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

Exploration d'ensembles de modèles II

Object Oriented 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), <u>models</u> represent <u>structured data</u>, and the <u>model of the data structure</u> is known as a <u>metamodel</u>. 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

MODEL DRIVEN ENGINEERING: A MODEL MODELING MODELING

Model Driven Engineering (MDE) has emerged as a central paradigm in software development. It's history is intertwined with that of Object Oriented Programming (OOP). Where the OOP paradigm can be summarized as *everything is an object*, MDE is similarly described as the paradigm where *everything is a model*. Two of the longest standing actors are the Object Management Group (OMG) which notably provides specifications for both the Unified Modeling Language and the Object Constraint Language, and Eclipse providing the Eclipse Modeling Framework (EMF). EMF allows for the generation and testing of code using UML specifications, the design and use domain specific languages, and the manipulation of data by means of model transformation. More recent efforts include JetBrains' Meta Programming System (MPS) which focuses on providing tools to develop and use Domain Specific Languages.

Models are used because they are: cheaper, safer, easier to manipulate and learn from. For software development, software engineers build models of the desired software. Such models allow the engineers to quickly iterate and test their design, but also serve as a language to communicate with the software developers who will implement the software. Beyond software development, models are similarly used: scaled-down models of buildings are easier to make and iterate upon and can be used to communicate with stake holders, weather models provide forecasts which are crucial to farming, digital-twins allow for the monitoring and interaction with complex systems.

In the following sections we'll present UML and OCL. We'll then present the state-of-the-art for MDE, and the tools we are building upon.

1.1 The Unified Modeling Language

UML provides a range of languages for modeling purposes. We need languages to describe metamodels and models. For models, UML provides Object Diagrams, and for metamodels, Class Diagrams and the Object Constraint Language. Additionally, we will use C-like syntax for Class specifications.

1.1.1 Meta-Object Facility

The Unified Modeling languguage is build ontop of the Meta-Object Facility standard. Formally: the UML metamodel conforms to MOF meta-metamodel.

The MOF standard also describes a four layer model hierarchy illustrating its application. The MOF meta-metamodel exists as the top layer of abstraction in the MOF architecture, the M3 layer. The MOF architecture has three lower layers of abstraction, the lowest being that of the system under study, the M0 layer. The UML metamodel exists at the M2 layer, and the M1 layer contains the UML conforming models and metamodels created by the user to represent the system under study. The UML languages we'll use to specify metamodels and models will are outlined in the following sections on class diagrams, object diagrams and the object constraint language.
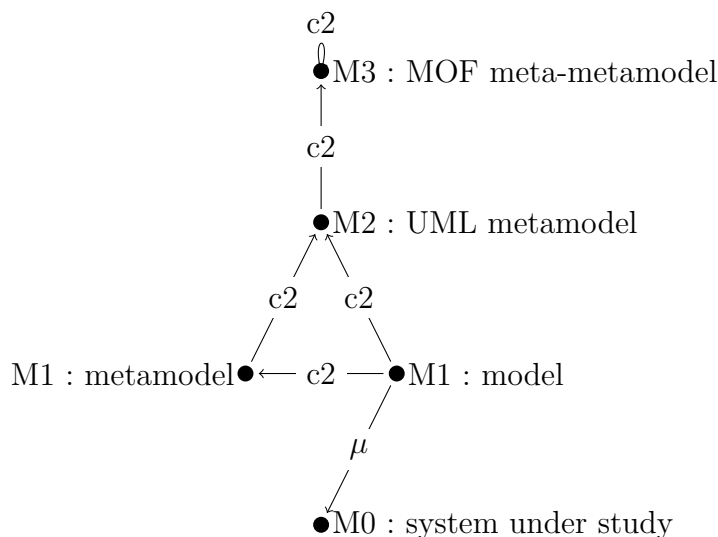
Figure 1.1: Meta-Object Facility model hierarchy

1.1.2   Class Diagrams

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 opposite, child) to express relationships between people.
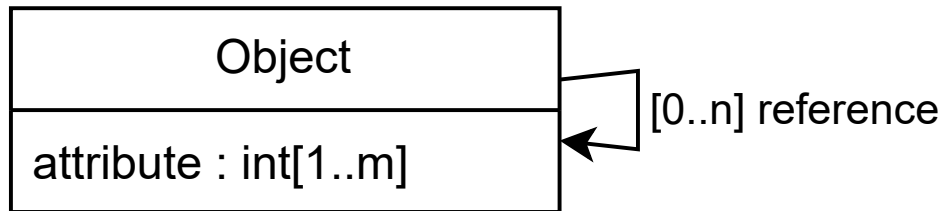
Figure 1.2: UML Class Diagram as Metamodel

Here we 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.

```
1 class Person {
2     age : int ,
3     parents : Person [2..2] set
4 }
```

Here we have a C-like description of a si

UML Collection Types

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

UML Reference Types

UML also gives us different kinds of relations between objects, notably containment relations and and opposite relations.

Opposite The opposite of *child-of* for instance, is *parent-of*. Sometimes references may need to be one-way for encapsulation.

Containment means that the *contained* object is only related to one *container* object through this instance of the relation.

1.1.3   Object Diagrams

describe instances of the classes defined in a class diagram. For example, Figure 1.3 shows an instance conforming to the class diagram in Figure 1.2. 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 1.3: UML Instance Diagram as Model

This diagram is also built from rectangles and lines connecting them. Here each rectangle represents an object, and the title area is used to give the :Type and optionally

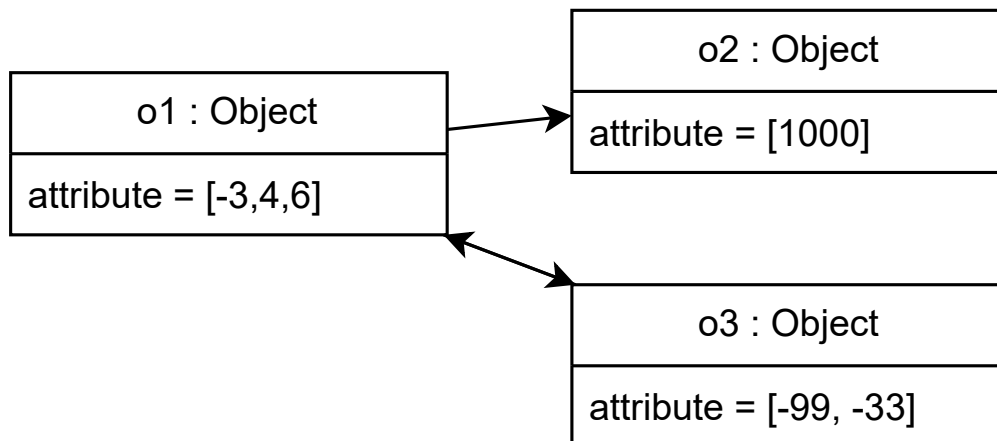a name. Below the title area, we again use the rectangle to list the attributes and the information associated. The arrows between the rectangles represent instances of the references between objects. In Figure 1.3 we see a model which conforms to the metamodel from Figure 1.2, for which we chose $n = 2$ and $m = 3$. This here shows objects of the type described by the class, and the values assigned to their properties. In this model, `o2 : Object` is an instance of the `Object` class named `o2`, as attribute it has a collection of integers with a single value: 1000. The object `o1` has as attribute a collection of 3 integers and is connected to both other objects, which are the maximums allowed by our choice of $n = 2$ and $m = 3$.

1.2 The Object Constraint Language

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.

Given an instance such as the one shown in Figure 1.3, 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.

1.2.1 OCL by example

OCL is designed to be easy to read an write. So for an initial introduction we'll use a simple example, and identify the core parts of the language. The primary use of OCL is to add additional constraints to a class diagram.

This class diagram describes people, giving their age, and listing their children.

However, 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:

```
1 context Person inv:
2   self.children.age.forall(a| a < self.age)
```

Figure 1.4: Simple class diagram describing people

This constraint states that for every Person instance, all of their children must be younger. The `context` keyword specifies the class to which the constraint applies, and `inv` stands for <u>invariant</u>, 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.

self: identidying objects in the model

The expression `self` refers to the current object of the set identified by the context. It identified the main variable of the expression. `self.attribute` returns its attribute values, and `self.reference` retrieve related objects. Chained queries like `self.reference.attribute` retrieve the attributes of referenced objects.

Building OCL expressions by chaining operations

OCL expressions a built using function composition.

$$source.operation(arguments)$$

OCL operations generally are applied OCL expressions identified as *source*, which can be understood as the first argument for the operation. Operation arguments generally take the form of OCL expressions.

Navigation: relation to the metamodel

The primary operation on object models, is accessing their properties, and navigating the graph of objects. OCL provides language to navigate the model and retrieve it's information.

Property Access OCL uses a `.property` notation for property access.

For example to access the age and the children of a person, in ocl you would write.

<div align="center">

`self.age`

</div>

<div align="center">

`self.children`

</div>

Navigation

To access the children's age, we need to navigate away from the object designated `self`, to the objects representing their parents, and access their age attribute.

In the general case, if the source expression results in a collection of objects, we can navigate their references to a new collection of objects and access their properties.

<div align="center">

`source.prop`

</div>

1.2.2   OCL Operations summary

OCL 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()`.

This cheat sheet outlines all the available OCL expressions, organized by the types the can be applied to.

1.2.3   Typing OCL Expressions

Typing is the process of assigning a data type to expressions, variables, and operations in a language. It ensures that operations are performed on compatible data types, preventing logical errors and enhancing code clarity. In the context of Object Constraint Language (OCL), typing plays a critical role in defining constraints and queries over UML models, ensuring that expressions are both syntactically and semantically valid. OCL is statically typed, meaning types are checked at compile-time rather than runtime.

This diagram specifies the type system for OCL.

`InvalidType` represents incorrect OCL expressions. `VoidType` represents *absent* values. Such as when `self.children` points to no one. `Class` represents the use of class

Figure 1.5: snippet from the OCL Cheat Sheet provided by eScribis

names from the metamodel in the model constraint expressions. `DataType` represents the data that can be found in the model, inferred from it, and the literals in the ocl expressions. `PrimitiveType` represents the type of data that can be used, such as: integers, reals, booleans, strings. `CollectionType` represents collections of data.

