

ÉCOLE DOCTORALE STIM

« SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DES MATÉRIAUX »

Année 2009

N° attribué par la bibliothèque

--	--	--	--	--	--	--	--	--	--

Langages et transformation de modèles en programmation par contraintes

THÈSE DE DOCTORAT

Discipline : INFORMATIQUE

*Présentée
et soutenue publiquement par*

Ricardo SOTO

*le 25 juin 2009 à l'UFR Sciences & Techniques, Université de Nantes,
devant le jury ci-dessous*

Président	:	Éric MONFROY, Professeur	UTFSM, Chili
Rapporteurs	:	François FAGES, Directeur de Recherche	INRIA Rocquencourt, France
		Michel RUEHER, Professeur	Université de Nice, France
Examineurs	:	Arnaud LALLOUET, Professeur	Université de Caen, France
		Éric MONFROY, Professeur	UTFSM, Chili

Directeur de thèse : Pr. Laurent GRANVILLIERS

Laboratoire : LABORATOIRE D'INFORMATIQUE DE NANTES ATLANTIQUE.
UMR CNRS 6241. 2, rue de la Houssinière, BP 92 208 – 44 322 Nantes, CEDEX 3.

N° ED 503-053

LANGAGES ET TRANSFORMATION DE MODÈLES EN
PROGRAMMATION PAR CONTRAINTES

*Languages and Model Transformation in Constraint
Programming*

Ricardo SOTO



favet neptunus eunti

Université de Nantes

Ricardo SOTO

*Langages et transformation de modèles en programmation par
contraintes*

XVIII+154 p.

This document was edited with `these-LINA` v. 2.7 L^AT_EX2e class of the “Association of Young Researchers on Computer Science (L^QG_TN)” from the University of Nantes (available at : <http://login.irin.sciences.univ-nantes.fr/>). This L^AT_EX2e class is under the recommendations of the National Education Ministry of Undergraduate and Graduate Studies ([circulaire n° 05-094 du 29 March 2005](#)) of the [University of Nantes](#) and the Doctoral School of « [Technologies de l'Information et des Matériaux\(ED-STIM\)](#) »

Print : template-theses.tex – 16/11/2009 – 11:35.

Last class review: these-LINA.cls,v 2.7 2006/09/12 17:18:53 mancheron Exp

Abstract

Constraint Programming is an optimization technology that associates rich modeling languages with efficient solving engines. It combines methods from different domains such as artificial intelligence, mathematical programming, and graph theory. A main challenge in this field is to provide high-level languages for facilitating the problem modeling phase. Another important concern is to design robust architectures to map high-level input models to different and efficient solving models. Handling these two concerns is remarkably hard since many aspects have to be investigated, for instance, the expressiveness and the abstraction level of the language as well as the techniques used to transform the high-level model into each of the solver's languages. In this thesis, we propose a new perspective to face those challenges. We introduce a novel constraint programming architecture in which the problem is seen as a set of high-level constrained objects defined through a new modeling language. The model transformation is performed by a model-driven process in which the elements of languages are defined as concepts of a model of models called metamodel. This new architecture allows one to tackle the modeling and the model transformation phases in a higher-level of abstraction and consequently to reduce the inherent complexity behind them.

Keywords: Constraint Programming, Constraint Modeling Languages, Model Transformation

Résumé

La programmation par contraintes est une technologie pour l'optimisation qui associe des langages de modélisation riches avec des moteurs de résolution efficaces. Elle combine des techniques de plusieurs domaines tels que l'intelligence artificielle, la programmation mathématique et la théorie des graphes. Un défi majeur dans ce domaine concerne la définition de langages de haut-niveau pour faciliter la phase de modélisation des problèmes. Un autre aspect important est de concevoir des architectures robustes pour transformer des modèles de haut-niveau et obtenir des modèles exécutables efficaces, tout en visant plusieurs moteurs de résolution. Répondre à ces deux préoccupations est très difficile, car de nombreux aspects doivent être pris en compte, comme par exemple, l'expressivité et le niveau d'abstraction du langage ainsi que les techniques utilisées pour traduire le modèle de haut-niveau dans chacun des langages de résolution. Dans cette thèse, nous proposons une nouvelle perspective pour faire face à ces défis. Nous introduisons une nouvelle architecture pour la programmation par contraintes dans laquelle le problème est défini comme un ensemble d'objets contraints dans un nouveau langage de modélisation haut-niveau. La transformation des modèles est réalisée à l'aide de l'ingénierie des modèles. Les éléments des langages sont alors considérés comme des concepts définis dans un modèle de modèles appelé métamodèle. Cette nouvelle architecture permet d'aborder les phases de modélisation et de transformation de modèles en raisonnant à un niveau d'abstraction supérieur et, par conséquent, de réduire la complexité inhérente à ces deux phases.


Mots-clés: Programmation par contraintes, Langages de modélisation par contraintes, Transformation de modèles

ACM Classification

Categories and Subject Descriptors : D.3.2 [Programming Languages]: Language Classifications—*Constraint and logic languages*; D.3.3 [Programming Languages]: Language Constructs and Features—*Classes and objects, Constraints*; D.2.2 [Software Engineering]: Design Tools and Techniques—*User interfaces*.

A' Stefanie

Remerciements

 Je tiens tout d'abord à remercier vivement mon directeur de thèse Laurent Granvilliers pour sa qualité humaine, sa disponibilité et son énergie débordante tout au long de mes travaux de thèse de doctorat. Pendant ces trois années, j'ai eu le plaisir de rédiger plusieurs articles avec lui, de profiter de sa rigueur scientifique et de son appréciation objective. Cela a ainsi été très enrichissant aussi bien personnellement que professionnellement.

Je souhaite aussi exprimer toute ma gratitude envers les membres de mon jury, en particulier Éric Monfroy pour l'avoir présidé ; Michel Rueher et François Fages, rapporteurs, pour avoir lu mon manuscrit avec soin et fait des remarques pertinentes. Un grand merci à Arnaud Lallouet pour ses questions qui m'ont aidé à orienter mes futurs travaux de recherche.

J'adresse aussi mes sincères remerciements aux membres du département d'informatique de l'Université Pontificale Catholique de Valparaíso. Je remercie particulièrement Broderick Crawford, Éric Monfroy et Carlos Castro qui ont fortement contribué à ma venue en France. Je tiens aussi à remercier Jaime Zavala pour son aide précieuse dans la préparation de mon séjour.

Je voudrais également remercier Raphaël Chenouard pour nos échanges fructueux. Nos contributions sont naturellement présentées dans ce manuscrit. En plus d'être un ami, il m'a aidé et soutenu sans relâche tout au long de ma thèse.

De nombreuses autres personnes ont contribué à faire ces années de thèse à Nantes une période agréable. Anthony, Eduardo, Jim, *gracias* pour votre amitié, votre disponibilité et votre soutien. Thomas et Nico, merci pour votre aide et pour ces innombrables conversations fructueuses. Les membres permanents de l'équipe MEO du LINA ont également toute ma gratitude : Christophe, Fred et Alex, merci. Charlotte et Lorraine merci aussi pour votre amitié et votre amabilité.

Je tiens aussi à remercier mes amis proches : Stéphane, Angel, Wence, Denisse, merci à vous pour tous ces moments agréables qu'on a vécu en France. Je remercie aussi mes amis du Chili qui m'ont toujours soutenu malgré la distance : Alfredo, Negro, Pato et Andrés, je vous serre dans mes bras.

J'adresse une pensée à ma famille, Papa, Maman, Xime et Vero, merci pour votre énorme soutien et ces longues conversations téléphoniques. Je remercie aussi mes beaux-parents pour leur soutien et leurs nombreuses visites qui ont rendu notre séjour en France encore plus plaisant.

Enfin, comment ne pas remercier ma femme sans qui rien n'aurait été possible. Tu sais que ton appui inconditionnel, tes encouragements et ton amour ont été le moteur pour mener à bien ces trois dures années de thèse. Tous ces moments difficiles qu'on a vécu ensemble et la force que t'a eu pour y faire face, tu sais c'est admirable. Stefanie, encore une fois je te remercie, je t'aime et à juste titre je te dédie cette thèse.

Table of Contents

List of Tables	XIII
----------------	------

List of Figures	XV
-----------------	----

—Body of the Dissertation—

1 Introduction	1
1.1 From the Roots of CP to Modern Architectures	2
1.2 Motivations & Contributions	3
1.3 Outline	6

Part I — State-of-the-art

2 Solving Techniques	9
2.1 Constraint Satisfaction Problems	9
2.2 Solving CSPs	9
2.2.1 Basic Search Algorithms	10
2.2.2 Filtering techniques	11
2.2.3 Solving Algorithms	12
2.2.4 Solving numerical CSPs	14
2.2.5 Variable and Value Ordering Heuristics	18
2.3 Summary	18
3 Languages and Systems	19
3.1 Constraint Logic Programming	19
3.2 Libraries	21
3.3 Modeling Languages	23
3.4 Programming Languages	28
3.5 Mathematical Programming	29
3.6 Object-oriented languages	31
3.7 Comparing s-COMMA with related approaches	34
3.8 Summary	35

Part II — The s-COMMA platform

4 Modeling Language & Graphical Artifacts	39
4.1 A Tour of the s-COMMA language	39
4.1.1 The SEND + MORE = MONEY Problem	39
4.1.2 The Packing Squares Problem	40

4.1.3	The Stable Marriage Problem	44
4.1.4	The Social Golfers Problem	46
4.1.5	The Production Problem	48
4.1.6	The Engine Problem	49
4.2	Modeling Features	51
4.2.1	Constants	51
4.2.2	Variable assignments	52
4.2.3	Classes	53
4.2.4	Attributes	54
4.2.5	Constraint Zones	56
4.2.6	Heuristic Orderings & Consistency Techniques	60
4.2.7	Extensibility	63
4.3	The s-COMMA GUI	67
4.4	Summary	71
5	Mapping Models to Solvers	73
5.1	From s-COMMA GUI to s-COMMA	74
5.2	From s-COMMA to Flat s-COMMA	77
5.2.1	Parsing	77
5.2.2	Semantic Checking	83
5.2.3	Refactoring Phase	87
5.3	From Flat s-COMMA to solvers	92
5.3.1	Hand-Written Translators	92
5.3.2	Model-Driven Translators	98
5.3.3	Discussion	107
5.4	Summary	109
 Part III — The Transformation Framework for CP		
6	Overview	113
6.1	The Model-Driven Transformation Framework	113
6.2	A Motivating Example	114
6.3	Summary	117
7	From Source to Target	119
7.1	From source to pivot	119
7.2	Pivot refactoring	122
7.2.1	Refactoring phase	122
7.3	From pivot to target	127
7.4	Transformation process	129
7.4.1	Selecting the refactoring steps.	129
7.5	Experiments	130
7.6	Summary	131
8	Conclusion	133
8.1	s-COMMA	133

8.2 Transformation framework for CP languages 134

8.3 Future research directions 134

—Appendixes—

A Grammars **139**

A.1 s-COMMA Grammar 139

A.2 Flat s-COMMA Grammar 142

Bibliography **145**

Hypertext References **151**

List of Tables

—*Body of the Dissertation*—

Part I — State-of-the-art

3.1	Comparing s-COMMA with five approaches. The meaning of each row is as follows. Object-Orientation: the language provides object-oriented capabilities. GUI: the system offers a graphical interface. Solver-Independence: the architecture is able to perform the problem resolution through different solvers. Mapping tool: the system provides a framework to add new solvers to the platform. Extensibility: the language can be extended for instance to support new global constraints or functions. Solving Options: the definition of heuristics orderings and consistency levels of constraints are allowed.	34
-----	---	----

Part II — The s-COMMA platform

4.1	Binary and unary operators. Higher precedence means lower priority. T represents integer, real, or boolean types. N represents integer or real types.	57
5.1	Translation times (seconds).	108
5.2	Solving times (seconds) and model sizes (number of tokens).	109

Part III — The Transformation Framework for CP

7.1	Times of complete transformation chains.	131
7.2	Time of complete transformation chains of the n-queens problem.	131

—*Appendixes*—

List of Figures

—Body of the Dissertation—

1.1	A solution of the 8-queens problem.	1
1.2	The transformation process in \mathbf{s} -COMMA.	4
1.3	The transformation framework for CP.	5

Part I — State-of-the-art

2.1	Solving the 4-queens problem using GT.	10
2.2	Solving the 4-queens problem using BT.	11
2.3	Enforcing arc consistency.	12
2.4	Solving the 4-queens problem using FC.	13
2.5	Solving the 4-queens problem using MAC.	13
2.6	Enforcing hull consistency.	16
2.7	The CSP $\mathcal{P} = \langle \langle x \rangle, \langle D_x \in [-2, 2] \rangle, \langle x^2 < 2 \rangle \rangle$	17
2.8	Enforcing box consistency.	17
3.1	An ECL ⁱ PS ^e model of the n-queens problem.	21
3.2	A Gecode/J model of the n-queens problem.	24
3.3	A Gecode/J model of the n-queens problem using global constraints.	25
3.4	A MiniZinc model of the n-queens problem.	28
3.5	An Alma-0 model of the n-queens problem.	30
3.6	An AMPL model of the n-queens problem.	31
3.7	A \mathbf{s} -COMMA model of the n-queens problem.	34

Part II — The \mathbf{s} -COMMA platform

4.1	A \mathbf{s} -COMMA model of the cryptarithmic puzzle $SEND + MORE = MONEY$. . .	40
4.2	A \mathbf{s} -COMMA model of the packing squares problem.	41
4.3	An object-oriented \mathbf{s} -COMMA model of the packing squares problem.	43
4.4	Data file of the stable marriage problem.	44
4.5	A \mathbf{s} -COMMA model of the stable marriage problem.	45
4.6	Data file of the social golfers problem.	46
4.7	Model file of the social golfers problem.	47
4.8	A \mathbf{s} -COMMA model of the production problem.	48
4.9	Data file of the production problem.	49
4.10	The Engine Problem.	49
4.11	A \mathbf{s} -COMMA model of the engine problem.	50
4.12	The CylSystem class of the engine model.	50

4.13	The <code>Injection</code> class of the engine model.	51
4.14	Constants.	52
4.15	Variable assignments.	52
4.16	Variable assignments guided by indexes.	52
4.17	Composition and inheritance.	53
4.18	Importing models.	53
4.19	Decision variables.	54
4.20	Decision variables, domains and enumerated domains.	54
4.21	Sets.	55
4.22	Objects and constrained objects.	55
4.23	A constraint zone.	56
4.24	Constraint zone overriding.	56
4.25	<code>forall</code> loops.	58
4.26	Nested <code>forall</code> loops.	58
4.27	The <code>sum</code> loop.	58
4.28	Conditionals.	59
4.29	Optimization statement.	59
4.30	A compatibility constraint.	60
4.31	The industrial mixer problem.	61
4.32	Value and variable orderings.	61
4.33	Consistency level.	62
4.34	Ordering heuristics & consistency level.	62
4.35	Adding constraints to <code>s-COMMA</code>	63
4.36	Removing symmetries from the social golfers problem.	64
4.37	The Sudoku problem.	65
4.38	Adding new functions.	65
4.39	Using the new functions in the Sudoku problem.	66
4.40	Adding new heuristic orderings and consistency levels.	66
4.41	The tuned mixer class.	67
4.42	Class and data artifacts.	67
4.43	The stable marriage problem on the <code>s-COMMA</code> GUI.	68
4.44	Attributes on the <code>s-COMMA</code> GUI.	69
4.45	Constraints on the <code>s-COMMA</code> GUI.	70
4.46	Data files on the <code>s-COMMA</code> GUI.	70
4.47	Some shortcuts of the <code>s-COMMA</code> GUI.	71
5.1	The <code>s-COMMA</code> architecture.	73
5.2	<code>s-COMMA</code> GUI Java packages.	74
5.3	The <code>AttributeDialog</code> class.	75
5.4	The <code>ClassArtifact</code> class.	75
5.5	The <code>SCommaClass</code> class.	76
5.6	The <code>getCode</code> method.	76
5.7	Tokens and rules in the ANTLR lexer of <code>s-COMMA</code>	78
5.8	The lexer rule to define numbers.	79
5.9	Three parser rules in ANTLR.	79
5.10	Introducing a proper tree node.	80

5.11	Parser rules of <code>s-COMMA</code> .	80
5.12	Parser rules of <code>s-COMMA</code> .	81
5.13	The rule to recognize expressions.	82
5.14	A syntactic error.	83
5.15	Tree walker of <code>s-COMMA</code> .	84
5.16	A Java procedure to check class redeclarations.	84
5.17	A semantic error.	84
5.18	Two <code>s-COMMA</code> classes.	85
5.19	The rule to check attributes in the second pass.	85
5.20	The rule to check constraints in the second pass.	86
5.21	Loop unrolling.	87
5.22	Enumeration substitution.	88
5.23	Composition flattening.	88
5.24	Flattening arrays containing objects.	88
5.25	Conditional removal.	89
5.26	Conditional evaluation.	89
5.27	Compatibility removal.	89
5.28	Expression evaluation.	90
5.29	A Flat <code>s-COMMA</code> model of the stable marriage problem.	91
5.30	The mapping tool.	92
5.31	Tokens and the <code>IDENT</code> rule in the ANTLR lexer of Flat <code>s-COMMA</code> .	93
5.32	Parser rules of Flat <code>s-COMMA</code> .	93
5.33	Parser rules of Flat <code>s-COMMA</code> .	94
5.34	Tree walker of Flat <code>s-COMMA</code> .	94
5.35	The initial procedure of the main Java class of the Gecode/J translator.	95
5.36	Code generation of the Gecode/J constructor.	95
5.37	Code generation of Gecode/J variables.	96
5.38	The tree walker for the code generation of constraints.	96
5.39	Two procedures for the code generation of constraints.	97
5.40	A Gecode/J model of the stable marriage problem.	97
5.41	A general MDA for model transformation.	98
5.42	Model-driven translation in <code>s-COMMA</code> .	98
5.43	An extract of the KM3 file of Flat <code>s-COMMA</code> .	99
5.44	Constraints in the KM3 file of Flat <code>s-COMMA</code> .	100
5.45	Operands in the KM3 file of Flat <code>s-COMMA</code> .	101
5.46	ATL rules for the Flat <code>s-COMMA</code> to Gecode/J transformation.	101
5.47	ATL rules for the Flat <code>s-COMMA</code> to Gecode/J transformation.	102
5.48	ATL rules for decomposing matrices containing sets.	103
5.49	ATL helper to generate a Gecode/J vector.	104
5.50	ATL helper to generate an addition.	105
5.51	Three templates of the TCS file of Flat <code>s-COMMA</code> .	105
5.52	The model-driven transformation process on the example of Flat <code>s-COMMA</code> (FsC) to Gecode/J.	106
5.53	Direct code generation.	107

Part III — The Transformation Framework for CP

6.1	The transformation framework.	114
6.2	A \mathfrak{s} -COMMA model of the social golfers problem.	115
6.3	The social golfers problem expressed in ECL^iPS^e	116
7.1	Three classes of the KM3 file of \mathfrak{s} -COMMA.	119
7.2	Attributes and variables in the KM3.	120
7.3	Constraint zones and statements in the KM3.	120
7.4	Some templates of the TCS file of \mathfrak{s} -COMMA.	121
7.5	Two ATL rules for a transformation from \mathfrak{s} -COMMA to pivot.	122
7.6	A fragment of the pivot metamodel.	123
7.7	An example of transformation rule.	123
7.8	The composition flattening transformation rule.	125
7.9	Composition flattening on the social golfers problem.	125
7.10	The enumeration substitution transformation rule.	126
7.11	Enumeration substitution on the social golfers problem.	126
7.12	The forall unrolling transformation rule.	126
7.13	Auxiliary variable insertion transformation rules.	127
7.14	Auxiliary variable insertion process.	127
7.15	A fragment of the ECL^iPS^e metamodel.	128
7.16	Five templates of the TCS file of ECL^iPS^e	128
7.17	The transformation process on the example of \mathfrak{s} -COMMA to ECL^iPS^e	129
7.18	An Ant script for selecting transformations.	130

—Appendixes—

CHAPTER 1

Introduction

Constraint Programming (CP) is known to be an efficient software technology for solving combinatorial and continuous problems. Under this framework, problems are formulated as Constraint Satisfaction Problems (CSP). Such a representation describes a problem in terms of variables and constraints. Variables are unknowns lying in a set of values called domain, and constraints are relations among these variables restricting the values that they can adopt. The goal is to find a variable-value assignment that satisfies the whole set of constraints.

