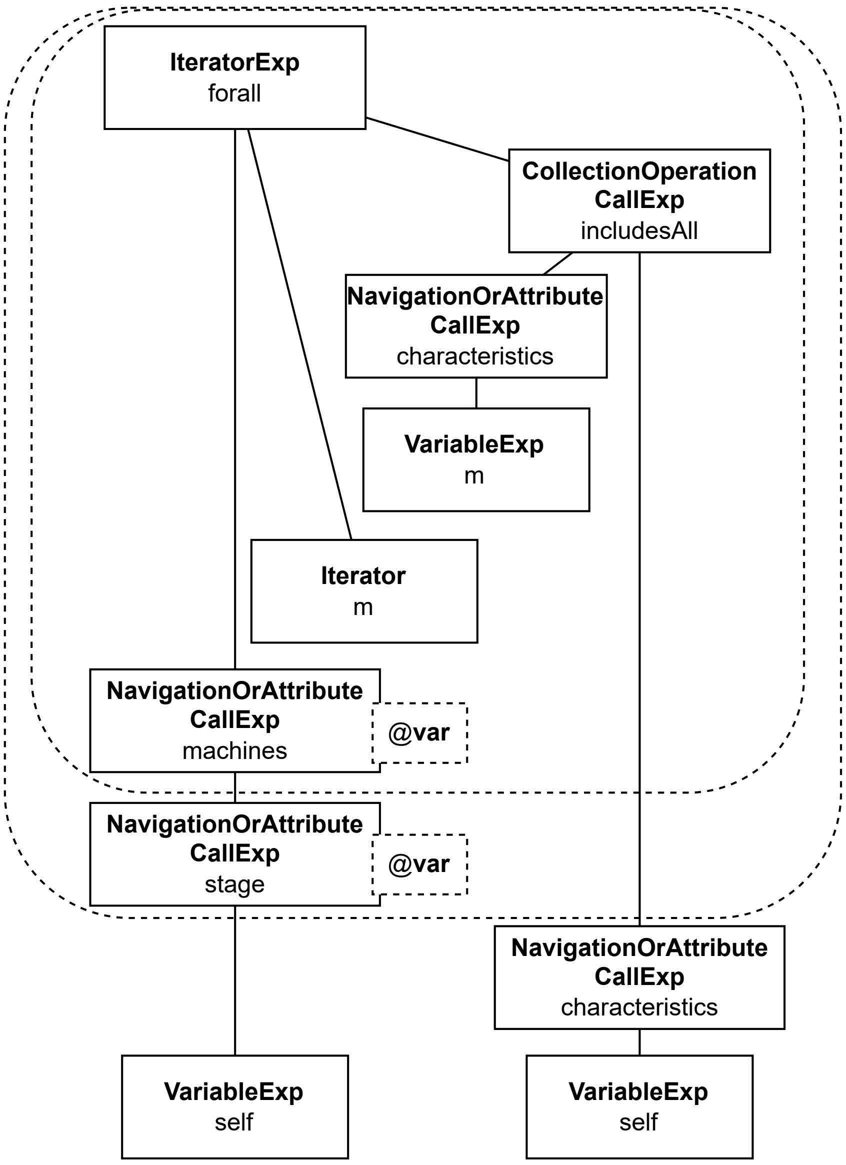
IteratorExp
forall
CollectionOperation CallExp
includesAll
NavigationOrAttribute CallExp
characteristics
VariableExp
m
Iterator
m
NavigationOrAttribute CallExp
machines
@var
NavigationOrAttribute CallExp
stage
@var
NavigationOrAttribute CallExp
characteristics
VariableExp
self
VariableExp
self

Figure 2.1: AST of `SameCharacteristicConstraint` from 1 Scenario S1, S2 & S3. [AST]

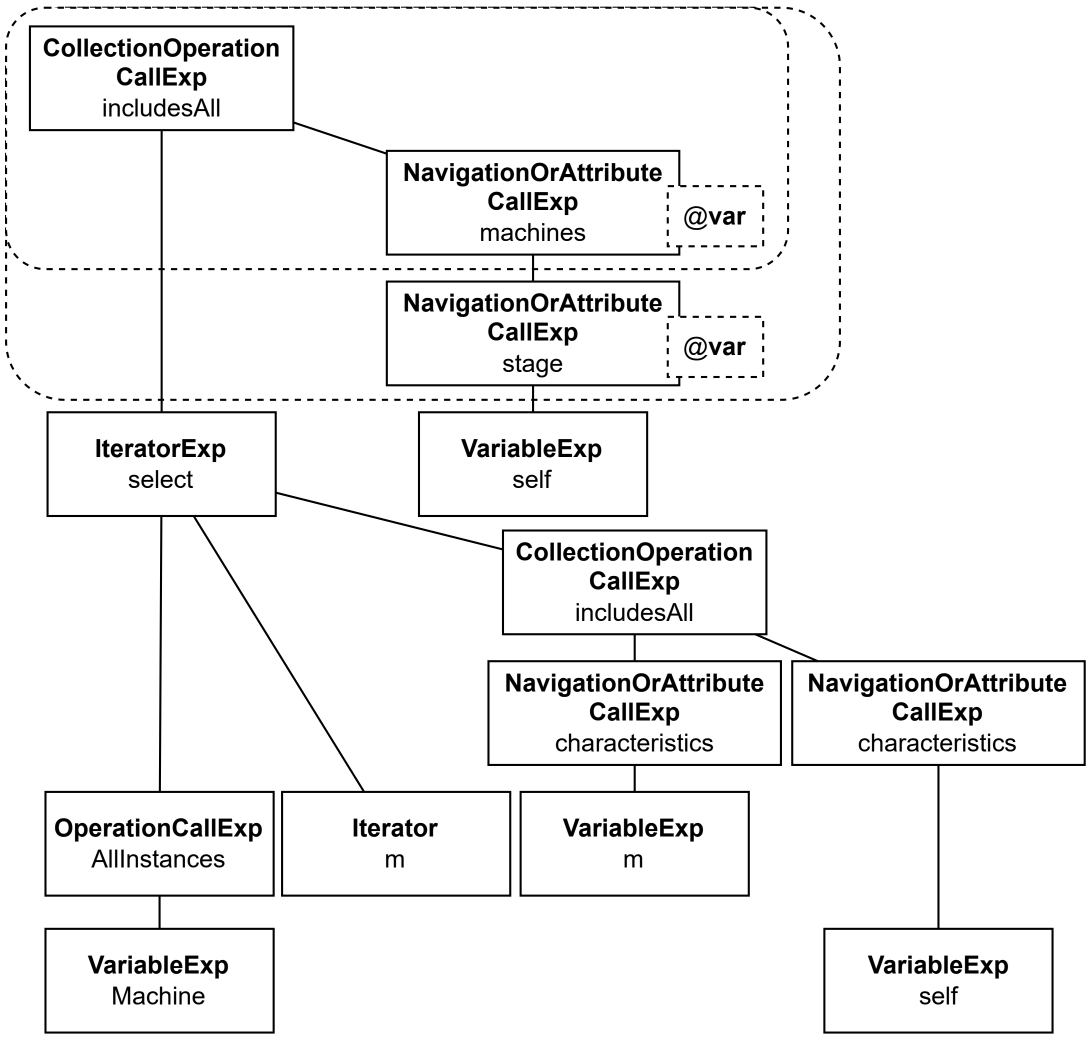

Figure 2.2: AST of SameCharactertic Constraint from 2 Scenario S2 & S3 [AST]

Chapter 3

Contribution : UML CSP

I. We can model Integer Collections type properties (it is possible to encode a lot of finite domain problems for different property types)

II. This encoding works for both integer attributes and object references

III. References has have extra "layers of models", leveraging the limited domains for pointer variables

3.1 Encoding Properties

A. Encoding Properties. The variables represent the properties–attributes and references– of the objects in the instance. Each class property is encoded as a matrix of integer variables, denoted $Class.property$. Each row in this matrix corresponds to one object of the class; for example, the i-th row is noted as $Class_i.property$ where $i \in [1, o]$ and $o = |Class|$ is the number of objects of that class. The number of columns p in this table is derived from the property's cardinality, which is given by n and m from Figure 1.1.

$$Class.property = \{x_{11}, ..., x_{op}\}$$

$$\forall x \in Class.property,\ domain(x) = \{d\} \cup \{lb..ub\}$$

Each property variable x_{ij} in the matrix has a domain defined by a lower bound lb, an upper bound ub, and a special dummy value d, where we set $d = lb - 1$. The property type, e.g., reference or attribute, determines the specific domain bounds. Attributes with an integer type may require large ranges, making domain enumeration impractical. This limits our ability to use certain global constraints like $global_cardinality(c)$onstraint, which counts the occurrences of domain values and therefore require finite, reasonably small domains. By default we chose a 16-bit range for these values: $lb = -32768$ and $ub = 32767$,

Object.attribute			
$Object_i$	**attribute**		
0	d	d	d
1	-3	4	6
2	1000	a_{22}	a_{23}
3	-99	-33	a_{33}

Object.reference		
$Object_i$	**reference**	
0	nullptr	nullptr
1	3	2
2	ptr_{21}	ptr_{22}
3	1	ptr_{32}

Table 3.1: Encoding of the instance from Figure 1.2 as tables of integer variables

meaning $d = -32769$, but these bounds can be refined by annotating the model accordingly [?]. For reference properties, the domain is defined as $\{1, \ldots, o\} \cup \{nullptr\}$, where o is the number of instances of the target class. These variables, named `ptr`, acts as pointers: values in $[1, o]$ identify object rows, and 0 (i.e., dummy value nullptr) denotes the absence of a reference. To support nullptr, an extra row is added to each table to represent a dummy object.

Table 3.1 shows the encoding of the instance from Figure 1.2, assuming $n = 2$ and $m = 3$ from the metamodel in Figure 1.1. The left side represents attributes, while the right side represents references. Each object (plus one dummy object) gets a row. The attribute variables a_{ij} are assigned the domain $\{lb, \ldots, ub\} \cup \{d\}$, and reference variables ptr_{ij} are assigned the domain $\{1, \ldots, o\} \cup \{nullptr\}$ with $o = 3$.

Model construction proceeds in two steps. First, we create matrices of variables with their full domains. Second, we instantiate some of these variables using data from the actual instance. In our current setting, we assign the exact values from the instance. Variables that remain uninstantiated may either be assigned dummy values or left free to explore during search, depending on the objective of the analysis. To choose between these behaviors and to reduce the size of the CSP, in previous work we've proposed an annotation system for OCL [?], which allows the user to identify variables. These annotations split the OCL expressions into parts which can be dispatched between our CP interpretation, and that of a standard interpreter. This reduces the scope and size of the CSP, notably in terms of modeled properties.

3.2 CP Models for UML Collection Types

As described in Section **??**, properties in class diagrams (e.g., Figure 1.1) can be annotated with collection types: `Sequence`, `Bag`, `Set`, or `OrderedSet`. These types can be enforced through constraint models to ensure consistency and reduce symmetries in the data..

For `Sequence`, permutations of the same multiset (e.g., $\{1, 1, 2\}$) yield distinct sequences. However, in our encoding, sequences such as $\{1, 2, d, 1\}$ and $\{1, 2, 1, d\}$ are treated as equivalent, since they encode the same effective ordering of values (e.g., the position of the dummy value d is ignored). To correctly model sequences, we impose an ordering where all dummy values are grouped at the end.

Let $X = \{x_1, .., x_p\}$ be the variable array for a property in the matrix $= Class.property$. The `Sequence` constraint is defined as: $Sequence(X) \iff \forall i \in [1, p[, (x_i = d) \Rightarrow (x_{i+1} = d)$. This ensures dummy values appear only at the end. We reformulate it using the `regular` global constraint applied to a Boolean mask $S = \{s_1, \ldots, s_p\}$:

$$Sequence(X) : \begin{cases} regular(S, DFA) \\ s_i = [\![x_i \neq d]\!] \end{cases} \tag{3.1}$$

The automaton *DFA* (Figure 3.1) accepts patterns of the form 1*0*, ensuring non-dummy values precede dummy ones. Here, S acts as a mask distinguishing actual values (1) from dummies (0) while avoiding symmetry.

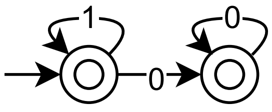

Figure 3.1: DFA packing dummy values for instance variables

For the `Bag` and `Set` types, all permutations of values are considered equivalent. To remove ordering symmetries, we sort the values in decreasing order, effectively pushing dummy values to the end:

$$Bag(X) : \Big\{ \forall i \in [1, p[: x_i \geq x_{i+1} \tag{3.2}$$

To model `Set`, we additionally enforce uniqueness among non-dummy values. Indeed, the sequence $\{d, d, d\}$ would be interpreted as an empty set. To this end, we define a relaxed variant of the `alldifferent` global constraint:

$$\begin{aligned} & alldifferent\_except\_d(X) \\ \iff & \ \forall\, i, j\, (i < j) \in [1, |X|], (x_i \neq x_j) \ \vee \ (x_i = x_j = d) \end{aligned}$$

$$Set(X) : \begin{cases} alldifferent\_except\_d(X) \\ Bag(X) \end{cases} \tag{3.3}$$

For `OrderedSet`, both value order and uniqueness matter. We combine the constraints used for `Sequence` and `Set`: dummy values must be packed at the end, and non-dummy values must be pairwise distinct. Formally:

$$OrderedSet(X) : \begin{cases} alldifferent\_except\_d(X) \\ Sequence(X) \end{cases} \tag{3.4}$$

This ensures a well-formed sequence without repetitions, where dummy values are ignored in uniqueness checks and appear only at the end of the array. These CP encodings ensure that model properties respect their specified UML and OCL collection types, enabling correct interpretation and reducing symmetry in instance generation.

CHAPTER 4

CONTRIBUTION : OCL NAVIGATION

4.1 CP model for OCL queries on the instance

Querying an instance involves navigating the object graph through references and retrieving attribute values. In OCL, navigation refers to the operation that, given source collection of objects and a reference property, returns a collection of target objects through that reference. We conflate this with attribute operations–as defined in the OCL specification–which return a collection of attribute values from a source collection of objects. In our encoding, both navigation and attribute access results are uniformly represented as integer variables.

Consider an OCL expression of the form `src.property`, where `src` is itself an expression like `self.reference` or `self.reference.reference`.

Let $Ptr = \{ptr_1, ..., ptr_z\}$ be the variables encoding the evaluation of `src`, with $dom(ptr_i) = \{1..o\} \cup \{nullptr\}$. Let T be the flattened array representing the `Class.property` matrix for `property`, where `property` refers to either an attribute or reference of the referenced class. Let p be the number of columns in the matrix. Let $Y = \{y_1, \ldots, y_{z \cdot p}\}$ be the variables representing the result of `src.property`. To link Y with T and Ptr, we define the navigation constraint:

$$nav(Ptr, T, Y) \iff \forall i \in [1, z], \forall j \in [1, p] : y_{(i-1)p+j} = T_{ptr_i \times p+j}$$

This constraint links the source pointers to the appropriate rows in the property table. This is reformulated in CP as a conjunction of *element* constraints, using intermediate variables for encoding the pointer arithmetic ($ptr_i \times p + j$):

$$nav(Ptr, T, Y) : \begin{cases} \forall i \in [1, z], \forall j \in [1, p] : \\ \quad ptr'_{ij} = ptr_i \times p + j \\ \quad element(y_k, T, ptr'_{ij}),\ k = (i-1)p + j \end{cases} \tag{4.1}$$

The intermediate variables introduced are functionally dependent on the Ptr variables and do not require enumeration during search. Given Ptr and T, the value of Y can be

Object.reference.attribute						
$Object_i$	**reference.attribute**					
1	-99	-33	a'_{13}	1000	a'_{15}	a'_{16}
2	a'_{21}	a'_{22}	a'_{23}	a'_{24}	a'_{25}	a'_{26}
3	-3	4	5	a'_{34}	a'_{35}	a'_{36}

Table 4.1: Encodings of `self.reference.attribute` for all objects of Figure 1.2 as a table of integer variables

determined. However, given an instantiation of Y, this model cannot fully determine Ptr and T, but it can filter to some extent. Thus, OCL query variables depend on the instance variables, and a query result may correspond to multiple instances.

It is important to note that the intermediate variables introduced by this reformulation are functionally dependent on the Ptr variables of the constraint. This means, we do not need to enumerate upon these variables during search. Similarly, given an instantiation of Ptr and T, in the context of model verification for example, we can determine Y. However, given an instantiation of Y, this model alone cannot determine Ptr and T, but it can filter to some extent. In the overall CSP this means that the variables encoding the OCL queries are all functionally dependent on the instance variables, but a query that solves the problem isn't associated with one instance.