To color our previous ocl expression on the Person metamodel:

$$self.children.forall(c|c.age < self.age)$$

`.children` is of type `CollectionType` `.age` resolves to an integer which is a `PrimiteType`. `forall` and `<` resolve to a boolean which is a `PrimitiveType`.
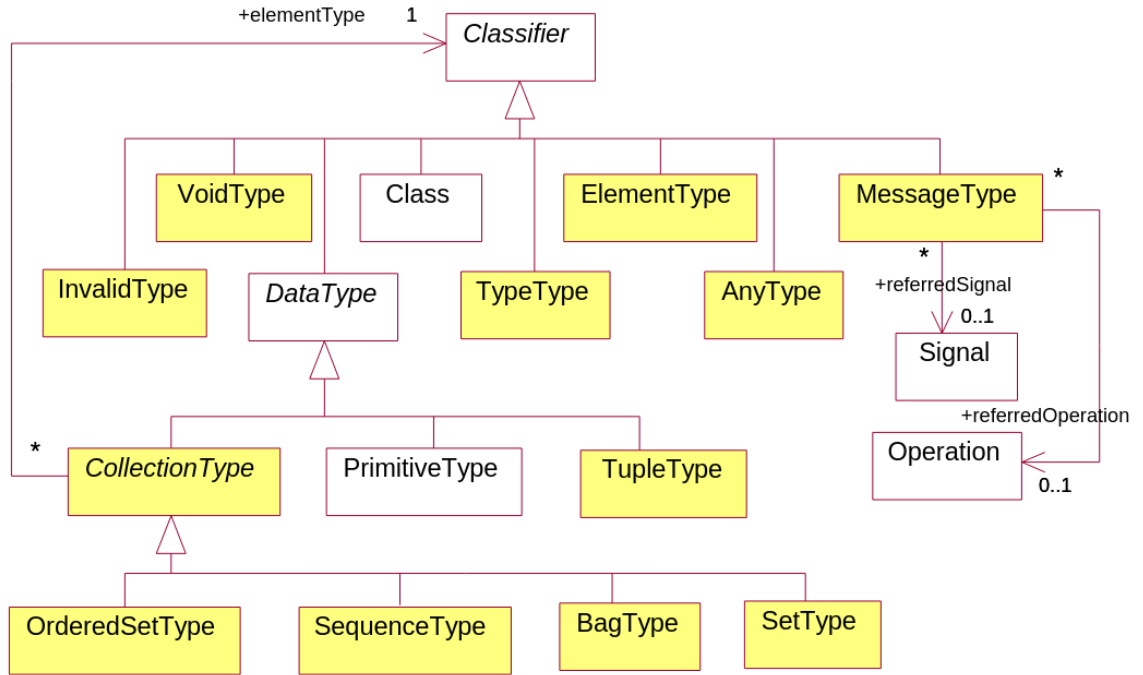
Figure 1.6: OCL Types

This is also a decent metric for OCL coverage. In this thesis we will be mainly focusing expressions which resolve to a CollectionType and the Integer and Boolean PrimitiveType. We will also have some cases of Class type, notable when handling `Person` in the expression `Person.allInstances()` We will also have to manage InvalidType and VoidType.

There are also limitations with the standard OCL typing system, especially around invalid and void expressions, and efforts such as OCL propose a typing system based on relational logic. Similar to Alloy. In relational logic all expressions are resolve to sets. Invalid and Void types are essentially encoded as the empty set .

$\rightarrow$ **MC** we leverage the type system to define CSP / type is checked before building csp

1.2.4   OCL Expression Metamodel

OCL being a domain specific language in the model driven ecosystem, we naturally have a class diagram modeling the language. $\rightarrow$ **MC** explain The OCL metamodel describes the "kinds of words" which appear in the OCL language, and the patterns in which they appear.
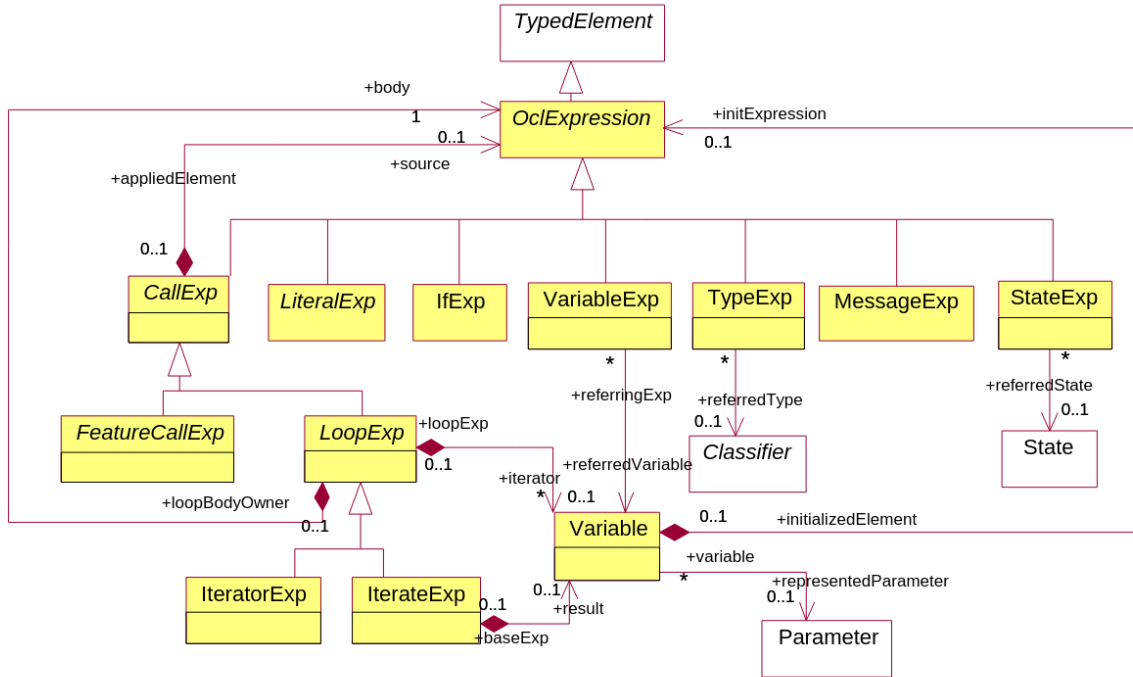
Figure 1.7: OCL Expression Metamodel (simplified)

The top level concept is that of an OCL expression: `OclExpression`. As previously stated, all OCL expressions are typed, hence they all inherit from the concept of `TypedElement`.

To color our previous ocl expression on the Person metamodel:

$$self.children.forall(c|c.age < self.age)$$

`self` and `c` are `VariableExp`. `.children` and `.age` are `FeatureCallExp`. `forall` is a `LoopExp`. `<` is a `CallExp`.

The CallExp represents most of the words in ocl, notably model navigation and operation calls. The simplest operations represented by `CallExp` are those applied to primitives, such as arithmetic operations for integers, and logical operations on booleans and integers: $+, <, \land$. The `FeatureCallExp` represents property access, providing attribute values and the objects resulting from references. Naturally the type of whats returned from property access varies greatly. The `LoopExp` groups operations applied to expressions of type `Collection`, notably iterate, and its special cases such as: sum, `size`, forall, etc. . . which make up the majority of the collection operations.

Query Expression: `self.ref.select(r | predicate(r)).prop.sum()` OCL expres-

sion resulting in a collection of integers or objects.

1.3 EMF and ATL: State-of-the-Art

1.3.1 Eclipse Modeling Framework

The Eclipse Modeling Framework is a suite of tools for the Eclipse ecosystem, allowing users design data structures, manipulate structured data and generate code.
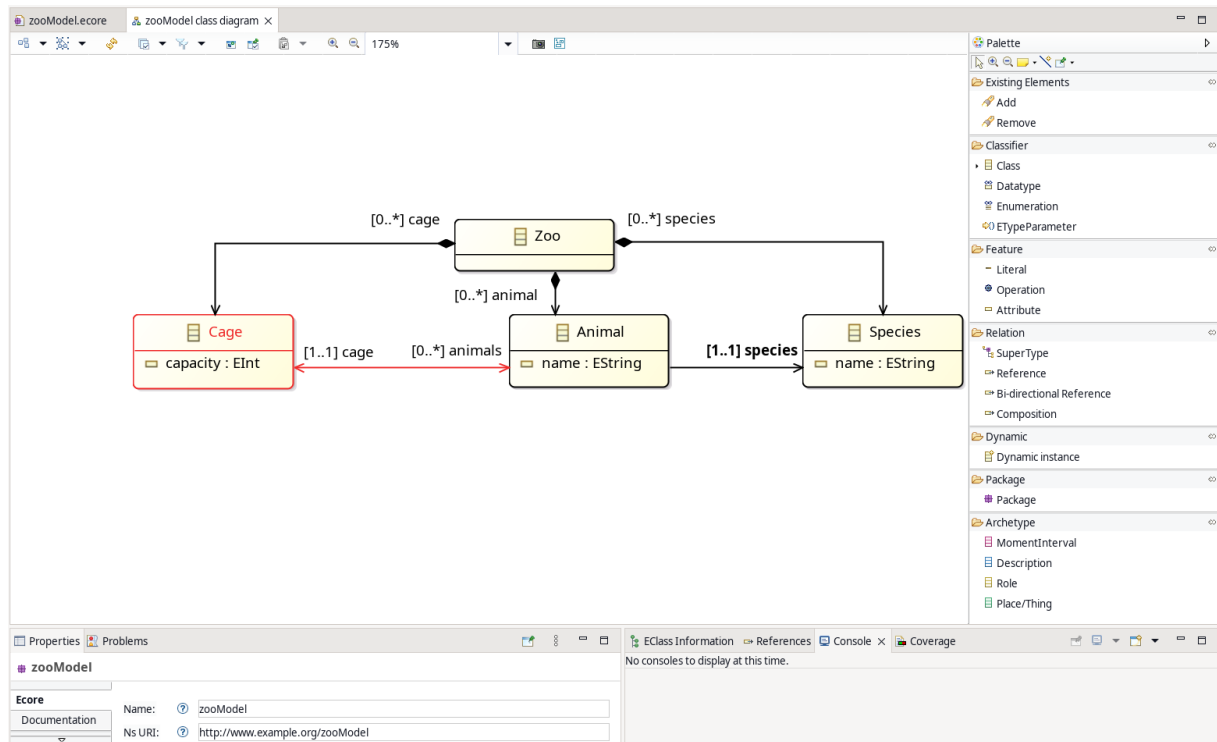
Figure 1.8: Using the Eclipse Modeling Framework to edit an Ecore metamodel using a Class Diagram editing tool

In this screenshot we can EMF being used to create a Class Diagram to design a zoo application. From this metamodel, we'll be able to generate a tool to edit object models representing instances, and serialize them in XMI files.

XMI XML Metadata Interchange, a file format specified by OMG for the serialization of MOF Models. EMF provides tools to load and save models from and to XMI files. Loading an XMI model into and EMF editor generally requires the Ecore metamodel it conforms to, and generates EObjects which can be manipulated. In this listing we see a

```
1  <Model>
2      <object att="information␣one" ref="//@object.1"/>
3      <object att="information␣two" ref="//@object.0"/>
4  </Model>
```

Listing 1: Minimal Object Model in the XMI format

simple model, with two linked objects, each object holds some information.