As an example, let us consider the 8-queens problem, which consists in placing eight chess queens on a 8x8 chessboard such that none of them is able to capture any other using the standard chess queen's moves. A solution requires that no two queens share the same row, column, or diagonal.

Eight variables can be identified, Q_1, \dots, Q_8 , where Q_i denotes the row position of the queen placed in the i th column of the chessboard. The domain for each of these variables is given by the integer interval domain $[1, 8]$, which represents the potential positions of the queens on the rows of the chessboard. Once the variables have been identified with their corresponding domains, we can formulate the constraints of the problem as the following inequalities for $i \in [1, 7]$ and $j \in [i + 1, 8]$:

- To avoid that two queens are placed in the same row: $Q_i \neq Q_j$.
- To avoid that two queens are placed in the same South-West–North-East diagonal: $Q_i + i \neq Q_j + j$.
- To avoid that two queens are placed in the same North-West–South-East diagonal: $Q_i - i \neq Q_j - j$.

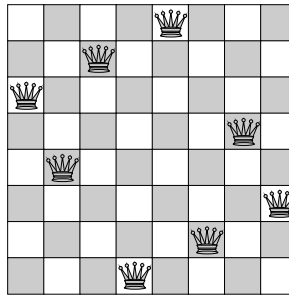


Figure 1.1 – A solution of the 8-queens problem.

A solution to this problem is depicted in Figure 1.1, it corresponds to the sequence (3,5,2,8,1,7,4,6), the first queen from the left is placed on the third row from the top, the second queen is placed on the fifth row, the third queen is placed on the second one and so on.

1.1 From the Roots of CP to Modern Architectures

The resolution process of CSPs involves two main aspects. A language to express the problem, and algorithms to perform the solving process. In some sense, this integration was firstly performed around 1963 by Ivan Sutherland, who developed a language for specifying constraints on drawings [Sut63]. After this landmark, a natural separation occurred between these two aspects, and the research work was divided [FM06] into two main streams: the language stream and the algorithm stream.

In the language stream, the notion of constraint was incorporated in several programming languages and systems. For instance, around 1967, Elcock developed a declarative language called Absys [Elc90] based on the manipulation of equational constraints. Burstall employed a form of constraint in a program for solving cryptarithmic puzzles [Bur69]. Then, the advances in the programming languages field allowed to incorporate constraints in different paradigms. For instance, Borning combined objects, constraints, and visual environments in the ThingLab simulation laboratory [Bor81]. Constraint were also mixed with logic programming in the form of constraint logic programming (CLP) [JMSY92]. Some examples are Prolog III [Col90], CLP(\mathbb{R}) [JL87], and CHIP [Van89].

In the algorithm stream, the research work was heavily influenced by the artificial intelligence (AI) domain. The focus was to develop more efficient search and heuristic methods. For example, Waltz introduced in the mid-1970s a filtering algorithm to accelerate the resolution process of the scene labeling CSP [Wal75]. Then, Montanari developed other kind of filtering mechanisms, technically called local consistencies, and a general framework for reasoning about constraints [Mon74] was established. The algorithm stream followed growing and new AI communities working around the concept of “reasoning” were developed such as constraint-based reasoning [FM92] and case-based reasoning [AP94].

The separation of both streams continued until the early 1990s when a group of scientists from different fields attempt to reintegrate them to create a new single paradigm called “constraint programming”. The idea was to create a new technology under the following principle: The user states the constraints and a general purpose constraint satisfaction engine solve them. From those days many constraint programming systems have been developed, always integrating the two aforesaid streams and sometimes involving other approaches, for example ECLⁱPS^e [WNS97] and GNU Prolog [DC00] for constraint logic programming or Oz [SSW94], a multiparadigm language combining constraint-based inference and distributed computing. Also, several libraries have been introduced, generally built on top of well-known programming languages such as ILOG Solver [Pug94] and Gecode [ST06] using C++; and CHOCO [vzw] running under Java.

At the beginning of the current decade, an important issue arose. The community realized that just a reduced number of experts mastered the CP technology. One of the main reasons was the complexity of the CP’s usage. The fruitful use of existing tools implied to have a considerable level of CP expertise, for instance to deal with encoding aspects of host languages or to tune search strategies to perform efficient solving processes, in conclusion, the modeling concerns to state problems were not enough. This important issue encouraged the creation of the so-called modeling languages, such as OPL [Van99], where a more “user-understandable” language is given. The user deals with a higher-level language without needing to overcome the encoding aspects of a host language or to specify a search strategy.

Three years ago, constraint programming systems evolved and the last generation of CP architectures has been proposed, some examples are Essence [FGJ⁺07], Zinc [RGMW07] and

MiniZinc [NSB⁺07]. This new architecture considers three layers, a modeling language on the top, a set of CP systems on the bottom and a mapping tool on the middle. The modeling language allows users to state problems in a high-level of abstraction. The mapping system takes this model and translates it to one of the underlying CP systems, which calculates the solution. These CP systems, generically called solvers, normally have a lower level of abstraction compared to the modeling language. An interesting feature of this architecture is the capability of processing one model with different solvers. This feature is useful for experimentation tasks, considering that there exists many kind of models and there is no solver having the best resolution for all.

1.2 Motivations & Contributions

The research of high-level languages and flexible architectures for model transformation is an important challenge in the CP field. The task is hard since many aspects must be investigated. The definition of high-level languages requires to consider several concerns. For instance, providing support for a wide range of problems depends on the definition of suitable levels of expressiveness. The design of elegant modeling styles is essential for getting concise and clear models. Extensibility mechanisms are important to enlarge the expressiveness of languages, and tuning capabilities are useful for achieving efficient solving processes. Software features to improve reuse and model management are desirable particularly for handling larger problems. Building flexible and efficient architectures for model transformation involves the study of additional concerns. For instance, the correct selection of tools and techniques is a key decision to implement flexible and modular mappings. Another important aspect is the openness of this architecture, i.e. it must be possible to plug new solvers to the underlying layer.

The development of languages and systems for CP is a long story. Various evolutions, improvements and combinations of previous approaches can be regarded. However, most of the aforementioned aspects are recent and they have not been studied enough. In this thesis, we present a new vision for handling those concerns. Software engineering practices are complemented with several innovations to provide high-level problem modeling. Powerful techniques from the model engineering world ensure modular and flexible mappings toward the solver resolution. This new approach consists of three main components: the `s-COMMA` language, the `s-COMMA` GUI, and a middle tool for transforming models to solver programs.

`s-COMMA` is the modeling language of the architecture [SG07b]. Its design is based on the experience of the software engineering world. Features from object-oriented languages such as modularity, composition, and inheritance are introduced to support reuse and the management of constraint models. The core of the language is a combination of a high-level object-oriented language with a constraint language. The constraint language includes usual data structures, control operations, and first-order logic to define constraint-based formulas. The object-oriented part of the language has been simplified to avoid the complex encoding concerns present in programming languages. As a consequence, the language is able to elegantly capture the structure of problems in single objects. This new modeling style is just the first innovation of our approach. The second innovation of `s-COMMA` concerns its tuning capabilities. A simple formalism is provided to perform customized solving processes [SG08b]. This formalism is unique in object-oriented constraint modeling and it profits of the object-oriented style to configure solving options in multiple manners. The third innovation of our approach is about extensibility. An extension mechanism is provided to adapt the modeling language to further upgrades of the

solver layer. This mechanism allows us to add new functionalities such as new global constraints, new functions, or new tuning options [SG07a].

The \mathfrak{s} -COMMA GUI [CGS08] is the associated authoring tool of the architecture. The visual language provided can be seen as the graphical representation of the \mathfrak{s} -COMMA language. The design of this new language has also been influenced by software engineering practices. In fact, the object-oriented style of \mathfrak{s} -COMMA has been naturally represented by means of an extension of the UML class artifact. This new language is the fourth innovation of the architecture, being the support of a visual and a more concise perception of models.

The mapping tool is the third component of the architecture. This tool is responsible for transforming an input model into an executable solver program. A main challenge must be faced at this stage. The transformations must be flexible and easy to implement in order to permit the integration of new solvers to the platform. This issue is evidently a model transformation concern. Accordingly, as the fifth innovation of the architecture, the mapping tool has been enhanced with the incorporation of a model-driven architecture [CGS08]. This approach provides proper metamodeling and transformation techniques to build flexible mapping tools.

The transformation process performed in \mathfrak{s} -COMMA is similar to that of Zinc or Rules2CP [FM08], except for the transformation of graphical artifacts. In \mathfrak{s} -COMMA we consider a three-step transformation phase (see Figure 1.2). Firstly, graphical artifacts are transformed to the corresponding \mathfrak{s} -COMMA model. This model must then be transformed to the Flat \mathfrak{s} -COMMA [SG08a] intermediate language to be closer, in terms of language constructs, from the solver language. In this process, several high-level constructs –not supported at the solver level– are transformed to simpler ones. For instance, loops are unrolled, conditionals are refactored, or object-oriented compositions are flattened. This allows one to simplify both the translation process and the integration of new solver transformations. Finally, this intermediate model is directly transformed to the executable solver program.

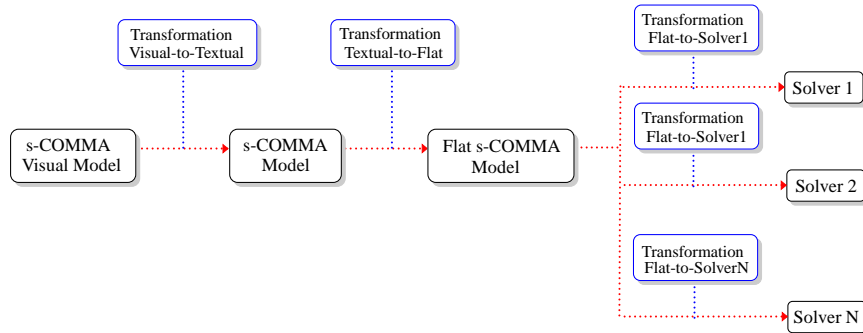


Figure 1.2 – The transformation process in \mathfrak{s} -COMMA.

The \mathfrak{s} -COMMA platform is the result of an investigation of several important concerns in the development of modern architectures for CP. Many innovations and benefits can be found in this new approach. A high-level language is provided to smoothly capture the structure of problems. An accurate graphical representation of this language is given to achieve a more concise representation of problems. As opposed to previous approaches, the expressiveness of \mathfrak{s} -COMMA can be extended to support new functionalities. The use of tuning mechanisms in object-oriented modeling is another innovation of \mathfrak{s} -COMMA, it permits performing customized solving processes. The platform also provides support for experimentation tasks, as the possibility of processing

a same model with different solvers is present. Finally, the structure of the architecture can be updated. New solvers can be connected to the platform in order to enlarge the experimentation possibilities.

A second work is presented in this thesis as well [CGS09]. This new approach can be seen as an improvement of the solver-independent architecture. We introduce a new framework allowing to define bridges between different modeling and solver languages. The main motivation behind this work concerns the fact that defining a universal modeling language¹ for CP is hard, and the users usually have their own preferences. Therefore, we believe that a transformation framework to define mappings between many modeling languages and many solvers would be desirable. This new approach involves important advantages. For instance, users may choose their favorite modeling language and the best known solving technology for a given problem provided that the transformation between languages is implemented. Additionally, it may be easy to create a collection of benchmarks for a given language from different source languages. This feature may speed up prototyping of one solver, avoiding the rewriting of problems in its modeling language.

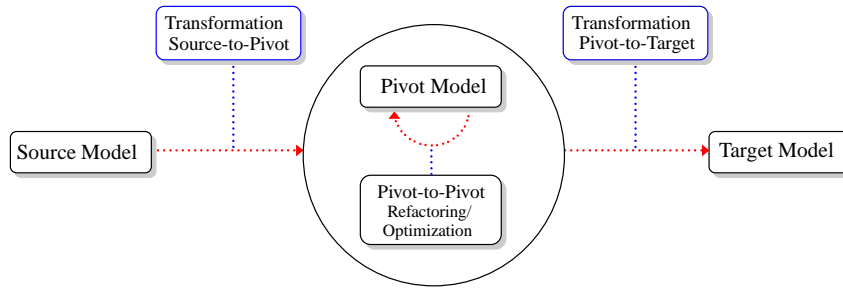


Figure 1.3 – The transformation framework for CP.

We implement this transformation framework by means of an architecture completely built using a model-driven approach. A generic and flexible pivot model (intermediate model) has been introduced, to which different languages can be mapped. This architecture allows one to perform a complete transformation in three main steps: from the source to the pivot model, refactoring/optimization of the pivot model, and from the pivot to the target model (see Figure 1.3). Refactoring and optimization steps are always implemented over the pivot so as to guarantee independence from external languages. This refining phase is comparable to the one performed from `s-COMMA` to `Flat s-COMMA`, but more flexible since the process is not fixed, i.e. it is possible to select the refining steps to be applied in a transformation. For instance, if loops are supported at the target level it is useless to unroll them. This feature allows one to make use of the constructs provided at the target level and therefore to reduce the differences (in terms of model structure) between the source and the target model.

The work done on this transformation framework can be seen as a natural continuation of the architecture implemented in `s-COMMA`. Two main innovations can be observed with respect to previous work. The possibility of using different modeling languages as the source of a transformation, and the possibility of selecting the appropriate refining phases in a transformation. The first feature speeds up prototyping of solvers and motivates model sharing, and the second one enables users to generate models targeting a desired solving technology.

¹ The definition of a standard language has been established as an important future challenge at the CP 2006 conference. At the CP 2007 conference, MiniZinc has been proposed as a standard language.

1.3 Outline

This thesis is composed of three main parts: Part one is devoted to the state of the art and it is divided into two chapters. The first chapter gives an overview of techniques developed for solving CSPs. We include the main procedures and we illustrate them by means of several examples. The second chapter gives a summary of languages and systems for modeling and solving CSPs. The spectrum is very wide, from programming to modeling languages and from logic to object-oriented paradigms. We also introduce various models of the n-queens problem in order to contrast the different approaches.

Part two presents the `s-COMMA` platform. The first chapter of this part is devoted to the modeling features of `s-COMMA`. A tour of the `s-COMMA` language is firstly given, followed by a detailed illustration of the modeling constructs supported. The chapter ends with a presentation of the `s-COMMA` GUI and its graphical artifacts. The second chapter of this part focuses on the whole transformation chain, from graphical artifacts to solver models. We present the main elements involved in the system (e.g. parsers, metamodels and transformation rules) and the tools and techniques for implementing them.

The second approach we developed is presented in Part three. The first chapter presents the architecture of the transformation framework and motivates its implementation through an example concerning several transformation issues. The following chapter focuses on the implementation of the main parts of the transformation framework. We explain the structure of the architecture and the transformation process from source to target models. The thesis ends with the conclusion and the future work.

PART I

State-of-the-art

CHAPTER 2

Solving Techniques

Constraint satisfaction involves various solving approaches, which are mainly based on artificial intelligence. In this chapter, we give an overview of these approaches. We firstly introduce some basic notations and then we present the foundations of techniques to solve CSPs. We consider the basic search algorithms as well as more advanced procedures that involve filtering mechanisms.

2.1 Constraint Satisfaction Problems

Definition 2.1 (Constraint Satisfaction Problem). A Constraint Satisfaction Problem \mathcal{P} is defined by a triple $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where:

- \mathcal{X} is a set of variables $\{x_1, x_2, \dots, x_n\}$.
- \mathcal{D} is a set of domains $\{d_1, d_2, \dots, d_n\}$ such that d_i is the domain of x_i defined as a subset of some set E_i called universe, for $i = 1, \dots, n$.
- \mathcal{C} is a set of constraints $\{c_1, c_2, \dots, c_m\}$ such that c_j is a relation over a set of variables $\{x_{j_1}, \dots, x_{j_{n_j}}\}$ called its scope, defined as the set $\Gamma_j \subseteq d_{j_1} \times \dots \times d_{j_{n_j}}$, for $j = 1, \dots, m$. $c_j(x_{j_1}, \dots, x_{j_{n_j}})$ is also used to denote a constraint c_j over its scope $x_{j_1}, \dots, x_{j_{n_j}}$.

A solution to a CSP is an assignment $\{x_1 \rightarrow a_1, \dots, x_n \rightarrow a_n\}$ such that:

- $a_i \in d_i$ for $i = 1, \dots, n$.
- $(a_{j_1}, \dots, a_{j_{n_j}}) \in \Gamma_j$, for $j = 1, \dots, m$.

If the CSP has a solution we say that it is consistent; otherwise we say that it is inconsistent.

There exist different classes of CSPs, for instance:

- A finite domain CSP corresponds to a CSP in which each domain is a finite subset of \mathbb{Z} (universe of variables). The constraints are generally defined as arithmetic, logic, or set expressions.
- A numerical CSP corresponds to a CSP in which each domain is an interval containing values from \mathbb{R} . The constraints are generally defined as linear and non linear equations or inequalities.

2.2 Solving CSPs

Solving CSPs requires to explore the space of potential solutions. Such an exploration can be performed using a tree data structure, where the root is the initial problem and each node corresponds to a sub-problem. The tree is built by splitting the domain of variables to obtain

those sub-problems. There exist different strategies for traversing the tree such as deep-first search and breadth-first search, and also various algorithms for generating and exploring the tree. The most basic one is the Generate and Test algorithm.

2.2.1 Basic Search Algorithms

Generate and Test

The Generate and Test (GT) algorithm consists in generating a potential solution and checking whether it satisfies all the constraints. This process is done by generating a tree that represents the Cartesian product of domains.

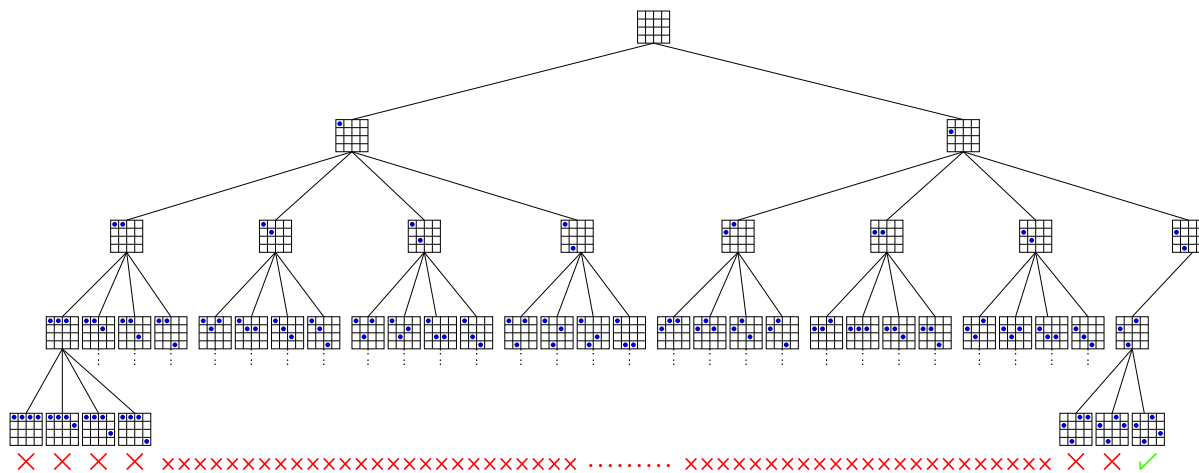


Figure 2.1 – Solving the 4-queens problem using GT.

Let us illustrate the GT process by means of the 4-queens problem, a smaller version of the 8-queens problem introduced in Chapter 1. Figure 2.1 depicts an extract of the process done by the GT algorithm to reach a solution for this problem¹. The figure shows that constraints are checked only when all the variables of the problem have been instantiated. Thus, failures cannot be detected as soon as only the variables relevant to a constraint have been instantiated. This approach is simple to implement, however the searching cost is too expensive.