Table 4.1 shows the results of the query `self.reference.attribute` using the navigation CP model (4.1) on the instance from Table 3.1. Result variables a'_{ij} share the same domains as a_{ij} but follow the reference and attribute order, introducing gaps due to ordering, e.g., a'_{13} might be the third variable, but yield a different third value (e.g., 1000) if $a'_{13} = d$. Similar effects occur in other OCL reformulations like union and append. Despite these gaps, value order and duplicates are preserved. These outputs are interpreted using the same models used for casting to collection types, such as `asSequence()`, discussed in Section 5.1.

Table 4.1 shows us the result of query `self.reference.attribute` using the navigation model CSP 4.1 on the instance from Table 3.1 The variables in this table, noted a'_{ij}, have the same domain as the a_{ij} variables. We also find the instantiated values, with respect to the order in the reference and the attribute: the variables of the first referred object come first, in the same order as in their original table. This introduces gaps between values, as illustrated by the first line: the third variable is a'_{13}, but if $a'_{13} = d$ the third value is 1000. Our reformulations of other OCL operations, such as union and the sequence operation append, similarly introduce gaps. However, despite these gaps, the order and multiplicity across values are preserved. The models to get a correct interpretation these collections are the same as the models to cast to a collection type, such as `asSequence()`, and are explored in Section 5.1. We will therefore need models to interpret these as the correct collection type.

4.2 NavCSP experimentation

Navigating a model adds a great deal of complexity. The pointer navigation Equation 4.1 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.

OCL Query Dimensions

To evaluate the navigation provided by Equation 4.1, we will look at the size of the CSP modeling the following OCL expression:

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

Such that `self.ref` is reflexive variable reference, modeled with N pointer variables, identifying objects of the same type. 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`.

OCL Query Size

In Figure 4.1 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 4.1 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.

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

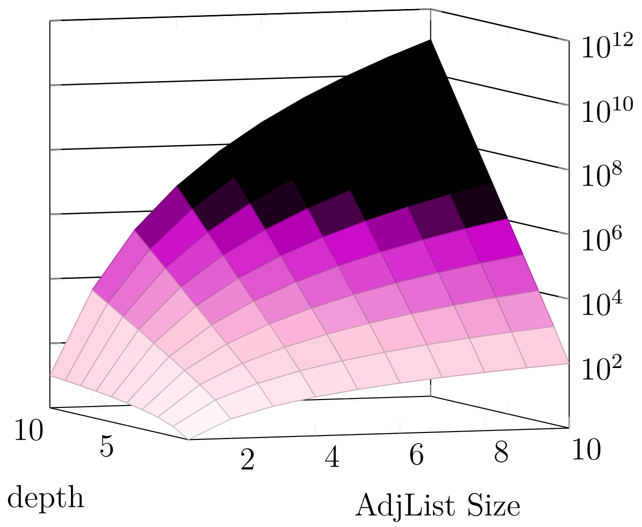

Figure 4.1: Number of query atoms in relation to `AdjList` size and navigation depth [AST]

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.

Subset Sum Problem

[1] 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 4.1, the lightest area being

[1]`https://github.com/ArtemisLemon/navCSP_SubsetSum`

quickly solvable, the darker area being quickly verifiable and the black area being too big to model.

Chapter 5

Contribution : OCL CSP

5.1 CP Models for OCL Collection Type Casting Operations

To illustrate OCL type casting, consider the following invariant, taken from the Zoo Model used in the experimentation:

```
context Cage inv:
    self.animals.species.asSet().size < 2
```

If `self.animals.species` evaluates to the sequence {1,1,1}, applying `asSet()` yields the set {1}, indicating that the cage contains a single species of animal. In the following, we define CP models to capture such collection type conversions.

Consider an expression of the form `src.asOP()`, where `src` is a collection-valued expression such as `self.attribute`, and `asOP()` denotes a type-casting operation applied to the source collection (e.g., `asSequence()`, `asSet()`, etc.). Let $X = \{x_1, ..., x_z\}$ be the array of variables modeling the values of `src`, and let $Y = \{y_1, ..., y_z\}$ represent the resulting collection after applying `asOP()`.

A. asBag(): Consider the expression `src.asBag()`, where the result is evaluated as a multiset Y that preserves all values from the source collection X, including repeated elements. Because OCL bags are insensitive to permutations, multiple orderings of the same values are semantically equivalent. To avoid such symmetries in the model, we impose a canonical form by sorting Y in descending order. This also ensures that any dummy values d used to pad the collection appear at the end. For example, given $X = \{1, 2, d, 1\}$, we enforce the canonical bag representation $Y = \{2, 1, 1, d\}$. This transformation is modeled using the global constraint $sort(X, Y)$, which sorts X into Y.

$$asBag(X, Y) : \Big\{ sort(X, Y^{\text{rev}}) \tag{5.1}$$

Y^{rev} denotes the reverse of Y, used to enforce descending order.

B. asSet(): Consider the expression `src.asSet()`. The `asSet()` operation removes duplicate elements from the source collection X while disregarding order. In our encoding, this corresponds to extracting the distinct values from X and placing them into the result array Y in a canonical form. Since the number of unique elements in X is not known beforehand, Y is defined with the same arity as X, and any unused positions are filled with a dummy value d. For instance, given an instantiation $X = \{1, 2, 1, d\}$, the result of `asSet()` would be $Y = \{2, 1, d, d\}$.

$$asSet(X, Y) : \begin{cases} sort(X, S^{\text{rev}}) & \\ X' = S \parallel \{d\} & \\ Y' = Y \parallel \{d\} & \\ p_1 = 1 & \\ \forall i \in]2, z+1] : & p_i = p_{i-1} + [\![x'_{i-1} \neq x'_i]\!] \\ & element(x'_i, Y', p_i) \\ \forall i \in [1, z] : & y_i \geq y_{i+1} \end{cases} \tag{5.2}$$

To enforce the `asSet()` semantics, we first sort the source array X in descending order into an auxiliary array S. We then define an array of position variables P and compute the position p_i of each variable s_i in a new array Y, ensuring that repeated values in S map to the same position. The first occurrence of a new value increments the position counter: $p_i = p_{i-1} + [\![s_{i-1} \neq s_i]\!]$. To support cases where all positions in Y are filled with unique values, we append a dummy value d to S, yielding $X' = S \parallel \{d\}$. In the case where all values in X are distinct (e.g., $X = \{2, 3, 1, 4\}$), the dummy has no room in Y. We resolve this by appending a dummy value to Y as well, forming $Y' = Y \parallel \{d\}$. This dummy will occupy the first unused position in Y, and all subsequent positions are forced to d by a descending sort constraint $y_i \geq y_{i+1}$. The final mapping from positions p_i to Y' is enforced via an $element(c)$onstraint over X' and Y'.

C. asSequence(): The `asSequence` operation retains all values from the source collection, including duplicates, and reorders them such that all non-dummy values appear first in their original relative order, followed by the dummy values. For example, if $X = \{1, d, 2, d, 1\}$, then `asSequence` yields $Y = \{1, 2, 1, d, d\}$. To enforce this transformation, we introduce the following CP model:

$$asSeq_{x2y}(X, Y) : \begin{cases} stable\_keysort(\langle B, X\rangle, \langle B', Y\rangle, 1) \\ \quad b_i = [\![x_i = d]\!], \forall i \in [1, z] \\ \quad b'_i = [\![y_i = d]\!], \forall i \in [1, z] \end{cases} \tag{5.3}$$

Index	B	X	**Sorted Index**	B'	Y
1	0	1	1	0	1
2	1	d	3	0	2
3	0	2	5	0	1
4	1	d	2	1	d
5	0	1	4	1	d

Table 5.1: Example of `asSequence()` transformation using stable sort. Dummy values are in red.

Here, B and B' are arrays of integer variables of size z, of domain $0, 1$, used as booleans indicating which variables in X and Y are equal to the dummy value d. The *stable_keysort*(T, S, k) constraint takes a matrix T and produces a sorted matrix S, ordering rows based on the first k columns, which form the sort key. In our case, we construct the matrices $\langle B, X\rangle$ and $\langle B', Y\rangle$, and sort on the first column, which separates dummy and non-dummy values while preserving the original order of the non-dummy elements.

To illustrate, let $X = \{1, d, 2, d, 1\}$, yielding $B = \{0, 1, 0, 1, 0\}$. We apply a stable sort to B, considering the pairs (b_i, x_i), and sorting by the key b_i. This ensures that all 0s (non-dummy values) appear before all 1s (dummy values), and the relative order of elements with the same key (e.g., all 0s) is preserved (see Table **??**). The sorted Boolean array becomes $B' = \{0, 0, 0, 1, 1\}$. Applying the permutation used to sort B to the array X results in $Y = \{1, 2, 1, d, d\}$.

One of the strategies during the search process involves enumerating the variables representing the top-level nodes in the OCL abstract syntax tree (AST). For example, in the expression `src.asSequence().sum()<3`, we explore possible values for `.sum()`, which helps filter the values of `src.asSequence`. To extend this filtering process down to `src`, an additional model is needed to manage the refinement. To filter from Y to X, we use a cumulative constraint, commonly applied in task scheduling. In this approach, we treat the intervals between values in Y as blocking tasks that prevent certain values from X during scheduling. By scheduling the tasks derived from X around the blocking intervals from Y, we filter down the possible values for X, effectively refining the search space according to the constraints set by Y.

$$asSeq_{y2x}(X,Y) : \begin{cases} & sort(Y,Y') \\ \text{let } T_y & \text{be the set of tasks such that:} \\ \forall i \in [1,z[& : s_i = y'_i + 1 \\ & \quad d_i = \max(0, y'_{i+1} - y'_i - 1) \\ & \quad h_i = z \\ \text{let } T_x & \text{be the set of tasks such that:} \\ \forall i \in [1,z] & : s_i = x_i \\ & \quad d_i = 1 \\ & \quad h_i = 1 \\ & cumulative(T_y \cup T_x, z) \end{cases} \tag{5.4}$$

Equation (5.4) defines how to filter values of X based on the sequence Y using a cumulative constraint:

1. First, Y is sorted into Y' to identify ordered non-dummy values.
2. From Y', we define blocking tasks T_y representing disallowed intervals. Each task (associated to value y'_i in Y'):
   - Starts at $s_i = y'_i + 1$,
   - Has a duration $d_i = \max(0, y'_{i+1} - y'_i - 1)$,
   - Has a height of $h_i = z$, fully consuming the resource and thus excluding X from that interval.
3. For each variable $x_i \in X$, a task is created in T_x starting at x_i, with duration 1 and height 1.
4. The cumulative constraint on $T_y \cup T_x$ ensures tasks from X are only scheduled in the non-blocked intervals.

In Figure 5.1, blocking tasks (highlighted in red and blue) are created from the intervals between values in Y', representing values that are prohibited for X. The white space represents the available slots for scheduling tasks from X. This model effectively restricts the possible values for X by ensuring that certain values, determined by the sorted sequence Y', are "blocked" from being selected, refining the search space. Combining both

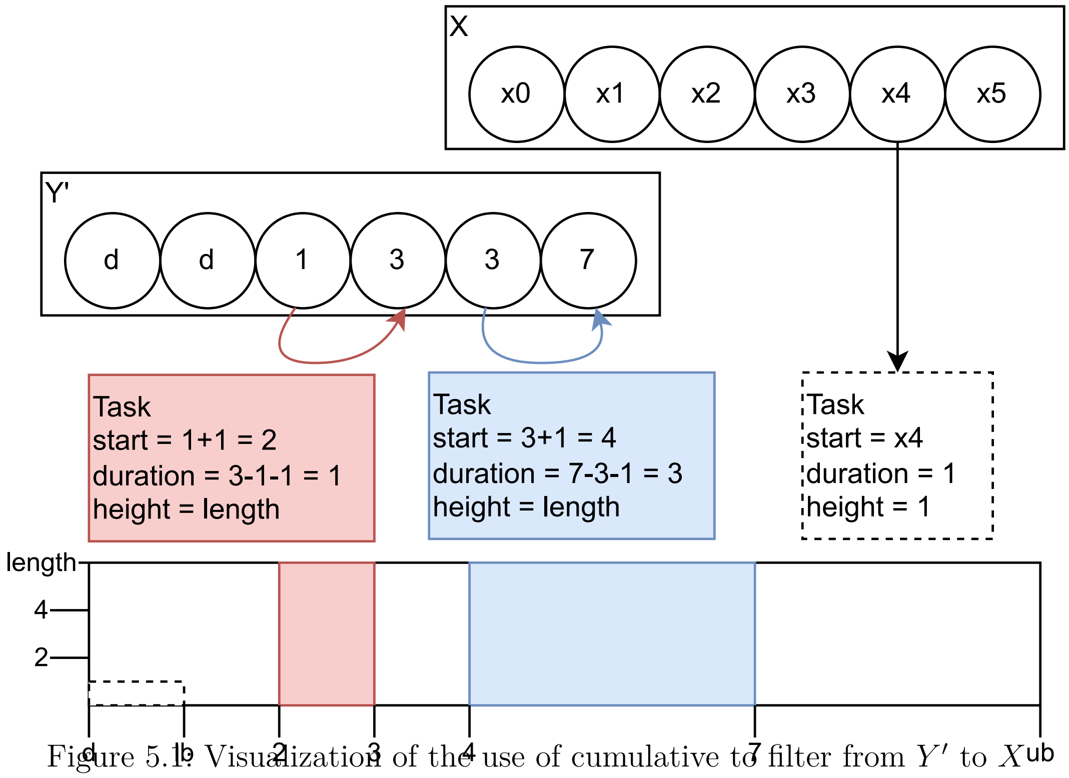

Figure 5.1: Visualization of the use of cumulative to filter from Y' to X

the X to Y and Y to X models give us the complete model for `asSequence`.

$$asSequence(X,Y): \begin{cases} asSeq_{x2y}(X,Y) \\ asSeq_{y2x}(X,Y) \end{cases} \tag{5.5}$$

D. asOrderedSet(): Consider the expression `src.asOrderedSet()`. The `asOrderedSet()` operation removes duplicates from X while preserving the relative order of first occurrences. Unused positions in Y are filled with dummy values d. For example if $X = \{1, 2, d, 1\}$, then `asOrderedSet` returns the array $Y = \{1, 2, d, d\}$. To enforce this behavior, we use the following CP model:

$$asOrdSet(X,Y): \begin{cases} stable\_keysort(<X,Y'>,<S,T>,1) \\ t_1 = s_1 \\ \forall i \in]1,z] : t_i = \begin{cases} s_i & \text{if } s_i \neq s_{i-1} \\ d & \text{otherwise} \end{cases} \\ asSequence(Y',Y) \end{cases} \tag{5.6}$$

The idea is to sort X into S to group identical values. We build T by keeping the first occurrence of each value in S and replacing subsequent duplicates with the dummy value

d. We then invert the sort to obtain Y', restoring the original structure. Finally, we apply `asSequence` to push all dummy values to the end, yielding the final ordered set Y.

Given $X = \{2, 1, 2, 3\}$, sorting yields $S = \{1, 2, 2, 3\}$, filtering gives $T = \{1, 2, d, 3\}$, reversing the sort results in $Y' = \{2, 1, d, 3\}$, and packing dummies yields $Y = \{2, 1, 3, d\}$.

E. Filtering Dummy Values in OCL Collection Operations For many OCL collection operations, the filtering process from X to Y can be enhanced by introducing a dedicated constraint to handle dummy values. This filtering mechanism can be integrated into models such as 5.1, 5.2, 5.5, and 5.6. The filtering approach is inspired by the strategy used in Equation (3.1), employing a *regular* constraint over a masked array:

$$dChannel(X, Y) : \begin{cases} regular(S, NFA) \\ \text{where } s_i = [\![s'_i \neq d]\!], i \in [1, z] \\ \text{with } S' = X \;\|\; c \,\|\; Y^{\text{rev}} \end{cases} \tag{5.7}$$

The mask encodes non-dummy values with 1s and dummy values with 0s. The *regular* constraint is applied over the concatenated sequence $S' = X \,\|\, c \,\|\, Y^{\text{rev}}$, where c is a counter variable ranging from 0 to z. The non-deterministic finite automaton (NFA), shown in Figure 5.2, ensures that the number of 0s (i.e., dummies) in X is matched by the same number of leading 0s in Y^{rev}.

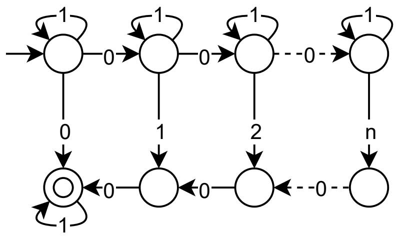

Figure 5.2: Non-Deterministic Finite Automaton accepting strings where Y starts with the same number of 0 found in X.