Ecore is the EMF format for class models (metamodels). Ecore is an XML-like language based on XMI. Loading an Ecore file entails instantiating EObjects of type EClasses, EAttributes and EReferences corresponding to the classes of the class model and their properties, and generating code to manipulate them and their corresponding objects. EMF provides tools so visualize these metamodels as Class Diagrams.

TODO: Add Ecore metamodel.

EObject java interface representing objects. Provides access to the EClass it conforms to. Provides access to the information via *getters* and *setters*, and the EReference or EAtrribute representing the class property.

EClass java interface representing classes. EClasses provide access to the properties.

EReference java interface representing properties of type reference. Provides access to the name, type, and collection cardinality.

EAttribute java interface representing properties of type attribute. Provides access to the name, type, and collection cardinality.

1.3.2 Atlas Transformation Language

The Atlas Transformation Language (ATL) provides a powerful framework for specifying model-to-model transformations within the Eclipse Modeling Framework (EMF). ATL is designed to automate the conversion of models from one metamodel to another, enabling the creation of model transformation pipelines—a critical feature for modern Model-Driven Engineering (MDE) workflows. ATL extends OCL (Object Constraint Language) to define transformation rules that operate across multiple classes. These rules establish bindings, which associate properties between source and target model elements using OCL queries. This allows ATL to express complex mappings between metamodels in a declarative style, focusing on what should be transformed rather than how. These rules associate properties across the classes using OCL queries, in what is called: a binding.

Just like OCL, ATL can be written declaratively, but ATL also allows for imperative

sections, which are sometimes necessary to specify *steps* required to transform an object into another.

Person
+ name : string
+ family_name : string

author [1..1]

Book
+ name : string
+ year : int

BookEntry
+ name : string
+ author : string
+ year : int

((a)) Source metamodel: people and books

((b)) Target metamodel: library book entry
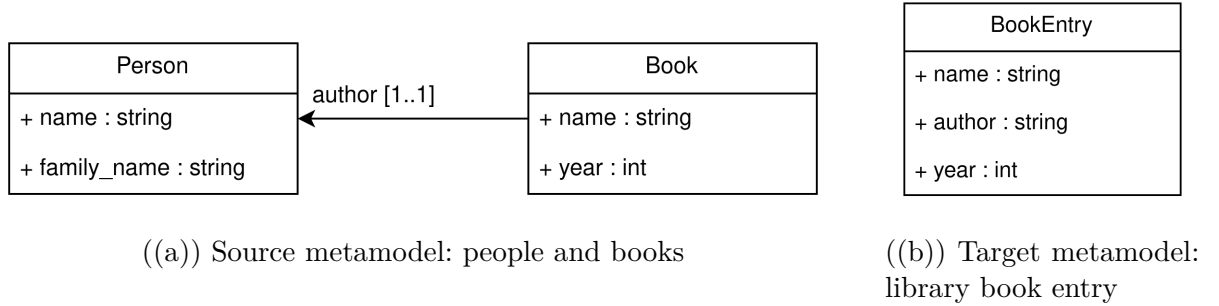
In these two figures we find a source and a target metamodels. The source metamodel describes books and their relation to people. And the target metamodel is a specification for a book entry in a library.

```
1   // ATL transformation rule
2   rule BookToBookEntry {
3   from
4       b : BookMM!Book
5   to
6       be : LibraryMM!BookEntry (
7           title <- b.title ,
8           authorName <- b.author.family_name.toUpperCase () + b.author.
                name ,
9           year <- b.year
10      )
11  }
12 }
```

In this example, we focus on a transformation rule, which produces `BookEntry` objects from `Book` objects and their associated `Person` objects. A rule has two parts: a `from` section which identifies source classes and `to` which identify target classes, and binds their properties to those of the source. In the binding `authorName <- b.author.name + b.author.first_name`, we link the `authorName` property of an entry `be`, to the conjunction of the `name` and `family_name` of the `Person` identified by the `author` property of the book `b`.

25

CONSTRAIN PROGRAMMING: AN EXACT AND EXPLAINABLE ARTIFICIAL INTELLIGENCE

Constraint Programming [?] is a powerful paradigm that offers a generic and modular approach to modeling and solving combinatorial problems.

2.1 Constraint Satisfaction 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 $c(X(c), R)$ 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.

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.

2.2.1 Global Constraints used in this thesis

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. This constraint is also known as nth in systems like ECL^iPS_e and swi-prolog.

- 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 $b = max(s_0 + d_0, .., s_n + d_n)$,

2.2.2 Propagation of Element

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.

2.2.3 Propagation on boolean variables and CNF models

→ **MC** DDPL is a propagator for the gobal constraint "CNF model"

2.2.4 Propagation on real variables and systems of linear equations

→ **MC** Simplex is a propagator for the gobal constraint "system of linear equations"

2.3 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 $(x_i = v)$.

2.3.1 Choco (and Gecode and OR-tools)

$\rightarrow$ **MC** API for using CP from source code $\rightarrow$ **MC** parse flatzinc

2.3.2 SWI-prolog and ECL^iPS_e

$\rightarrow$ **MC** Solvers using prolog like language to model problems

$\quad \rightarrow$ **MC** Combine traditional prolog techniques with constraint programming techniques

2.3.3 SAT4j and Cassowary

$\rightarrow$ **MC** focus on SAT and LP propagation: DDPL and Simplex

$\quad \rightarrow$ **MC** Often employed by CP solvers for globals such as "CNF model"

MODEL SEARCH USING ARTIFICIAL INTELLIGENCE:OBJECT ORIENTED CONSTRAINT PROGRAMMING

3.1 Model Search

Model search exists at the interesection of Model Driven Engineering and Artificial Intelligence. EMF has a need for tools assisting in the modeling process. Model search is a powerful tool with many applications. One of the open questions in MDE is *how to effectively find models that conform to a metamodel*. When the metamodel has model constraints the problem may become complex.

Because MDE leverages formal languages for modeling purposes a natural choice of AI are those solving Constraint Satisfaction Problems.

3.1.1 Model Search

In → MC cite Kleiner there is a formal definition for model search.

Relaxed Metamodel A relaxed metamodel is a metamodel for which a subset of the constraints are not enforced. These constraints include property cardinalities and model constraints.

Partial Model A partial model conforms to a relaxed metamodel, and partially conforms to the metamodel which was relaxed. This generally means it is populated with Objects, but missing information on the values of referrences and attributes.

$$\exists o \in \texttt{Object} | o.\texttt{prop} = \emptyset$$

For a `Person`, this means not having their age, name, or know which other people they

are related to.

Partially-conforms-to The relation between a Partial Model and the metamodel which was relaxed.

Model Search
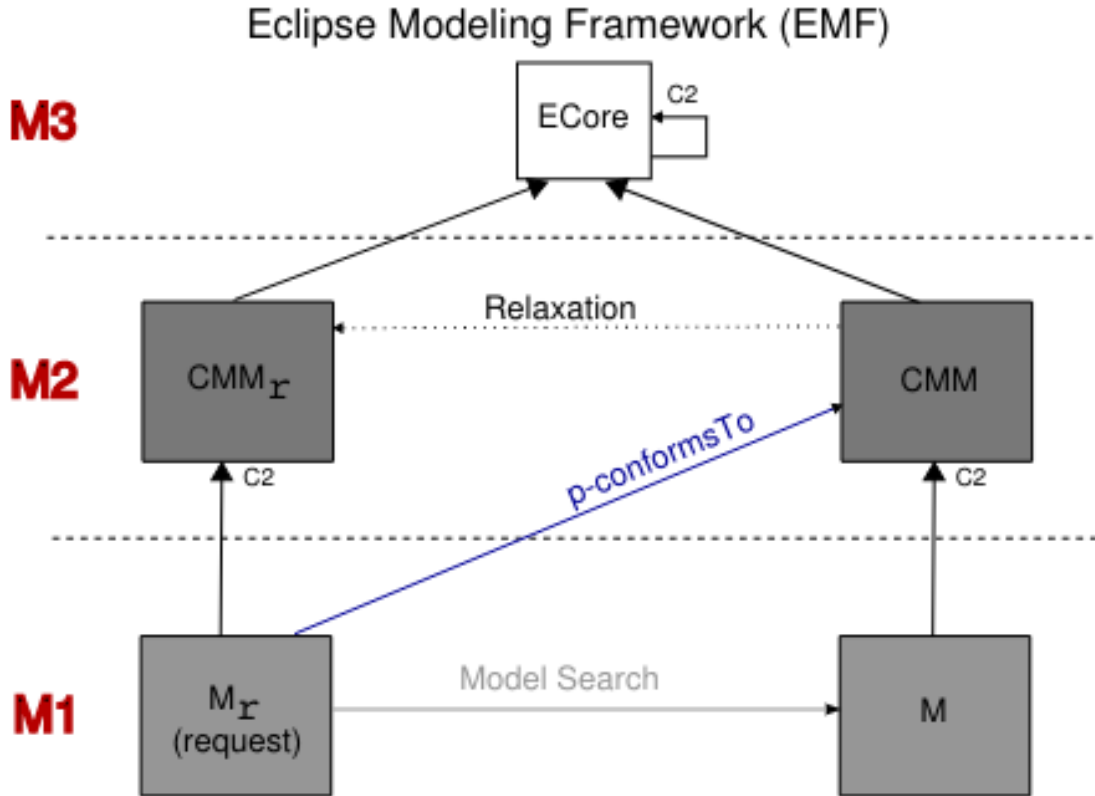
Figure 3.1: Model Search

3.1.2 Applications of Model Search

Model Instantiation is the process of generating a model that conforms to a metamodel. Generally this means providing a partial model, with all the object instances. The count of objects per class is often a parameter for model instanciation.

Model Completion is the process of taking a model with missing information and infering possible information. Generally this means providing a partial model, with all

the object instances and some immutable data for the objects. The missing data is the reason the model only partially conforms.

Model Repair is the process of making a conforming model from a non-conforming model. Repairing a model using model search implies creating a partial model by removing information from the object properties. Automatically removing objects doesn't align with with our model repair use-cases, and is generally uncommon.

model space exploration: classical model transformation tools associate one source model to one target model. Model space exploration involves transformations transformations towards sets of models. Domain Space Exploration describes a system of model transformations, model space exploration uses transformations over sets to model the system of model transformations.