Backtracking

Backtracking (BT) [Luc91, GB65] is another approach for the exploration/generation of the search tree. In this method the potential solutions are generated incrementally by repeatedly choosing a value for another variable and as soon as all the variables involved in a constraint are instantiated, the constraint is checked. Thus, if a partial solution violates a constraint, the algorithm returns to the most recently instantiated variable that still has alternatives available (to achieve a solution), eliminating as a consequence the conflicting subspace.

Figure 2.2 depicts the search process performed by the BT procedure on the 4-queens problem. The figure shows that BT is able to detect failures as soon as two variables are instantiated (at

¹Figures 2.1, 2.2, 2.4 and 2.5 have been adapted from [W4W].

the middle level of the tree), that is much earlier than in the GT approach. Despite this, the BT approach is not able to detect failures before assigning the values to all the variables involved in a conflicting constraint. This problem can be addressed by using filtering techniques.

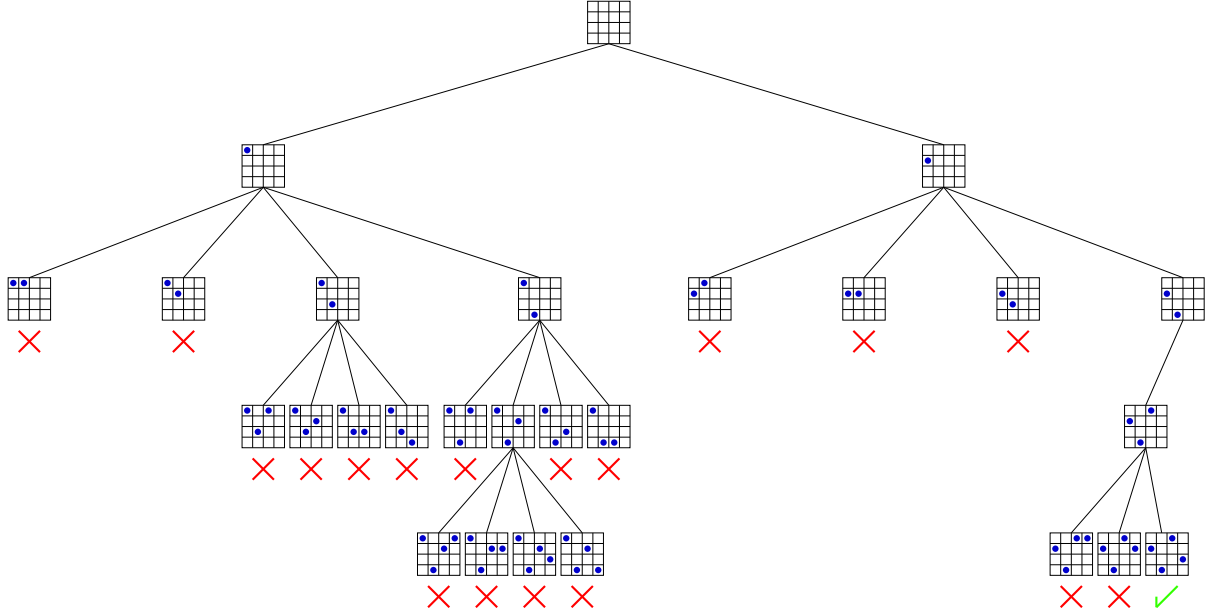


Figure 2.2 – Solving the 4-queens problem using BT.

2.2.2 Filtering techniques

The performance of basic search algorithms can be improved by reducing the variables' domains of each generated sub-problem. This is possible by calculating a consistency property on the constraints. The idea is to enforce such a property on each sub-problem by using a constraint propagation algorithm. The most used notion of consistency is the arc consistency [Mac77].

Definition 2.2 (Arc Consistency). *Let $c_j(x_{j_1}, \dots, x_{j_{n_j}})$ be a constraint and let k be an integer, $k \in \{j_1, \dots, j_{n_j}\}$. We say that c_j is arc consistent wrt. x_k iff:*

$$\forall a_k \in d_k : \exists a_{j_1} \in d_{j_1}, \dots, \exists a_{k-1} \in d_{k-1}, \exists a_{k+1} \in d_{k+1}, \dots, \exists a_{j_{n_j}} \in d_{j_{n_j}} \text{ such that } (a_{j_1}, \dots, a_{j_{n_j}}) \in \Gamma_j$$

A constraint is said to be arc consistent if it is arc consistent wrt. to all its variables. A CSP is said to be arc consistent if all its constraints are arc consistent.

Arc consistency allows one to verify that for each value of a domain it exists at least one value in the domain of the other variables such that the constraint involved is satisfied. This property can be calculated by a constraint propagation algorithm in order to reduce the domains of variables. As an example, let us consider the placement of the first queen on the cell (1,1) of the chessboard (see Figure 2.3). Three cells have been eliminated to make the sub-problem arc consistent. The value 1 has been removed from the domain of Q_2 since there is no corresponding

value in the domain of Q_1 such that the constraint $Q_1 \neq Q_2$ is satisfied (considering that the domain of Q_1 became $\{1\}$ after the instantiation). In the same way, the value 1 has been removed from the domain of Q_3 and Q_4 . This process is done for each constraint of the problem allowing to avoid several potential wrong instantiations. Let us note that there exist different algorithms to enforce arc consistency, for instance AC-3 [Mac77], AC-4 [MH86] and AC-5 [VDT92].

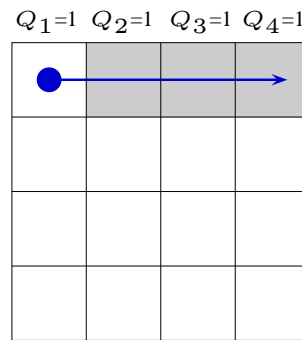


Figure 2.3 – Enforcing arc consistency.

There also exist stronger consistency notions, which may eliminate a larger number of conflicting values from domains, but at higher cost in terms of computations. Some examples are the path consistency [Mon74] and the k -consistency [Fre78].

2.2.3 Solving Algorithms

A search algorithm can be combined with constraint propagation to obtain a more efficient solving procedure. The most common approach is to combine the BT algorithm with the arc consistency. Some examples are Forward Checking (FC) and Maintaining Arc Consistency (MAC).

Forward Checking

Forward checking [McG79] is able to prevent future conflicts by performing arc consistency on the not yet instantiated variables. This is done by removing temporarily the values of the variables that will further cause a conflict with the current variable assignment. Hence, the algorithm immediately detects that the current partial solution is inconsistent and consequently the search space can be pruned earlier than using simple backtracking.

Figure 2.4 illustrates this process: values from domains are removed since the second level of the tree. Once a queen is stated, its future conflicting values are temporarily removed, for instance the queen stated at the position (1,1) removes all values corresponding to the first row and the NW-SE diagonal. Then, in the left subtree, the second queen is placed at the position (3,2) which is immediately set as inconsistent since it does not leave available place for the third queen. The propagation follows for every queen on the chessboard, allowing to avoid most of wrong instantiations done by the BT approach.

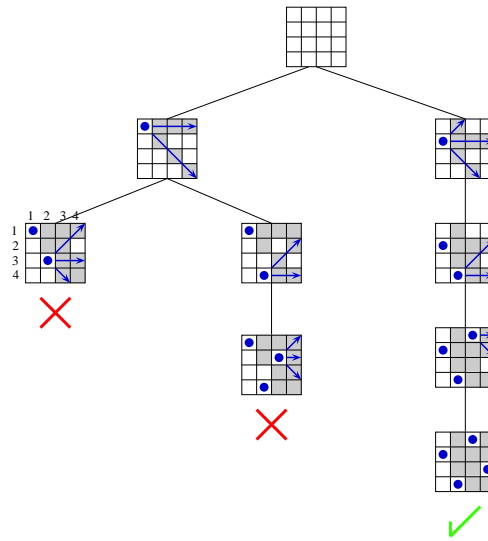


Figure 2.4 – Solving the 4-queens problem using FC.

Maintaining Arc Consistency

The Maintaining Arc Consistency (also called Full Look Ahead) [Gas74, SF94] is a stronger solving algorithm. It checks the conflicts between future variables in addition to the test between the current and the future variables.

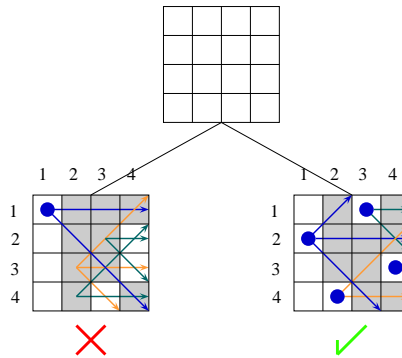


Figure 2.5 – Solving the 4-queens problem using MAC.

Figure 2.5 illustrates this process, where we can see that the MAC algorithm is able to prune the search space earlier than the forward checking, but doing much work on each variable assignment. For instance, when the first queen is placed at the position (1,1) the conflicts between the current position and the future positions are removed. After that, the algorithm checks the conflicts among the future variables starting with the first available position on the second column that is, the cell (3,2). The algorithm finds out that the position (3,2) is inconsistent since it does not leave available place for the third queen, thus the position (3,2) is removed. The algorithm follows with the cell (4,2), the next available position on the second column. This placement

leaves the cell (2,3) as the unique available position on the third column, which is then set as inconsistent since it does not leave available place for the fourth queen. The process follows until the result is reached on the right subtree.

2.2.4 Solving numerical CSPs

In the presence of constraints over real numbers, the already presented filtering techniques cannot be applied due to three main limitations:

- Deciding the consistency of constraints over real numbers is not possible in a general context [Ric68].
- The representation of reals in numerical computations is not exact since it is commonly done by means of floating-point numbers, which correspond to a finite set of rational numbers [Gou00].
- The use of floating point numbers may lead to rounding errors.

As a consequence, specific filtering techniques have been defined in order to deal with real numbers. These techniques mainly rely on the calculation of approximations over domains represented by intervals bounded by floating-point numbers. Some techniques are based on hull consistency [Lho93, Lv93, BO97] and on box consistency [BMV94].

2.2.4.1 Interval arithmetic

Before presenting the filtering techniques dedicated to numerical CSPs, let us give an overview of interval arithmetic [Moo66].

Definition 2.3 (Floating-point Interval). *An interval I bounded by floating-point numbers is defined as:*

$$I = [a, b] = \{r \in \mathbb{R} \mid a \leq r \leq b, \text{ with } a, b \in \mathbb{F}\}$$

We denote $\inf(I)$ as the lower bound and $\sup(I)$ as the upper bound of the interval. The four basic operations to be used on floating-point intervals are the following:

$$\begin{aligned} [a, b] \oplus [c, d] &= [\lfloor a + c \rfloor, \lceil b + d \rceil] \\ [a, b] \ominus [c, d] &= [\lfloor a - d \rfloor, \lceil b - c \rceil] \\ [a, b] \otimes [c, d] &= [\min(\lfloor ac \rfloor, \lfloor ad \rfloor, \lfloor bc \rfloor, \lfloor bd \rfloor), \max(\lceil ac \rceil, \lceil ad \rceil, \lceil bc \rceil, \lceil bd \rceil)] \\ [a, b] \oslash [c, d] &= [\min(\lfloor a/c \rfloor, \lfloor a/d \rfloor, \lfloor b/c \rfloor, \lfloor b/d \rfloor), \max(\lceil a/c \rceil, \lceil a/d \rceil, \lceil b/c \rceil, \lceil b/d \rceil)], 0 \notin [c, d] \end{aligned}$$

Definition 2.4. *Given $a \in \mathbb{R}$, we denote a^+ as the smallest element of \mathbb{F} greater than a , and a^- as the greatest element of \mathbb{F} smaller than a .*

Definition 2.5 (Canonical Interval). *We say that a nonempty interval I is canonical if :*

$$I = [a, b] \text{ such that } b \leq a^+, \text{ with } a, b \in \mathbb{F}$$

Definition 2.6 (Hull Operator). *The hull of a set $S \subseteq \mathbb{R}$ is defined as the smallest interval enclosing S :*

$$\text{hull}(S) = [\lfloor \inf(S) \rfloor, \lceil \sup(S) \rceil]$$

Definition 2.7 (Interval Extension). *An interval function $F : \mathbb{I}^n \rightarrow \mathbb{I}$ is an interval extension of a real function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ iff :*

$$\forall B \in \mathbb{I}^n : \{f(x) | x \in B\} \subseteq F(B)$$

There are various implementations of interval extensions. The natural interval extension of a real function f is defined as the function F in which each real constant is replaced by its hull and each real operation is replaced by its corresponding interval operation. As an example let us consider the following function f defined over real numbers:

$$f(x, y) = x^2 - (x \times y) + 2 | x, y \in \mathbb{R}$$

the natural extension F of the function f is defined as follows:

$$F(X, Y) = X^2 \ominus (X \otimes Y) \oplus [2, 2] | X, Y \in \mathbb{I}$$

Given $x \in X = [0, 2]$ and $y \in Y = [1, 3]$ we have:

$$F(X, Y) = [0, 4] \ominus [0, 6] \oplus [2, 2]$$

$$F(X, Y) = [-6, 4] \oplus [2, 2]$$

$$F(X, Y) = [-4, 6] \supseteq \{f(x, y) | x \in X, y \in Y\}$$

2.2.4.2 Consistency notions

In this section we present two of the consistency notions devoted to numerical CSPs: hull consistency (also called 2B-consistency) and box consistency.

Definition 2.8 (Hull Consistency). *Given a real constraint $c_j(x_{j_1}, \dots, x_{j_{n_j}})$, a box $B = I_1 \times \dots \times I_n \subseteq \mathbb{I}^n$, the box $B' = I_{j_1} \times \dots \times I_{j_{n_j}}$, an integer $k \in \{j_1, \dots, j_{n_j}\}$, we say that the constraint c_j is hull consistent wrt. x_k iff :*

$$I_k = \text{hull}(\pi_k(\Gamma_j \cap B')),$$

where π_k corresponds to the projection of c_j on x_k . We say that the constraint c_j is hull consistent wrt. B' if that relation is true for $k \in \{j_1, \dots, j_{n_j}\}$.

Definition 2.9 (Box Consistency). *Given a real constraint c_j of the form $f_j(x_{j_1}, \dots, x_{j_{n_j}}) = 0$, F_j a natural interval extension of f_j , a box $B = I_1 \times \dots \times I_n \subseteq \mathbb{I}^n$, the box $B' = I_{j_1} \times \dots \times I_{j_{n_j}}$, an integer $k \in \{j_1, \dots, j_{n_j}\}$, we say that the constraint c_j is box consistent wrt. x_k iff :*

$$I_k = \text{hull}(\{a_k \in I_k | 0 \in F_j(I_{j_1}, \dots, I_{k-1}, \text{hull}(\{a_k\}), I_{k+1}, \dots, I_{j_{n_j}})\})$$

We say that the constraint c_j is box consistent wrt. B' if that relation is true for $k \in \{j_1, \dots, j_{n_j}\}$.

For the sake of simplicity we define the box consistency only wrt. equalities, but this definition can be easily extended for inequalities, considering that $f \leq 0 \Leftrightarrow f = z, z \in [-\infty, 0]$.

The box consistency property is generally weaker than hull consistency (a comparison can be found in [CDR99]). Let us note that there also exist additional consistencies for numerical CSPs, for instance 3B-consistency, kB-consistency [Lho93], and CID-consistency [TC07].

2.2.4.3 Filtering algorithms

In this section, we illustrate two filtering algorithms by using the already presented consistencies.

Enforcing hull consistency

The hull consistency can be enforced by using interval arithmetic in two main phases: forward evaluation and backward propagation. As an example, let us consider the not hull consistent CSP $\mathcal{P} = \langle \langle x, y, z \rangle, \langle D_x \in [4, 9], D_y \in [2, 7], D_z \in [3, 8] \rangle, \langle x = y + z \rangle \rangle$.

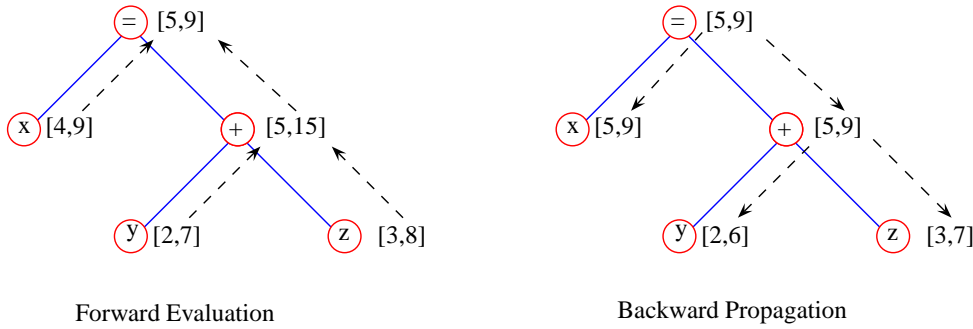


Figure 2.6 – Enforcing hull consistency.

Figure 2.6 depicts the process performed by the hull consistency algorithm. Such a process begins with the forward evaluation, which is a bottom-up tree traversal to evaluate its terms. The expression $y + z$ is evaluated by considering the interval addition operation, giving as a result the interval $[5, 15]$. The root of the tree corresponds to an equal symbol, which operates as an intersection. Thus, the result of this node is given by $[4, 9] \cap [5, 15] = [5, 9]$. The forward evaluation is followed by the backward propagation, where the constraint is projected on a top-down tree traversal. Starting with the root, the interval $[5, 9]$ is intersected with its child nodes, both nodes become $[5, 9]$, and the hull consistent domain of x is obtained. Then, to calculate the hull consistent domain of y , we reorganize the equation as follows: $y = [5, 9] \ominus z$. Using the interval subtraction operation, and replacing z by its domain, the result of the equation is given by $[5, 9] \ominus [3, 8] = [-3, 6]$. The new interval is intersected with the previous domain of y to obtain the hull consistent domain of y ($[-3, 6] \cap [2, 7] = [2, 6]$). The hull consistent domain of z is calculated in the same way.

Enforcing box consistency

For the sake of simplicity we consider a simple algorithm using box consistency (the original procedure includes the interval Newton method [Neu90]). This algorithm begins by testing whether the domain contains solutions. If the domain is inconsistent it is rejected; otherwise its lower canonical interval $[\inf(D), \inf(D)^+]$ is tested. If the canonical interval satisfies the constraint, $\inf(D)$ is the new lower bound. Otherwise, D is bisected and the procedure is performed again with the interval $[\inf(D)^+, \frac{\inf(D) + \sup(D)}{2}]$ and the interval $[\frac{\inf(D) + \sup(D)}{2}, \sup(D)]$. As an example,

let us consider the not box consistent CSP $\mathcal{P} = \langle \langle x \rangle, \langle D_x \in [-2, 2] \rangle, \langle x^2 < 2 \rangle \rangle$ shown in Figure 2.7.

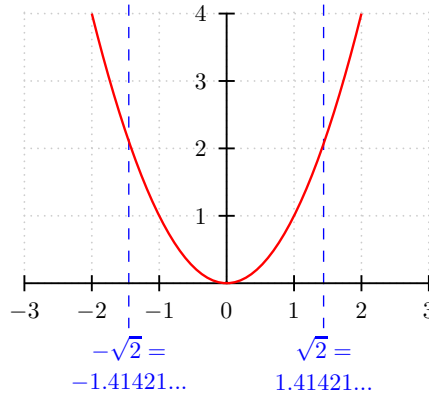


Figure 2.7 – The CSP $\mathcal{P} = \langle \langle x \rangle, \langle D_x \in [-2, 2] \rangle, \langle x^2 < 2 \rangle \rangle$.

Figure 2.8 illustrates the process performed by the algorithm. The process begins by testing the domain $[-2, 2]$ which contains consistent values but its canonical lower bound ($[-2, -2^+]$) is inconsistent, so it is bisected into the intervals $[-2^+, 0]$ and $[0, 2]$. The same process is done with the lower interval, which is bisected again into the intervals $[-2^{++}, -1]$ and $[-1, 0]$. The lower interval $[-2^{++}, -1]$ is bisected again, and the new lower interval is rejected since no solution is found. The process continues until both the lower and the upper canonical intervals are consistent. The lower bound of the consistent lower canonical interval and the upper bound of the consistent upper canonical interval correspond to the bounds of the box consistent domain.