Given a partial instantiation such as $X = \{x_1, d, x_3, d, x_5\}$, this constraint allows filtering to deduce $Y = \{y_1, y_2, y_3, d, d\}$.

Chapter 6

Usage Patterns for Off-the-Shelf AI in Model Transformation Tools

6.1 Introduction

Models and model transformations are a core part of model driven engineering, a well-known paradigm for structured software and system development. As the field finds more applications, the modeling and transformation requirements gain in complexity. This growth in complexity can come in the form of a satisfaction or optimization problem, for which algorithms and meta-heuristics have been provided by the artificial intelligence community. Integrating established artificial intelligence techniques allows engineers to augment their transformations, e.g. with smart rule application or by deducing part of the target model. Additionally, machine learning techniques can leverage data to infer complex transformations.

To understand the modeling processes and tools, particular models can be used, namely mega-models, i.e. diagrams describing modeling artefacts, and their relations. Using a language based on common patterns in these diagrams, allows for categorisation of the application of techniques within model transformations.

The objective of this paper is to find categories for applications of off-the-shelf artificial intelligence components in model transformations. Two applications can be said to be in the same category, if they fall under the same description. In order to provide concise and unambiguous descriptions of these applications, we propose to create a language, designed for the sole purpose of describing the application of artificial intelligence in the field of model transformations. The language can also be used to identify interesting applications which haven't been observed.

This paper is organised as follows. In section 6.2 we present the context of study. We define our pattern language in section 6.3 and in section 6.4 we apply existing approaches found in the literature. Finally, in section 6.5 and section 6.6 we discuss future possible patterns and conclude.

6.2 Context

6.2.1 Transformations and Transformation Tools

Models, Meta-Models and Transformations

Models are a concept widely used. Many kinds of model exist for: weather, economy, information. A good model follows a few guidelines: easy, representative, one could say with a goal in mind. Models exist in relation to the systems under study they represent. The aim of this representation is to make learning from or building the system easier. For instance, whole numbers can be considered as very simple models representing quantities; Numbers can be manipulated effortlessly compared to the quantities they represent.

Meta-Models, are themselves models, however specifically representing a set of models. Models within the represented set are said to be conforming models. Following the example with whole numbers; while each number is a model, we can represent the set of all numbers with $\mathbb{N}$, and describe the conformity relation as inclusion in this set: $n \in \mathbb{N}$.

$$\begin{tikzcd} MM \\ M \arrow[u, "c2"] \end{tikzcd}$$

((a)) Model conforming to Meta-Model

$$\begin{tikzcd} M_s & MTa \arrow[l, dash] \arrow[r] & M_t \end{tikzcd}$$

((b)) Transformation applications

$$\begin{tikzcd} MM_s & MT \arrow[l, dash] \arrow[r] & MM_t \end{tikzcd}$$

((c)) Transformation specifications

Figure 6.1: Basic patterns of models and transformations

In Figure 6.1(a) we can see the first part of the patterns seen in this paper. Depicted are two artefacts M and MM, respectively a model and the meta-model it conforms to. The conformity relation, is shown as an arrow labeled $\underline{\text{c2}}$ indicating the model M conforms to MM.

Transformations, are the operation applied to models, and are specified at the same level of abstraction as meta-models. In the case of [?] transformations are akin to whole number functions: sets of relations between antecedent and image, each of those relations is referred to as a **transformation application**. A **Transformation Specification** is commonly an artefact of a Transformation Framework, such as QVT [?] or ATL [?] file. Naturally these are also models; a transformation specification in ATL is a model conforming to the ATL meta-model. This implies that transformation themselves can be source and target of higher-order transformations [?].

Transformation Specifications, such as ATL or QVT, link between meta-models, generally as a collection of rules matching parts of source and target models.

In Figure 6.1(b) and Figure 6.1(c) we see again two artefacts, both models or both meta-models (respectively), related by a transformation-application, or a transformation-specification. These similar patterns, summarise each a level of abstractions: between models and between meta-models.

These concepts, modeling and transformations, are applied in many fields, beyond software engineering (the scope of this paper) or even engineering.

MDE Transformation Tools

There are a wide selection of transformation languages and tools, for models and for graphs. These tools and their use is described the by the core word of the pattern language presented in this paper (see section 6.3.1). Furthermore, the mega-models used to depict these tools, are some of the earliest mega-models [?] and informed how the pattern is depicted in this paper.

Transformation within engineering, and MDE can have a number of qualities and a number of ways to describe them. First the notion of a transformation's direction in regards to abstraction: transformations can move vertically, from abstract models (e.g. UML) to concrete ones (e.g. Java) or the other way around, and transformations can move horizontally at the same level of abstraction (Java to C++). Another important aspect of transformations is the relation between the source and target meta-models: in cases where they are different, we talk of Exogenous Transformations (UML to Java, Java to C++) allowing use to move between contexts, other wise we talk of Endogenous Transformations (Java to Java) which generally allows for tasks like optimisations, refactoring, etc.

Notable transformation tools of MDE found in this paper are QVT [?], ATL [?, ?], Viatra [?], JESS [?,?] and Hensin [?,?]. They each provide different concepts for specifying transformations, such as conditions for rule application, or graph representations and matching. Aside from transformation languages and engines, OCL is commonly used: it provides a query language to map meta-model classes and attributes, and can provide verification of the models and transformations.

6.2.2 Off-the-Shelf AIs

The definition of AI is fairly wide and what people picture when faced with the term has evolved over time. Today, Neural Networks and Evolutionary Algorithms, are first to mind, allowing us to find solutions to complex problems within reasonable effort, time and accuracy. These contrast with more mature AI tools based on logical induction or satisfiability which while often requiring more effort and time, can achieve exact and explainable solutions.

We will call AI, a generalised approach which can solve many problems. A **Problem Specification** is the encoding of the problem in a form suitable as input to the AI tool. Concretely in this paper we consider the following families of AI tools:

Logic Programming (LP) [?, ?, ?, ?, ?] has been a mainstay of AI since the beginning of the field. Notable tools from this field are Prolog and ASP, providing simple yet expressive languages to define NP class problems, and a variety of algorithms to find solutions. Problems and Solutions are encoded using axioms and rules of inference, allowing to search for solutions and prove them.

Similarly, **Constraint Programming** (CP or CSP) [?, ?, ?] provides simple languages to model complex problems, and additional algorithms and heuristics to search for solutions. Constraints can be satisfied, by searching for solutions. Problems are modeled using variables which can be numeric, sets, graphs, etc.., depending on the solver, and constraints between these variables such as: equality or inferiority between numbers, or inclusion for sets.

Genetic Algorithms (GA) [?, ?, ?, ?, ?] inspired by the mechanism of genetic evolution and natural selections, encode the problem and it's solutions in an evaluation function and a population of genomes, each individual genome being a possible solution. The genomes with the best value are selected to be mutated and mixed, providing a form of local search.

Particle Swarm Optimisation (PSO) [?, ?, ?] is another algorithm inspired by nature has found application in model transformations, itself a model of flocking birds. Like GA uses a population representing solutions, except solutions are encoded as positions in space, searching a position of high value.

Simulated Annealing (SA) [?, ?] provides a meta-heuristic for global search (as opposed to GA or PSO), again by modeling a natural process: the heat treatment of metals, with the aim to crystallise optimal solutions. Similarly to PSO, solutions are expressed as positions in space.

Q-Learning (QL) [?, ?] is a method to estimate the value of actions which can be

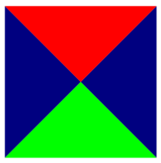

Figure 6.2: Example pattern language tile

$$\begin{tikzcd}
MM_s \arrow[r, dash] & \text{MT} \arrow[r] & MM_t \\
\text{c2} \arrow[u] & \text{execution-of} \arrow[u] & \text{c2} \arrow[u] \arrow[d] \\
M_s \arrow[u, dash] \arrow[r, dash] & \text{MTa} \arrow[u, dash] \arrow[r] & M_t
\end{tikzcd}$$

Figure 6.3: Transformation Pattern

performed given a certain state. In the context of model transformations, this allows us to represent models as states and transformations as actions, and explore the eventual values of transformation actions.

Neural Networks (NN) [?] are one of the AI getting the most attention today, specifically Long Short-Term Memory (LSTM) structures which show great promise with their application in large language models such as GPT. This version of neural networks, as do others, use a vectorised representation as input and output connected by a network of neurons.

6.3 Pattern Language

In brief terms, to give a first intuition on our understanding of pattern languages: a set of patterns which can be tiled to make bigger patterns, can be described as a language. A simple pattern similar to the one proposed by this paper, is portrayed in Figure 6.2, along with the rule that two patterns, can by placed side-by-side if their touching shapes are the same colour. The square title it's self is a pattern of coloured triangles, and by tiling the patterns together we can make larger patterns.

Our patterns share similar structure to this example, however require more nuance than the colors proposed here and additional structural rules for the larger pattern, and hence more elaborate rules such as: there must be a continuous chain of tiles connected along their blue sides, where none of the greens sides have a neighbour, but to motivate and understand this rule, we need to understand the semantics of our patterns.

In this paper, patterns used as tiles are mega-models depicting relations between models. Said mega-models are composites of Figure 6.1(a), Figure 6.1(b) and Figure 6.1(c) (which serve the role of the letters for the words of our language). The mega-models, or words of the language proposed here, have a similar structure to Figure 6.2: the blue areas housing mega-models of conformant-models and their meta-models as seen in Figure 6.1(a), the red and green areas, holding respectively mega-models or models related by transformation specification Figure 6.1(c) or application Figure 6.1(b).

The pattern makes use of two dimensions, patterns can be placed side-by-side from left to right, or placed above patterns. The primary dimension is drawn between a source and target model, made up of a continuous chain of transformations-like operations. The line from source to target model must remain unbroken and the lowest side of the composed pattern. Above transformations specifications, using secondary dimension: higher-order transformations can be chained. Allowing to create or augment the transformations applied.

We propose this pattern language over mega-models to help identify and categorise how artificial intelligence has been applied to model transformations. The vocabulary is made up of simple patterns representing the processes performed by transformation technologies and AIs, while the grammar describes how these actions and patterns can be chained and composed.

To develop this language, cases of artificial intelligence in model transformations [**?**, **?**, **?**, **?**] were mega-modeled using concepts from established mega-modeling languages [**?**, **?**], allowing the identifications and isolation of common patterns.

6.3.1 Words of the Pattern Language

We have three core patterns - words in the language - found in the literature. Two of these words, AI Learning and AI Processing specifically describe the use of AI in the model transformation process. Both of these processes can be implemented by a variety of the off-the-shelf AI suggested in subsection 6.2.2. These are in addition to the core word of our field: Transformation.

Transformation

The core pattern for this paper, described in Figure 6.3, is naturally that of transformations. In our context, a transformation implies a collection of artefacts and relations.

In Figure 6.3 we can see two models and the meta-models they each conform to. On the left source models, on the right target models. Between the meta-models the transformation specification MT and between the models a transformation application MTa. Additionally in this mega-model is represented the relation between definition and application, named here execution of.

This pattern is representative of standard transformation tools such as ATL or QVT, but also represents other tools and algorithms which can serve the purpose of transformations, such as general programming languages, logic programming languages, or even neural networks, or even whole transformations processes including a variety of these or AI Processes outside of the focus of the pattern.

AI Learning

This pattern mirrors the transformation pattern; in the previous pattern, the applications came as executions of the definition, in this pattern the definition is derived from applications or generally, mappings between sets of example source and target models. In the diagram, the direction of the arrow between Specification MT and Application MTa has been inverted. In our scope the responsibility of these higher-order transformations is conferred to artificial intelligence, in the hopes of allowing the easy definition of complex transformations.

In Figure 6.4 we find a similar pattern to transformation, at the top between meta-models we have a transformation specification MT. However at the model level-of-abstraction, the lower side of the pattern, we predominantly have sets of models and mappings between them, as many machine learning techniques rely on large amounts of example models, or in the case of reinforcing a transformation: prior executions. These sets M_s and M_t are depicted as conforming to their respective meta-models, which is to mean each model of either set is a conformant-model, and between them a form of mapping MTa, which is to say a set of source-target relations, like a function.

This figure is oriented with higher-levels of abstraction (meta-models) at the top, like with Figure 6.3 and Figure 6.5, however when in use, the pattern is flipped vertically. That way the vertical dimension of the diagram represents the process in place to generate an artifact that can substitute a transformation specification.

In our examples the resulting transformation specification can take on a few forms, as previously mentioned for the transformation mega-model, notable examples will be traditional definitions in ATL or JESS [?, ?], blackbox functions like neural networks [?] or an augmented version of a previous definition [?, ?].

$$\begin{tikzcd}
MM_s \arrow[r, dash] & \text{MT} \arrow[r] \arrow[d, "\text{learning-of}"] & MM_t \\
\{M_s\} \arrow[u, "\text{c2}"] \arrow[r, dash] & \{\text{MTa}\} \arrow[r] & \{M_t\} \arrow[u, "\text{c2}"']
\end{tikzcd}$$

Figure 6.4: Transformation Learning Pattern

$$\begin{tikzcd}
MM & \text{Problem specification} & MM' \\
M \arrow[u, "\text{c2}"] \arrow[r, dashed, dash] & \text{problem resolution} \arrow[u, "\text{execution-of}"] \arrow[r, dashed] & M' \arrow[u, "\text{c2}"']
\end{tikzcd}$$

Figure 6.5: Problem Solving Pattern

AI Processing

In Figure 6.5 we can see two models M and M' which each conform to their respective meta-models MM and MM'. The left side of the pattern shows the unsolved model, while on the right a model solution to the problem specification. The problem specification, is represented as a relation between the meta-model the source and target model conform to.

In the context of a problem like N-Queens, the unsolved model would be the instance of the board and the N-Queens, while the solved model would be the N-Queens on the board, none attacking another. The meta-model for both the unsolved and solved model are the same concept of chess boards and pieces, which is why we say they both conform to the same meta-model, and the rules of the game are the problem specification: two queens can't be on the same diagonal, two queens can't be in the same column, and two queens can't on the same line, of the chess board.

6.3.2 Connecting Patterns

Connecting, and merging these patterns will allow us to describe how AI is used to augment MT, while reusing simple concepts.

There are two main dimensions to composite patterns: along the main horizontal axis a sequence of transformation applications Figure 6.6. And along the vertical axis, higher-order transformations resulting in those applied Figure 6.4. The overall pattern, along it's lowest sequence, describes a timeline as a single source model is transformed into it's final target model. Along this line, patterns can be connected by sharing the same model.

To start a pattern, one can either choose a transformation Figure 6.3 or a problem resolution Figure 6.5 or merge them Figure 6.7 (which is explained in a following subsubsection).

Connecting Models and Meta-Models

The primary way of connecting patterns is by chaining transformation operations, meaning the output of the previous transformation is given as input to the following transformations. This behaviour is standard in Model Transformation Frameworks.

At the end of a sequence we can add a new instance of a transformation Figure 6.3 or a problem resolution Figure 6.5 or their combinations Figure 6.7 if and only if the target of the previous block and the source of the following block have matching models and meta-models. We can extend this concept to sets of meta-models, as a model can conform to multiple meta-models, and in this context the target meta-model of the previous transformation must be include in the set of source meta-models or the following, or vice-versa.

Connecting Transformation

This allows to add patterns above transformation specifications in the dominant chain of transformations. Above any transformation specification, we can choose to place a learning block, meaning the transformation specification was achieved through a learning process, based on sets of example models conforming to the source and target meta-models of the transformation specification we want to learn, and mappings between both sets demonstrating the desired transformations. These sets of example source models and target models can also be part of a transformation process, meaning we can place a transformation block above a learning block.

This implies a possible laddering of Learning-Of and Execution-Of patterns, which can portray the iterative process of reinforcing a transformation for example.

Merging Executions

Executions-of can be merged, which results in the MT + Problem Specification and applications. This is generally the result of a problem specification language and solver being additionally able to perform the transformation. A transformation can be expressed as a decision problem, in Prolog for example - a logic programming language and associated resolution techniques - one can use facts, and predicates to describe models and meta-models. While rules can be used to describe the transformation: in the head of rules

$$\begin{tikzcd}
MM_s \arrow[r, "\text{MT}"] & MM_t \arrow[r, "\text{Problem Specification}"] & MM_t \\
M_s \arrow[u, "\text{c2}"] \arrow[r, "\text{MTa}"] \arrow[ur, phantom, "\text{execution-of}"] & M_p \arrow[u, "\text{c2}"] \arrow[r, "\text{resolution}"] \arrow[ur, phantom, "\text{execution-of}"] & M_t \arrow[u, "\text{c2}"]
\end{tikzcd}$$

Figure 6.6: Two stage transformation, with first a standard transformation, followed by a problem resolution

are predicates for the target meta-model, while the body uses predicates of the source meta-model, using facts that fit those predicates. A problem specification, as previously described doesn't allow to transform from one meta-model to another, and a model transformation isn't responsible for problem resolutions, therefore even when the same process does both, we distinctly display both aspects.