3.2 State of the Art

3.2.1 ATL^c

ATL^c provides an extension to the ATL allowing users to add OCL invariants to the targets of transformation specifications. ATL^c then translates the conjunction of these invariants into a CSP which can be solved by a range of solvers, such as: Choco and Cassowary (Simplex). In ATL^c invariants, only integer attributes can be variables, and variables are identified by the last `PropertyCallExp` of a query expression.
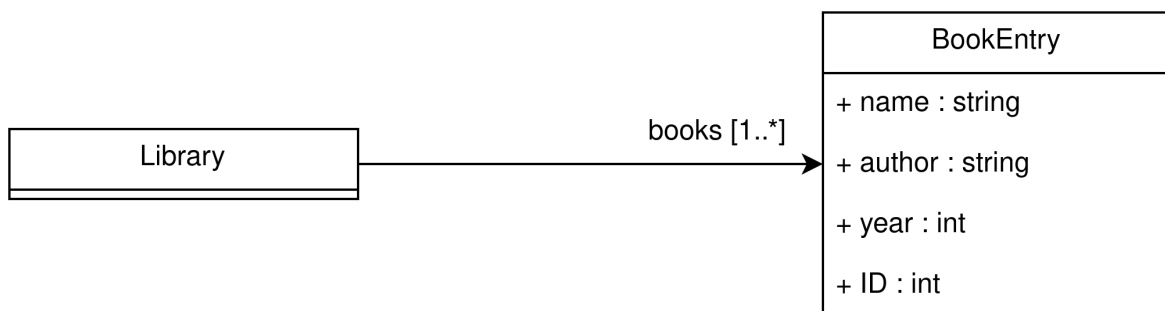
Figure 3.2: Target metamodel: Library

```
1    rule CreateLibrary {
2    to
3        lib : LibraryMM!Library (
```

```
4                   books <- BookMM!Book.allInstances()->collect(b | thisModule.
                        BookToBookEntry(b))
5            )
6            cstr : Constraints!Constraint(
7                    value <- lib.books.ID->isUnique()
8            )
9  }
```

Here we complete the transformation from the ATL → **MC** section, by providing a rule to create a library. In the `to` section, we see two targets: a `Library` object, for which the reference `books` is populated by a standard binding, and a `Constraint` object. The `cstr` target allows the user to add invariants to a transformation rule. Here the invariant states that *books in a library must have unique IDs*. This invariant replaces a specification of how to make a unique ID. ATL^c assumes `ID` is the variable in this invariant, not the `books` relation, as `ID` is the last `PropertyCallExp` in the query `lib.books.ID`.

Describing a satisfactory model in such away, can be easier and shorter than describing how to make a satisfactory model.

→ **MC** Can leverage different kinds of CP solvers, such as Cassowary (simplex) for positioning GUI elements.

→ **MC** Another effort similar to this one is QVTc: passing parts of a transformation to a solver to use model search to finish a transformation (uses Choco to make an ECL^iPS_e spec)

→ **MC** Another effort similar to this one is Grimm: refining, uses Choco, adds heuristics to make statistically ineteresting targets

→ **MC** In the context of Model Search, the standard ATL transformation provides the relaxed-metamodel and produces the partial model. The relaxed-metamodel is completed with the ATL^c invariants, and a solver is leveraged to finish the transformation.

3.2.2 EMFtoCSP

→ **MC** Prolog is a natural way to implement a language

→ **MC** SWI-prlog and ECL^iPS_e also provide constraint programming techniques ontop of prolog techniques

→ **MC** EMF2CSP uses ECL^iPS_e to neatly describe OCL, and leverage gobal constraints such as *element* (like us)

→ **MC** Important note (un-like us): This method will generate a different problem

for each combination of collection sizes. Whereas we add the "what size is the collection" problem to the CSP.

$\rightarrow$ **MC** related work OCL#

3.2.3 Alloy & Kodkod

Alloy provides a framework for modeling problems and solving them using *off-the-shelf* SAT solvers. The alloy language is inspired by the Z modeling language and the Object Constraint Language. It provides both a means to specifiy class models and model constraints.

Alloy Language

```
1  sig Species {}
2
3  sig Animal {
4      cage: one Cage,
5      species: one Species
6  }
7
8  sig Cage {
9      capacity: Int,
10     animals: seq Animal
11 //  animals: set Animal
12 }{
13     #animals <= capacity
14     #animals.species.elems <2
15 //  one animals.species
16 }
```

Listing 2: Minimal Object Model in the XMI-like format

In this listing we see our example zoo model. In alloy a class can be modeled using signatures. We can see, signatures are similarly named lists of properties. Properties are similarly typed using primitives for attributes, and other signatures for references. Properties in alloy hold a single value by default, but can declared to be a collection, such as set and sequence. Signatures can also be used as the context for model constraints. This is similar to how contraints are defined using OCL.

```
1 sig S {R : one T}
2 sig T {}
3 run {#S=2 #T=2}
```

Listing 3: Simple Alloy model, with two related concepts, with two instances each

Kodkod

Kodkod is an API to model relational first order logic problems and translate them to Conjunctive Normal Form (CNF) model. Alloy uses this API to translate models for solving with SAT solvers.

Here we have a simple alloy specification which simply illustrates how types and relations are modeled when looking at the resulting *relational first order logic* specification. This specification also describes the search space, by stating there should be two objects of type S, and two of type T

$$
\begin{aligned}
&\{a_1, a_2, a_3, a_4\} \\
S :_1\ &[\{\langle a_1\rangle, \langle a_2\rangle\}, \{\langle a_1\rangle, \langle a_2\rangle\}] \\
T :_1\ &[\{\langle a_3\rangle, \langle a_4\rangle\}, \{\langle a_3\rangle, \langle a_4\rangle\}] \\
R :_2\ &[\{\}, \{\langle a_1, a_3\rangle, \langle a_1, a_4\rangle, \langle a_2, a_3\rangle, \langle a_2, a_4\rangle\}] \\
&\forall s \in S : |R.s| = 1
\end{aligned}
\tag{3.1}
$$

Here we have a *relational first order logic* model of the alloy specification. The first line lists the atoms. There is one atom for each object of the model. When working with integers, each possible int value implies an additional atom. The next two lines represent the unary relations *is of type S* and *is of type T*. These relations are encoded as a set, for which we have the lower bound (objects that must be in the set) and upper bound (objects that maybe in the set). For these two relations S and T have identical upper and lower bounds, meaning its a constant of this problem. They identify atoms a_1 and a_2 as being of type S, and a_3 and a_4 of type T.

Next is the binary relation R, which describes the links from objects of type S to those of type T. If we look at the upper bound for the possibilities we see → MC stuff like $\langle a_1, a_3\rangle$, meaning maybe a_1 is linked to a_3. We have no information about existing relations between S and T objects, so the lower bound is empty.

The final line uses relational language such as R.s: *R join s*, to predicate on the relations. The s in $\forall s \in S$, will be instantiated as singletons relations, which can be joined

with the `R` relation, to get the subset of `R` corresponding to the s object. Ultimately the alloy expression `one T` gets translated to $|R.s| = 1$, or the size of the subset resulting from the join $R.s$ is equal to one.

→ **MC** If you want to have a sequence of T, we need to add atoms for integers (a_5,a_6,a_7,etc..). And the relation `R` becomes a ternary relation where the tuples now include the integer atom representing the position of the link in the sequence: $\langle a_6, a_1, a_3 \rangle$ which would mean the first (1=a_6) link between from a_1:S is to a_3:T.

→ **MC** Similar to this work: OCLsolve

Contribution : OCL Variable Declaration `VarOperationExpression`

4.1 Problem

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.

4.1.1 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

Figure 4.1 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)[1], that connects classes by unidirectional references, instead of bidirectional associations.

---

[1] https://eclipse.dev/modeling/emf/
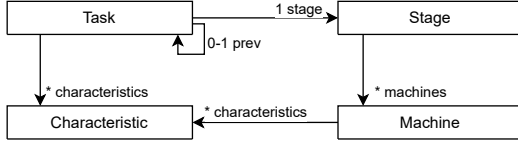
Figure 4.1: Class Diagram for RMS Task constraints
[class diagram of RMS]

```
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 4: 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 Figure 4.1 cannot encode all constraints that are required for an instance to be a correct RMS instance. Additional constraints can be specified using OCL.

4 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 4 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 4 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.[2]

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.

4.2 Denoting Variables

4.2.1 Base Syntax

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:

$$\texttt{source.var('property')}$$

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.

---

[2]The constraint uses the `stageNum` constant integer attribute that indicates the position of the stage in the manufacturing line, omitted in Figure 4.1.

In **??** we apply the operator to `SameCharacteristicConstraint` in 4 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).[3] Whereas, if one wants to enforce OCL, `var` is used as a hint to build the corresponding CSP.

4.2.2 Parameters to guide modeling and and search

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.

- **max_card** allows a user to set the maximum number of elements in the property. Required if a property's max cardinality is set to infinite.

- **min_card** allows a user to set the minimum number of elements in the property.

- **lb** allows a user to set a lower bound for the domain of the variables encoding the property.

- **ub** allows a user to set an upper bound for the domain of the variables encoding the property.

- **enumerate-domain** applies an encoding which enumerates the domain, allowing for additional filtering models.

- **branch-priority** make the solver prefer to branch off these variables.

- **branch-max** make the solver prefer to try large values for these variables.

- **branch-min** make the solver prefer to try small values for these variables.

- **fill** allows a user to preserve the values that may already be present for the property.

---

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

- **temp** guides model repair, positive temperatures allow for values to change, negative values allow for values to resist change.

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.

4.2.3   Vocabulary for Annotated Expressions

Variable Property: these imply adding variables and possibly constraints to the UML CSP. Non-annotated properties will be called constant. From annotation of the OCL we infer which properties are 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.

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 subexpressions, and variable expressions can be decomposed into variable and constant subexpressions. 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`
machines and characteristics are reached through variable navigations, the first being from
`Stage` to `Machines`, the second from `Machines` to `Characteristics` .

4.3   Refactoring OCL around annotations

4.3.1   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. **??** shows the AST of `SameCharacteristicConstraint` from **??** Scenario S3. We show `var` annotations as dotted rectangles.

?? 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 **??**, the scope defined by each `var` annotation is indicated by a dotted rounded rectangle. The `var` annotation requires everything <u>inside</u> 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 **??** (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 **??** has the same AST as in **??**, 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.

4.3.2 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 **??**, all have their annotations low in the tree **??**. 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: <u>All the machines connected to a task (via a stage), each individually match the task's characteristics</u> this is the same as requiring that: <u>The set of machines that match the task, includes the set of machines connected to the task.</u>

In 5 we can see the result of this rewrite for all three scenarios. The beginning of the

```
-- 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 **??** refactored around the annotations.

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 4.3 we can see the AST resulting from the parsing of the expression of 5 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

43

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.