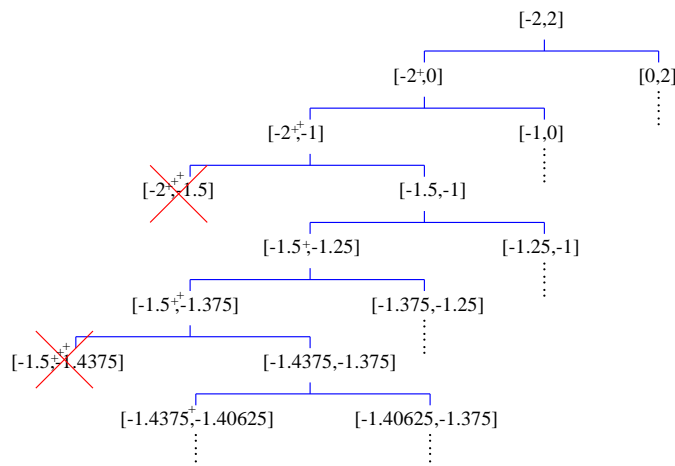


Figure 2.8 – Enforcing box consistency.

2.2.5 Variable and Value Ordering Heuristics

Search algorithms start the process by selecting a variable to enumerate or to bisect. The order in which this choice is done is referred to as the variable ordering. Several experiments have demonstrated that a correct ordering decision can be crucial to perform an effective solving process. There exist several heuristics for selecting the variable ordering:

- Fail-first: to select the variable with the smallest domain. This choice is motivated by the assumption that a success can be achieved by first trying the variables that have a bigger chance to fail, in this case, the values with a smaller number of available alternatives. This heuristic is known to be more adapted to discrete domains.
- Most-constrained variable: this choice can be justified by the fact that the instantiation of such a variable should lead to a bigger tree pruning through the constraint propagation.
- Reduce-first: to select the variable with the biggest domain. This heuristic is known to be more adapted to continuous domains.
- Round-robin: to select the variables in some rational and equitable order, for instance from the first variable defined in the model to the last one.

After selecting the variable to enumerate or bisect, the algorithms have to select a value from the variable's domain. This selection is called the value ordering and it can also have a considerable impact. For example, if the right value is chosen on the first try for each variable, a solution can be found without performing backtracks. However, if the CSP is inconsistent or the whole set of solutions is required, the value ordering is irrelevant. The literature presents different ways to perform this selection which, depending on the problem nature, may lead to a more efficient constraint propagation [Apt03].

For instance, continuous domains are generally bisected, i.e. each interval is split to obtain two size-equivalent intervals. It is also possible to enumerate a set of little intervals, whose size corresponds to the precision of variables. The discrete domains are, in general, enumerated, however it is also possible to bisect them as usually done in continuous domains. After the enumeration, it is possible to choose the first value as well as the smallest, the median or the maximal value. There also exist more complex value ordering heuristics which are in general either based on estimating the number of solutions or estimating the probability of a solution [van06].


2.3 Summary

In this chapter we have presented the main techniques for solving CSPs. We have illustrated basic search algorithms as well as more advanced procedures such as the ones involving constraint propagation. Constraint propagation is a filtering mechanism capable of improving the efficiency of search algorithms by enforcing a consistency property. Different kinds of consistency notions exist, which can be applied depending on the nature of the CSP.

In the next chapter, we present a large list of languages and systems for modeling and solving CSPs. Most of them embed in their internal solving engines the algorithms and techniques presented in this chapter.

CHAPTER 3

Languages and Systems

anguages and systems for modeling and solving CSPs have been developed under different principles. As we have mentioned, the first system dates back from the 1960s, followed by a large list where very different paradigms became involved. For instance, the use of logic programming as the support for the CLP paradigm or the use of objects for the simulation of problems under constraints. From an implementation point of view, different ways have been proposed, for instance, using libraries upon a host programming language or building a new programming language with support for constraints. The development of a pure modeling language instead of a programming language is a more recent concern, the idea is to provide a more “user-understandable” language. In the following paragraphs we give an overview of languages and systems for constraint satisfaction organized in six groups: CLP systems, libraries, modeling languages, programming languages, mathematical programming systems, and object-oriented languages. To give a general view of similarities and differences of such languages, at each section’s end a model of the n-queens problem is introduced.

3.1 Constraint Logic Programming

Constraint Logic Programming is the paradigm that extends logic programming to support constraint solving. This extension is known to be natural, as the declarativeness of logic programming is suitable for stating constraints, and the built-in backtracking engine can be used to simplify the implementation of search mechanisms. This idea was pioneered by Colmerauer, in the development of Prolog II [Col82]. Such an approach was then generalized in the CLP scheme established by Jaffar and Lassez in [JL87]. Then, many other systems including additional features were developed, some examples are presented in the following.

Prolog III-IV

Prolog III [Col90] is the successor version of the pioneering Prolog II system. This approach was one of the first in replacing the logic programming unification mechanism by the more general mechanism of constraint solving which, from a technical standpoint, is one of the basic principles of CLP. The last version of this set of successors is called Prolog IV [Col96], a CLP system designed to support constraints over different domains such as integers, reals and booleans.

CLP(\mathbb{R})

CLP(\mathbb{R}) [JMSY92] is another precursor CLP tool. It was defined as an instance of the CLP scheme established by Jaffar and Lassez. The implementation was designed to support constraint

over reals by means of an algebraic built-in constraint solver able to deal with linear arithmetic and non-linear constraints.

CHIP (Constraint Handling In Prolog)

CHIP [Van89] is also considered a pioneering CLP system together with the already presented CLP(\mathcal{R}) and Prolog III systems. It was originally developed as an extension of Prolog, being the first one in including global constraints. The current version, CHIP V5, is also available as a C and C++ library. CHIP V5 includes several features such as support for different kinds of constraints, interfaces to graphical components and relational databases. The system also integrates Xpress-MP [24] as its solver for linear programming.

ECLⁱPS^e

ECLⁱPS^e [WNS97] is a more recent CLP system. It provides a very wide range of features for solving problems under constraints, including the most typical such as lists, arrays and records, support for sets, and control statements such as conditionals and for loops. ECLⁱPS^e also provides a set of libraries, for example, for handling continuous CSPs, for CHR (Constraint Handling Rules) [Frü98] and for mathematical programming. Some of them can be combined to solve problems by means of a hybrid style. The definition of customized search procedures and variable and value orderings is also supported.

GNU Prolog

GNU Prolog is another system belonging to the CLP group [DC00]. GNU Prolog has been designed to support finite domain CSPs, however it can be interfaced to handle CSP over reals. It provides a large list of predefined Prolog predicates and constraints as well as support for common constructs such as lists, sets, and conditionals. Optimization problems and ordering heuristics are also supported. An interface has been included to call external routines written in C.

SICStus Prolog

The SICStus Prolog system [COC97] is based on a solver platform for finite domains, continuous domains and CHR. The host language provides typical data structures such as lists and arrays, and also more complex such as sets and Prolog-like objects. Support for conditional statements, optimization problems and variable ordering heuristics is available as well. It is also worth mentioning that SICStus Prolog has one of the most efficient implementations of global constraints. The system also provides multiple interfaces, for instance, for C, C++, .NET and Java.

Mercury

Originally, Mercury [SHC96] was designed as a logic/functional programming language. Currently, as part of the G12 project [SGM⁺05], it also provides support for CLP. An interesting

aspect of Mercury is that allows users to specify non-declarative code in a specific module. This facility avoids to define interfaces with other programming languages which normally add an overhead to the resolution process.

Example in ECLⁱPS^e

Figure 3.1 depicts an ECLⁱPS^e model for the n-queens problem. The file is composed of a call to a required library and a Prolog-like predicate called `queens`. This predicate is used to state the problem, and its header owns two arguments, `N` and `Board`. The first argument holds the quantity of queens and the second one is an array representing the row positions of the queens on the chessboard. The size of this array is given by `N` (line 5) and the domain of its variables is given by the interval `1..N` (line 6). Between lines 8 and 14, two for loops ensure that the constraints of the problem are applied over all the queens, `param` is used to define parameters, i.e. the variables stated outside the loop scope that must remain constant across iterations. Inside those loops, the three constraints of the problem are posted. The first constraint forbids two queens placed in the same row (line 10), the second one avoids two queens placed in the same South-West – North-East diagonal (line 11), and the third one avoids two queens in the same North-West – South-East diagonal (line 12). The `#\=` symbol corresponds to the not equal operator over integer expressions. At line 16, `Board` is converted to a list called `Vars` (due to the `labeling` predicate cannot be used over arrays). At the end of the file, the solving process is launched.

```

1.  :- lib(ic).
2.
3.  queens(N, Board) :-
4.
5.      dim(Board, [N]),
6.      Board[1..N] :: 1..N,
7.
8.      ( for(I,1,N), param(Board,N) do
9.          ( for(J,I+1,N), param(Board,I) do
10.              Board[I] #\= Board[J],
11.              Board[I]+I #\= Board[J]+J,
12.              Board[I]-I #\= Board[J]-J
13.          )
14.      ),
15.
16.      Board =.. [_|Vars],
17.      labeling(Vars).

```

Figure 3.1 – An ECLⁱPS^e model of the n-queens problem.

3.2 Libraries

Libraries provide a language for stating problems under constraints in the form of built-ins embedded in a host programming language. These built-ins are generally implemented by means of specific classes and methods, for instance, a given class is used to state variables and

methods define relations over them. This approach is a common way for implementing constraint systems since there is no need to implement a new language. However, the user is forced to have a background about the host language to use the library correctly, which is normally more complex and verbose compared to a pure modeling language.

ILOG Solver

ILOG Solver [Pug94] is a constraint-based optimization engine written as a C++ library. ILOG Solver provides a rich set of built-ins, for instance to support finite domain and floating-point variables. The library also supports optimization problems, the specification of heuristic orderings, and customized search procedures. Currently, the ILOG solver belongs to the ILOG CP suite, which is distributed together with ILOG Scheduler (for scheduling problems) and with ILOG Dispatcher (for vehicle routing problems).

Gecode & Gecode/J

Gecode [ST06] is another library written on top of C++. It has been designed to support finite domain variables. The constraint set is very large involving different kinds of constraints, over integer, boolean, and set variables. The Gecode system supports the definition of variable and value orderings as well as the specification of customized search and branching strategies. Gecode programs can be written in Java by using the Gecode/J interface.

Koalog

Koalog Solver [w7w] is a Java library for constraint satisfaction and constraint optimization. It supports finite domain constraints and finite set constraints. The specification of variable heuristics is supported, and customized search mechanisms can be built by defining specialized solver objects.

Choco

Choco [w2w] is a constraint programming solver written as a Java library. A large set of constraints is provided to be applied over integer, real and set variables. Support for optimization problems is given, and the search process can be customized by selecting predefined or user-defined variable and value ordering heuristics.

Example in Gecode/J

Figure 3.2 depicts a Gecode/J model for the n-queens problem. A Java class is used to state the entire problem. Such a class is composed of several elements: package and import statements (lines 1 to 4), a constructor (lines 9 to 25), a copy constructor required by the Gecode engine (lines 27 to 30), a procedure to show the results (lines 32 to 40), and a main method (lines 42 to 50). The constructor of the class is used to state the constants, variables and constraints of the problem. For instance, the constant holding the number of queens is defined at line 11 (it is set to 8, at line 44 in the main method of the class), and the array representing the positions of

the queens is stated at line 12. This array is initialized with five parameters: the reserved word **this** indicates the current class instance, **n** corresponds to the size of the array, **IntVar.class** corresponds to the class of objects contained in the array, and finally '**1,n**' defines the domain of the array. The three constraints of the problem are stated between lines 16 and 21. They are encapsulated in two forall loops and stated by means of the **post** method. Such a method defines a constraint between two expression objects. The '**new Expr().p(board.get(i))**' Gecode/J expression corresponds to the **Board[i]** expression in ECLⁱPS^e. The **IRT_NQ** parameter represents to the not equal operator, and **p** and **m** represent the '+' and '-' operators, respectively. At line 24, the labeling process is determined by a call to the **branch** method. This method requires the array to be processed, and the variable and value ordering heuristics.

At the end of the file, the main method sets several options, for instance, the size of the problem (line 44) and the use of the Gecode/J graphical interface (line 45). The process is launched by calling the **doSearch** method.

Another version for this problem can be stated by using a global constraint [vK06]. Figure 3.3 depicts this new model, where the three constraints of the problem has been replaced by calls to the **alldifferent** global constraint.

Note

A global constraint can be seen as a constraint that encapsulates a set of other constraints. For instance, the **alldifferent**(X_1, \dots, X_n) constraint specifies that the values assigned to the variables X_1, \dots, X_n must be pairwise distinct [Rég94]. This same constraint can be represented as a set of single inequality constraints. A main advantage of global constraints is that they can be associated to more powerful filtering algorithms since they can take into account the simultaneous presence of single constraints to further reduce the domains of the variables.

In Gecode/J, the **alldifferent** constraint is represented by the **distinct** method. The $board_i \neq board_j$ constraint is stated as **distinct(this, board)** (line 22). The second and third constraint (lines 23 and 24) are similar, but involve an array (**pos** and **neg**) which have been filled with the necessary offsets (lines 14 to 20) to represent the $board_i + i \neq board_j + j$ and the $board_i - i \neq board_j - j$ constraint, respectively. This model is probably less intuitive for understanding, however it is more efficient since the filtering algorithm of the **alldifferent** constraint is able to enforce the local consistency in a more effective way.

3.3 Modeling Languages

Modeling languages aim at simplifying the definition of constraint problems. They attempt to move users away from complicated encoding concerns present in typical libraries or programming languages. The core of the language is generally more comprehensible, as simpler syntax and semantics are provided. In some approaches, the specification of search procedures is permitted, but not mandatory.

Alice

Alice [Lau78] is also known as a precursor system in constraint programming. It dates back to 1978, as a result of the J.L. Lauriere Ph.D. Thesis. In this approach, variables and constraints

```

1.  package examples;
2.  import static org.gecode.Gecode.*;
3.  import static org.gecode.GecodeEnumConstants.*;
4.  import org.gecode.*;
5.
6.  public class Queens extends Space {
7.      public VarArray<IntVar> board;
8.
9.      public Queens(Options opt) {
10.         super();
11.         int n = opt.size;
12.         board = new VarArray<IntVar>(this, n, IntVar.class, 1, n);
13.
14.         for(int i=0;i<=n-1;i++) {
15.             for(int j=i+1;j<=n-1;j++) {
16.                 post(this, new Expr().p(board.get(i)),IRT_NQ,
17.                     new Expr().p(board.get(j)));
18.                 post(this, new Expr().p(board.get(i)).p(i),IRT_NQ,
19.                     new Expr().p(board.get(j)).p(j));
20.                 post(this, new Expr().p(board.get(i)).m(i),IRT_NQ,
21.                     new Expr().p(board.get(j)).m(j));
22.             }
23.         }
24.         branch(this, board, BVAR_SIZE_MIN, BVAL_MIN);
25.     }
26.
27.     public Queens(Boolean share, Queens queens) {
28.         super(share, queens);
29.         board = new VarArray<IntVar>(this, share, queens.board);
30.     }
31.
32.     public String toString() {
33.         int i;
34.         String st = "";
35.         for (i=0;i<board.size();i++){
36.             if(board.get(i).assigned())
37.                 st += board.get(i).val() + " ";
38.         }
39.         return st;
40.     }
41.
42.     public static void main(String[] args) {
43.         Options opt = new Options();
44.         opt.size = 8;
45.         opt.gui = true;
46.         opt.parse(args);
47.         opt.name = "Queens";
48.         Queens queens = new Queens(opt);
49.         opt.doSearch(queens);
50.     }
51. }

```

Figure 3.2 – A Gecode/J model of the n-queens problem.

```
1. package examples;
2. import static org.gecode.Gecode.*;
3. import static org.gecode.GecodeEnumConstants.*;
4. import org.gecode.*;
5.
6. public class Queens extends Space {
7.     public VarArray<IntVar> board;
8.
9.     public Queens(Options opt) {
10.         super();
11.         int n = opt.size;
12.         board = new VarArray<IntVar>(this, n, IntVar.class, 1, n);
13.
14.         int pos[] = new int[n];
15.         for (int i=0; i<n; i++)
16.             pos[i] = i;
17.
18.         int neg[] = new int[n];
19.         for (int i=0; i<n; i++)
20.             neg[i] = -i;
21.
22.         distinct(this, board);
23.         distinct(this, pos, board);
24.         distinct(this, neg, board);
25.
26.         branch(this, board, BVAR_SIZE_MIN, BVAL_MIN);
27.     }
28.     ...
```

Figure 3.3 – A Gecode/J model of the n-queens problem using global constraints.

are posted in a declarative style and the solutions are computed by an internal solving engine. This engine involves a graph, which is responsible for managing the variables and domains as well as the constraint propagation.

OPL

OPL [Van99] is a leading modeling language. Its syntax and semantics have been used as the base of modern modeling languages. The whole OPL language is composed of many high-level constructs, e.g. data structures such as arrays and records, finite domain variables, loops and conditional statements, and a set of built-ins for resource allocation. Heuristics for defining variable and value orderings are also supported. An interesting feature of OPL and perhaps its main novelty, is that searching strategies can be specified using the same elegant way as the used for stating the problem.

Zinc

Zinc [RGMW07] is a recent modeling language belonging to the G12 project. The Zinc syntax can be seen as an extension of OPL with support for user-defined predicates and functions. Typical data structures, sets, control abstractions, and finite and continuous domains are provided. The platform is supported by a solver-independent architecture where Zinc models can be mapped to three ECLⁱPS^e models: a constraint programming model, a local search model, and a mathematical programming model. An intermediate model called FlatZinc is also involved to facilitate the translation from source to target models.

MiniZinc

MiniZinc [NSB⁺07] is a smaller version of Zinc where user-defined types, functions and some coercions have been excluded. MiniZinc is also built upon a solver-independent architecture allowing mappings from MiniZinc to ECLⁱPS^e and Gecode. The mapping process is supported by a term rewriting-based transformation system called Cadmium [BDPS08] which allows to specify the translations from source to target models, a FlatZinc intermediate model is also used to facilitate the translation.

Essence

Essence [FGJ⁺07] is a language for specifying combinatorial problems. Its syntax can be seen as a combination of natural language and discrete mathematics. Essence supports typical modeling constructs and features for finite domain problems. Also, it provides the possibility of defining nested types of arbitrary depths (e.g. a set of sets of sets) on which constraints can operate. The architecture is solver-independent on which Essence models can be mapped either to ECLⁱPS^e or Minion [GJM06]. An intermediate OPL-like model called Essence' is used to facilitate the mapping chain. This model can be generated by means of the Conjure [FJMHM05] transformation system.

ESRA

ESRA [FPÅ04] is another modeling language based on the OPL's syntax. It has been designed for finite domain problems and supports common modeling constructs such as enumerations and arrays, and control abstractions such as forall loops. ESRA uses the notion of relation (e.g. injection, bijection), which often allows to define more concise and shorter models compared to OPL. ESRA models can be compiled into OPL and SICStus Prolog models.

NP-SPEC

NP-SPEC [CIP⁺00] is a logic-based language for the specification of problems belonging to the complexity class NP. A NP-SPEC model is divided into two sections, one section holds the data and the other the problem specification. The problem is mainly defined by means of Prolog-like predicates, first-order constraints on finite domains, and rules. NP-SPEC models are translated and then solved in the ECLⁱPS^e platform.

\mathcal{F}

\mathcal{F} [Hni03] extends OPL by introducing, among others, the notion of function problem, i.e. problems where the objective is to find functions from a source set to a target set such that some constraints are satisfied. In this architecture, \mathcal{F} models are mapped to an intermediate language called \mathcal{L} and then solved with ILP or CP techniques.