6.4 Patterns in Transformation Literature

In a first pass through the literature, some notable applications of AI in model transformations were collected. From these examples and taking inspiration from prior mega-modeling languages, we elaborated a set of patterns which we classified in the following categories.

To test and refine the pattern language and the identified categories, we categorised approaches from two surveys [?, ?]. From these two surveys we started analysing papers within the scope of model transformations of which the first 14 are represented here.

6.4.1 Model Space Exploration

Transformations in most cases are deterministic: given the same input, the result of the transformation are all the same. It is conceivable for a desired transformation to have many acceptable targets for which, specifying a transformation to a single one of these, brings along a difficult decision problem. In the case of transformation with non trivial choices, in ATL you need to encode this into the transformation specification, if the requirements are too complex, constraint problems and their solvers can be applied to navigate this complex decision problem.

$$\begin{tikzcd} MM_s \arrow[rr, "\text{MT + Problem}", ""{name=T, below}] & & MM_t \\ M_s \arrow[u, "c2"] \arrow[rr, "\text{MTa + resolution}", ""{name=B, above}] & & M_t \arrow[u, "c2"'] \arrow[from=B, to=T, "\text{execution-of}"] \end{tikzcd}$$

Figure 6.7: Transformation and problem resolution performed as one execution

In Figure 6.6 we have a sequence Figure 6.3 and Figure 6.5. The source model is first transformed to a prototype model conforming to the target meta-model, which is subsequently transformed into a model satisfying the constraints.

Coupling Solvers with Model Transformations [?, ?]

This effort is an extension to the ATL engine ATOL, which allows engineers to use OCL to not only make verifiable assertions about the models, but express constraint problems to be solved.

Use-cases for this, illustrated by Figure 6.6, are when transforming to the provided target meta-models for JavaFX and Constraints, allowing for rapid development of applications based around graphical user interfaces. Users to define a meta-model and model for their application, and a transformation towards user-interface elements and constraints specifying desired positions for these elements. A first transformation from application model to JavaFX is performed, providing a prototype target model of which the attributes are refined using constraints.

The constraint meta-model allows to create solver independent models, permitting the use a variety of solvers, such as Choco and Cassowary..

In Figure 6.7 we see the merging of both steps in Figure 6.6 into one. Both the transformation and problem specification are combined, and the combined specification is executed.

Transformations as logic problem

In this sections we have two implementations of the logic solvers, first two efforts using Prolog to perform transformations and another implementing their own solver optimised for logic applied over models, and applying it to model transformations.

These two efforts, both using a Prolog Engine: both represent models, ensure their conformity, and perform transformations in the prolog language as facts and rules. [?, ?]

$$\begin{tikzcd}
MM_s \arrow[r, dash] & \text{MT} \arrow[r] & MM_t \\
\{M_s\} \arrow[u, "\text{c2}"] \arrow[d, "\text{c2}"] \arrow[r, dash] & \text{traces} \arrow[u, dashed, "\text{execution-of}"] \arrow[r] & \{M_t\} \arrow[u, "\text{c2}"'] \arrow[d, "\text{c2}"'] \\
MM_s \arrow[r, dotted, dash] & \text{augmented MT} \arrow[u, dashed, "\text{learning-of}"] \arrow[r, dotted] & MM_t \\
M_s \arrow[u, "\text{c2}"] \arrow[r, dash] & \text{MTa} \arrow[u, dashed, "\text{execution-of}"] \arrow[r] & M_t \arrow[u, "\text{c2}"']
\end{tikzcd}$$

Figure 6.8: Transformation augmented by Reinforcement Learning

$$\begin{tikzcd}
\{M_s\} \arrow[d, "\text{c2}"] \arrow[r, dash] & \text{mapping} \arrow[r] & \{M_t\} \arrow[d, "\text{c2}"'] \\
MM_s \arrow[r, dash] & \text{MT} \arrow[u, "\text{learning-of}"] \arrow[r] & MM_t \\
M_s \arrow[u, "\text{c2}"] \arrow[r, dash] & \text{MTa} \arrow[u, "\text{execution-of}"] \arrow[r] & M_t \arrow[u, "\text{c2}"']
\end{tikzcd}$$

Figure 6.9: Transformation learned from examples

However logical programming is able to express problems outside of model transformation, and thus can be used to express additional properties of the target model and transformation. These additional properties can be geometric constraints as in the example of [?].

An issue for MDE is allowing domain experts to model, and to subsequently transform those models for other domain experts. And prolog and logical programming isn't the best candidate modeling language for modelers. [?] adds a layer allowing users to use MDE standards such as MOF, QVT, and OCL to declare models, transformations and constraints.

Another solution to the layer above logic programming, connecting to MDE technologies is proposed in [?] where the models, transformations, and constraints are specified using the Viatra graph transformations language. This solution however also uses their own logic solver specialised in constraints over models, CSP(M).

6.4.2 Learning Rule Application

Another augmentation AIs can provide model transformations, is learning when to apply transformation rules; Some transformations might propose small number of rules which have to be repeatedly applied to progress towards the target, or a large number of rules of which a subset achieves the desired goals. This can have the effect of encoding decision problems into the transformation specification.

Figure 6.8 shows us three distinct groups of source and target artefacts. The source and target meta-models represented each twice, source and target execution examples, and the final source and target models. Between the source and target elements we have transformation-likes, most importantly an initial transformation specification MT, and a final augmented specification. the initial (or intermediate) specifications are executed over sets of examples $\{M_s\}$ and $\{M_t\}$, and from the traces left by those executions improvements for the MT can be learnt. The final execution MTa performs the task of producing the target M_t from the source M_s.

This figure shows a general case where source and target meta-models can be different, but most of the examples listed here perform and evaluate over sequences of transformations within a single meta-model, which allows to create optimising transformations.

Sequencing Transformation Rules [?, ?, ?, ?]

These first examples, tackle the use case of optimising code during compilation towards embedded systems. In this stage of optimisation, the program code is transformed to better suit the objectives. Programs and programming languages are similar concepts to models and meta-models, and share a similar conformity relation, but are distinct in their technical spaces. The optimisation of the code can be done by a collection of algorithms, in our context transformations, performing actions such as removing variables, refactoring functions, numbering global values, register coalescing, etc... [?] The order in which these transformations are performed can have a significant impact on the desired qualities of the code, such as code size, as such AI techniques are employed to determine the best sequence of operation for the transformation.

To use genetic algorithms, the order of operations is encoded into a genome, and a population of varying orders compete to produce the optimal code. The most successful orders, are mutated and combined with other successful orders, and the population is tested again, until a suitable order is found.

Reinforcement Learning applied to Model Transformations

These efforts explore the applications for reinforcement learning in model transformations. In the context of simple transformation rules which can be repeatedly applied to a model in order to progress towards a desired target, Q-Learning provides a framework to learn the the value of each available transformation rule given the current state of the model.

A simple use case which portrays these kinds of transformations, is that of Pacman [?]. The meta-model is that of pacman games: describing walls, pacman, ghosts and food. The models each represent a state of a pacman game: a maze of walls, remaining food in position on the floor, ghosts at their positions, and the pacman at theirs. Transformations describe going from one game state, to the next, such as pacman moving into an adjacent square with food and eating it. The objective is of the transformations, at each step, is to approach a model of a victorious game, with no food on the floor.

In this context, Q-learning allows the transformation to know which is the best rule to apply for a given model, and to chain those transformations to optimise the model.

The use case of [?] describes model repair, a specific kind of model transformation, which can result in the model's conformity to a meta-model, or an optimisation relating to the semantics of the model, such as constraints. Their transformations take the errors returned by EMF[1] tools on models being created, finds transformation rules to apply to improve the model. Each state represents a model with an error in focus, and each action an available repair operation for the error. Q-Learning is again a strong candidate to give value to the repairs done by transformation actions for a given state.

Selecting Transformation Rules to Apply

[?] stands out from the others in this section by allowing for exogenous transformations, and by performing a search for each transformation. The search this effort provides is that of which rules to apply, which again differs from previous examples, by encoding the application of a rule as a decision variable as opposed to it's position in the sequence or the expected return, furthermore allowing to use a sub-set of a larger number rules as opposed to multiple applications, encouraging the possibility for exogenous transformations. Having encoded rule application as decision variables and the transformation as an objective of a decision problem, the MOEA tool (using GA, PSO, etc..) is handed the responsibility of finding the correct set of rules to apply, allowing for the transformation.

6.4.3 Transformation by Example

The efforts listed in these applications of AI to model transformation, all share a similar goal of performing transformations, leveraging data from previously transformed models, allowing the reuse of previous efforts, and lessening efforts to automate future transformations. However different approaches use techniques from all across AI research, from classical AI atop of logic, to state-of-the-art neural network designs. The resulting transformation specifications are just as varied, ranging from specifications in transformation

[1]Eclipse Modeling Framework

languages such as ATL or JESS to instances of a neural network.

Figure 6.9 describes a situation where a transformation specification MT is derived from sets of example source $\{M_s\}$ and target $\{M_t\}$ models conforming to their respective meta-models MM_s and MM_t, and a mapping between them. This learned specification is then executed MTa to transform new models M_s into acceptable targets M_t.

Learning Black-Box Transformations

[?] employs a neural network to perform the transformation, allowing to approximate black box transformation specifications from examples of source models and their targets.

In this context we refer to the structure and the parameters of the neural network as a transformation specification, while the path of activation though the network caused by the input, resulting in activation at the output as an transformation application. This requires a transformation between classing modeling languages and vectorised representations which the AI can manipulate. The encoder neural network, takes a vectorised form of the model and transforms it to a model understood by the neural network. The decoder then performs the opposite, taking the neural network's understanding of the source, and translating to a vectorised form of the target model.

Learning Transformations as a Logic Problem

In the previously explored transformations as logical problems in section 6.4.1, source models are defined as fact, and logical inference rules allow to prove facts representing the target. The engineer would give source models and transformation rules, to get targets models. With the ALEPH ILP system, the application of which is explored in [?] as an extension to the work in [?], from solely providing facts - about background knowledge accompanied with positive and negative facts about the targets - we can find hypothesis to explain to positive facts from our knowledge, while not proving the negative facts.

In the context of model transformations, we can provide facts representing source and target models, and ask the ALEPH system to determine which logical inference rules can prove the target facts from the source facts, providing the logical transformation between them.

Learning Transformation Specifications [?, ?]

This effort tries to produce the code for transformations specifications (in JESS but applicable to others like ATL), using genetic programming. Like other genetic algorithms, this involves encoding the problem as a genome, however in this case the genome is the code, and the genetic operations guarantee that the derived programs are syntactically and semantically valid. Therefore one starts with example source and target models, a population random but correct transformation rules, and a function to compare models produced by the population of transformations and example desired targets. The result of the process, is a transformation, or collection of rules, in the target transformation specification language.

Reusing Examples of Transformations [?, ?, ?]

In these approaches, example source models and transformations are dissected, in hopes to find reusable elements when transforming new models. The resulting parts of source models, and the transformation rules applicable to those parts, are used to define the dimensions of a space representing the solutions to the problem of model transformations. This allows the use of local (PSO) and global (SA) search algorithms, which encode their solutions spatially.

6.5 Discussions

The analysis we made in the previous section can be used as a guide to identify approaches that have not been yet explored. In following we show a few examples of possible outcomes, exemplifying new pattern combinations (5.1), new instances for existing patterns (5.2), or new words for our language (5.3).

6.5.1 Reusing Learned Transformations

While we have seen approaches for learning transformations (e.g. according to the pattern in Figure 6.9), we have not observed tools that focus on reusing learned transformations for transforming different source and/or target metamodels. E.g., in Figure 6.10 we can see the pattern for a possible approach for reuse. A traditional transformation engine is used in a prepossessing stage (prepMT), translating to the source metamodel of the learned transformation.

$$\begin{tikzcd}
 & \{M_s\} \arrow[r, "\text{mapping}"{name=A}] \arrow[d, "c2"] & \{M_t\} \arrow[d, "c2"] \\
MM_a \arrow[r, "\text{prepMT}"{name=B}] & MM_s \arrow[r, "\text{MT}"{name=C}] & MM_t \\
M_a \arrow[u, "c2"] \arrow[r, "\text{prepMTa}"{name=D}] & M_s \arrow[u, "c2"] \arrow[r, "\text{MTa}"{name=E}] & M_t \arrow[u, "c2"]
\arrow[from=C, to=A, "\text{learning-of}"]
\arrow[from=D, to=B, "\text{execution-of}"]
\arrow[from=E, to=C, "\text{execution-of}"]
\end{tikzcd}$$

Figure 6.10: Reusing transformation learned from examples

6.5.2 Exogenous Transformations by Searching for Sets of Rules

The example of using evolutionary techniques to search for a selection of rules to apply [?] suggests the ability to perform transformations between different meta-models and weaving meta-models. Other examples in the section sought to repeatedly apply rules as part of a sequence, meaning the target models must be very similar to their sources, in order for the same transformations to be repeatedly applied. When presenting Figure 6.8 this behaviour was alluded to; the figure describes a general case with independent source and target meta-models, however all examples collected limit themselves to endogenous transformations. Selecting which rules to apply from a large set, or the combinations of rules from several specifications, would allow for a wide range of targets for a single specification.

6.5.3 Learning Constraint Problem Specifications for Model Transformations

While several transformation approaches require users to write constraints, writing a consistent and efficient constraint problem specification is non-trivial. Sometimes it is easier for domain experts to describe constraints using examples which do and don't satisfy them. In the field of constraints, there are efforts to derive conjectures [?] (a parallel to ALEPH's hypotheses) about data in the form of constraint problems.

While learning deterministic transformations was observed, learning problem specifications observed augmenting transformations has not been. In the case of learning transformations as a logic problem [?], the transformation was indeed able to be produced as a problem specification, but the use case didn't present the complexity of AI-augmented transformations.

$$\begin{tikzcd}
MM & \text{Problem Specification} \arrow[d, "\text{learning-of}"] & MM' \\
\{M\} \arrow[u, "\text{c2}"] \arrow[r, dash] & \{\text{solutions}\} \arrow[r] & \{M'\} \arrow[u, "\text{c2}"]
\end{tikzcd}$$

Figure 6.11: Problem Learning Pattern

Hence, in our pattern language such component would require a new word, for Problem Learning (Figure 6.11), completing a set of 4 words describing all combinations of execution/learning of transformations/problem-specifications.

6.6 Conclusions and Future Work

In this paper we propose a classification of applications of AI in model transformations. To allow this we used a language of common patterns, based on existing mega-modeling languages, to represent AI operations in model transformations processes. The language proposed allowed us to clearly form distinct categories for the use of AI in model transformations, and we showed that a few simple patterns are sufficient to represent AI augmented model transformations found in the literature, revealing three main trends: extending the specification language to include decision problems (subsection 6.4.1), learning increasingly nuanced conditions for rule application (subsection 6.4.2), learning transformations specifications and rules (subsection 6.4.3).

These patterns cover a wide set of applications of AI such as inferring model transformations, completing part of target models that cannot be easily computed by transformations or even replacing the transformation by a logic program. These different tasks can be organised in a few categories. We also identified possible patterns that have not been explored yet, such as problem learning for model transformation, and searching for exogenous rule application conditions.

Future efforts on our behalf involve leveraging the pattern language to systematically explore the space of possible future AI usages, and to investigate conditions for tool replaceability and refactoring of transformation toolchains.

Chapter 7

Towards Enforcing Structural OCL Constraints using Constraint Programming

7.1 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 [?], EMF2CSP [?] and Alloy [?]. 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 [?] 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 [?] 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

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

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 **??** we present a problem on a UML model as our running case. In section 2.2 we describe how we annotate CP variables in OCL. In **??** we show how we model the annotated UML and OCL problem as a CSP. In **??** we determine the size and performance of the resulting CP model. In **??** we discuss limitations and future work. In **??** we look at related work. Finally, we conclude in **??**.

7.2 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.

7.2.1 Reconfigurable Manufacturing Systems

An RMS [?] is an industrial solution to the problem of varying product demand. In the most common version of the problem (from [?]), 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.

A Class Diagram for RMS

?? uses a class diagram to describe the concepts of an RMS, and how they relate (inspired by [?]). 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.