4.4   Discussions
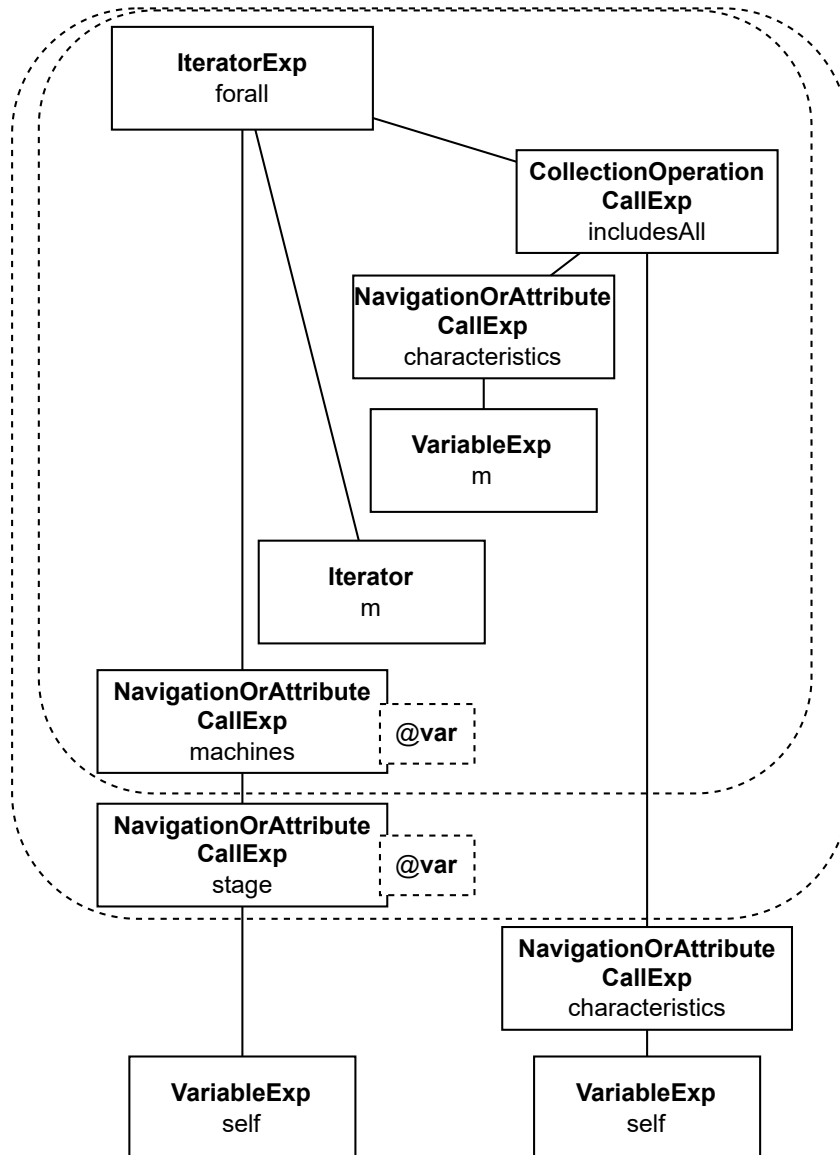
Figure 4.2: AST of `SameCharacteristicConstraint` from **??** Scenario S1, S2 & S3. [AST]
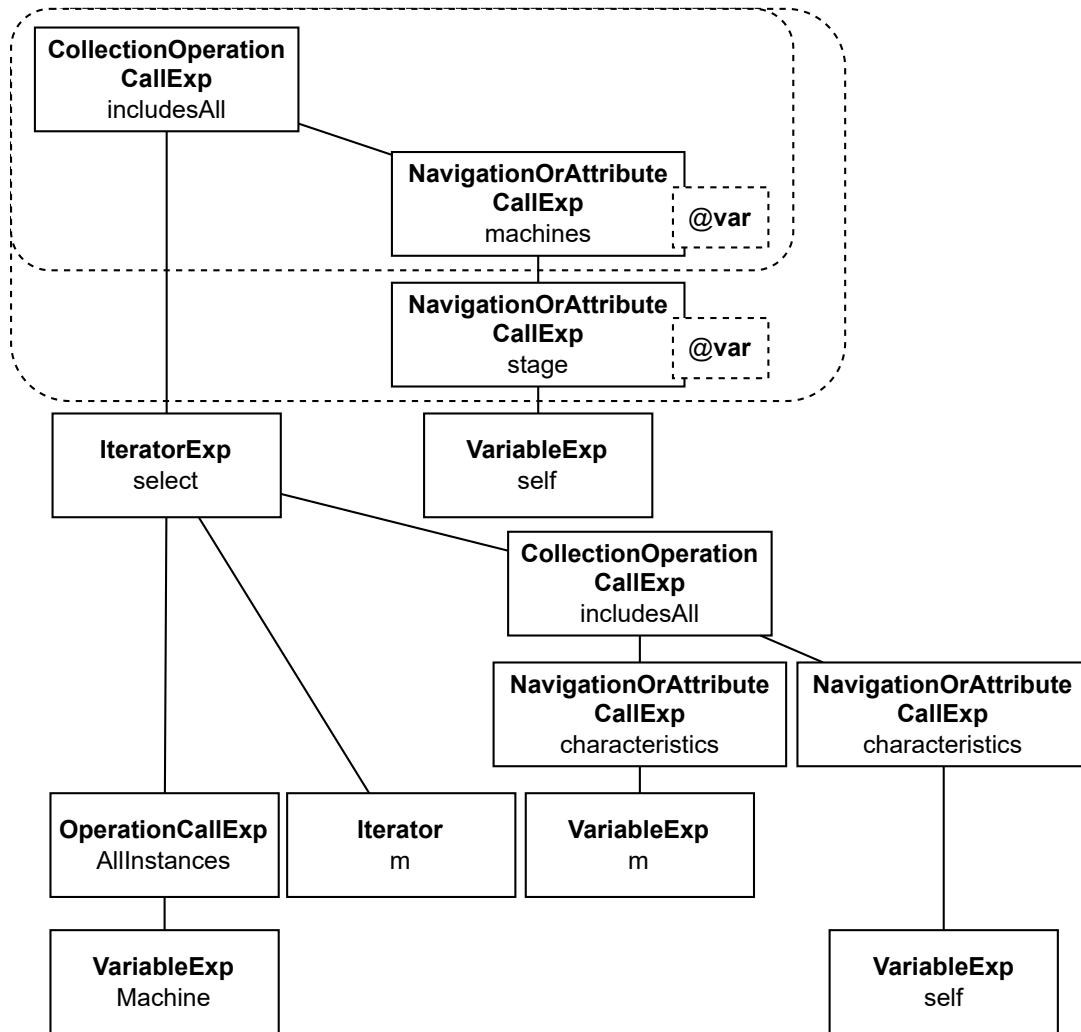
Figure 4.3: AST of SameCharacterticConstraint from 5 Scenario S2 & S3 [AST]

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

5.1   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 1.2.

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

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

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 5.1: Encoding of the instance from Figure 1.3 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 <u>nullptr</u>) denotes the absence of a reference. To support <u>nullptr</u>, an extra row is added to each table to represent a dummy object.

Table 5.1 shows the encoding of the instance from Figure 1.3, assuming $n = 2$ and $m = 3$ from the metamodel in Figure 1.2. The left side represents attributes, while the right side represents references. Each object (plus one <u>dummy object</u>) 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.

5.1.1 Additional *enumerated* encoding for References

References are a special case of property, with a grately restricted domain. The values in the domain of pointer variables identify rows of property tables. The domain of relation variables is generally small enough to be enumerated. Therefore, we give them

accompanying variables counting the occurrences for each value of the domain.

$$\begin{cases} \forall Class_o.ref \in Class.reference \\ \implies gcc(Class_o.ref, Class_o.refOcc) \end{cases} \tag{5.1}$$

$$Class.ref = \{r_{00}, ..., r_{nm}\}$$

$$Class.refOcc = \{occ_{00}, ..., occ_{nm}\}$$

$$\forall r \in Class.ref, \ domain(r) = \{0..n'\}$$

$$\forall occ \in Class.refOcc, \ domain(occ) = \{0..m\}$$

$$\forall n : object \in Class, \ gcc(Class_o.ref, Class_o.refOcc)$$

5.2 Constraint Models for UML Reference Types

When two references are opposites, such as <u>child</u> and <u>parent</u>, we need to do stuff:

$$\forall i, j \in \{0..n\} * \{0..n'\} A.refOcc_{ij}, \neq 0 \iff B.oppOcc_{ji} \neq 0$$

s

5.3 Constraint Models for UML Collection Types

As described in Section **??**, properties in class diagrams (e.g., Figure 1.2) 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 $S = \{s_1, \ldots, s_p\}$:

$$Sequence(X) : \begin{cases} regular(S, DFA) \\ s_i = [\![x_i \neq d]\!] \end{cases} \tag{5.2}$$

The automaton *DFA* (Figure 5.1) 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.
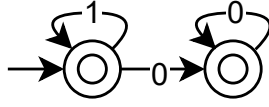
Figure 5.1: DFA packing dummy values for instance variables

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:

$$Bag(X) : \begin{cases} \forall i \in [1, p[: x_i \geq x_{i+1} \end{cases} \tag{5.3}$$

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:

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

$$Set(X) : \begin{cases} alldifferent\_except\_d(X) \\ Bag(X) \end{cases} \tag{5.4}$$

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:

$$OrderedSet(X) : \begin{cases} alldifferent\_except\_d(X) \\ Sequence(X) \end{cases} \tag{5.5}$$

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.

5.4 Using information from Variable Annotations

CONTRIBUTION : OCL NAVIGATION

6.1   CP model for OCL queries on the instance

Querying an instance involves navigating the object graph through references and retrieving attribute values. In OCL, <u>navigation</u> 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 <u>attribute operations</u>–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 ($ptr_i \times p + j$):

$$nav(Ptr, T, Y) : \begin{cases} \forall i \in [1, z], \forall j \in [1, p] : \\ \quad ptr'_{ij} = ptr_i \times p + j \\ \quad element(y_k, T, ptr'_{ij}), \ k = (i-1)p+j \end{cases} \tag{6.1}$$

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 6.1: Encodings of `self.reference.attribute` for all objects of Figure 1.3 as a table of integer variables

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.

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 6.1 shows the results of the query `self.reference.attribute` using the navigation CP model (6.1) on the instance from Table 5.1. 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 <u>union</u> and <u>append</u>. 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 **??**.

Table 6.1 shows us the result of query `self.reference.attribute` using the navigation model CSP 6.1 on the instance from Table 5.1 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 <u>third variable</u> is a'_{13}, but if $a'_{13} = d$ the <u>third</u>

value is 1000. Our reformulations of other OCL operations, such as <u>union</u> and the sequence operation <u>append</u>, 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 **??**. We will therefore need models to interpret these as the correct collection type.