Rules2CP

Rules2CP is a new modeling language [FM08]. The main idea behind this approach is to combine the business rules knowledge representation paradigm with a CLP-based language. This combination may motivate the use of the CP technology in a wider audience since the extensive knowledge of business rules in the industry. Rules2CP models are compiled to SICStus Prolog via rewriting rules.

Example in MiniZinc

A MiniZinc model for the n-queens problem is shown in Figure 3.4. This model is divided into two files, a data file and a model file. The data file is used to assign values to the constants of the model. For instance, the constant `n` is defined as an integer in the first line of the model and set to 8 in the data file. The `board` array holding the positions of the queens is defined at line 2. It contains decision variables lying in the domain `1..n`. The three constraints of the problem are posted between lines 6 and 8, the `'!='` symbol corresponds to the not equal operator and `/\` represents the 'and' logical operator. The two forall loops required to traverse the array are embedded in just one forall. Finally, the `solve satisfy` statement is used to launch the solving process.

Data File

```
1.  n=8;
```

Model File

```
1.  int: n;
2.  array [1..n] of var 1..n: board;
3.
4.  constraint
5.      forall (i in 1..n, j in i+1..n) (
6.          board[i]      != board[j]      /\
7.          board[i] + i != board[j] + j /\
8.          board[i] - i != board[j] - j;
9.      );
10.
11. solve satisfy;
```

Figure 3.4 – A MiniZinc model of the n-queens problem.

3.4 Programming Languages

Many programming languages with support for constraint satisfaction have been developed, some of them have been specifically written for constraint satisfaction (e.g., CoJava, Comet) and others include support for constraints as an additional feature (e.g. Alma-0, OZ). In these languages the encoding possibilities are larger than in pure modeling languages, not only a declarative part is in general given, but also an imperative part. Thus, more freedom is given to programmers, however the learning process for non-experts may be slower compared to a pure modeling language.

Alma-0

Alma-0 [ABPS98] is an imperative programming language with support for declarative programming. The language allows to define arrays, records, and control statements such as conditionals and loops. The declarative part is devoted to problems involving search, being possible to define first-order constraints and Prolog-like predicates. The Alma-0 architecture merges techniques used to compile both imperative languages (RISC architecture) and logical languages (WAM Machine) in order to execute optimized programs.

Oz

Oz [SSW94] is the language of the Mozart Programming System. Oz can be seen as a multi-paradigm language since it supports several programming styles such as declarative and object-oriented programming as well as concurrent and constraint programming. The constraint programming component has been developed for sets and finite domain constraints. Support for optimization problems is given and the definition of custom search strategies is permitted. Another interesting feature of the platform is the Oz Explorer, a GUI (Graphical User Interface) for the interactive exploration of search spaces.

Comet

Comet [MV02] is an object-oriented programming language for combinatorial optimization problems. The COMET semantics supports typical data structures such as arrays and control abstraction such as forall loops. A rich language is used to post constraints and to define search strategies, which are defined in a style as elegant as in OPL. However, today Comet is a more general approach compared to OPL since it includes not only a language and a CP solver, but also a local search solver.

Minion

Minion [GJM06] is a solver for finite domain constraint problems. It has been designed to be interfaced with a modeling language such as Essence or OPL mainly since no syntactic sugar for modelers is provided. The input format is based on matrix models that is, the CSP is represented by one or more matrices of decision variables on which constraints are applied, e.g. on the rows, columns or planes. The solving engine supports optimization problems and different kinds of constraints such as global and reified constraints. Support for ordering heuristics is also given.

CoJava

CoJava [BN06] is an extension of the Java programming language that provides support for constraint optimization problems. The syntax of CoJava is identical to that of Java, and the support for CSPs and optimization problems is implemented in the form of a specific class. This class provides the necessary methods to define variables, domains, constraints and objective functions. CoJava problems are compiled and transformed into a mathematical model to be solved in AMLP.

Example in Alma-0

Figure 3.5 depicts an Alma-0 model for the n-queens problem. The constant giving the number of queens is stated at the beginning of the file. A new type called **board** is declared at line 2. Such a type defines the array representing the positions of the queens. The procedure to state the model begins at line 3, its input parameter is an array called **x** of type **board**. Within this procedure, the constraints of the problem are embedded in the required iteration loops.

3.5 Mathematical Programming

There exist several toolkits for mathematical programming. They mainly focus on solving optimization problems, their solving engines are based on mathematical programming procedures, and some of them have been boosted with constraint satisfaction mechanisms (e.g. Numerica, RealPaver). An important advantage of this field is that problems can be stated by means of a standard language, facilitating problem sharing, writing and experimentations [Pug04].

```

1.  CONST N = 8;
2.  TYPE board = ARRAY[1..N] OF [1..N];
3.  PROCEDURE Queens(Var x: board);
4.    VAR i;
5.    BEGIN
6.      FOR i := 1 TO N DO
7.        FOR j := i+1 TO N DO
8.          x[i] <> x[j];
9.          x[i] + i <> x[j] + j;
10.         x[i] - i <> x[j] - j;
11.        END;
12.      END
13. END Queens;

```

Figure 3.5 – An Alma-0 model of the n-queens problem.

AMPL

AMPL [FGK90] is a modeling language for mathematical programming. It supports linear and nonlinear optimization problems involving discrete or continuous variables. The language provides separation of model and data, data structures, and control abstractions such as loops and conditionals. The platform can be interfaced with a large list of solvers, e.g. CPLEX [w8w], MINOS [v4w], Xpress-MP [v2w] and SNOPT [v3w]. AMPL can also be linked to problem analysis tools such as MProbe [v6w] to identify the shape of functions. This information can be useful for modeling or for selecting an appropriate solving tool.

GAMS

GAMS [BKM92] is another modeling language for mathematical programming. As AMPL, GAMS is supported by a compiler and a large set of underlying solvers, some of them are MOSEK [v5w], LINGO [v3w], Xpress-MP and CPLEX. The core of the syntax supports typical mathematical programming modeling constructs, e.g. arrays, sets and control features such as loops and conditionals. Several contributions have been developed to complement the GAMS platform, for instance an interface with MATLAB [v7w] and tools for analyzing models and the given solutions.

Numerica

Numerica [VMD97] is a modeling language for global optimization based upon common mathematical notation, like AMPL and GAMS. An interesting feature of Numerica is related to its solving engine, it combines numerical analysis with consistency techniques for an efficient solving process. The use of intervals leads to another important aspect: the correctness of its computed results, i.e. no wrong solutions are produced in Numerica (modulo hardware or software errors).

RealPaver

RealPaver [GB06] is a constraint satisfaction system for modeling and solving linear and nonlinear systems. As in Numerica, the reliability of solutions is guaranteed by the use of intervals.

The modeling language is closer to AMPL, providing support for discrete and continuous variables, data structures such as arrays, and mathematical notation for posting constraints. The hull and the box consistency techniques can be used to tune the performance of search processes.

Example in AMPL

An AMPL model for the n-queens problem is depicted in Figure 3.6. The problem is modeled using the integer programming formulation, which is more appropriate for mathematical programming tools. Here, the chessboard is represented as a matrix containing binary variables (line 6). The size of the board is given by the sets stated at lines 3 and 4. In this formulation, four constraints are needed. The first constraint called `column_attack` avoids two queens sharing the same column. The `sum` function performs an addition of the column values of the matrix `board`. The `row_attack` constraint avoids two queens sharing the same row, and the last two constraints check the diagonals of the chessboard.

```

1.  param n := 8;
2.
3.  set ROWS := {1..n};
4.  set COLUMNS := {1..n};
5.
6.  var board {ROWS,COLUMNS} binary;
7.
8.  column_attacks {j in COLUMNS}:
9.    sum {i in ROWS} board[i,j] = 1;
10.
11. row_attacks {i in ROWS}:
12.   sum {j in COLUMNS} board[i,j] = 1;
13.
14. diagonal1_attacks {k in 3..2*t-1}:
15.   sum {i in ROWS, j in COLUMNS: i+j=k} board[i,j] <= 1;
16.
17. diagonal2_attacks {k in -(n-2)..(n-2)}:
18.   sum {i in ROWS, j in COLUMNS: i-j=k} board[i,j] <= 1;

```

Figure 3.6 – An AMPL model of the n-queens problem.

3.6 Object-oriented languages

An object-oriented language can also be merged with constraints in the form of constrained objects. In other words, a constrained object is an instance of a class that encapsulates the variables and constraints of a problem (or of a sub-problem). This approach is useful for modeling problems whose structure can be organized in many parts, as each one of these parts can be represented by a class. It is said that the benefits given by this combination are closer to those gained by writing software in an object-oriented language, e.g. encapsulation (of variables and constraints), modularity, reuse, etc. From the beginnings of constraint satisfaction, objects have been mixed with constraints through different ways.

Sketchpad

Sketchpad [Sut63] is considered a main contribution to the computer science field, not only in constraint satisfaction systems, but also in computer-aided drafting and object-oriented programming. Sketchpad was the first system in using a complete graphical user interface where the notion of objects and constraints was present. The system allowed the user to state master drawings (which can be regarded as a primitive form of a class) which could be instantiated to generate duplicates (objects), so if the master drawing changed, all the instances would change too. Constraints could be applied on drawings, for instance to fix the length of a line or the angle between two lines.

ThingLab

ThingLab [Bor81] was a direct successor of Sketchpad. The main idea behind ThingLab was to define a computer-based environment for constructing interactive graphic simulations, i.e. the simulation of an electrical circuit or a mechanical linkage. ThingLab allowed to perform these simulations by stating objects subject to constraints in a graphical user interface. Compared to Sketchpad, the major innovations were the support for multiple inheritance and the definition of local procedures for satisfying the constraints.

Gianna

Gianna [Pal95] is a visual modeling environment where the object-oriented concepts have been merged with the notion of constraint graph. A Gianna model is a graph formed by the association of several graphical components, each one representing an object-oriented entity. The associations define constraints as well as relations between the entities. For instance an association between classes is a class relation, and an association between objects is an object relation. An association between class attributes is a class constraint, and an object constraint is determined by an association of object attributes.

COB

COB [JT02] is a more recent language for constrained objects. It has been designed for modeling problems under constraints mainly from the engineering field. The language allows one to encapsulate the variables and the constraint of the problem as well as CLP predicates to define modular models. A graphical interface for COB exists, allowing users to design engineering problems using class diagrams. This graphical model is transformed into COB code, which is then compiled to a CLP solving engine.

Hinrichs et al. Approach

In [HLP⁺04], Hinrichs et al. present an object-oriented language involving constraint semantics devoted to automated constrained configurations. The approach can be seen as an extension

of the Common Information Model [w9w] (a common language for representing resource configuration in the industry) with an embedded language for posting first-order formulas as the constraints of the problem. The constructs supported by the language are limited to the automated configuration domain, and an internal theorem prover based solver performs the resolution phase.

SysML

SysML [w1w] is an extension of the UML, defined for modeling systems from the engineering field. As main novelty with respect to UML, SysML incorporates two new diagrams: the requirement diagram and the parametric diagram. The first diagram allows one to handle the requirements of the system and the second one permits modeling mathematical equations as constraints on the properties of such systems, for instance on their reliability or their performance. SysML models can be exported in XMI files and then pre-processed by an intermediate component called XaiTools. This tool is able to generate executable models to be launched in Mathematica [w5w] or in the Ansys [w3w] analysis tool.

s-COMMA

s-COMMA is an object-oriented modeling language for CP problems. The core of the language supports several modeling constructs, such as arrays, enumerations, finite and continuous domain variables and sets. Control abstractions such as loops and conditionals as well as global constraints and optimization statements are also supported. A specific simple formalism has been included to define variable and value orderings as well as the consistency levels for constraints. Additionally, an interesting extension mechanism allows the integration of new solver procedures. The whole system is supported by a solver-independent architecture where models can be mapped to many solvers (Gecode/J, ECLⁱPS^e, GNU Prolog and RealPaver). The integration of new solvers is possible by means of standard model transformation mechanisms. The platform also offers the s-COMMA GUI, which allows users to state problems using an extension of the UML class diagram.

Example in s-COMMA

Figure 3.7 depicts a s-COMMA model for the n-queens problem. Model from data independence is provided in s-COMMA. The data file is used to define and to assign values to constants (e.g. `n:=8`). In the model file, the problem is stated through classes. For this problem, just one main class called `Queens` is declared. Inside this class, the `board` array is defined, it contains `n` decision variables with domain `[1,n]`. Between lines 5 and 10, a constraint zone called `noAttack` is stated. Constraint zones are used to group constraints and statements. In the `noAttack` constraint zone, the two required forall loops have been embedded in one forall declaration. Within this loop the three constraints of the problem are posted.

Data File

```
1.  n:=8;
```

Model File

```
1.  main class Queens {
2.
3.      int board[n] in [1,n];
4.
5.      constraint noAttack {
6.          forall(i in 1..n, j in i+1..n) {
7.              board[i]  <> board[j];
8.              board[i]+i <> board[j]+j;
9.              board[i]-i <> board[j]-j;
10.         }
11.     }
12. }
```

Figure 3.7 – A s-COMMA model of the n-queens problem.

3.7 Comparing s-COMMA with related approaches

In this section, we give a more precise comparison between s-COMMA and its related approaches. We select the closest systems and we compare their features to give a more clear vision of how s-COMMA is positioned. In Table 3.1, s-COMMA is contrasted with five approaches considering six important features.

Table 3.1 – Comparing s-COMMA with five approaches. The meaning of each row is as follows. Object-Orientation: the language provides object-oriented capabilities. GUI: the system offers a graphical interface. Solver-Independence: the architecture is able to perform the problem resolution through different solvers. Mapping tool: the system provides a framework to add new solvers to the platform. Extensibility: the language can be extended for instance to support new global constraints or functions. Solving Options: the definition of heuristics orderings and consistency levels of constraints are allowed.

	Gianna	COB	Essence	Zinc	MiniZinc	s-COMMA
Object-Orientation	✓	✓	-	-	-	✓
GUI	✓	✓	-	-	-	✓
Solver-Independence	-	-	✓	✓	✓	✓
Mapping Tool	-	-	Hand-Written	TR+CHR	TR+CHR	Model-Driven
Extensibility	-	-	-	-	-	✓
Solving Options	-	-	-	-	✓	✓

Gianna and COB are the first systems included in the comparison. They belong to the same group as *s*-COMMA sharing some features such as object-oriented capabilities¹ and graphical interfaces. However, as opposed to *s*-COMMA, their modeling styles are not purely object-oriented. The COB language merges objects with CLP predicates and Gianna combines objects with constraints graphs. Additionally, they lack of solver-independence, a mapping-tool, extensibility, and the possibility of defining solving options.

Zinc, MiniZinc and Essence are the state-of-the-art systems and they are supported by a solver-independent architecture. The Essence execution platform allows to map specifications into ECLⁱPS^e and Minion solver. A model transformation system called Conjure is involved, but the integration of solver translators is not its scope. Translators from Essence' to solver code are written by hand. Zinc and MiniZinc can be mapped to the underlying solver layer via Cadmium, a transformation system based on Term-Rewriting (TR) [BN98] and Constraint Handling Rules (CHR) [Frü98]. *s*-COMMA is also built upon a solver-independent architecture, where models can be mapped to different solvers by means of model-driven translators.

Model-driven translators offer important advantages. The tools for implementing them are widely supported by the model engineering community. A considerable amount of documentation and several implementation examples are available at the Eclipse IDE site [10]. Tools such as Eclipse plug-ins are also available for developing and debugging applications. It is not less important to mention that ATL [KvJ07] (the language used for defining the model transformations) is considered a standard solution for model transformation in Eclipse. We believe this is a key issue to motivate and facilitate the addition of new solvers to the platform. Another important advantage is the separation of model and syntax concerns (we illustrate this in Section 5.3.2). This independence allows one to define clear and concise transformation rules, which are the base of our mapping tool.

From a language standpoint, *s*-COMMA is as expressive as MiniZinc and Essence, in fact these approaches provide similar constructs and modeling features. However, additional important features of *s*-COMMA remarkably differences it from those languages, for instance, the object-oriented modeling style, the extensibility mechanisms, and the possibility of modeling problems using a visual language.

3.8 Summary

In this chapter, we have presented a large list of constraint satisfaction systems. We have classified these systems in six groups: CLP systems, libraries, modeling languages, programming languages, mathematical programming systems, and object-oriented modeling languages including support for constraints. Several differences arise among these different approaches. The CLP paradigm extends logic programming by adding support for constraint solving. Libraries are built upon a host programming language, which provides its full semantics to the user. However, it is mandatory to master this language to successfully use the library. Programming languages have a larger expressiveness as well, they commonly provide a declarative and an imperative part to state models. The use of a modeling language is generally easier compared with a library or a programming language. Modeling languages provide a more understandable language, in which

¹It is important to clarify that object-oriented capabilities are also provided by languages such as CoJava, and in libraries such as Gecode or ILOG SOLVER. The main difference here is that the host language provided is a programming language but not a high-level modeling language. As we have explained, advanced programming skills may be required to deal with these tools.

complex encoding concerns are in general absent. Mathematical programming tools target optimization problems. Their core is supported by mathematical programming solving techniques and some of them include constraint satisfaction mechanisms. Finally, an object-oriented language can also be combined with constraints. The idea is to involve the benefits of object-orientation in a constraint satisfaction context.

At the end of the chapter, we have compared `s-COMMA` with five constraint satisfaction systems. We have shown how it is positioned with respect to its closest approaches through six features: object-orientation, GUI, solver-independence, mapping tool, extensibility and solving process customization. In the following chapter we present all these features in detail, we start by a giving an overview of the `s-COMMA` language to finish with a presentation of the `s-COMMA` GUI.

PART II

The s-COMMA platform

Modeling Language & Graphical Artifacts

s-COMMA is a new language for modeling CP problems. Such a language can be seen as a fusion of a high-level object-oriented language with a constraint language. This fusion has been complemented with useful features such as: solver-independence, extensibility, and a mechanism to customize the solving process.

The combination of these features provides interesting advantages. Users can model problems using a high-level modeling language. The object-oriented style provided can be used to organize problems in sub-problems to be captured in single classes. The extensibility mechanism allows one to extend the expressiveness of **s**-COMMA i.e., new functionalities can be added to the base language. A simple mechanism to tune models can be used to customize the solving process. A graphical user interface is also included in the platform. Visual models can be stated in the **s**-COMMA GUI by means of UML-based class diagram artifacts.

In this chapter we describe the various features of the **s**-COMMA language and the trade-offs we faced in its design. We begin by giving a tour of the **s**-COMMA language over six well-known CP problems. The tour is followed by a presentation of every modeling construct presented in the language. Then, the formalism to customize the solving process is introduced, followed by the extensibility mechanisms. At the end of the chapter, we illustrate the **s**-COMMA GUI and its main drawing and modeling components.

4.1 A Tour of the **s**-COMMA language

Let us begin the tour of the **s**-COMMA language by using the famous *SEND + MORE = MONEY* cryptarithmic puzzle. The idea is to find distinct digits for the letters *S, E, N, D, M, O, R, Y* such that the equation *SEND + MORE = MONEY* is satisfied.

4.1.1 The *SEND + MORE = MONEY* Problem

Figure 4.1 depicts the corresponding **s**-COMMA model for this problem. A main class called **Send** is used to state the whole model. Within this class, we identify **s,e,n,d,m,o,r,y** as the variables of the problem. Since these variables represent digits, their domains are given by the integer type. The integer domain $[0, 9]$ is used for the variables **e,n,d,o,r,y** and the integer domain $[1, 9]$ for variables **s** and **m**. These variables represent leading digits of the sum, being unable to take 0 as value. At line 6, a constraint zone called **equality** is stated to post the constraints of the problem.

```

1.  main class Send {
2.
3.      int e,n,d,o,r,y in [0,9];
4.      int s,m in [1,9];
5.
6.      constraint equality {
7.          1000*s + 100*e + 10*n + d
8.          + 1000*m + 100*o + 10*r + e
9.          = 10000*m + 1000*o + 100*n + 10*e + y;
10.         alldifferent();
11.     }
12. }

```

Figure 4.1 – A *s*-COMMA model of the cryptarithmic puzzle $SEND + MORE = MONEY$.