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

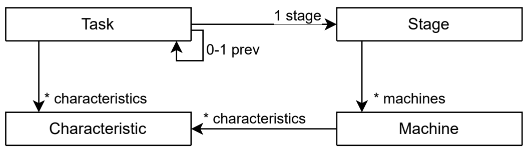

Figure 7.1: Class Diagram for RMS Task constraints
[class diagram of RMS]

```
1 context Task inv SameCharacteristicConstraint:
2     self.stage.machines.forall(m | m.characteristics
3         ->includesAll(self.characteristics))
4
5 context Task inv PrecedenceConstraint:
6     self.stage.stageNum >= self.prev.stage.stageNum
```

Listing 3: RMS Task constraints in OCL.

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 [?], tasks and machines can be linked to any number of characteristics.

OCL Constraints for RMS

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

?? 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 **??** 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 ?? 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.

7.2.2 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

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

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.

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.

7.3 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

```
-- Scenario S1
(*@\label{lst:ocl:var:char:s}@*) context Task inv
    SameCharacteristicConstraint:
     self.var('stage').machines
          ->forall(m | m.characteristics
               ->includesAll(self.characteristics))
-- Scenario S2
(*@\label{lst:ocl:var:char:m}@*) context Task inv
    SameCharacteristicConstraint:
     self.stage.var('machines')
          ->forall(m | m.characteristics
               ->includesAll(self.characteristics))
-- Scenario S3
(*@\label{lst:ocl:var:char:sm}@*) context Task inv
    SameCharacteristicConstraint:
     self.var('stage').var('machines')
          ->forall(m | m.characteristics
               ->includesAll(self.characteristics))
```

Listing 4: Denoting variables in `SameCharacteristicConstraint` from **??** using `.var()` in accordance with the three scenarios.

Section **??**, 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 **??**. 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 1 we apply the operator to `SameCharacteristicConstraint` in **??** 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.

7.3.1 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 2.1 shows the AST of `SameCharacteristicConstraint` from 1 Scenario S3. We show `var` annotations as dotted rectangles.

Figure 2.1 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 2.1, the scope

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

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 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 2.1 (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 1 has the same AST as in Figure 2.1, 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.

7.3.2 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 1, all have their annotations low in the tree Figure 2.1. 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 2 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
```

```
-- Scenario S1
(*@\label{lst:ocl:var:derive:s}@*) context Task inv
    SameCharacteristicConstraint:
     inv: Stage.AllInstances()
         ->select(s|
             s.machines.forall(c | c.characteristics
                 ->includesAll(self.characteristics))
         ->includesAll(self.var(stage)))
-- Scenario S2
(*@\label{lst:ocl:var:derive:m}@*) context Task inv
    SameCharacteristicConstraint:
     inv: Machine.AllInstances()
         ->select(m| m.characteristics
             ->includesAll(self.characteristics)
         ->includesAll(self.stage.var(machines)))
-- Scenario S3
(*@\label{lst:ocl:var:derive:sm}@*) context Task inv
    SameCharacteristicConstraint:
     inv: Machine.AllInstances()
         ->select(m| m.characteristics
             ->includesAll(self.characteristics)
         ->includesAll(self.var(stage).var(machines)))
```

Listing 5: Annotated `SameCharacteristicConstraint` from Listing 1 refactored around the annotations.

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 2.2 we can see the AST resulting from the parsing of the expression of 2 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.

7.4 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 (**??**), and then encode the OCL using additional CSPs on the variables of the model instances (**??**).

7.4.1 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 **??**. 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 (**??**) 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.

7.4.2 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).ch` machines and characteristics are reached through variable navigations, the first being from `Stage` to `Machines`, the second from `Machines` to `Characteristics` .

`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.

`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 Equation 4.1 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.

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 7.1: NavCSP: `NavigationOrAttributeCallExp` semantics modeled in CSP in the context of integer variables modeling pointers

The incoming pointers of the `AdjList` ($src.pointer_i$ in Equation 4.1), 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 **??** we will count the number of query atoms to evaluate the query model.

7.4.3 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`.

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 **??** described in **??**. In this case the accessed property is a reference, allowing us to navigate further. Navigating from a variable reference modeled with pointers means using Equation 4.1 described in **??**. We can see it applied in **??** 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.

7.5 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 Equation 4.1.

7.5.1 NavCSP

Navigating a model adds a great deal of complexity. The pointer navigation Equation 4.1 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.

OCL Query Dimensions

To evaluate the navigation provided by Equation 4.1, we will look at the size of the CSP modeling the following OCL expression:

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

Such that `self.ref` is reflexive variable reference, modeled with N pointer variables, identifying objects of the same type. 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`.

OCL Query Size

In Figure 4.1 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 4.1 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.

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.

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

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

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 4.1, the lightest area being quickly solvable, the darker area being quickly verifiable and the black area being too big to model.

7.5.2 RMS use-case

In our running example, our instance will have 4 stages (S) and 43 task (T), directly taken from [?]. 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 7.1: Size and resolution times of the use-case CSPs

In **??** 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 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

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

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.

7.6 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 [?] as it applies SAT techniques [?] for model finding.

Automation of the Refactoring. As hinted as in subsection 2.2.2, 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 `NavigationOrAttrib` 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 compiler, and integrate it into a model transformation framework such as ATLc [?], 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 [?] with incremental model transformations [?]. 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.

7.7 Related Work

The most similar work to our overall objective is by Le Calvar et al. [?], 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 [?] 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 [?] 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 [?, ?]. 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.

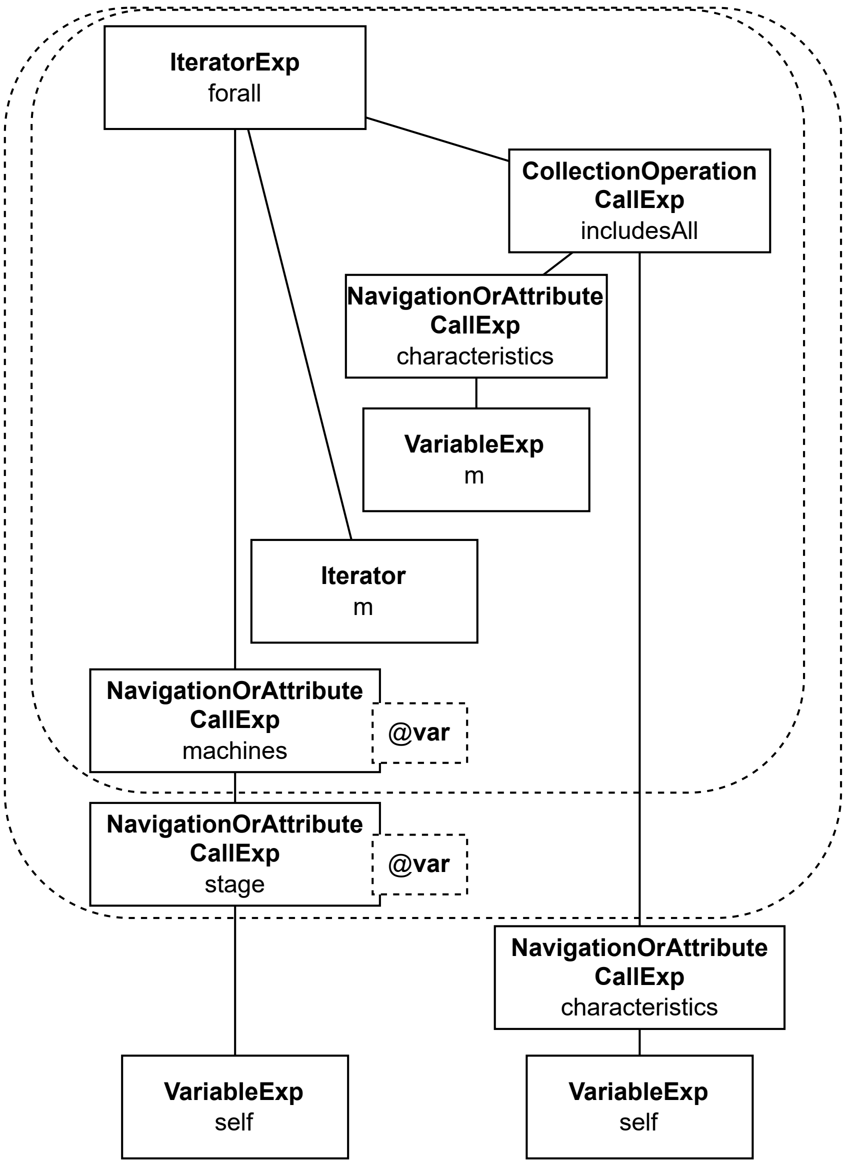

Figure 7.2: AST of `SameCharacteristicConstraint` from 1 Scenario S1, S2 & S3. [AST]

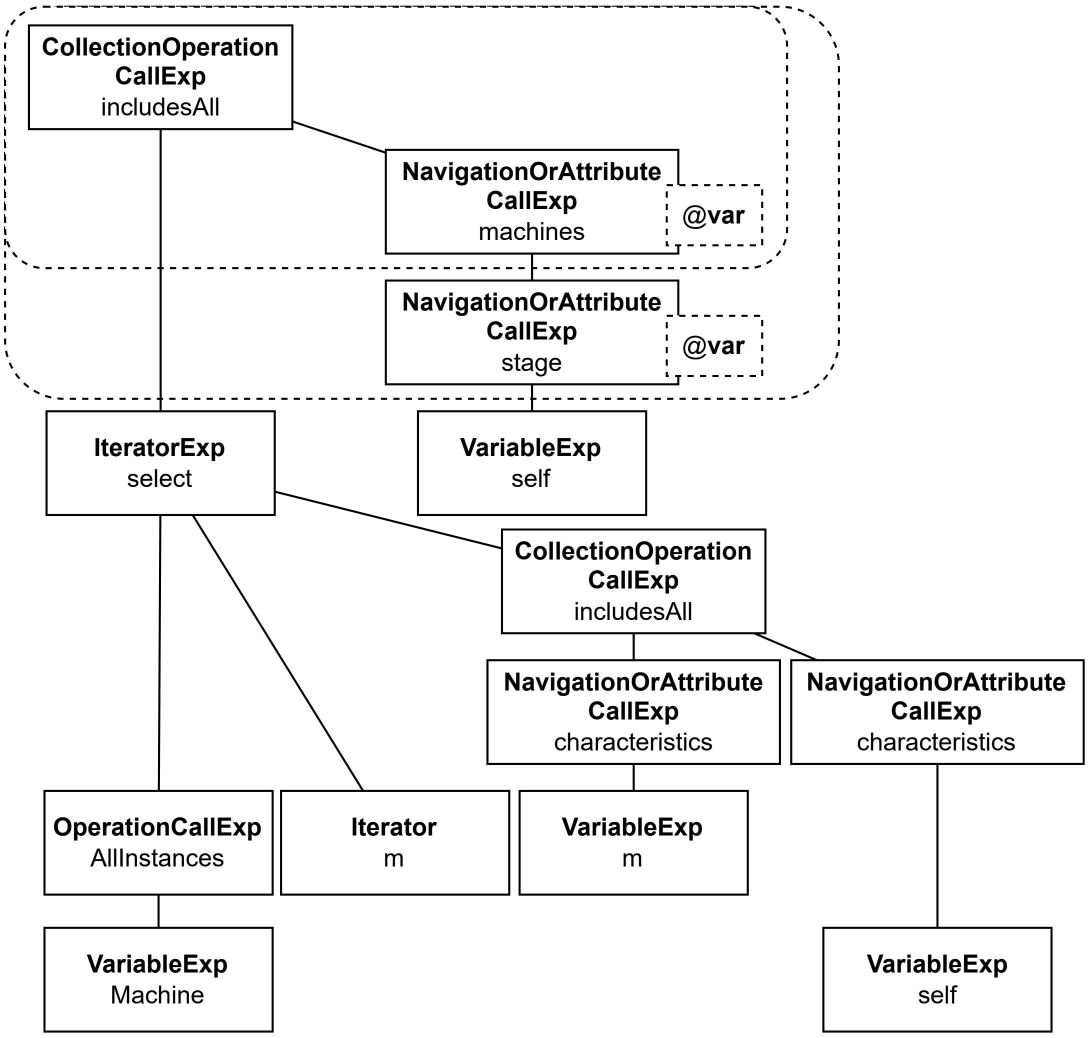

Figure 7.3: AST of SameCharactecticConstraint from 2 Scenario S2 & S3 [AST]

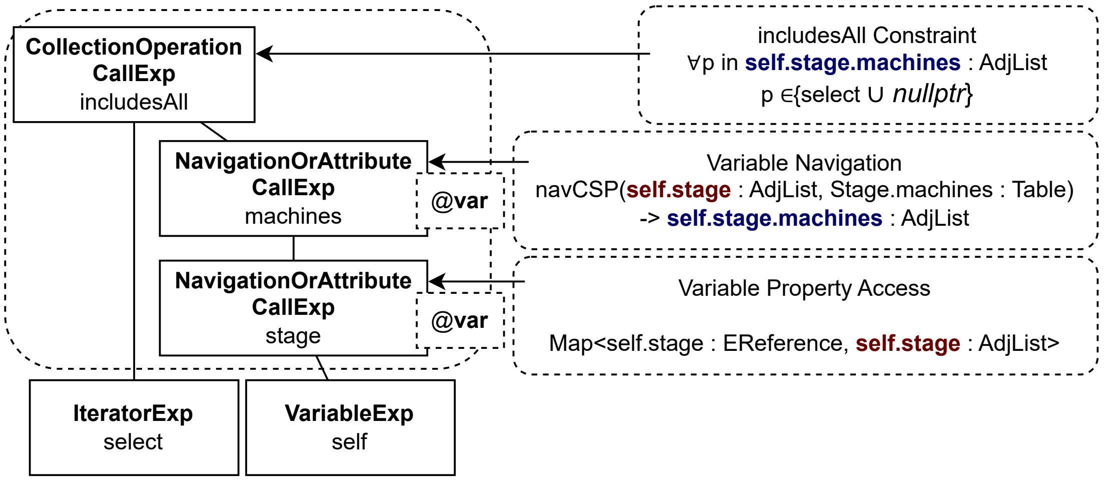
CollectionOperation CallExp includesAll
NavigationOrAttribute CallExp machines
@var
NavigationOrAttribute CallExp stage
@var
IteratorExp select
VariableExp self
includesAll Constraint
∀p in self.stage.machines : AdjList
p ∈{select ∪ nullptr}
Variable Navigation
navCSP(self.stage : AdjList, Stage.machines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

Figure 7.4: Compilation to OCL CSP of the AST from Figure 2.2
[AST]

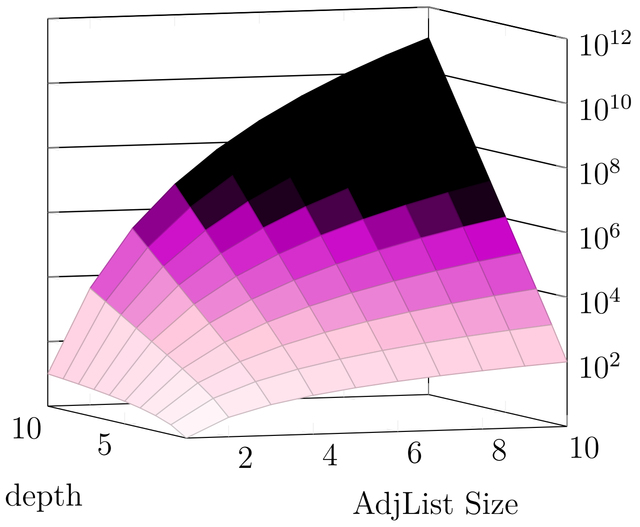

Figure 7.5: Number of query atoms in relation to `AdjList` size and navigation depth
[AST]

Chapter 8

Modeling OCL Collection Types and Type Casting using Constraint Programming

8.1 Introduction

In the context of Model-Driven Engineering (MDE), models represent structured data, and the model of the data structure is known as a metamodel. The Unified Modeling Language (UML) [1] provides visual languages, such as class and object diagrams, to define both models and meta-models. The Object Constraint Language (OCL) [2] complements UML by enabling the specification of constraints over models, based on the underlying metamodel concepts. The Eclipse Modeling Framework (EMF) [3] supports UML and OCL, enabling validation of models against their meta-models and associated constraints. It also includes model transformation tools such as ATL [?, ?], an OCL-based language that expresses mappings between meta-models. ATLc [?] extends ATL by introducing model space exploration capabilities to facilitate transformation specification. It leverages constraint solvers to generate and visualize model instances, which users can then adjust or repair using solver feedback. The primary use of ATLc is to create a Graphical User Interface for a model, allowing the user to easily edit the model. This generally breaks some of the user defined OCL constraints, and our work hopes to provide a way to repair the models around the user's choices. The core problem is: given a metamodel, a partial model and model constraints as input, the objective is to find model instances that satisfy the metamodel and model constraints. ATLc does so by interpreting part of their OCL expressions upon an instance as a constraint satisfaction problem (CSP), which can

[1]`https://www.omg.org/spec/UML/2.4`
[2]`https://www.omg.org/spec/OCL/2.4`
[3]`https://projects.eclipse.org/projects/modeling.emf.emf`

be solved by engines like Cassowary (for linear programming) or Choco (for constraint programming). However, ATLc is currently limited to single-valued model attributes, using integers or reals. Our work seeks to generalize this approach to support collection-valued properties: attributes and relations.

Among existing tools, Alloy [?] stands out as a tool offering a dedicated language for defining meta-models and constraints. Alloy is often used for specification testing–such as verifying security protocols or code–by searching for models that satisfy given constraints. It can also be used for checking specifications by searching for valid instances or counterexamples. Alloy has also been applied to model transformation and model repair [?], with some approaches translating UML/OCL into Alloy specifications [?,?]. The core difference with our approach lies in the underlying solving technique: Alloy is based on SAT solving, while we use Constraint Programming (CP). Choosing between SAT and CP for model search tasks is not straightforward, and through our experimentation, we aim to shed some light on that choice in the context of model search. Related work leveraging CP, global constraints and similar models also exists [?], however UML/OCL coverage doesn't include the general case of collection properties discussed in this paper, and required for the experimentation.

Section **??** presents the context of our work. Section **??** describes the CP model for representing UML instances and evaluating queries. Section 3.2 details the CP models used to enforce collection types and break symmetries within instances. In Section 5.1, we present CP models for casting between different collection types. Section **??** reports some experimental results, and Section 6.6 provides a discussion and concluding remarks.

8.2 Context: UML & CP

Our work is based on translating UML/OCL object models into Constraint Programming (CP) models, leveraging domain variables and global constraints.

A. Constraint Programming [?] is a powerful paradigm that offers a generic and modular approach to modeling and solving combinatorial problems. A CP model consists of a set of variables $X = \{x_1, \dots, x_n\}$, a set of domains $\mathcal{D}$ mapping each variable $x_i \in X$ to a finite set of possible values $dom(x_i)$, and a set of constraints $\mathcal{C}$ on X, where each constraint c defines a set of values that a subset of variables $X(c)$ can take. Domains can be either bounded, defined as an interval $\{lb..ub\}$, or enumerated, explicitly listing all possible values (e.g., $1, 10, 100, 1000$). This distinction impacts the choice of constraints:

for instance, the global cardinality constraint is more effective with enumerated domains. An assignment on a set $Y \subseteq X$ of variables is a mapping from variables in Y to values in their domains. A solution is an assignment on X satisfying all constraints.
CP solvers use backtracking search to explore the search space of partial assignments. The main concept used to speed up the search is constraint propagation by *filtering algorithms*. At each assignment, constraint filtering algorithms prune the search space by enforcing local consistency properties like *domain consistency* (a.k.a., *Generalized Arc Consistency* (GAC)). A constraint c on $X(c)$ is domain consistent, if and only if, for every $x_i \in X(c)$ and every $v \in dom(x_i)$, there is an assignment satisfying c such that $(x_i = v)$.
Global constraints provide shorthand to often-used combinatorial substructures. More precisely, a global constraint is a constraint that captures a relationship between several variables [**?**, **?**], for which an efficient filtering algorithm is proposed to prune the search tree. In other words, the "global" qualification of the constraint is due to the efficiency of its filtering algorithm, and its capacity to filter any value that is not globally consistent relative to the constraint in question. Global constraints are thus a key component to solving complex problems efficiently with CP. Some notable examples of global constraints used in this paper are:

- Element is useful when "selecting a variable from a list" is part of the problem. Let $X = [x_1, \ldots, x_n]$ be an array of integer variables, $z \in \{1, \ldots, n\}$ be an integer variable representing the index, and y be an integer variable representing the selected value. $element(y, X, z)$ [**?**,**?**] holds iff $y = x_z$ and $1 \leq z \leq n$, this means that variable y is constrained to take the value of the z-th element of array X.

- Regular expression constraints are very expressive when describing sequences of variables, and offers powerful filtering. Let $X = [x_1, \ldots, x_n]$ be an array of integer variables and A be a finite automaton. $regular(X, A)$ [**?**] enforces that the sequence of values in X must form a valid word in the language recognized by the automaton A.

- The $stable\_keysort(X, Y, z)$ [**?**,**?**] defined over two matrices of integer variables holds iff (1) there exists a permutation π s.t. each row y_k of Y is equal to the row $x_{\pi(k)}$ of X ($k \in \{1, \ldots, i\}$); (2) the sequence of rows in Y, truncated to the first z columns, is lexicographically non-decreasing; (3) if two rows in X have equal key values for the first z columns, then their relative order in Y must match their original order in X. Table **??** illustrates with an instance that satisfied this constraint.

- Cumulative is generally used for scheduling tasks defined by their start time, duration and resource usage: $< s_i, d_i, r_i >$. It requires that at any instant t of the schedule, the summation of the amount of resource r of the tasks that overlap t, does not exceed the upper limit C. The values of t range from: a the earliest possible start time s_i, to b the latest possible end time $s_j + d_j$. Let $S = [s_1, \dots, s_n]$ be the start times of n tasks, $D = [d_1, \dots, d_n]$ their durations, $R = [r_1, \dots, r_n]$ their resource demands, and C the total capacity of the resource (a constant). $\mathit{cumulative}(S, D, R, C)$ [?, ?] holds iff $\forall t \in [a, b], \sum_{i|s_i \leq t < s_i + d_i} r_i \leq R$ [?]. where $a = \mathit{min}(s_0, .., s_n)$ and $b = \mathit{max}(s_0 + d_0, .., s_n + d_n)$,

B. Class diagrams identify concepts and their properties. In a family tree for instance, the core concept is Person, with attributes such as age and references such as parent (or its inverse, child) to express relationships between people.

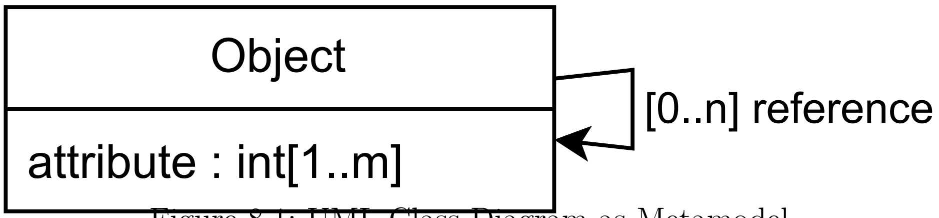

Figure 8.1: UML Class Diagram as Metamodel

Figure 1.1 present a generic metamodel. It describes a class named `Object`, which has two properties: `attribute`: a collection of integers, with at least one and at most m elements, `reference`: a collection of up to n references to other `Object` instances. These illustrate the two main types of properties in object-oriented modeling: Attributes, which store intrinsic data values (e.g., numbers or strings), References, which define relationships between objects in the model.

UML allows properties to be collections, and distinguishes four standard collection types, based on two dimensions: order and uniqueness.

- `Sequence`: ordered, allows duplicates – e,g., [2,3,1,1],
- `Bag`: unordered, allows duplicates – e.g., [1,1,2,3],
- `Set`: unordered, unique elements only – e.g., [1,2,3],
- `OrderedSet`: ordered, unique elements – e.g., [2,3,1].

An important note is that ordered doesn't pertain to the values. In [2,3,1,1]: 2 is the first value, and 1 is the last value. The intended collection type can be indicated in the class diagram using annotations such as `ordered`, `unique`, or `seq` (for sequences).

C. Object Diagrams describe instances of the classes defined in a class diagram. For example, Figure 1.2 shows an instance conforming to the class diagram in Figure 1.1. It includes three objects, each identified by a unique ID (e.g., `o1`, `o2`, `o3`). For instance, object `o1` has as attribute a collection of 3 integers and is connected to other objects (e.g., `o2` and `o3`).

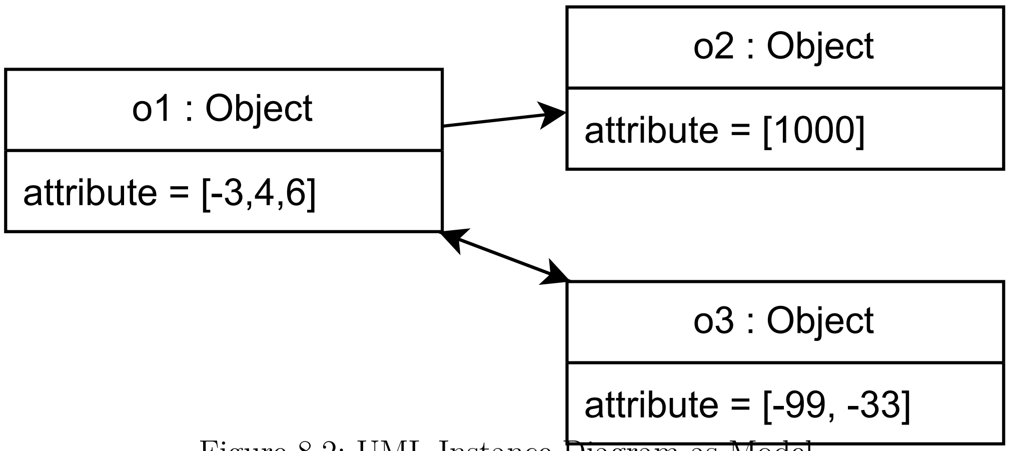

Figure 8.2: UML Instance Diagram as Model

D. The Object Constraint Language (OCL) is a declarative language used to specify additional rules and constraints on UML models that cannot be expressed using diagrams alone. It enables the formalization of conditions that instances of the model must satisfy, serving as a powerful complement to class and object diagrams. For example, in the context of a family tree, a constraint such as "a child must be younger than their parents" cannot be represented directly in a class diagram. However, it can be expressed in OCL as follows:

```
context Person inv:
  self.parents.age.forall(a| a > self.age)
```

This constraint states that for every Person instance, all of their parents must be older. The `context` keyword specifies the class to which the constraint applies, and `inv` stands for invariant, i.e., a condition that must always hold true. This invariant states that for every Person, the age of each parent must be greater than the person's age. OCL Supports navigation expressions (e.g., `self.parents.age`) and collection operations (e.g., `forall`, `exists`, `size`) that apply to attributes and references. The expression `self` refers to the current object, `self.attribute` returns its attribute values, and `self.reference` retrieve related objects. Chained queries like `self.reference.attribute` retrieve the attributes of referenced objects.

OCL also supports a rich set of operations on primitive types and collections. Examples include: Boolean expressions (`forall`, `exists`, `not`, `and`, `or`), Arithmetic and comparison (`+`, `-`, `>`, `<`), and Collection operations (`sum`, `size`, `includes`, `asSet`, `asSequence`, etc). Each collection type comes with its own operations and can be explicitly cast using operations like `asSet()`.

Given an instance such as the one shown in Figure 1.2, OCL is typically used to verify whether it satisfies the specified constraints. In this work, however, we aim to use OCL as a means to guide model search, thereby enabling the completion or correction of partial or inconsistent data. To this end, we propose an approach that reformulates OCL specifications as constraint satisfaction problems (CSPs). This paper focuses on how OCL's collection typing, defined in the Class Diagram, and type casting operations can be modeled using global constraints over bounded domains.

E. Alloy and the Alloy Analyzer [?] are at the forefront of the related works; furthermore, they are commonly found as a tool employed by related work, such as the Viatra Generator [?, ?]. Alloy is a textual specification language not too dissimilar to UML class diagrams and OCL, most notably: the native collection type in Alloy is sets. Alloy also has utilities for the sequence type and offers a similar set of operations. The Alloy Analyzer allows the user to test their specification by finding conforming models, which can help prove or disprove the specification. UML models and the problems upon them have also been translated to Alloy, to leverage the analyzer. [?, ?] The underlying tool is Kodkod [?], a relational first-order logic API for SAT solvers, which is used to compile Alloy models to CNF for solving by a third-party solver. Their solution for integers is encoding them as bit-vectors (5=101), and modeling arithmetic accordingly. This could in some cases become a limitation when modeling with integers and sequences, which guided our choice to explore models using domain variables and sequences as the native collection type.

8.3 CP Models for UML Instances and OCL Queries

To solve problems on UML instances using constraint programming (CP), we must first define a CP model that represents the instance. Since constraints are expressed in OCL, this model must also encode how OCL expressions query the instance.

A. Encoding Properties. The variables represent the properties–attributes and references–of the objects in the instance. Each class property is encoded as a matrix of integer variables, denoted *Class.property*. Each row in this matrix corresponds to one object of the class; for example, the i-th row is noted as $Class_i.property$ where $i \in [1, o]$ and $o = |Class|$ is the number of objects of that class. The number of columns p in this table is derived from the property's cardinality, which is given by n and m from Figure 1.1.

$$Class.property = \{x_{11}, ..., x_{op}\}$$

$$\forall x \in Class.property,\ domain(x) = \{d\} \cup \{lb..ub\}$$

Each property variable x_{ij} in the matrix has a domain defined by a lower bound lb, an upper bound ub, and a special dummy value d, where we set $d = lb - 1$. The property type, e.g., reference or attribute, determines the specific domain bounds. Attributes with an integer type may require large ranges, making domain enumeration impractical. This limits our ability to use certain global constraints like $global_cardinality(c)$onstraint, which counts the occurrences of domain values and therefore require finite, reasonably small domains. By default we chose a 16-bit range for these values: $lb = -32768$ and $ub = 32767$, meaning $d = -32769$, but these bounds can be refined by annotating the model accordingly [?]. For reference properties, the domain is defined as $\{1, \ldots, o\} \cup \{nullptr\}$, where o is the number of instances of the target class. These variables, named `ptr`, acts as pointers: values in $[1, o]$ identify object rows, and 0 (i.e., dummy value nullptr) denotes the absence of a reference. To support nullptr, an extra row is added to each table to represent a dummy object.

Table 3.1 shows the encoding of the instance from Figure 1.2, assuming $n = 2$ and $m = 3$ from the metamodel in Figure 1.1. The left side represents attributes, while the right side represents references. Each object (plus one dummy object) gets a row. The attribute variables a_{ij} are assigned the domain $\{lb, \ldots, ub\} \cup \{d\}$, and reference variables ptr_{ij} are assigned the domain $\{1, \ldots, o\} \cup \{nullptr\}$ with $o = 3$.

Model construction proceeds in two steps. First, we create matrices of variables with

Object.attribute			
$Object_i$	**attribute**		
0	d	d	d
1	-3	4	6
2	1000	a_{22}	a_{23}
3	-99	-33	a_{33}

Object.reference		
$Object_i$	**reference**	
0	nullptr	nullptr
1	3	2
2	ptr_{21}	ptr_{22}
3	1	ptr_{32}

Table 8.1: Encoding of the instance from Figure 1.2 as tables of integer variables

their full domains. Second, we instantiate some of these variables using data from the actual instance. In our current setting, we assign the exact values from the instance. Variables that remain uninstantiated may either be assigned dummy values or left free to explore during search, depending on the objective of the analysis. To choose between these behaviors and to reduce the size of the CSP, in previous work we've proposed an annotation system for OCL [?], which allows the user to identify variables. These annotations split the OCL expressions into parts which can be dispatched between our CP interpretation, and that of a standard interpreter. This reduces the scope and size of the CSP, notably in terms of modeled properties.

B. CP model for OCL queries on the instance. Querying an instance involves navigating the object graph through references and retrieving attribute values. In OCL, navigation refers to the operation that, given source collection of objects and a reference property, returns a collection of target objects through that reference. We conflate this with attribute operations–as defined in the OCL specification–which return a collection of attribute values from a source collection of objects. In our encoding, both navigation and attribute access results are uniformly represented as integer variables.

Consider an OCL expression of the form `src.property`, where `src` is itself an expression like `self.reference` or `self.reference.reference`.

Let $Ptr = \{ptr_1, ..., ptr_z\}$ be the variables encoding the evaluation of `src`, with $dom(ptr_i) = \{1..o\} \cup \{nullptr\}$. Let T be the flattened array representing the `Class.property` matrix for `property`, where `property` refers to either an attribute or reference of the referenced class. Let p be the number of columns in the matrix. Let $Y = \{y_1, \ldots, y_{z \cdot p}\}$ be the variables representing the result of `src.property`. To link Y with T and Ptr, we define the navigation constraint:

$$nav(Ptr, T, Y) \iff \forall i \in [1, z], \forall j \in [1, p] : y_{(i-1)p+j} = T_{ptr_i \times p+j}$$

This constraint links the source pointers to the appropriate rows in the property table.

Object.reference.attribute						
$Object_i$	**reference.attribute**					
1	-99	-33	a'_{13}	1000	a'_{15}	a'_{16}
2	a'_{21}	a'_{22}	a'_{23}	a'_{24}	a'_{25}	a'_{26}
3	-3	4	5	a'_{34}	a'_{35}	a'_{36}

Table 8.2: Encodings of `self.reference.attribute` for all objects of Figure 1.2 as a table of integer variables

This is reformulated in CP as a conjunction of *element* constraints, using intermediate variables for encoding the pointer arithmetic ($ptr_i \times p + j$):

$$nav(Ptr, T, Y) : \begin{cases} \forall i \in [1, z], \forall j \in [1, p] : \\ \quad ptr'_{ij} = ptr_i \times p + j \\ \quad element(y_k, T, ptr'_{ij}),\ k = (i-1)p + j \end{cases} \tag{8.1}$$

The intermediate variables introduced are functionally dependent on the Ptr variables and do not require enumeration during search. Given Ptr and T, the value of Y can be determined. However, given an instantiation of Y, this model cannot fully determine Ptr and T, but it can filter to some extent. Thus, OCL query variables depend on the instance variables, and a query result may correspond to multiple instances.

Table 4.1 shows the results of the query `self.reference.attribute` using the navigation CP model (4.1) on the instance from Table 3.1. Result variables a'_{ij} share the same domains as a_{ij} but follow the reference and attribute order, introducing gaps due to ordering, e.g., a'_{13} might be the third variable, but yield a different third value (e.g., 1000) if $a'_{13} = d$. Similar effects occur in other OCL reformulations like union and append. Despite these gaps, value order and duplicates are preserved. These outputs are interpreted using the same models used for casting to collection types, such as `asSequence()`, discussed in Section 5.1.

8.4 CP Models for UML Collection Types

As described in Section **??**, properties in class diagrams (e.g., Figure 1.1) can be annotated with collection types: `Sequence`, `Bag`, `Set`, or `OrderedSet`. These types can be enforced through constraint models to ensure consistency and reduce symmetries in the data..

For `Sequence`, permutations of the same multiset (e.g., $\{1, 1, 2\}$) yield distinct sequences. However, in our encoding, sequences such as $\{1, 2, d, 1\}$ and $\{1, 2, 1, d\}$ are treated as equivalent, since they encode the same effective ordering of values (e.g., the position of the dummy value d is ignored). To correctly model sequences, we impose an ordering where all dummy values are grouped at the end.

Let $X = \{x_1, .., x_p\}$ be the variable array for a property in the matrix $= Class.property$. The `Sequence` constraint is defined as: $Sequence(X) \iff \forall i \in [1, p[, (x_i = d) \Rightarrow (x_{i+1} = d)$. This ensures dummy values appear only at the end. We reformulate it using the `regular` global constraint applied to a Boolean mask $S = \{s_1, \dots, s_p\}$:

$$Sequence(X) : \begin{cases} regular(S, DFA) \\ s_i = [\![x_i \neq d]\!] \end{cases} \tag{8.2}$$

The automaton *DFA* (Figure 3.1) accepts patterns of the form 1*0*, ensuring non-dummy values precede dummy ones. Here, S acts as a mask distinguishing actual values (1) from dummies (0) while avoiding symmetry.

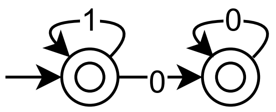

Figure 8.3: DFA packing dummy values for instance variables

For the `Bag` and `Set` types, all permutations of values are considered equivalent. To remove ordering symmetries, we sort the values in decreasing order, effectively pushing dummy values to the end:

$$Bag(X) : \Big\{ \forall i \in [1, p[: x_i \geq x_{i+1} \tag{8.3}$$

To model `Set`, we additionally enforce uniqueness among non-dummy values. Indeed, the sequence $\{d, d, d\}$ would be interpreted as an empty set. To this end, we define a relaxed variant of the `alldifferent` global constraint:

$$\begin{aligned} & alldifferent\_except\_d(X) \\ \iff & \forall\, i, j\ (i < j) \in [1, |X|], (x_i \neq x_j) \lor (x_i = x_j = d) \end{aligned}$$

$$Set(X) : \begin{cases} alldifferent\_except\_d(X) \\ Bag(X) \end{cases} \tag{8.4}$$

For `OrderedSet`, both value order and uniqueness matter. We combine the constraints used for `Sequence` and `Set`: dummy values must be packed at the end, and non-dummy values must be pairwise distinct. Formally:

$$OrderedSet(X) : \begin{cases} alldifferent\_except\_d(X) \\ Sequence(X) \end{cases} \tag{8.5}$$

This ensures a well-formed sequence without repetitions, where dummy values are ignored in uniqueness checks and appear only at the end of the array. These CP encodings ensure that model properties respect their specified UML and OCL collection types, enabling correct interpretation and reducing symmetry in instance generation.

8.5 CP Models for OCL Collection Type Casting Operations

To illustrate OCL type casting, consider the following invariant, taken from the Zoo Model used in the experimentation:

```
context Cage inv:
    self.animals.species.asSet().size < 2
```

If `self.animals.species` evaluates to the sequence {1,1,1}, applying `asSet()` yields the set {1}, indicating that the cage contains a single species of animal. In the following, we define CP models to capture such collection type conversions.

Consider an expression of the form `src.asOP()`, where `src` is a collection-valued expression such as `self.attribute`, and `asOP()` denotes a type-casting operation applied to the source collection (e.g., `asSequence()`, `asSet()`, etc.). Let $X = \{x_1, ..., x_z\}$ be the array of variables modeling the values of `src`, and let $Y = \{y_1, ..., y_z\}$ represent the resulting collection after applying `asOP()`.

A. asBag(): Consider the expression `src.asBag()`, where the result is evaluated as a multiset Y that preserves all values from the source collection X, including repeated elements. Because OCL bags are insensitive to permutations, multiple orderings of the same values are semantically equivalent. To avoid such symmetries in the model, we impose a canonical form by sorting Y in descending order. This also ensures that any dummy values d used to pad the collection appear at the end. For example, given $X = \{1, 2, d, 1\}$, we enforce the canonical bag representation $Y = \{2, 1, 1, d\}$. This transformation is modeled using the global constraint $sort(X, Y)$, which sorts X into Y.

$$asBag(X, Y) : \Big\{ sort(X, Y^{\text{rev}}) \tag{8.6}$$

Y^{rev} denotes the reverse of Y, used to enforce descending order.

B. asSet(): Consider the expression `src.asSet()`. The `asSet()` operation removes duplicate elements from the source collection X while disregarding order. In our encoding, this corresponds to extracting the distinct values from X and placing them into the result array Y in a canonical form. Since the number of unique elements in X is not known beforehand, Y is defined with the same arity as X, and any unused positions are filled with a dummy value d. For instance, given an instantiation $X = \{1, 2, 1, d\}$, the result of `asSet()` would be $Y = \{2, 1, d, d\}$.

$$asSet(X, Y) : \begin{cases} sort(X, S^{\text{rev}}) \\ X' = S \parallel \{d\} \\ Y' = Y \parallel \{d\} \\ p_1 = 1 \\ \forall i \in]2, z+1] : \quad p_i = p_{i-1} + [\![x'_{i-1} \neq x'_i]\!] \\ \qquad\qquad\qquad\quad element(x'_i, Y', p_i) \\ \forall i \in [1, z] : \qquad y_i \geq y_{i+1} \end{cases} \tag{8.7}$$

To enforce the `asSet()` semantics, we first sort the source array X in descending order into an auxiliary array S. We then define an array of position variables P and compute the position p_i of each variable s_i in a new array Y, ensuring that repeated values in S map to the same position. The first occurrence of a new value increments the position counter: $p_i = p_{i-1} + [\![s_{i-1} \neq s_i]\!]$. To support cases where all positions in Y are filled with unique values, we append a dummy value d to S, yielding $X' = S \parallel \{d\}$. In the case where all values in X are distinct (e.g., $X = \{2, 3, 1, 4\}$), the dummy has no room in Y. We resolve this by appending a dummy value to Y as well, forming $Y' = Y \parallel \{d\}$. This dummy will occupy the first unused position in Y, and all subsequent positions are forced to d by a descending sort constraint $y_i \geq y_{i+1}$. The final mapping from positions p_i to Y' is enforced via an $element(c)$onstraint over X' and Y'.

C. asSequence(): The `asSequence` operation retains all values from the source collection, including duplicates, and reorders them such that all non-dummy values appear first in their original relative order, followed by the dummy values. For example, if $X = \{1, d, 2, d, 1\}$, then `asSequence` yields $Y = \{1, 2, 1, d, d\}$. To enforce this transformation, we introduce the following CP model:

$$asSeq_{x2y}(X, Y) : \begin{cases} stable\_keysort(\langle B, X \rangle, \langle B', Y \rangle, 1) \\ b_i = [\![x_i = d]\!], \forall i \in [1, z] \\ b'_i = [\![y_i = d]\!], \forall i \in [1, z] \end{cases} \tag{8.8}$$

Here, B and B' are arrays of integer variables of size z, of domain $0, 1$, used as booleans indicating which variables in X and Y are equal to the dummy value d. The $stable\_keysort(T, S, k)$

Index	B	X	**Sorted Index**	B'	Y
1	0	1	1	0	1
2	1	d	3	0	2
3	0	2	5	0	1
4	1	d	2	1	d
5	0	1	4	1	d

Table 8.3: Example of `asSequence()` transformation using stable sort. Dummy values are in red.

constraint takes a matrix T and produces a sorted matrix S, ordering rows based on the first k columns, which form the sort key. In our case, we construct the matrices $\langle B, X\rangle$ and $\langle B', Y\rangle$, and sort on the first column, which separates dummy and non-dummy values while preserving the original order of the non-dummy elements.

To illustrate, let $X = \{1, d, 2, d, 1\}$, yielding $B = \{0, 1, 0, 1, 0\}$. We apply a stable sort to B, considering the pairs (b_i, x_i), and sorting by the key b_i. This ensures that all 0s (non-dummy values) appear before all 1s (dummy values), and the relative order of elements with the same key (e.g., all 0s) is preserved (see Table **??**). The sorted Boolean array becomes $B' = \{0, 0, 0, 1, 1\}$. Applying the permutation used to sort B to the array X results in $Y = \{1, 2, 1, d, d\}$.

One of the strategies during the search process involves enumerating the variables representing the top-level nodes in the OCL abstract syntax tree (AST). For example, in the expression `src.asSequence().sum()<3`, we explore possible values for `.sum()`, which helps filter the values of `src.asSequence`. To extend this filtering process down to `src`, an additional model is needed to manage the refinement. To filter from Y to X, we use a cumulative constraint, commonly applied in task scheduling. In this approach, we treat the intervals between values in Y as blocking tasks that prevent certain values from X during scheduling. By scheduling the tasks derived from X around the blocking intervals from Y, we filter down the possible values for X, effectively refining the search space according to the constraints set by Y.

$$asSeq_{y2x}(X, Y) : \begin{cases} & sort(Y, Y') \\ \text{let } T_y & \text{be the set of tasks such that:} \\ \forall i \in [1, z[& : s_i = y'_i + 1 \\ & d_i = \max(0, y'_{i+1} - y'_i - 1) \\ & h_i = z \\ \text{let } T_x & \text{be the set of tasks such that:} \\ \forall i \in [1, z] & : s_i = x_i \\ & d_i = 1 \\ & h_i = 1 \\ & cumulative(T_y \cup T_x, z) \end{cases} \tag{8.9}$$

Equation (5.4) defines how to filter values of X based on the sequence Y using a cumulative constraint:

1. First, Y is sorted into Y' to identify ordered non-dummy values.

2. From Y', we define blocking tasks T_y representing disallowed intervals. Each task (associated to value y'_i in Y'):

   - Starts at $s_i = y'_i + 1$,
   - Has a duration $d_i = \max(0, y'_{i+1} - y'_i - 1)$,
   - Has a height of $h_i = z$, fully consuming the resource and thus excluding X from that interval.

3. For each variable $x_i \in X$, a task is created in T_x starting at x_i, with duration 1 and height 1.

4. The cumulative constraint on $T_y \cup T_x$ ensures tasks from X are only scheduled in the non-blocked intervals.

In Figure 5.1, blocking tasks (highlighted in red and blue) are created from the intervals between values in Y', representing values that are prohibited for X. The white space represents the available slots for scheduling tasks from X. This model effectively restricts the possible values for X by ensuring that certain values, determined by the sorted sequence Y', are "blocked" from being selected, refining the search space. Combining both

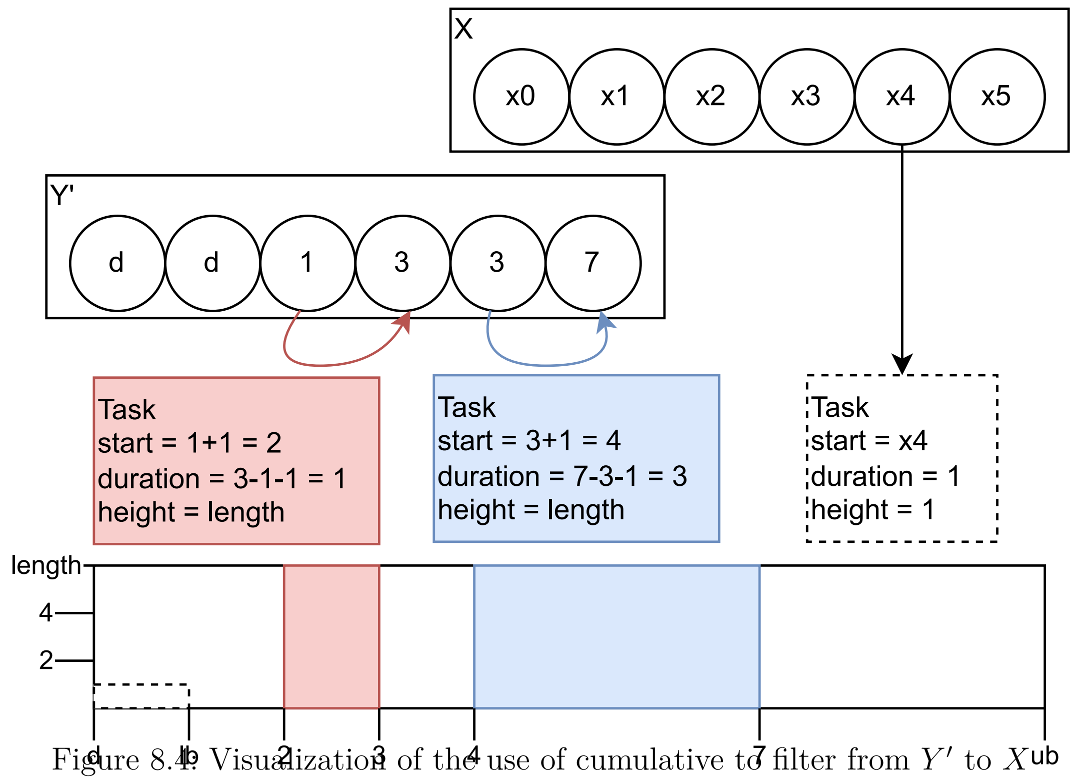

Figure 8.4: Visualization of the use of cumulative to filter from Y' to X

the X to Y and Y to X models give us the complete model for `asSequence`.

$$asSequence(X, Y) : \begin{cases} asSeq_{x2y}(X, Y) \\ asSeq_{y2x}(X, Y) \end{cases} \tag{8.10}$$

D. asOrderedSet(): Consider the expression `src.asOrderedSet()`. The `asOrderedSet()` operation removes duplicates from X while preserving the relative order of first occurrences. Unused positions in Y are filled with dummy values d. For example if $X = \{1, 2, d, 1\}$, then `asOrderedSet` returns the array $Y = \{1, 2, d, d\}$. To enforce this behavior, we use the following CP model:

$$asOrdSet(X, Y) : \begin{cases} stable\_keysort(< X, Y' >, < S, T >, 1) \\ t_1 = s_1 \\ \forall i \in]1, z] : t_i = \begin{cases} s_i & \text{if } s_i \neq s_{i-1} \\ d & \text{otherwise} \end{cases} \\ asSequence(Y', Y) \end{cases} \tag{8.11}$$

The idea is to sort X into S to group identical values. We build T by keeping the first occurrence of each value in S and replacing subsequent duplicates with the dummy value

d. We then invert the sort to obtain Y', restoring the original structure. Finally, we apply `asSequence` to push all dummy values to the end, yielding the final ordered set Y.

Given $X = \{2, 1, 2, 3\}$, sorting yields $S = \{1, 2, 2, 3\}$, filtering gives $T = \{1, 2, d, 3\}$, reversing the sort results in $Y' = \{2, 1, d, 3\}$, and packing dummies yields $Y = \{2, 1, 3, d\}$.

E. Filtering Dummy Values in OCL Collection Operations For many OCL collection operations, the filtering process from X to Y can be enhanced by introducing a dedicated constraint to handle dummy values. This filtering mechanism can be integrated into models such as 5.1, 5.2, 5.5, and 5.6. The filtering approach is inspired by the strategy used in Equation (3.1), employing a *regular* constraint over a masked array:

$$\mathit{dChannel}(X, Y) : \begin{cases} \mathit{regular}(S, \mathit{NFA}) \\ \text{where } s_i = [\![s'_i \neq d]\!], i \in [1, z] \\ \text{with } S' = X \parallel c \parallel Y^{\text{rev}} \end{cases} \tag{8.12}$$

The mask encodes non-dummy values with 1s and dummy values with 0s. The *regular* constraint is applied over the concatenated sequence $S' = X \parallel c \parallel Y^{\text{rev}}$, where c is a counter variable ranging from 0 to z. The non-deterministic finite automaton (NFA), shown in Figure 5.2, ensures that the number of 0s (i.e., dummies) in X is matched by the same number of leading 0s in Y^{rev}.

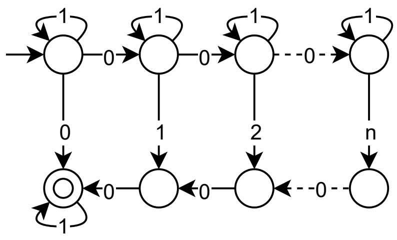

Figure 8.5: Non-Deterministic Finite Automaton accepting strings where Y starts with the same number of 0 found in X.

Given a partial instantiation such as $X = \{x_1, d, x_3, d, x_5\}$, this constraint allows filtering to deduce $Y = \{y_1, y_2, y_3, d, d\}$.

8.6 Experimental Results

In order to perform our experiments we use our EMF interpreter, which employs the Choco Solver, to build and solve a CSP representing an instance and the associated model constraints. As input the interpreter takes a metamodel (fig.1.1) in ecore format, and a model (fig. 1.2) in xmi format producing what we call the UML CSP. The UML CSP will employ the models 3.1 to 3.4 to enforce the collection property types. With as input the UML CSP and the model constraint written in ATL OCL, the interpreter builds the OCL CSP. The OCL CSP is composed of models such as 4.1 and 5.1 to 5.6.

As previously discussed, Alloy encodes models into SAT using bit-vectors for integers, while our approach translates to Constraint Programming (CP) using domain variables. In this experiment, the key parameter is the integer domain size, analogous to the bit-width in Alloy. To configure integer size in Alloy, we specify a bit-width using the solver call, e.g., `run{} for 8 int` sets a bit-width of 8. In CP, this corresponds to setting explicit lower and upper bounds for the integer variables.

Zoo Model. We conduct experiments on a zoo model, which includes three core classes: `Cage` with a sequence of `animals` and an integer attribute `capacity`. `Animal` linked to one Species and one Cage. `Species` has an integer attribute `space` indicating how much space one animal requires.

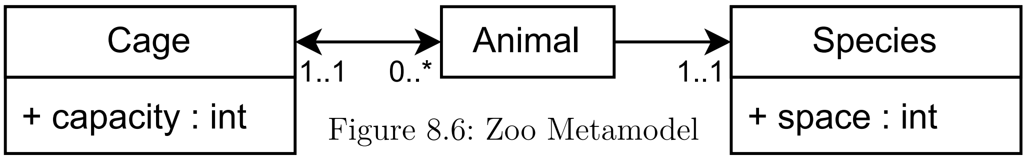

Figure 8.6: Zoo Metamodel

We want to ensure that each cage contains only one species and does not exceed its capacity. Space requirements are expressed in a unit such as square meters. For instance, 3 lions needing $1000m^2$ each, 2 gnous needing $3000m^2$ each, and 2 cages with capacities of $3000m^2$ and $6000m^2$. Encoding such values in Alloy requires sufficient bit-width: 14 bits for encoding in square meters (m^2), 12 bits for decameters squared (dam^2), and 8–10 bits for hectometers squared (hm^2). In our CP model, we define domain bounds accordingly. The model has two cages, two species, and five animals. The associations (which animal is in which cage) are unknown and must be inferred. Constraints ensure:

1. A cage contains at most one species.
2. The total required space of animals in a cage does not exceed its capacity.

To express the species constraint, we use collection type casting as discussed earlier. For the capacity constraint, we navigate over the sequence of animals to retrieve each individual's space requirement from their species, and then sum these values.

	bit-width	8	10	12	14	16
Alloy Seq	variables	84k	382k	Atm	Atm	Atm
	clauses	316k	1.5M			
	build (sec)	0.1	0.6			
	solve (sec)	< 0.1	0.3			
Alloy Set	variables	35k	157k	703k	3.1M	Mem
	clauses	128k	589k	2.7M	11.2M	
	build (sec)	< 0.1	0.2	2.6	26	
	solve (sec)	< 0.1	0.1	0.5	2.3	
OCL in CP	variables	223	223	223	223	223
	constraints	126	126	126	126	126
	build (sec)	0.6	0.6	0.6	0.6	0.6
	solve (sec)	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1

Table 8.4: Comparison of Alloy and our work for models with integer properties