6.2 NavCSP experimentation

Navigating a model adds a great deal of complexity. The pointer navigation Equation 6.1 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 6.1, 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 6.1 we can see the number of <u>query atoms</u>, 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 6.1 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
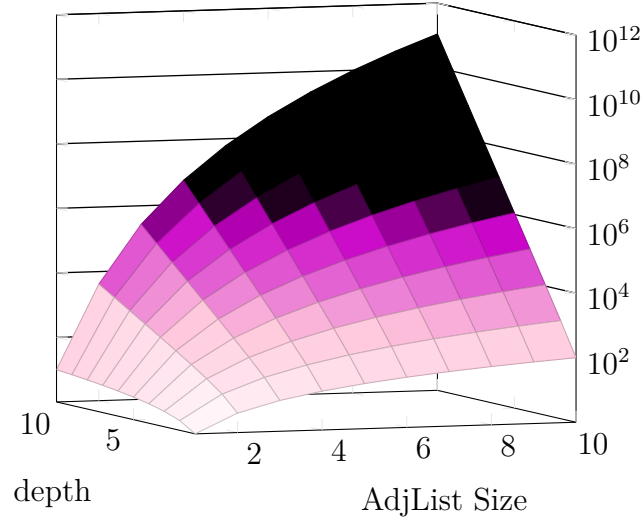
Figure 6.1: Number of query atoms in relation to `AdjList` size and navigation depth [AST]

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:

$$query\text{->}sum(attribute) = Target$$

$$and\ query\text{->}isUnique(attribute)$$

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 6.1, 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.

CONTRIBUTION : OCL CSP
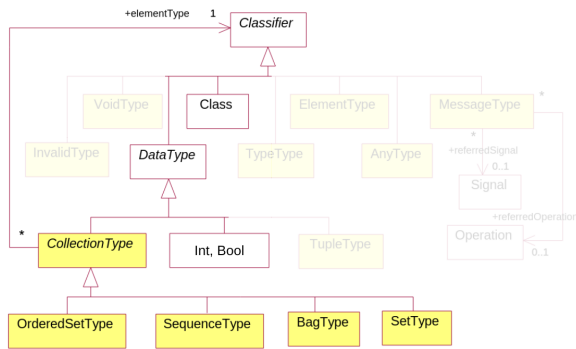
((a)) OCL Type Coverage
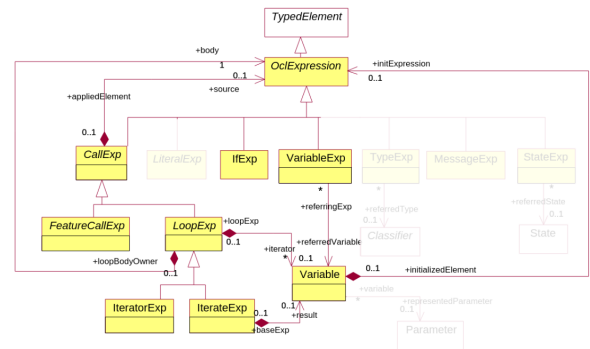
((b)) OCL Expression MMCoverage

The first section covers integer and boolean operations. Collection operations are split across result types: operations of section 2 result in integer, section 3 in booleans, section 4 in collections. Collection type casting operations are in section 5, and typed collection operations are found in the following sections.

src \out	bool	int	element	collection	Sequence	Bag
bool	and					
int	<	+				
collection	=, <> includes, forall	size count	any	select reject	asSequence sortedBy	asBag
Sequence			first at		subSequence union	
Bag						union intersection

7.1 CP Models for OCL Integer and Boolean Operations

OCL is fundamentally a 4-valued logic, with values: True, False, Void, and invalid. We have chosen to not interpret expressions for which a sub-expression is invalid, so we only need 3-logic values.

$\rightarrow$ **MC** OCL# also collapses to 3-valued logic using the empty set to encode void and invalid expressions.

7.1.1 3-valued logic of valid OCL

a	b	a and b	a or b	a xor b	a implies b	not a
2	2	2	2	0	2	0
2	1	1	2	1	1	0
2	0	0	2	2	0	0
1	2	1	2	1	2	1
1	1	1	1	1	1	1
1	0	0	1	1	1	1
0	2	0	2	2	2	2
0	1	0	1	1	1	2
0	0	0	0	0	2	2

In this table we find the semantics of boolean operations in valid OCL. the semantics can be simply model with arithmetic operations on int vars.

- $\neg a \equiv 2 - a$

- $a \wedge b \equiv min(a, b)$

- $a \vee b \equiv max(a, b)$

- $a \rightarrow b \equiv min(2 - a, b)$

- $a \oplus b \equiv min(2 - min(a, b), max(a, b))$

7.1.2 Property encoding to 3-valued logic

In our base encoding for properties, True equals 1, False equals 0, and no information is -1. We could implement a different encoding for booleans, or simply convert from one encoding to the other using an regular expression constraint based on the AFD here.
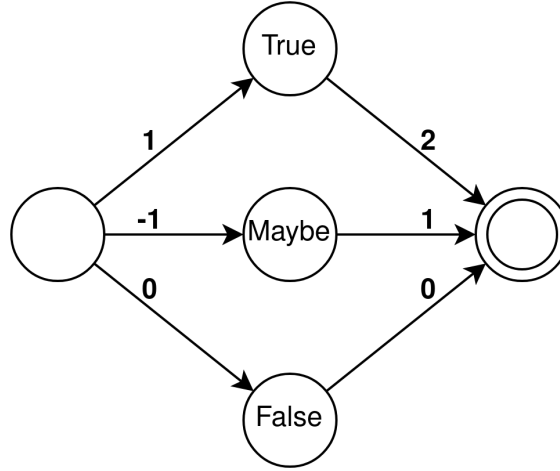
Figure 7.2: Part of the tree for `src.forall(a,b,c| (a<b and b<c) implies (a+b=c+c))`

7.2 CP Models for OCL Collection to Int Operations

7.2.1 size()

Consider the expression `src.size()`, where Formal Definition:

$$size(X,y) \iff y = \sum_{0<i\leq|X|} [\![x_i \neq d]\!] \tag{7.1}$$

Reformulation using Global Constraints:

$$size(X,y) : \begin{cases} count(d, X, y') \\ Y = |X| - y' \end{cases} \tag{7.2}$$

7.2.2 sum()

Definition:

$$sum(X,y) \iff y = \sum_{0<i\leq|X|} x_i * [\![x_i \neq d]\!] \tag{7.3}$$

Reformulation using Global Constraints:

$$sum(X,y) : \begin{cases} count(d, X, s') \\ y = sum(X) - ds' \end{cases} \tag{7.4}$$

7.2.3   count()

Definition:

$$count(X, y, z) \iff z = \sum_{0 < i \leq |X|} [\![x_i = y]\!] \tag{7.5}$$

Reformulation using Global Constraints:

$$count(X, y, z) : \left\{ count(y, X, z) \right. \tag{7.6}$$

7.2.4   max()

Definition:

$$max(X, y) \iff (\forall x \in X, y \geq x) \land (\exists x \in X, x = y) \tag{7.7}$$

Reformulation using Global Constraints:

$$max(X, y) : \left\{ maximum(y, X) \right. \tag{7.8}$$

7.2.5   min()

Definition:

$$min(X, y) \iff (\forall x \in X, y \leq x) \land (\exists x \in X, x = y) \tag{7.9}$$

Reformulation using Global Constraints:

$$min(X, y) : \begin{cases} x'_i = x_i - (2d + 1) * [\![x_i = d]\!] \\ minimum(y, X') \end{cases} \tag{7.10}$$

7.3   CP Models for OCL Collection to Bool Operations

Some operation applied to collections result in a boolean. It is common to know the resulting boolean at compile time, such as at the top of an invariant expression. In the cases where the resulting truth value is known, we often have a simpler model.

7.3.1 equals and different

$$coleq(X, Y) : \begin{cases} x_i = y_i \end{cases} \tag{7.11}$$

$$coleq(X, Y, z) : \begin{cases} b_i \iff (x_i = y_i) \\ product\_ctr(B, " = ", z) \end{cases} \tag{7.12}$$

$$coldiff(X, Y, z) : \begin{cases} b_i \iff (x_i \neq y_i) \\ sum\_ctr(B, " = ", z') \\ z \iff (z' > 0) \end{cases} \tag{7.13}$$

7.3.2 includes(e)

Consider the expression `src.includes(e)`, where `src` is an expression resulting in a collection of objects or integers, and (e) is an (expression resulting in an) object or an integer.

Reformulation using Global Constraints:

$$includes(X, y, z) : \begin{cases} count(y, X, c) \\ z = [\![c > 0]\!] \end{cases} \tag{7.14}$$

7.3.3 excludes(e)

Consider the expression `src.excludes(e)`, where `src` is an expression resulting in a collection of objects or integers, and (e) is an (expression resulting in an) object or an integer.

Reformulation using Global Constraints:

$$excludes(X, y, z) : \begin{cases} count(y, X, c) \\ z = [\![\neg c > 0]\!] \end{cases} \tag{7.15}$$

7.3.4 includesAll(e)

Consider the expression `src.includesAll(e)`, where `src` is an expression resulting in a collection of objects or integers, and (e) is (an expression resulting in) a collection of objects or an integers.

Reformulation using Global Constraints:

$$includesAll(X, Y, z) : \begin{cases} \forall y \in Y, includes(X, y, z') \\ product\_ctr(Z', " = ", z) \end{cases} \tag{7.16}$$

7.3.5 excludesAll(e)

Consider the expression `src.excludesAll(e)`, where `src` is an expression resulting in a collection of objects or integers, and (e) is (an expression resulting in) a collection of objects or an integers.

Reformulation using Global Constraints:

$$
excludesAll(X, Y, z) : \begin{cases} \forall y \in Y, excludes(X, y, z') \\ sum\_ctr(Z', " = ", z'')z = [\![z'' > 0]\!] \end{cases} \tag{7.17}
$$

7.3.6 isEmpty()

Consider the expression `src.isEmpty()`, where `src` is an expression resulting in a collection of objects or integers.

Reformulation using Global Constraints:

$$isEmpty(X, y) : \begin{cases} count(d, X, s) \\ y = [\![s = |X|]\!] \end{cases} \tag{7.18}$$

7.3.7 notEmpty()

Consider the expression `src.notEmpty()`, where `src` is an expression resulting in a collection of objects or integers.

Reformulation using Global Constraints:

$$notEmpty(X, y) : \begin{cases} count(d, X, s) \\ y = [\![s \neq |X|]\!] \end{cases} \tag{7.19}$$

7.3.8 forall(exp)

Consider the expression `src.forall(exp)`, where the source `src` is an expression resulting in a collection of objects or integers and `exp` is an sub-expression of type boolean which applies to a sub-collection of the collection from the source `src`. The expression as a whole resolves to true if all the sub-expressions resolves to true, and false otherwise.

Reformulation using Global Constraints:

$$forall(X, y, exp) : \begin{cases} itpred(X, B, exp) \\ product\_ctr(B, " = ", y) \end{cases} \tag{7.20}$$

Reformulation in OCL when source is a collection of objects, the sub-expression is constant and has one iterator:

```
src.oclType().allInstances().select(exp).includesAll(src)
```

The first part of this expression, `src.oclType().allInstances().select(exp)`, remains constant if `exp` is constant. It can either be interpreted before building the OCL CSP, or with only require one propagation for each of the constraints modeling it.