Remark

Constraint zones have been designed to group constraints under a descriptive name and to offer the possibility of overriding constraints in an inheritance context (see Section 4.2.5). Such a construct is another innovation of *s*-COMMA.

Between lines 7 and 9, the equation of the problem is represented as an equality constraint. Finally, the `alldifferent` global constraint is posted to define that all the variables involved in the problem must take different values.

4.1.2 The Packing Squares Problem

Let us continue the tour by presenting the packing square problem. This problem consists in completely covering a square base with a given set of squares, possibly having different sizes, with no overlappings among them.

A *s*-COMMA model for this problem is shown in Figure 4.2. Three constants are defined for this problem, which are imported from the data file `PackingSquares.dat`. The side size of the square base is given by `sideSize`, `squares` corresponds to the quantity of squares, and the array `size` contains their sizes.

Remark

In *s*-COMMA the data can be provided independently from the model file. This feature permits reusing models for different instances without change.

In the model file, two integer arrays of variables are defined to represent respectively the *x* and *y* coordinates of the square base. For example, `x[2]=1` and `y[2]=1` means that the second square must be placed in row 1 and column 1, indeed in the upper left corner of the square base. Both arrays are constrained, the decision variables must have values into the domain `[1,sideSize]`.

Data File

```

1.  int sideSize:=5;
2.  int squares:=8;
3.  int size:=[3,2,2,2,1,1,1,1];

```

Model File

```

1.  import PackingSquares.dat;
2.
3.  class PackingSquares {
4.
5.      int x[squares] in [1,sideSize];
6.      int y[squares] in [1,sideSize];
7.
8.      constraint inside {
9.          forall(i in 1..squares) {
10.             x[i] <= sideSize - size[i] + 1;
11.             y[i] <= sideSize - size[i] + 1;
12.          }
13.      }
14.
15.      constraint noOverlap {
16.          forall(i in 1..squares, j in i+1..squares) {
17.             x[i] + size[i] <= x[j] or
18.             x[j] + size[j] <= x[i] or
19.             y[i] + size[i] <= y[j] or
20.             y[j] + size[j] <= y[i];
21.          }
22.      }
23.
24.      constraint fitBase {
25.          (sum(i in 1..squares) (size[i]^2)) = sideSize^2;
26.      }
27.  }

```

Figure 4.2 – A s-COMMA model of the packing squares problem.

At line 8, a constraint zone called **inside** is declared. In this constraint zone, a **forall** loop contains the necessary constraints to ensure that each square is placed inside the base, one constraint acts over rows and the other one over the columns.

Remark

Loops have been designed to be used with loop variables (*i* and *j* in the example). A loop variable is valid only within the scope of its corresponding loop, and to simplify the model, no type is needed to declare it.

At line 15, the **noOverlap** constraint zone ensures that no overlapping occurs in the placement. Finally, the constraint zone called **fitBase** ensures the whole coverage of the square base. The **sum** loop is used to perform the addition of the areas of the square set.

Figure 4.3 depicts an analogous version of this model. An additional class called **Square** has been integrated to model the squares (line 3). This class contains the squares' attributes such as the *x* and *y* coordinates, and the size.

The data file of this model version is similar, the side size of the base and the quantity of squares have been defined. The third element of the data file corresponds to a variable assignment for the array **s** defined in the **PackingSquare** class at line 11. Variable assignments allow us to assign values to class attributes. The elements enclosed by '**{}**' symbols represent objects containing values for their attributes. In the example, a set of values is assigned to the third attribute of each **Square** object contained by **s**. The assignments are performed by respecting the order of arrays and class' attributes. For instance, the value 3 is assigned to the **size** attribute of the first object of the array. The value 2 is assigned to the **size** attribute of the second, third and fourth object of the array. The value 1 is assigned to the **size** attribute of remaining objects. The '**_**' symbol is used to omit assignments.

Remark

Variable assignments have been designed to perform direct assignments of values to decision variables. This feature offers the following benefits: (1) The definition of constructors¹ for each class is not necessary. (2) Calling a constructor each time an object is stated is not required. If we need to perform an assignment we do it directly in the data file. (3) The omission of these statements allows one to obtain a cleaner class definition. **s-COMMA** is unique in providing such a feature.

The main class of the problem is stated at line 9. This class is composed of an array and three constraint zones. The array contains the **Square** objects, and the constraint zones play the same role as in the previous packing squares model. Let us note that access to object attributes is achieved by using standard modeling notation, e.g. **s[2].x** corresponds to accessing the **x** attribute of the second object of the array called **s**.

¹A constructor is a special function used to set up the class attributes with values. It is used in most of object-oriented programming languages.

Data File

```

1.  int sideSize := 5;
2.  int squares  := 8;
3.  Square PackingSquares.s := [{_,_,3},{_,_,2},{_,_,2},{_,_,2},
                               {_,_,1},{_,_,1},{_,_,1},{_,_,1}];

```

Model File

```

1.  import PackingSquares.dat;
2.
3.  class Square {
4.      int x in [1,sideSize];
5.      int y in [1,sideSize];
6.      int size;
7.  }
8.
9.  main class PackingSquares {
10.
11.      Square s[squares];
12.
13.      constraint inside {
14.          forall(i in 1..squares){
15.              s[i].x <= sideSize - s[i].size + 1;
16.              s[i].y <= sideSize - s[i].size + 1;
17.          }
18.      }
19.
20.      constraint noOverlap {
21.          forall(i in 1..squares, j in i+1..squares){
22.              s[i].x + s[i].size <= s[j].x or
23.              s[j].x + s[j].size <= s[i].x or
24.              s[i].y + s[i].size <= s[j].y or
25.              s[j].y + s[j].size <= s[i].y;
26.          }
27.      }
28.
29.      constraint fitBase {
30.          (sum(i in 1..squares) (s[i].size^2)) = sideSize^2;
31.      }
32.  }

```

Figure 4.3 – An object-oriented s-COMMA model of the packing squares problem.

Remark

In this example, the representation is more natural since each square is independently handled as an object. The object-oriented style used here permit us to obtain a more modular model in which the structure of the problem has been captured in a single class composition.

4.1.3 The Stable Marriage Problem

The third problem of the tour is the stable marriage problem. Such a problem considers a group of n women and a group of n men who must marry. Each woman has a preference ranking for her possible husband, and each man has a preference ranking for his possible wife. The aim is to find a matching between groups such that the marriages are stable, i.e. there is no pair of people of opposite sex that like each other better than their respective spouses.

The data file of this problem is depicted in Figure 4.4. Two enumerations and two variable assignments can be identified. The `menList` enumeration holds the names of men and `womenList` holds the names of women. The `StableMarriage.man` variable assignment provides values for the `man` array defined at line 15 in the model file (see Figure 4.5). This variable assignment is composed of 5 objects, one for each man of the group. Each of these objects has two elements, the first element is an array (enclosed by ‘[]’) and the second one is the ‘_’ symbol. The first element sets the preferences of men, assigning the values to the `rank` array of `Man` objects (e.g. Richard prefers Tracy 1st, Linda 2nd, Wanda 3rd, etc).

Data File

```

1.  enum menList := {Richard,James,John,Hugh,Greg};
2.  enum womenList := {Helen,Tracy,Linda,Sally,Wanda};
3.  Man StableMarriage.man :=
4.      [Richard: {[Helen:5 ,Tracy:1, Linda:2, Sally:4, Wanda:3],_},
5.      James : {[Helen:4 ,Tracy:1, Linda:3, Sally:2, Wanda:5],_},
6.      John  : {[Helen:5 ,Tracy:3, Linda:2, Sally:4, Wanda:1],_},
7.      Hugh  : {[Helen:1 ,Tracy:5, Linda:4, Sally:3, Wanda:2],_},
8.      Greg  : {[Helen:4 ,Tracy:3, Linda:2, Sally:1, Wanda:5],_}];
9.
10. Woman StableMarriage.woman :=
11.     [Helen: {[Richard:1, James:2, John:4, Hugh:3, Greg:5],_},
12.     Tracy: {[Richard:3, James:5, John:1, Hugh:2, Greg:4],_},
13.     Linda: {[Richard:5, James:4, John:2, Hugh:1, Greg:3],_},
14.     Sally: {[Richard:1, James:3, John:5, Hugh:4, Greg:2],_},
15.     Wanda: {[Richard:4, James:2, John:3, Hugh:5, Greg:1],_}];

```

Figure 4.4 – Data file of the stable marriage problem.

The model file is stated through three classes, a class to represent men, a class to represent women and a main class to describe the stable marriages. The class representing men is composed of two attributes, the first one represents the preferences of a man, while the second one represents its wife. The `rank` array is indexed by the enumeration type `womenList` (line 2 of the data file), meaning that the 1st index of the array is `Helen`, the 2nd is `Tracy`, the 3rd is `Linda` and so on. The `wife` attribute is typed with an enumeration, therefore its domain is given by the values of that enumeration (`{Helen,Tracy,Linda,Sally, Wanda}`). The definition of the `Women` class is analogous.

Model File

```

1.  import StableMarriage.dat;
2.
3.  class Man {
4.      int rank[womenList];
5.      womenList wife;
6.  }
7.
8.  class Woman {
9.      int rank[menList];
10.     menList husband;
11. }
12.
13. main class StableMarriage {
14.
15.     Man man[menList];
16.     Woman woman[womenList];
17.
18.     constraint matchHusbandWife {
19.         forall(m in menList)
20.             woman[man[m].wife].husband = m;
21.
22.         forall(w in womenList)
23.             man[woman[w].husband].wife = w;
24.     }
25.
26.     constraint forbidUnstableCouples {
27.         forall(m in menList, w in womenList){
28.             man[m].rank[w] < man[m].rank[man[m].wife] ->
29.             woman[w].rank[woman[w].husband] < woman[w].rank[m];
30.
31.             woman[w].rank[m] < woman[w].rank[woman[w].husband] ->
32.             man[m].rank[man[m].wife] < man[m].rank[w];
33.         }
34.     }
35. }

```

Figure 4.5 – A s-COMMA model of the stable marriage problem.

The main class of the problem is stated at line 13. This class is composed of two arrays and two constraint zones. The first array models the group of men and the second one the group of women. The constraint zone called `matchHusbandWife` includes two `forall` loops, each one including a constraint. These constraints are satisfied whether the pairs man-wife match with the pairs woman-husband. The `forbidUnstableCouples` constraint zone contains two loops holding two logical formulas to guarantee that marriages are stable.

Remark

Enumerations have been designed for multiple usages. For instance, as type for decision variables (e.g. `womenList wife`), as the set of values to be traversed by a loop (e.g. `forall(m in menList)`) and for defining the size of arrays (e.g. `Man man[menList]`).

4.1.4 The Social Golfers Problem

The fourth problem of this overview corresponds to the Social Golfers Problem. This problem considers a group of n social golfers which play golf once a week, and always in groups of size g . The goal is to arrange a schedule for these players for w weeks, such that no two golfers play together more than once.

Figure 4.6 depicts the data file of this problem. It consists of one enumeration and three constants. The enumeration contains the name of the golfers and the constants hold the size of groups, the number of weeks, and the quantity of groups playing per week.

Data File

```
1.  enum name := {a,b,c,d,e,f,g,h,i};
2.  int s := 3; //size of groups
3.  int w := 4; //number of weeks
4.  int g := 3; //groups per week
```

Figure 4.6 – Data file of the social golfers problem.

The model file is divided into three classes (see Figure 4.7). One to model the groups, one to model the weeks and a main class to arrange the schedule of the social golfers. The `Group` class owns the `players` attribute corresponding to a set of golfers playing together, each golfer being identified by a name given in the enumeration from the data file. In this class, the constraint zone `groupSize` restricts the size of the golfers group. The `Week` class has an array of `Group` objects and the constraint zone `playOncePerWeek` ensures that each golfer takes part of a unique group per week. Finally, the `SocialGolfers` class has an array of `Week` objects and the constraint zone `differentGroups` states that each golfer never plays two times with the same golfer throughout the considered weeks.

Model File

```
1.  import SocialGolfers.dat;
2.
3.  class Group {
4.      name set players;
5.      constraint groupSize {
6.          card(players) = s;
7.      }
8.  }
9.
10. class Week {
11.     Group groupSched[g];
12.     constraint playOncePerWeek {
13.         forall(g1 in 1..g, g2 in g1+1..g)
14.             card(groupSched[g1].players intersect
15.                 groupSched[g2].players)= 0;
16.     }
17. }
18.
19. main class SocialGolfers {
20.
21.     Week weekSched[w];
22.
23.     constraint differentGroups {
24.         forall(w1 in 1..w, w2 in w1+1..w)
25.             forall(g1 in 1..g, g2 in 1..g)
26.                 card(weekSched[w1].groupSched[g1].players intersect
27.                     weekSched[w2].groupSched[g2].players) <= 1;
28.     }
29. }
```

Figure 4.7 – Model file of the social golfers problem.

4.1.5 The Production Problem

The fifth problem of the tour corresponds to an optimization problem. This problem considers a factory that must satisfy a determined demand of products. These products can be either manufactured inside the factory or purchased from an external market. The aim is to determine the quantity of products that must be produced inside the factory and the quantity to be purchased in order to minimize the total cost.

Model File

```

1.  import Production.dat;
2.
3.  class Product {
4.      int demand;
5.      int insideCost;
6.      int outsideCost;
7.      int consumption[resourceList];
8.      int inside in [0,5000];
9.      int outside in [0,5000];
10. }
11.
12. main class Factory {
13.
14.     int capacity[resourceList];
15.     Product productSet[productList];
16.
17.     constraint noExceedCapacity {
18.         forall(r in resourceList)
19.             capacity[r] >= sum(p in productList)
20.                 (productSet[p].consumption[r] *
21.                  productSet[p].inside);
22.     }
23.
24.     constraint satisfyDemand {
25.         forall(p in productList)
26.             productSet[p].inside + productSet[p].outside >= productSet[p].demand;
27.     }
28.
29.     constraint minimizeCost {
30.         [minimize] sum(p in productList)
31.             (productSet[p].insideCost * productSet[p].inside +
32.              productSet[p].outsideCost * productSet[p].outside);
33.     }
34. }

```

Figure 4.8 – A s-COMMA model of the production problem.

Figure 4.8 shows a s-COMMA model for this problem. The model is represented by two classes. The first one models the products while the second one models the factory. Within the **Product** class, several attributes are defined: the demand, the inside and the outside cost, the consumption, and the quantity that must be produced inside and outside the factory. The main class of the problem is stated at line 12. Two arrays are defined, the first one contains the amount of resources available for manufacturing the products and the second one contains the set of products. At line

17, a constraint zone called `noExceedCapacity` is stated to ensure that the resources consumed by the products manufactured inside do not exceed the total quantity of available resources. At line 24, a constraint zone is defined to satisfy the demand of all the products. Finally, at line 30, an optimization statement is posted to determine the quantity of products that must be produced inside the factory and the quantity to be purchased in order to minimize the total cost.

Data File

```

1.  enum resourceList := {flour, eggs};
2.  enum productList := {kluski, capellini, fettucine};
3.  int Factory.capacity := [200,400];
4.  Product Factory.productSet := [kluski:{1000,6,8,[flour:5,eggs:2],_,_},
                                   capellini:{2000,2,9,[flour:4,eggs:4],_,_},
                                   fettucine:{3000,3,4,[flour:3,eggs:6],_,_}];

```

Figure 4.9 – Data file of the production problem.

The data file of this problem is shown in Figure 4.9. It is composed of two enumerations and two variable assignments. The name of resources and products are held by the enumerations. The first variable assignment sets 200 as the `flour` capacity and 400 as the `eggs` capacity. The `Factory.productSet` variable assignment defines values for three products. Several values are set to those products. For instance, 1000 corresponds to the demand of the `kluski`, its inside cost is 6 and its outside cost is 8, finally, its manufacture requires 5 flour items and 2 egg units.

4.1.6 The Engine Problem

Let us finish the tour by presenting an academic problem from the engineering field. Consider the task of configuring a car engine subject to design constraints. The composition of the engine is depicted in Figure 4.10 using UML class diagram notation. Such a figure shows that the engine is the root of the system, it is built from a crankcase, a cylinder system, a block and a cylinder head at the second level. The cylinder system is a subsystem made of a valve system, an injection and a piston system. Both valve and piston systems have their own composition rules.

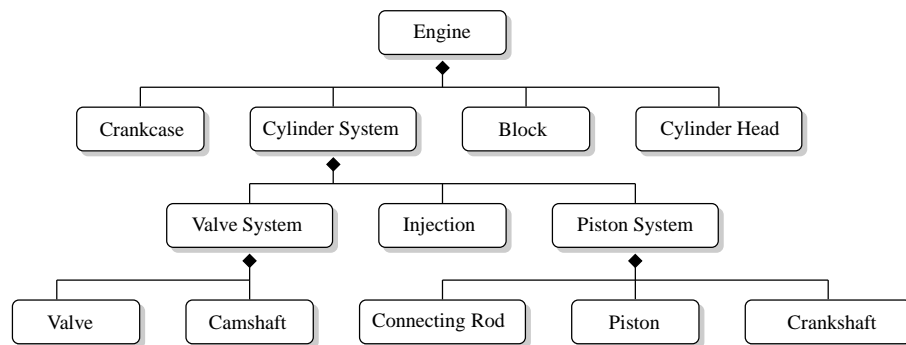


Figure 4.10 – The Engine Problem.

Figure 4.11 depicts the data file and the main class of the model. The attributes `cCase`, `cSyst`, `block` and `cHead` represent the subsystems of the engine. The last attribute defines its

`volume` and `dim` encapsulates a constraint between that attribute and the `volume` attribute of the `cCase` object.

Data File

```
1.  enum size := {small,medium,large};
2.  enum flow := {direct,indirect};
```

Model File

```
1.  main class Engine {
2.      CrankCase cCase;
3.      CylSystem cSyst;
4.      Block      block;
5.      CylHead    cHead;
6.      int         volume;
7.      constraint dim {
8.          volume > cCase.volume;
9.      }
10. }
```

Figure 4.11 – A `s-COMMA` model of the engine problem.

The `CylSystem` class is depicted in Figure 4.12. It has two integer variables, and three subsystems denoted by `inj`, `vSyst`, and `pSyst`. Its constraint zone encapsulates a conditional constraint. This constraint states that 6-cylinder-engines have to be a distance between cylinders bigger than 6, and in others kinds of engines this distance must be bigger than 3. In conditional constraints, whether the condition holds, the constraints belonging to the `if` block are activated; otherwise the constraints of the `else` block are activated.

```
1.  class CylSystem {
2.      int         quantity in [2,12];
3.      int         distBetCyl in [3,18];
4.      Injection   inj;
5.      ValveSystem vSyst;
6.      PistonSystem pSyst;
7.      constraint determineDistance {
8.          if (quantity = 6)
9.              distBetCyl > 6;
10.         else
11.             distBetCyl > 3;
12.         }
13. }
```

Figure 4.12 – The `CylSystem` class of the engine model.

The injection subsystem is depicted in Figure 4.13. It consists of three attributes: `gasFlow`, `admValve`, and `pressure`. The `compValues` constraint zone encapsulates a built-in compatibility constraint [GF03]. Such a constraint limits the combination of allowed values for a group of decision variables to a limited set. For example, only four combinations of values are permitted for the variables `gasFlow`, `admValve` and `pressure`. The possible values are described inside the

`compatibility` built-in constraint. Let us notice that the remaining classes of the model have been omitted since they are irrelevant for the purpose of this tour.

```

1.  class Injection {
2.      flow    gasFlow;
3.      size    admValve;
4.      int     pressure;
5.      constraint compValues {
6.          compatibility(gasFlow,admValve,pressure) {
7.              ("direct", "small", 80);
8.              ("direct", "medium", 90);
9.              ("indirect", "medium", 100);
10.             ("indirect", "large", 130);
11.          }
12.      }
13.  }
14.  ...

```

Figure 4.13 – The `Injection` class of the engine model.