```
context Cage inv:
    self.animals.species.asSet().size() < 2
and self.animals.species.space.sum() <= self.capacity
```

We compare our CP-based encoding of the OCL constraints with an equivalent Alloy model that incorporates sequences and type casting. Additionally, we also include an optimized Alloy variant that uses only sets, omitting sequence order over animals. The solver employed here is SAT4J.

Table **??** compares the Alloy (Sequence and Set) and our CP models across different integer bit-widths (8 to 16 bits). As a reminder: 8-bit allows for integers ranging from -128 to 127, and 16-bit ranges from -32768 to 32767. For Alloy, we report the number of SAT variables and clauses, while for the CP model we provide the number of variables and constraints. While these metrics aren't directly comparable across paradigms, they illustrate model growth with respect to integer encoding. We also show solver build times. Some Alloy configurations failed[4] due to memory limits ("Mem") or excessive atoms ("Atm"). Resolution times are low across all models, reflecting the relative simplicity of the problem. Alloy models explode when faced with integers: each extra bit roughly doubles the number of variables and clauses in the SAT model, you can approximate the model growth with regards to bit-width with: $2^{\text{bit-width}}$. The CP model's size however, is independent from the integer domain, and the model size remains constant. This trend can also be found

[4]The generated model is too large to fit within a 2.5 gigabyte memory limit

in resolution times, where incrementing bit-width doubles resolution time for Alloy. This experiment shows that our CP solution scales independently of the integer domain size. While Alloy models hit limitations due to memory or atom count at higher bit-widths, our approach continues to support larger domains and remains solvable. Moreover, our model can exceed 16-bit integers, with the only limitation being operations, such as addition and multiplication, that require wider domains to hold intermediate results.

8.7 Conclusions

In this paper, we extended the ATLc approach by generalizing the CP encoding of UML/OCL models to fully support collection types: `Sequence`, `Bag`, `Set`, and `OrderedSet`. We introduced modular and semantically faithful constraint models for collection type casting using global constraints such as `keysort`, `regular`, and `cumulative`. We also proposed a filtering constraint (`dChannel`) to help prune symmetries of dummy values. These encodings allow precise control over ordering, multiplicity, and filtering, leveraging constraint propagation mechanisms for effective instance generation and validation.

We evaluated our encodings against Alloy's SAT-based analysis, showing that our CP models scale independently of integer domain size, unlike Alloy where performance degrades with increasing bit-width. Our CP models remain tractable even beyond 16-bit integers, demonstrating better adaptability for data-intensive configurations. Overall, our method offers a flexible and scalable foundation for interpreting UML/OCL models through constraint programming. Future work includes extending support to more OCL operations and exploring optimization techniques for larger problem instances.

CONCLUSION

Lorem ipsum dolor sit amet, « consectetuer » adipiscing elit. Maecenas fermentum, elit non lobortis cursus, orci velit suscipit est, id mollis turpis mi eget orci. Ut aliquam sollicitudin metus. Mauris at sapien sed sapien congue iaculis. Nulla lorem urna, bibendum id, laoreet iaculis, nonummy eget, massa. Phasellus ullamcorper commodo velit. Class aptent taciti sociosqu ad litora torquent per « conubia nostra », per inceptos hymenaeos. Phasellus est. Maecenas felis augue, gravida quis, porta adipiscing, iaculis vitae, felis. Nullam ipsum. Nulla a sem ac leo fringilla mattis. Phasellus egestas augue in sem. Etiam ac enim non mauris ullamcorper scelerisque. In wisi leo, malesuada vulputate, tempor sit amet, facilisis vel, velit. Mauris massa est, sodales placerat, luctus id, hendrerit a, urna. Nullam eleifend pede eget odio. Duis non erat. Nullam pellentesque.

> Maître Corbeau, sur un arbre perché,
> Tenait en son bec un fromage.
> Maître Renard, par l'odeur alléché,
> Lui tint à peu près ce langage :
> « Hé ! bonjour, Monsieur du Corbeau.
> Que vous êtes joli ! que vous me semblez beau !
> Sans mentir, si votre ramage
> Se rapporte à votre plumage,
> Vous êtes le Phénix des hôtes de ces bois. »

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Maecenas fermentum, elit non lobortis cursus, orci velit suscipit est, id mollis turpis mi eget orci. Ut aliquam sollicitudin metus. Mauris at sapien sed sapien congue iaculis. Nulla lorem urna, bibendum id, laoreet iaculis, nonummy eget, massa[5]. Phasellus ullamcorper commodo velit. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Phasellus est. Maecenas felis augue, gravida quis, porta adipiscing, iaculis vitae, felis. Nullam ipsum. Nulla a sem ac leo fringilla mattis. Phasellus egestas augue in sem. Etiam ac enim non mauris ullamcorper scelerisque. In wisi leo, malesuada vulputate, tempor sit amet, facilisis vel, velit. Mauris massa est, sodales placerat, luctus id, hendrerit a, urna. Nullam eleifend pede eget odio. Duis non erat. Nullam pellentesque.

[5]**Pierre1901**.

Première section de l’intro

Lorem ipsum dolor sit amet, « consectetuer » adipiscing elit. Maecenas fermentum, elit non lobortis cursus, orci velit suscipit est, id mollis turpis mi eget orci. Ut aliquam sollicitudin metus. Mauris at sapien sed sapien congue iaculis. Nulla lorem urna, bibendum id, laoreet iaculis, nonummy eget, massa. Phasellus ullamcorper commodo velit. Class aptent taciti sociosqu ad litora torquent per « conubia nostra », per inceptos hymenaeos. Phasellus est. Maecenas felis augue, gravida quis, porta adipiscing, iaculis vitae, felis. Nullam ipsum. Nulla a sem ac leo fringilla mattis. Phasellus egestas augue in sem. Etiam ac enim non mauris ullamcorper scelerisque. In wisi leo, malesuada vulputate, tempor sit amet, facilisis vel, velit. Mauris massa est, sodales placerat, luctus id, hendrerit a, urna. Nullam eleifend pede eget odio. Duis non erat. Nullam pellentesque.

Une boite magique :

Titre de la boite

Praesent placerat, ante at venenatis pretium, diam turpis faucibus arcu, nec vehicula quam lorem ut leo. Sed facilisis, augue in pharetra dapibus, ligula justo accumsan massa, eu suscipit felis ipsum eget enim.

Laoreet iaculis, nonummy eget, massa. Phasellus ullamcorper commodo velit. Class aptent taciti sociosqu ad litora torquent per « conubia nostra », per inceptos hymenaeos. Phasellus est. Maecenas felis augue, gravida quis, porta adipiscing, iaculis vitae, felis. Nullam ipsum. Nulla a sem ac leo fringilla mattis. Phasellus egestas augue in sem. Etiam ac enim non mauris ullamcorper scelerisque. In wisi leo, malesuada vulputate, tempor sit amet, facilisis vel, velit. Mauris massa est, sodales placerat, luctus id, hendrerit a, urna. Nullam eleifend pede eget odio. Duis non erat. Nullam pellentesque.

Une boite simple :

Mauris lorem quam, tristique sollicitudin egestas sed, sodales vel leo. In hac habitasse platea dictumst. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed sed lorem lacus, at venenatis elit. Pellentesque nisl arcu, blandit ac eleifend non, sodales a quam.

Laoreet iaculis, nonummy eget, massa. Phasellus ullamcorper commodo velit. Class aptent taciti sociosqu ad litora torquent per « conubia nostra », per inceptos hymenaeos. Phasellus est. Maecenas felis augue, gravida quis, porta adipiscing, iaculis vitae, felis. Nullam ipsum. Nulla a sem ac leo fringilla mattis. Phasellus egestas augue in sem. Etiam ac enim non mauris ullamcorper scelerisque. In wisi leo, malesuada vulputate, tempor sit amet, facilisis vel, velit. Mauris massa est, sodales placerat, luctus id, hendrerit a, urna. Nullam eleifend pede eget odio. Duis non erat. Nullam pellentesque.

COLLEGES SCIENCES
BRETAGNE POUR L'INGENIEUR
LOIRE ET LE NUMERIQUE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Titre : titre (en français)..............

Mot clés : de 3 à 6 mots clefs

Résumé : Eius populus ab incunabulis primis ad usque pueritiae tempus extremum, quod annis circumcluditur fere trecentis, circummurana pertulit bella, deinde aetatem ingressus adultam post multiplices bellorum aerumnas Alpes transcendit et fretum, in iuvenem erectus et virum ex omni plaga quam orbis ambit inmensus, reportavit laureas et triumphos, iamque vergens in senium et nomine solo aliquotiens vincens ad tranquilliora vitae discessit. Hoc inmaturo interitu ipse quoque sui pertaesus excessit e vita aetatis nono anno atque vicensimo cum quadriennio imperasset. natus apud Tuscos in Massa Veternensi, patre Constantio Constantini fratre imperatoris, matreque Galla. Thalassius vero ea tempestate praefectus praetorio praesens ipse quoque adrogantis ingenii, considerans incitationem eius ad multorum augeri discrimina, non maturitate vel consiliis mitigabat, ut aliquotiens celsae potestates iras principum molliverunt, sed adversando iurgandoque cum parum congrueret, eum ad rabiem potius evibrabat, Augustum actus eius exaggerando creberrime docens, idque, incertum qua mente, ne lateret adfectans. quibus mox Caesar acrius efferatus, velut contumaciae quoddam vexillum altius erigens, sine respectu salutis alienae vel suae ad vertenda opposita instar rapidi fluminis irrevocabili impetu ferebatur. Hae duae provinciae bello quondam piratico catervis mixtae praedonum.

Title: titre (en anglais)..............

Keywords: de 3 à 6 mots clefs

Abstract: Eius populus ab incunabulis primis ad usque pueritiae tempus extremum, quod annis circumcluditur fere trecentis, circummurana pertulit bella, deinde aetatem ingressus adultam post multiplices bellorum aerumnas Alpes transcendit et fretum, in iuvenem erectus et virum ex omni plaga quam orbis ambit inmensus, reportavit laureas et triumphos, iamque vergens in senium et nomine solo aliquotiens vincens ad tranquilliora vitae discessit. Hoc inmaturo interitu ipse quoque sui pertaesus excessit e vita aetatis nono anno atque vicensimo cum quadriennio imperasset. natus apud Tuscos in Massa Veternensi, patre Constantio Constantini fratre imperatoris, matreque Galla. Thalassius vero ea tempestate praefectus praetorio praesens ipse quoque adrogantis ingenii, considerans incitationem eius ad multorum augeri discrimina, non maturitate vel consiliis mitigabat, ut aliquotiens celsae potestates iras principum molliverunt, sed adversando iurgandoque cum parum congrueret, eum ad rabiem potius evibrabat, Augustum actus eius exaggerando creberrime docens, idque, incertum qua mente, ne lateret adfectans. quibus mox Caesar acrius efferatus, velut contumaciae quoddam vexillum altius erigens, sine respectu salutis alienae vel suae ad vertenda opposita instar rapidi fluminis irrevocabili impetu ferebatur. Hae duae provinciae bello quondam piratico catervis mixtae praedonum.