7.3.9 exists(exp)

Consider the expression `src.exists(exp)`, where the source `src` is an expression resulting in a collection of objects or integers and `exp` is an sub-expression of type boolean which applies to a sub-collection of the collection from the source `src`. The expression as a whole resolves to true if at least one of the sub-expressions resolves to true, and false otherwise.

Reformulation using Global Constraints:

$$exists(X, y, exp) : \begin{cases} itpred(X, B, exp) \\ sum\_ctr(B, " = ", y') \\ y = [\![y' = 1]\!] \end{cases} \qquad (7.21)$$

Reformulation in OCL when source is a collection of objects, the sub-expression is constant and has one iterator:

`src.oclType().allInstances().select(exp).intersect(src.asSet()).size()>0`

The first part of this expression, `src.oclType().allInstances().select(exp)`, remains constant if `exp` is constant. It can either be interpreted before building the OCL CSP, or with only require one propagation for each of the constraints modeling it.

7.3.10   one(exp)

This operation can have at most one iterator variable.

Consider the expression `src.one(exp)`, where the source `src` is an expression resulting in a collection of objects or integers and `exp` is an sub-expression of type boolean which applies to the elements of the collection from the source `src`. The expression as a whole resolves to true if one of the sub-expressions resolves to true, and false otherwise.

Reformulation using Global Constraints:

$$one(X, y, exp) : \{ \tag{7.22}$$

7.3.11 Iterating with predicates

$$itpred(X, B, exp) : \begin{cases} \forall b_{ijk...} \in B, \forall x_i, x_j, x_k, ... \in X \\ \\ b_{ijk...} \iff (((x_i \neq d) \land (x_j \neq d) \land ...) \implies exp(b_{ijk...}, x_i, x_j, x_k, ...)) \end{cases}$$
$$(7.23)$$

Consider the expression `src.forall(a,b,c| (a<b and b<c) implies (a+b=c+c))`.

$$exp(x_i, x_j, x_k) : \begin{cases} (x_i < x_j \land x_j < x_k) \implies (x_i + x_j = x_k + x_k) \end{cases} \quad (7.24)$$
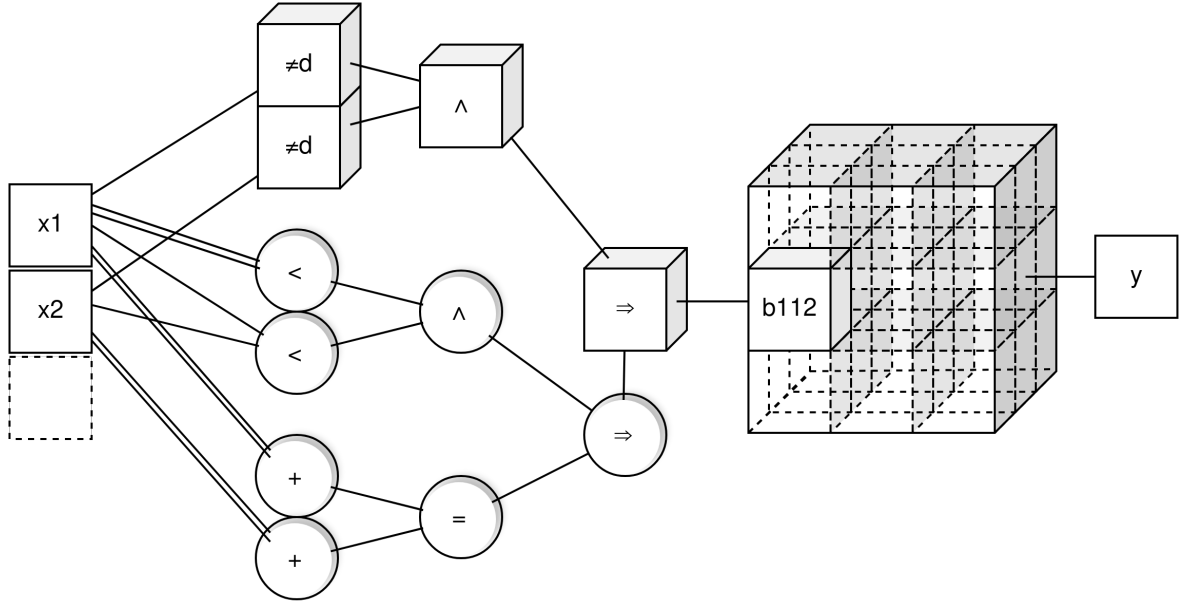
Figure 7.3: Part of the tree for `src.forall(a,b,c| (a<b and b<c) implies (a+b=c+c))`

7.4 CP Models for OCL Collection Operations

7.4.1   select(exp)

This operation can have at most one iterator variable.

Consider the expression `src.select(exp)`, where `src` is an expression resulting in a collection of objects or integers and `exp` is an expression of type boolean.

Reformulation using Global Constraints:

$$select(X, Y, exp) : \left\{ y_i = d - (d - x_i) * [\![exp(x_i)]\!] \right. \tag{7.25}$$

7.4.2 reject(exp)

This operation can have at most one iterator variable.

Consider the expression `src.reject(exp)`, where `src` is an expression resulting in a collection of objects or integers and `exp` is an expression of type boolean.

Reformulation using Global Constraints:

$$reject(X, Y, exp) : \left\{ y_i = d - (d - x_i) * [\![\neg exp(x_i)]\!] \right. \tag{7.26}$$

7.4.3 sortedBy(exp)

This operation can have at most one iterator variable.

Consider the expression `src.sortedBy(exp)`, where exp is an expression of type int.

Reformulation using Global Constraints:

$$sortedBy(X, Y, exp) : \begin{cases} stable\_keysort(\langle K, X \rangle, \langle K', Y \rangle, 1) \\ k_i = exp(x_i), \forall i \in [1, z] \end{cases} \tag{7.27}$$

7.4.4 any()

This operation can have at most one iterator variable.

Consider the expression `src.any(exp)`, where `src` is an expression resulting in a collection of objects or integers and `exp` is an expression of type boolean.

Reformulation using Global Constraints:

$$any(X, y, exp) : \begin{cases} select(X, Y, exp) \\ asBag(Y, Y') \\ y = y'_1 \end{cases} \tag{7.28}$$