Remark

In `s-COMMA`, all classes are public. Currently, we see no need to consider further visibility notions such as private or protected. This will force modelers to consider an additional concern and as a consequence to make more difficult the modeling tasks. However, whether these options become a necessity we may include them.

4.2 Modeling Features

In the previous section we have introduced some `s-COMMA` models to give an overview of its features. In this section, we provide a more extended presentation of such features. We introduce first the elements to be stated in data files such as constants and variable-assignments, and then the elements belonging to model files such as classes, attributes and constraint zones¹. We also include in this section the formalism to tune the solving process and the extension mechanisms.

4.2.1 Constants

Constants, also called parameters or data variables, are the variables that have a fixed value in the model. In `s-COMMA`, constants are declared in the data file and they have to be prefixed by a type. The available types for constants are: real, integer, boolean, and enumeration. As shown in Figure 4.14, constants can be included in one-dimensional and two-dimensional arrays. Boolean values can be defined by means of ‘0’ and ‘1’ digits or by using the tokens ‘true’ and ‘false’. Enumerations can contain real values, integer values or strings.

¹The grammar of the `s-COMMA` modeling language can be found in the appendix.

```

int  anIntegerConstant      := 5;
int  aOneDimArrayOfIntegerConstants := [1,2,3];
int  aTwoDimArrayOfIntegerConstants := [[1,2,3],[1,2,3],[1,2,3]];

real aRealConstant          := 5.2e-5;
real aOneDimArrayOfRealConstants := [1.1,2.2,3.3];
real aTwoDimArrayOfRealConstants := [[1.1,2.2,3.3],[1.1,2.2,3.3],[1.1,2.2,3.3]];

bool aBooleanConstant := false;
enum anEnumeration     := {France, Italy, Germany};

```

Figure 4.14 – Constants.

4.2.2 Variable assignments

A variable assignment permits setting values to variables in order to convert them into constants. Variable assignments are also stated in the data file, and they allow to assign values to many elements, for instance to decision variables, arrays containing decision variables, and objects. Figure 4.15 shows two examples. In the first one, the value 2.5 is given to the attribute **a** of the class **Test**. In the second one, the value 200 and the value 400 are assigned to the first and second cell of the **capacity** array, respectively.

```

real Test.a      := 2.5;
int  Factory.capacity := [200,400];

```

Figure 4.15 – Variable assignments.

As we have mentioned, variable assignments are performed by respecting the order of the involved elements. For instance, on the assignment of the array **capacity**, the value 200 is given to the first cell of the array, and 400 to the second cell of the array. However, whether the index of the array element is explicitly stated, this implicit ordered matching is omitted, and the assignments are guided by the indexes. For instance, Figure 4.16 depicts two variable assignments for the **productSet** array. Although the organization of both assignments differs, the resultant assignments are equivalent.

```

Product Factory.productSet := [kluski:{1000,6,8,[flour:5,eggs:2],_,_},
                               capellini:{2000,2,9,[flour:4,eggs:4],_,_},
                               fettucine:{3000,3,4,[flour:3,eggs:6],_,_}];

Product Factory.productSet := [capellini:{2000,2,9,[eggs:4,flour:4],_,_},
                               kluski:{1000,6,8,[eggs:2,flour:5],_,_},
                               fettucine:{3000,3,4,[eggs:6,flour:3],_,_}];

```

Figure 4.16 – Variable assignments guided by indexes.

4.2.3 Classes

Classes are the main element of models. They encapsulate the attributes and the constraints of the problem allowing to organize models and to capture the structure of problems. The main class of the model is defined using the **main** reserved word, if there is no main class in the model, the last class declared is set as main. Two kinds of relations are permitted among classes: composition and inheritance. Composition allows a class to be composed of many objects, and inheritance permits to define a new class based upon a superclass. Figure 4.17 shows a composition relation between the engine and its subsystems. On the right side of the figure a specific turbo engine class has been defined as a subclass of the class **Engine**. The reserved word **extends** is used to inherit the attributes and constraint zones of a superclass.

```

class Engine {
    CrankCase cCase;
    CylSystem cSyst;
    Block      block;
    CylHead    cHead;
    ...
}

class TurboEngine extends Engine {
    boost in [5,8];
    ...
}

```

Figure 4.17 – Composition and inheritance.

Remark

To ensure termination, recursive composition (a class having as attribute an instance of itself) and recursive inheritance (a class inheriting from itself) are not allowed.

Let us note that modularity of **s-COMMA** models can be enhanced since single models can be stored in different files to be imported in a main file. Figure 4.18 depicts a model representing the design of a car. Each car's subsystem (the engine, the electric system, the exhaust system, etc.) has been modeled in a different file which has been then imported from the car model file.

```

import Engine.cma
import ElectSystem.cma
...

main class Car {
    Engine      eng;
    ElectricSystem elSyst;
    ExhaustSystem exSyst;
    SuspSystem   suSyst;
    DriveTrain   drSyst;
    Chassis      chass;
    ...
}

```

Figure 4.18 – Importing models.

Remark

Modularity, composition and inheritance are important strengths of the object-oriented style. In s-COMMA we can benefit from that and motivate the reuse of existing elements.

4.2.4 Attributes

Attributes are used to define object properties. In s-COMMA, attributes are stated within classes and they have to be prefixed with a type. Attributes may represent decision variables, sets or objects.

4.2.4.1 Decision Variables

Decision variables correspond to the unknowns of the problem. s-COMMA allows decision variables to be contained in one-dimensional and two-dimensional arrays (see Figure 4.19). The size of the arrays can be defined by an integer constant, an integer value or an integer constant expression. The latter stands for an expression composed only of integer values and/or integer constants.

Remark

To avoid non-terminating iteration over an array, no decision variable is permitted to define its size.

```
int  anIntegerDecisionVariable;
real aTwoDimArrayOfRealDecisionVariables[5,anIntegerConstant+1];
```

Figure 4.19 – Decision variables.

Decision variables and arrays of decision variables can be constrained to a determined domain (see Figure 4.20). The nature of values to define the domains depends on the nature of decision variables. For instance, integer values, integer constants and integer constant expressions are used to define domains for both integer and real decision variables. Real values, real constants and real constant expressions can only be used to define the domain of real decision variables. Decision variables with no domain stated adopt a default domain in the translation process, which depends on the solver used. An enumeration can be used as the type for a decision variable in order to adopt as domain the set of values contained in the enumeration.

```
int  anIntegerDecisionVariable in [0,anIntegerConstant + 1];
real aRealDecisionVariable    in [0.5,aRealConstant + 5.5];

enum menNames := {Richard,James,John,Hugh,Greg};
menNames husband;
```

Figure 4.20 – Decision variables, domains and enumerated domains.

4.2.4.2 Sets

A set can be seen as a special kind of decision variable for which the resolution process must search a set of values. Sets are used in many problems and specific relations can act over them (e.g. union, intersection, disjunction, etc.). Sets are defined with the reserved word **set**, and they can be contained in one-dimensional and two-dimensional arrays. The domains of sets can be given by integer values, integer constants, integer constant expressions, and enumerations. Three examples are depicted in Figure 4.21.

```
int set aSet in [0,9];
int set aTwoDimArrayOfSets[3,3] in [0,9];
name set players;
```

Figure 4.21 – Sets.

4.2.4.3 Objects and Constrained Objects

Objects are instances of classes and they must be typed with the corresponding class name. Objects embedding one or more constraints are called constrained objects. In Figure 4.22, the **p** object is an instance of the **Product** class, and **g** is a constrained object as its **players** attribute is subject to a constraint.

```
Product p;

class Product {
    int demand;
    int insideCost;
    int outsideCost;
    int consumption[resourceList];
    int inside in [0,5000];
    int outside in [0,5000];
}

Group g;

class Group {
    name set players;
    constraint groupSize {
        card(players) = s;
    }
}
```

Figure 4.22 – Objects and constrained objects.

4.2.5 Constraint Zones

Constraint zones are used to group constraints encapsulating them inside a class. A constraint zone can contain constraints, loops, conditional statements, compatibility constraints, an optimization statement, and global constraints. Figure 4.23 depicts a constraint zone of the packing squares problem.

```

constraint inside {
  forall(i in 1..squares){
    x[i] <= sizeArea - size[i] + 1;
    y[i] <= sizeArea - size[i] + 1;
  }
}

```

Figure 4.23 – A constraint zone.

The name of the constraint zone is chosen by the modeler. It can be used to describe the role of the constraint zone on the problem and also to allow the constraint zone to be overridden by a subclass. Constraint zone overriding can be seen as method overriding in object-oriented languages. In other words, when a class inherits from a superclass, the constraint zones of the superclass (having a same name) are no longer considered and they are replaced by the constraint zones of the subclass. In Figure 4.24, the constraint zone `distanceBetAxes` is overridden by the subclass `TurboEngine`, resulting in a replacement of the constraint `left + 2320 = right` by the constraint `left + 2840 = right`.

```

class Engine {
  ...
  constraint distanceBetAxes {
    left + 2320 = right;
  }
}

class TurboEngine extends Engine {
  ...
  constraint distanceBetAxes {
    left + 2840 = right;
  }
}

```

Figure 4.24 – Constraint zone overriding.

4.2.5.1 Constraints

Constraint are relations among variables, being posted using mathematical-like notation. s-COMMA supports most of common relations among values, constants, decision variables and sets (see Table 4.1).

Table 4.1 – Binary and unary operators. Higher precedence means lower priority. T represents integer, real, or boolean types. N represents integer or real types.

Operator	Operation	Precedence	Relation
\leftrightarrow	Bi-implication	1300	$(boolean \times boolean) \rightarrow boolean$
\rightarrow	Implication	1200	$(boolean \times boolean) \rightarrow boolean$
\leftarrow	Reverse implication	1200	$(boolean \times boolean) \rightarrow boolean$
or	Disjunction	1100	$(boolean \times boolean) \rightarrow boolean$
xor	Exclusive or	1100	$(boolean \times boolean) \rightarrow boolean$
and	Conjunction	1000	$(boolean \times boolean) \rightarrow boolean$
not	Unary negation	900	$boolean \rightarrow boolean$
$<$	Less than	800	$(T \times T) \rightarrow boolean$
$>$	Greater than	800	$(T \times T) \rightarrow boolean$
\leq	Less than or equal	800	$(T \times T) \rightarrow boolean$
\geq	Greater than or equal	800	$(T \times T) \rightarrow boolean$
$==, =$	Equality	800	$(T \times T) \rightarrow boolean$ or $(set \times set) \rightarrow boolean$
$!=, <>$	Inequality	800	$(T \times T) \rightarrow boolean$ or $(set \times set) \rightarrow boolean$
subset	Subset	700	$(set \times set) \rightarrow boolean$
superset	Superset	700	$(set \times set) \rightarrow boolean$
union	Union	600	$(set \times set) \rightarrow set$
diff	Difference	600	$(set \times set) \rightarrow set$
symdiff	Symmetric difference	600	$(set \times set) \rightarrow set$
$+$	Addition	500	$(N \times N) \rightarrow N$
$-$	Subtraction	500	$(N \times N) \rightarrow N$
$*$	Multiplication	400	$(N \times N) \rightarrow N$
$/$	Division	400	$(N \times N) \rightarrow N$
intersect	Intersection	300	$(set \times set) \rightarrow set$
$^$	Exponent	200	$(N \times N) \rightarrow N$
$-$	Unary subtraction	100	$N \rightarrow N$

4.2.5.2 Loops

Two kinds of loops are provided by s-COMMA, the **forall** loop and the **sum** loop. The **forall** loop is used to iterate over loop variables stated within constraints and the **sum** loop is used to perform the mathematical summation.

The **forall** loop can contain loops, conditionals, constraints, and global constraints. The loop header is declared in two parts. The left part defines the loop variable and the right part defines the set of values to be traversed by the loop variable. The right part can be stated by using a range of values. This range must be defined by integer values, integer constants, loop variables, or integer constant expressions (including loop variables). An enumeration, or a one-dimensional array can also be used to define the right part of the loop header. In these cases, the loop will cross from 1 until the size of the enumeration or array (see Figure 4.25).

<pre>forall(i in j+1..5+n) { a[i] > i; ... }</pre>	<pre>forall(i in anEnumeration) { a[i] > i; ... }</pre>	<pre>forall(i in aOneDimArray) { a[i] > i; ... }</pre>
---	--	---

Figure 4.25 – **forall** loops.

To compact models, it is possible to embed an arbitrary number of nested **forall** loops in a single **forall** definition (see Figure 4.26). Forall loops holding only one statement can omit their curly brackets.

```
forall(i in 1..5) {
  forall(j in i+1..5) {
    forall(k in j+1..5) {
      ...
    }
  }
}

forall(m in menList)
  woman[man[m].wife].husband = m;
```

Figure 4.26 – Nested **forall** loops.

The **sum** loop performs an addition of a set of expressions. Its header is defined in the same manner as in **forall** loops. Figure 4.27 depicts an example, where the expression ‘ $a[1]*1 + a[2]*2 + a[3]*3$ ’ has been compressed in a **sum** loop. To avoid ambiguities, parentheses around $a[i]*i$ are mandatory.

```
sum(i in 1..3) (a[i]*i)
```

Figure 4.27 – The **sum** loop.

4.2.5.3 Conditionals

Conditionals are stated by means of the **if** and the **if-else** statement. Loops, conditionals, constraints, an optimization statement, and global constraints can be stated inside the body of

conditionals. The condition can be stated through an expression containing values, constants or decision variables. Curly brackets are mandatory when the conditional holds more than one statement. Examples are shown in Figure 4.28.

```

if (quantity = 6)
    distBetCyl > 6;
else
    distBetCyl > 3;

if (quantity = 6) {
    distBetCyl > 6;
    ...
} else {
    distBetCyl > 3;
    ...
}

```

Figure 4.28 – Conditionals.

4.2.5.4 Optimization

Optimization statements allow to model optimization problems. Optimization statements are defined with a tag specifying the kind of optimization to be applied. The `[maximize]` tag is used for maximizing and the `[minimize]` tag for minimizing expressions. An example is shown in Figure 4.29.

```

constraint reduce {
    a + b > c;
    [minimize] a + b;
}

```

Figure 4.29 – Optimization statement.

4.2.5.5 Global Constraints

Two versions of the `alldifferent` constraint are provided. The `alldifferent()` forces that all the values defined in the class must be different, and the `alldifferent(anIntegerArray)` forces that all the values inside the given array must be different.

The `alldifferent` constraint is the unique global constraint included in *s-COMMA*. Additional global constraints can be added using the extension mechanisms presented in Section 4.2.7.

4.2.5.6 Compatibility Constraints

A compatibility constraint is used to limit the combination of allowed values for a group of decision variables to a group of given tuples. For instance, the compatibility constraint depicted in Figure 4.30 defines that only three possible tuples of values satisfy the constraint. This built-in constraint can also be seen as syntactic sugar for a boolean formula (depicted on the right side of the figure).

```

compatibility(a,b,c,d) {
    (3, 5, 8, 6);
    (1, 2, 5, 8);
    (9, 0, 3, 2);
}
    (a=3 and b=5 and c=8 and d=6) or
    (a=1 and b=2 and c=5 and d=8) or
    (a=9 and b=0 and c=3 and d=2)

```

Figure 4.30 – A compatibility constraint.

4.2.6 Heuristic Orderings & Consistency Techniques

The formalism to customize the solving options of object-oriented models is one of the many innovations of *s*-COMMA. Such a formalism permits the specification of the value and variable ordering as well as the consistency level of constraints.

4.2.6.1 Variable and Value Ordering

As mentioned in Section 2.2.5, variable and value orderings stand for the sequence in which the variables and values are selected for the variable-value assignment performed during the resolution process. Different heuristics exist for carrying out this process, *s*-COMMA includes the most solver-supported ones:

Variable orderings:

- **min-dom-size**: selects the variable with the smallest domain size.
- **max-dom-size**: selects the variable with the largest domain size.
- **min-dom-val**: selects the variable with the smallest value in its domain.
- **max-dom-val**: selects the variable with the greatest value in its domain.
- **min-regret-min-dif**: selects the variable that has the smallest difference between the smallest value and the second-smallest value of its domain.
- **min-regret-max-dif**: selects the variable that has the greatest difference between the smallest value and the second-smallest value of its domain.
- **max-regret-min-dif**: selects the variable that has the smallest difference between the largest value and the second-largest value of its domain.
- **max-regret-max-dif**: selects the variable that has the greatest difference between the largest value and the second-largest value of its domain.

Value orderings:

- **min-val**: selects the smallest value.
- **med-val**: selects the median value.
- **max-val**: selects the maximal value.

To exemplify the use of this feature let us introduce a fragment of the engineering design problem presented in [GF03]. The aim of this problem is to assemble an industrial mixer subject to configuration constraints. Figure 4.31 shows the composition of such a system.

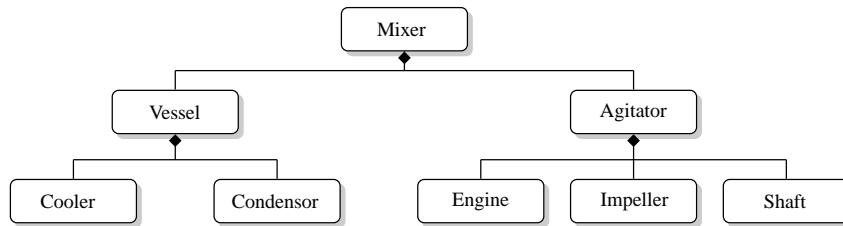


Figure 4.31 – The industrial mixer problem.

s-COMMA provides four possibilities for defining heuristic orderings: (1) to select the variable ordering, (2) to select the value ordering, (3) to select both or (4) not select any option, in this case the solving process will be performed using the default option given by the solver. Figure 4.32 depicts the four cases.

<pre>// variable ordering selected main class Mixer [min-dom-size] { ... // both selections main class Mixer [min-dom-size,min-val] { ...</pre>	<pre>// value ordering selected main class Mixer [min-val] { ... // no selection main class Mixer { ...</pre>
--	--

Figure 4.32 – Value and variable orderings.

Remark

Since the searching process is performed for the entire problem, we cannot consider different heuristics for each class. So, if more than one class includes any ordering option, just the option stated at the main class will be considered.

4.2.6.2 Consistency Level

As we have explained in Section 2.2.3, backtracking procedures can be complemented with consistency algorithms to detect failures earlier, thus avoiding the inspection of useless spaces. This task is in general performed by variants of the arc-consistency algorithm embedded in the search engine of the solver. s-COMMA provides the most-solver supported consistency levels, the bound and the domain consistency:

- **bound**: an arc-consistency algorithm is used to reduce the domain of involved variables, but just the bounds of the variables' domain are updated.
- **domain**: an arc-consistency algorithm is used to reduce the domain of involved variables, but the full domain of variables is updated.

To specify these options s-COMMA provides three possibilities: (1) to select the consistency level for a class, (2) to select the consistency level for an object; and (3) to select the consistency

level for a constraint. Let us note that cases 1 and 2 lead to a “cascade effect” i.e., the selected option will be inherited by objects and constraints belonging to the composition. Only objects and constraints with their own option do not inherit, they keep their own selected option.

Figure 4.33 depicts an example on which two classes of the mixer problem have been tuned. The **Mixer** class has been configured with a **domain** consistency level leading the “cascade effect” to set the objects and constraints of the mixer’s composition (Vessel, Agitator, Cooler, Condenser, etc.) with the **domain** option, except for the **Engine** object **e** and the constraint **e.power** $\geq 2*i.power$ which keep their own option (**bound** consistency).

Remark

The cascade effect provided by the object-oriented style of **s-COMMA** allows us to avoid the definition of solving options for constraints one by one.

```
// tuned class
main class Mixer [domain] {
  Vessel    v;
  Agitator  a;
  constraint design {
    a.i.rps <= v.diameter/a.i.diameter;
    a.i.diameter <= a.i.ratio*v.diameter;
    ...
  }

// tuned object & tuned constraint
class Agitator {
  [bound] Engine e;
  Impeller i;
  Shaft    s;
  constraint power {
    [bound] e.power >= 2*i.power;
    ...
  }
}
```

Figure 4.33 – Consistency level.