7.5 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:

```
1 context Cage inv:
2     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()`.

7.5.1 asBag()

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.

$$asBag(X, Y) : \left\{ sort(X, Y^{\mathrm{rev}}) \right. \tag{7.29}$$

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

7.5.2   asSet()

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 $Y = \{2, 1, d, d\}$.

$$asSet(X, Y) : \begin{cases} sort(X, S^{\mathrm{rev}}) \\ X' = S \parallel \{d\} \\ Y' = Y \parallel \{d\} \\ p_1 = 1 \\ \forall i \in]2, z+1] : \quad p_i = p_{i-1} + [\![x'_{i-1} \neq x'_i]\!] \\ \qquad\qquad\qquad element(x'_i, Y', p_i) \\ \forall i \in [1, z] : \quad\ \ y_i \geq y_{i+1} \end{cases} \tag{7.30}$$

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'.

7.5.3 asSequence()

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:

$$
asSeq_{x2y}(X, Y) : \begin{cases} stable\_keysort(\langle B, X \rangle, \langle B', Y \rangle, 1) \\ b_i = [\![x_i = d]\!], \forall i \in [1, z] \\ b'_i = [\![y_i = d]\!], \forall i \in [1, z] \end{cases} \tag{7.31}
$$

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

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

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.

$$
asSeq_{y2x}(X,Y) : \begin{cases}
\qquad\qquad sort(Y,Y') \\
\text{let } T_y \qquad \text{be the set of tasks such that:} \\
\forall i \in [1,z[\quad : s_i = y'_i + 1 \\
\qquad\qquad d_i = \max(0, y'_{i+1} - y'_i - 1) \\
\qquad\qquad h_i = z \\
\text{let } T_x \qquad \text{be the set of tasks such that:} \\
\forall i \in [1,z] \quad : s_i = x_i \\
\qquad\qquad d_i = 1 \\
\qquad\qquad h_i = 1 \\
\qquad cumulative(T_y \cup T_x, z)
\end{cases} \tag{7.32}
$$

Equation (7.32) 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 7.4, 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 7.4: Visualization of the use of cumulative to filter from Y' to X^{ub}
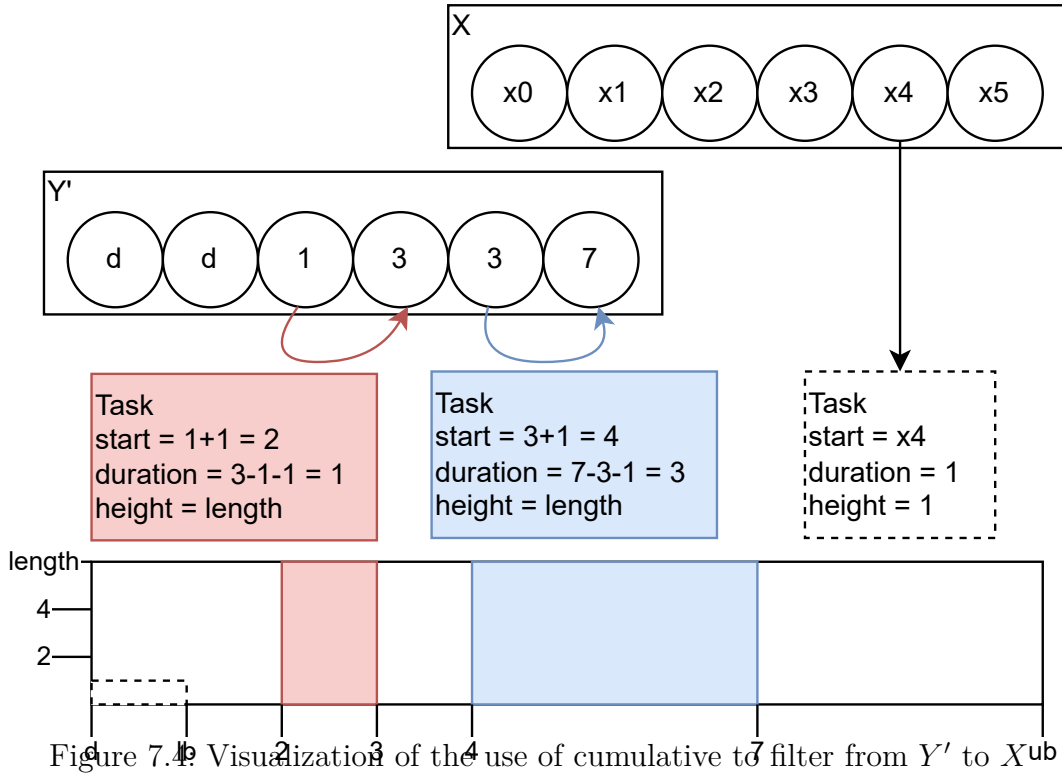
the X to Y and Y to X models give us the complete model for `asSequence`.

$$asSequence(X, Y) : \begin{cases} asSeq_{x2y}(X, Y) \\ asSeq_{y2x}(X, Y) \end{cases} \tag{7.33}$$

81

7.5.4 asOrderedSet()

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:

$$asOrdSet(X, Y) : \begin{cases} stable\_keysort(<X, Y'>, <S, T>, 1) \\ t_1 = s_1 \\ \forall i \in\,]1, z] : t_i = \begin{cases} s_i & \text{if } s_i \neq s_{i-1} \\ d & \text{otherwise} \end{cases} \\ asSequence(Y', Y) \end{cases} \tag{7.34}$$

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 $Y = \{2, 1, 3, d\}$.

7.5.5 Filtering Dummy Values in OCL Collection Operations

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 7.29, 7.30, 7.33, and 7.34. The filtering approach is inspired by the strategy used in Equation (5.2), employing a *regular* constraint over a masked array:

$$dChannel(X, Y) : \begin{cases} regular(S, NFA) \\ \text{where } s_i = [\![s_i' \neq d]\!], i \in [1, z] \\ \text{with } S' = X \parallel c \parallel Y^{\text{rev}} \end{cases} \tag{7.35}$$

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 7.5, ensures that the number of 0s (i.e., dummies) in X is matched by the same number of leading 0s in Y^{rev}.
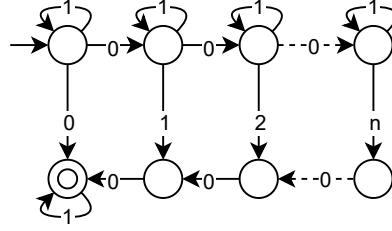

Figure 7.5: Non-Deterministic Finite Automaton accepting strings where Y starts with the same number of 0 found in X.

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

7.6 CP Models for OCL Sequence and Ordered Set Operations

7.6.1 src.prepend(e)

Consider the expression `src.prepend(e)`, where `src` is an expression resulting in a collection of objects or integers and `e` is an expression of resulting in an object or int.

Reformulation using Global Constraints:

$$prepend(X, Y, z) : \left\{ Y = z \parallel X \right. \tag{7.36}$$

7.6.2 src.append(e)

Consider the expression `src.append(e)`, where `src` is an expression resulting in a collection of objects or integers and `e` is an expression of resulting in an object or int.

Reformulation using Global Constraints:

$$append(X, Y, z) : \begin{cases} Y' = X \parallel z \\ asSequence(Y', Y) \end{cases} \qquad (7.37)$$

7.6.3   src.insertAt(i,e)

Consider the expression `src.insertAt(i,e)`, where `src` is an expression resulting in a collection of objects or integers, `i` is an expression resulting in an integer, and `e` is an expression of resulting in an object or int.

Reformulation using Global Constraints:

$$insertAt(X, Y, z, p) : \begin{cases} p'_i = i + [\![i \geq z]\!] \\ element(x_i, Y, p'_i) \\ element(z, Y, p) \end{cases} \tag{7.38}$$
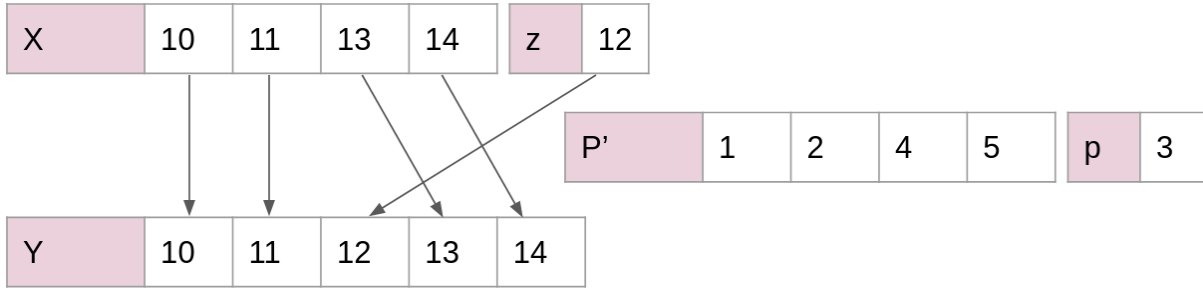

Figure 7.6: example solution for insert at

7.6.4 src.subSequence(s,f)

Consider the expression `src.subSequence(e)`, where `src` is an expression resulting in a collection of objects or integers and `s,f` are expressions of resulting integers giving the start and finish of the sub-sequence in the sequence.

Reformulation using Global Constraints:

$$subSequence(X, Y, s, f) : \begin{cases} X' = d \parallel X \\ p_1 = s + 1 \\ p_i = (s + i) * [\![s + i \le j]\!] \\ element(y_i, X', p_i) \end{cases} \tag{7.39}$$
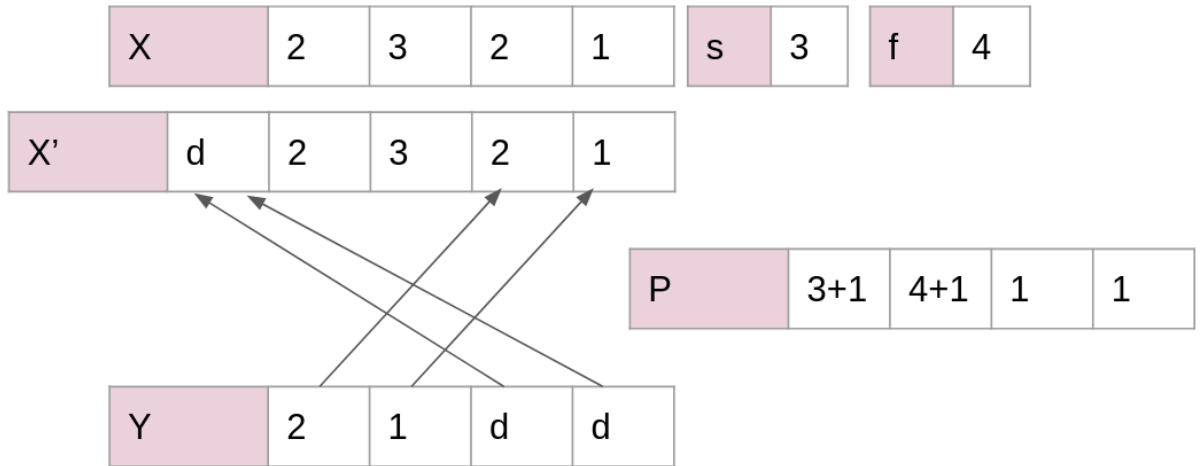

Figure 7.7: example solution for sub-sequence

7.6.5   src.at(p)

Consider the expression `src.at(p)`, where `src` is an expression resulting in a collection of objects or integers and `p` is an expression of resulting in an integer.

Reformulation using Global Constraints:

$$at(X, y, p) : \Big\{ element(y, X, p) \tag{7.40}$$

7.6.6 src.indexOf(p)

Consider the expression `src.indexOf(p)`, where `src` is an expression resulting in a collection of objects or integers and `p` is an expression of resulting in an integer.

Reformulation using Global Constraints:

$$indexOf(X, y, p) : \Big\{ element(y, X, p) \tag{7.41}$$

7.6.7 src.first()

Consider the expression `src.first()`, where `src` is an expression resulting in a collection of objects or integers.

Reformulation using Global Constraints:

$$first(X, y) : \begin{cases} y = x_1 \end{cases} \tag{7.42}$$

7.6.8   src.last()

Consider the expression `src.last()`, where `src` is an expression resulting in a collection of objects or integers.

Reformulation using Global Constraints:

$$last(X, y) : \begin{cases} size(X, p) \\ element(y, X, p) \end{cases} \tag{7.43}$$

7.6.9 src.reverse()

Consider the expression `src.reverse()`, where `src` is an expression resulting in a collection of objects or integers.

Reformulation using Global Constraints:

$$reverse(X, Y) : \begin{cases} size(X, s) \\ p_i = (s - i) * [\![i \leq s]\!] + i * [\![\neg (i \leq s)]\!] \\ element(x_i, Y, p_i) \end{cases} \tag{7.44}$$

7.7 CP Models for OCL Set Operations

7.7.1 union(exp)

Consider the expression `src.union(e)`, where `src` is an expression resulting in a set of objects or integers, and (e) is (an expression resulting in) a set of objects or an integers. Reformulation using Global Constraints:

$$union\_setset(X, Y, Z) : \left\{ X \parallel Y'asSet(Y', Y) \right. \tag{7.45}$$

Consider the expression `src.union(e)`, where `src` is an expression resulting in a set or bag of objects or integers, and (e) is (an expression resulting in) a bag of objects or an integers. Reformulation using Global Constraints:

$$union\_Xbag(X, Y, Z) : \left\{ X \parallel Y'asBag(Y', Y) \right. \tag{7.46}$$

7.7.2 Intersection

Reformulation using Global Constraints:

$$
intersection(X, Y, Z) : \begin{cases}
count(x_i, X, c_i) \\
count(x_i, Y, c'_i) \\
count(x_i, Z, c''_i) \\
c''_i = [\![x_i = d]\!] * a_i + [\![x_i \neq d]\!] * min(c_i, c'_i) \\
a_i \geq max(c_i, c'_i) \\
count(z_i, X, b_i) \\
count(z_i, Y, b'_i) \\
count(z_i, Z, b''_i) \\
b''_i = [\![z_i = d]\!] * a'_i + [\![z_i \neq d]\!] * min(b, b'_i) \\
a'_i \geq max(b_i, b'_i)
\end{cases}
\tag{7.47}
$$

7.7.3 set - set

$$
diff\_set(X, Y, Z) : \begin{cases} count(x_i, Y, s_i) \\ b_i \iff (s_i = 0) \\ z_i' = d - (d - x_i) * [\![b_i]\!] \\ sort(Z', Z^{\text{rev}}) \end{cases} \tag{7.48}
$$

7.7.4 Symmetric Difference

$$symdiff\_set(X,Y,Z) : \begin{cases} diff\_set(X,Y,S) \\ diff\_set(Y,X,T) \\ C = S \parallel T \\ sort(C,Z^{\text{rev}}) \end{cases} \tag{7.49}$$

7.7.5 including

7.7.6 excluding

7.8 CP Models for OCL Bag Operations

7.9 CP Models for Class Functions and Global Functions

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[1]. 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.

---

[1] **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.

Titre : Exploration d'Ensembles de Modèles II

Mot clés : Ingénierie Dirigée par les Modèles, Programmation par Contraintes, Exploration d'Ensembles de Modèles

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: Model Space Exploration II

Keywords: Model Driven Engineering, Constraint Programming, Model Space Exploration

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.