Let us note that the combination of consistency level with value and variable orderings is permitted (see Figure 4.34).

```
main class Mixer [min-dom-size,min-val,domain] {
  ...
}
```

Figure 4.34 – Ordering heuristics & consistency level.

Note

A given heuristic ordering or a given consistency level can only be used if the selected underlying solver has support for it.

4.2.7 Extensibility

Extensibility is another important feature of **s-COMMA**. New constraints, functions, ordering heuristics and consistency levels can be integrated by defining extension files. This mechanism ensures the semantics of the **s-COMMA** language adaptable to potential upgrades of the solver layer.

4.2.7.1 Adding constraints

Let us present this feature by recalling the social golfers problem. Consider that a programmer adds to the Gecode/J solver a new global constraint to enforce the $a <_{lex} b$ lexicographic ordering. This constraint operates over a set $a = \{x_0, x_1, \dots, x_n\}$ and a set $b = \{y_0, y_1, \dots, y_n\}$ of n integer values, ensuring that: $x_0 < y_0$; $x_1 < y_1$ when $x_0 = y_0$; $x_2 < y_2$ when $x_0 = y_0$ and $x_1 = y_1$; ...; $x_{n-1} < y_{n-1}$ when $x_0 = y_0, x_1 = y_1, \dots$, and $x_{n-2} = y_{n-2}$ [FHK⁺02]. The $a <_{lex} b$ constraint will be used to remove the symmetries [Pug93, CGLR96, GS00] (eliminate redundant solutions) of the already presented social golfers model.

To use this new constraint we can extend the semantics of the **s-COMMA** constraint language. This can be achieved by defining an extension file where the rules of the translation are stated. Such a file may be composed of one or more main blocks (see Figure 4.35). Main blocks hold the translation rules and denote the solver to which the mapping must be performed. For instance, the first main block defines the mapping rules for the Gecode/J solver.

```

1.  GecodeJ {
2.      Constraint {
3.          lexOrder(a,b) -> "gecodeJLexicalOrdering($a$, $b$)";
4.      }
5.  }
6.
7.  ECLiPSe {
8.      Constraint {
9.          ...
10.     }
11.     ...

```

Figure 4.35 – Adding constraints to **s-COMMA**.

Within the **GecodeJ** block, a **Constraint** block has been defined. This block owns the mapping rule of the new constraint to be added. This rule consists of two parts. The left part of the rule defines the statement used to call the new function from the **s-COMMA** language, and the right part defines the statement used to call the new built-in method from the solver file. In this way, the rule states that **lexorder(a,b)** will be translated to **gecodeJLexicalOrdering(a,b)** in the mapping process from **s-COMMA** to Gecode/J. To facilitate the translation of the input parameters, variables (**a** and **b**) must be tagged with '\$' symbols. In the example, the first parameter and the second parameter of the new **s-COMMA** constraint will be translated as the first parameter and the second parameter of the Gecode/J method call, respectively. The use of the new constraint in the social golfers problem is shown in Figure 4.36.

```

1.  import lexOrderings.ext;
2.  ...
3.
4.  main class SocialGolfers {
5.
6.      Week weekSched[w];
7.
8.      constraint differentGroups {
9.          forall(w1 in 1..w, w2 in w1+1..w)
10.             forall(g1 in 1..g, g2 in 1..g)
11.                 card(weekSched[w1].groupSched[g1].players intersect
12.                     weekSched[w2].groupSched[g2].players) <= 1;
13.      }
14.
15.      constraint removeSymmetries {
16.          forall(w1 in 1..weeks, g1 in 1..groups-1)
17.              lexOrder(weekSched[w1].groupSched[g1].players,
18.                      weekSched[w1].groupSched[g1+1].players);
19.
20.          forall(w1 in 1..weeks-1)
21.              lexOrder(weekSched[w1].groupSched[1].players,
22.                      weekSched[w1+1].groupSched[1].players);
23.      }
24.  }

```

Figure 4.36 – Removing symmetries from the social golfers problem.

4.2.7.2 Adding functions

To present the usefulness of this feature, let us introduce the Sudoku problem. This problem consists in filling a 9×9 matrix so that each column, each row, and each of the nine 3×3 sub-matrices contains different digits from 1 to 9. A model for this problem is depicted in figure 4.37. The data file is composed of two constants and a variable assignment. The constant **n** defines the size of the matrix and **s** the size of the sub-matrices. The variable assignment is used to fill some of the cases of a two-dimensional array called **puzzle**. This array is stated at line 5 of the model file and represents the matrix of the problem. The constraint zones of the model are defined next. The **differentInRowsAndColumns** constraint zone ensures that every row and column of the matrix contains different values, and **differentInSubMatrices** guarantees that all the 3×3 sub-matrices get different values.

Let us now consider that three new functions operating over two-dimensional arrays are added to Gecode/J. A function to get the rows, another to get the columns and a third one to get sub-matrices. Figure 4.38 depicts the corresponding extension file. The parameter **mat** corresponds to the matrix on which the function acts, **i** and **j** are the indexes of the row and of the column to be obtained, respectively. The third function has four parameters, the pair (**i1**,**j1**) represents the coordinates of the upper-left corner of the sub-matrix and the pair (**i2**,**j2**) represents the lower-right corner of the sub-matrix.

The resulting model using these new functions is depicted in Figure 4.39. Here, we can see that the model has been defined in a more concise and elegant way. In addition, the use of the **alldifferent** constraint will improve the resolution process of the problem.

Data File

```

1.  int s := 3;
2.  int n := 9;
3.  int Sudoku.puzzle := [[_, _, _, _, _, _, _, _, _],
                           [_, 6, 8, 4, _, 1, _, 7, _],
                           [_, _, _, _, 8, 5, _, 3, _],
                           [_, 2, 6, 8, _, 9, _, 4, _],
                           [_, _, 7, _, _, _, 9, _, _],
                           [_, 5, _, 1, _, 6, 3, 2, _],
                           [_, 4, _, 6, 1, _, _, _, _],
                           [_, 3, _, 2, _, 7, 6, 9, _],
                           [_, _, _, _, _, _, _, _, _]];

```

Model File

```

1.  import Sudoku.dat;
2.
3.  main class Sudoku {
4.
5.      int puzzle[n,n] in [1,n];
6.
7.      constraint differentInRowsAndColumns {
8.          forall(k in 1..n, i in 1..n, j in i+1..n) {
9.              puzzle[k,i] != puzzle[k,j];
10.             puzzle[i,k] != puzzle[j,k];
11.          }
12.      }
13.
14.      constraint differentInSubMatrices {
15.          forall(x1 in 1..s, y1 in 1..s, x2 in 1..s) {
16.              forall(y2 in 1..s, x3 in 1..s, y3 in 1..s) {
17.                  if(x2 != x3 and y2 != y3)
18.                      puzzle[(x1 - 1) * s + x2, (y1 - 1) * s + y2] !=
19.                      puzzle[(x1 - 1) * s + x3, (y1 - 1) * s + y3];
20.              }
21.          }
22.      }
23.  }

```

Figure 4.37 – The Sudoku problem.

```

1.  GecodeJ {
2.      Constraint {
3.          lexOrder(a,b) -> "gecodeJLexicalOrdering($a$, $b$)";
4.      }
5.      Function {
6.          getRow(mat,i)          -> "gecodeJGetRow($mat$, $i$)";
7.          getColumn(mat,j)       -> "gecodeJGetColumn($mat$, $j$)";
8.          getSubMatrix(mat,i1,i2,j1,j2) -> "gecodeJGetSubMatrix($mat$, $i1$, $i2$, $j1$, $j2$)";
9.      }
10. }
11. ...

```

Figure 4.38 – Adding new functions.

```

1.  main class Sudoku {
2.
3.      int puzzle[n,n] in [1,n];
4.
5.      constraint differentInRowsAndColumns {
6.          forall(i in 1..n) {
7.              alldifferent(getColumn(puzzle, i));
8.              alldifferent(getRow(puzzle, i));
9.          }
10.     }
11.
12.     constraint differentInSubMatrices {
13.         forall(i in 1..s, j in 1..s)
14.             alldifferent(getSubMatrix(puzzle,(i-1)*s + 1,i*s,(j-1)*s + 1,j*s));
15.     }
16. }

```

Figure 4.39 – Using the new functions in the Sudoku problem.

4.2.7.3 Adding heuristic orderings and consistency levels

Extensibility for heuristic orderings and consistency levels is also provided. Three new blocks can be added to the extension file: a **Variable-Ordering** block, a **Value-Ordering** block, and a **Consistency-Level** block. As an example, let us consider that new solving options are introduced in the Gecode/J solver. A variable ordering called **BVAR_NONE**, which selects the leftmost variable. A value ordering called **BVAL_SPLIT_MIN**, which selects the first value of the lower half of the domain; and the **ICL_VAL** consistency level, which performs the Gecode value consistency [WW]. The corresponding extension file and the **Mixer** class tuned with the new options are shown in Figure 4.40 and in Figure 4.41, respectively.

```

1.  GecodeJ {
2.      Constraint {
3.          ...
4.          Variable-Ordering {
5.              first -> BVAR_NONE;
6.          }
7.          Value-Ordering {
8.              lower-half -> BVAL_SPLIT_MIN;
9.          }
10.         Consistency-Level {
11.             value -> ICL_VAL;
12.         }
13.     }

```

Figure 4.40 – Adding new heuristic orderings and consistency levels.

```

1.  main class Mixer [first,lower-half] {
2.      [value] Vessel v;
3.      Agitator a;
4.      constraint design {
5.          [value] a.i.rps <= v.diameter/a.i.diameter;
6.          [value] a.i.diameter <= a.i.ratio*v.diameter;
7.      }
8.  }

```

Figure 4.41 – The tuned mixer class.

4.3 The s-COMMA GUI

The s-COMMA GUI is the graphical user interface for the s-COMMA language. The visual language of the s-COMMA GUI provides a more concise perception of models, allowing to state problems via two kinds of graphical artifacts: Data artifacts and class artifacts (see Figure 4.42).

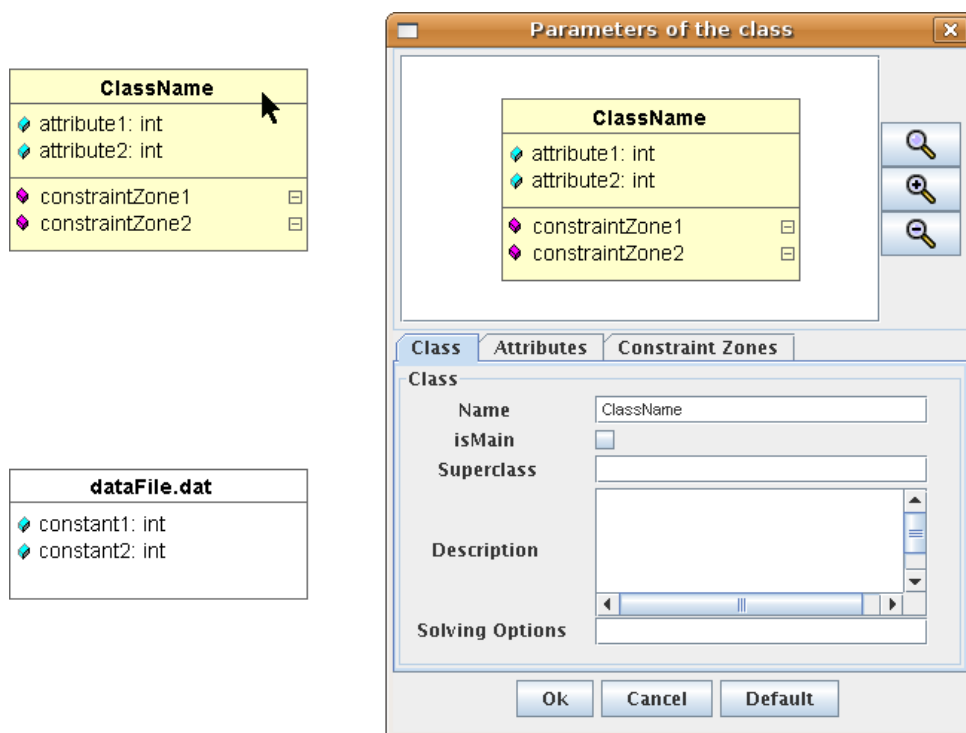


Figure 4.42 – Class and data artifacts.

Class artifacts correspond to the graphical representation of classes. Class artifacts have by default three compartments, the upper compartment for the class name, the middle compartment for attributes and the bottom one for constraint zones. By clicking on the class artifact its specification can be opened to define its properties, its attributes and constraint zones. Several class properties can be defined, for instance, the name, if the class is a main class, a superclass, a description and the solving options. Relationships can be used to define inheritance or

composition between classes. Data files are represented by data artifacts, being composed of two compartments, one for the file name and another for both the constants and variable assignments.

Note

The graphical artifacts of the s-COMMA GUI have been designed as an extension of the UML class artifact provided by the UML Infrastructure Library Basic Package. This ensures the s-COMMA GUI notation to be entirely supported by the UML Infrastructure Specification [19].

Figure 4.43 shows a snapshot of the s-COMMA GUI where the stable marriage problem is represented by a class diagram. This diagram is composed of three class artifacts, one to represent men, another to represent women, and a third one to describe the stable marriages. The composition relationships are depicted through connections among classes. The right-panel of the tool shows the corresponding s-COMMA textual version, which is instantly generated once graphical artifacts are stated on the drawing frame.

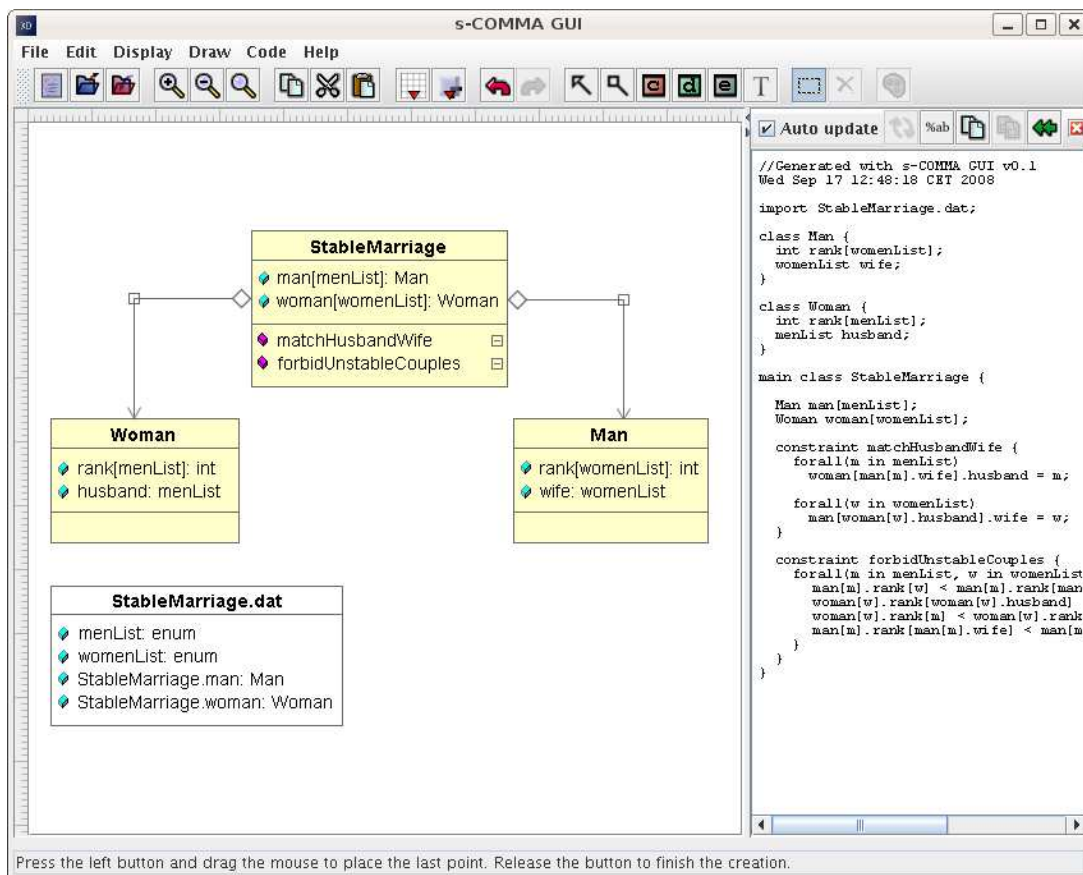


Figure 4.43 – The stable marriage problem on the s-COMMA GUI.

The `StableMarriage` class has two attributes, one array to represent the group of men and other array to represent the group of women. Attributes can be stated using the attribute panel of the class window illustrated in Figure 4.44. The attribute panel permits to add, modify and delete attributes. Each attribute can be defined by giving its type, name, and domain. To define attributes as one-dimensional arrays the left array field must be filled with its size. Matrices are defined filling both array fields, the left one for the row size and the right one for the column size. In the example, the attribute `man` is an array having `Man` as its type and `menList` as its size. The domain fields are not filled since the attribute is an object array. The check box allows one to define set variables and the last field is used to define an optional consistency level to be used for objects.

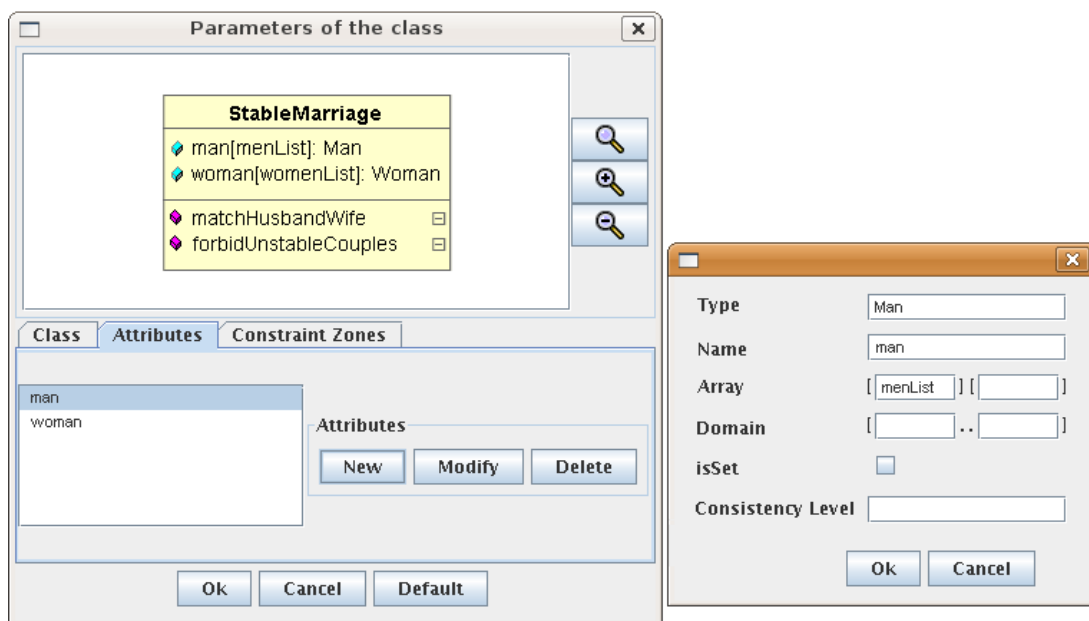


Figure 4.44 – Attributes on the s-COMMA GUI.

Constraint zones are stated in a similar way. Figure 4.45 shows the constraint zone panel, where constraint zones can be added, modified and deleted. Shortcut buttons are provided to generate a code framework to be then completed by the user, for instance to state loops, conditionals, optimization statements, and compatibility constraints. The constraints must be written by hand.

Both constants and variable assignments are stated in the data window. They are defined giving a type, a name, and a value. Figure 4.46 shows the enumeration constant `menList`, the value field is filled with the names of the group of men.

The s-COMMA GUI includes typical operations for handling projects, managing some preferences and printing draws and codes. Also, common shortcuts such as cut, copy, paste, undo and redo are provided. Buttons for changing the properties of the drawing frame (zoom-in, zoom-out, scaling the grid) have been considered as well (see Figure 4.47).

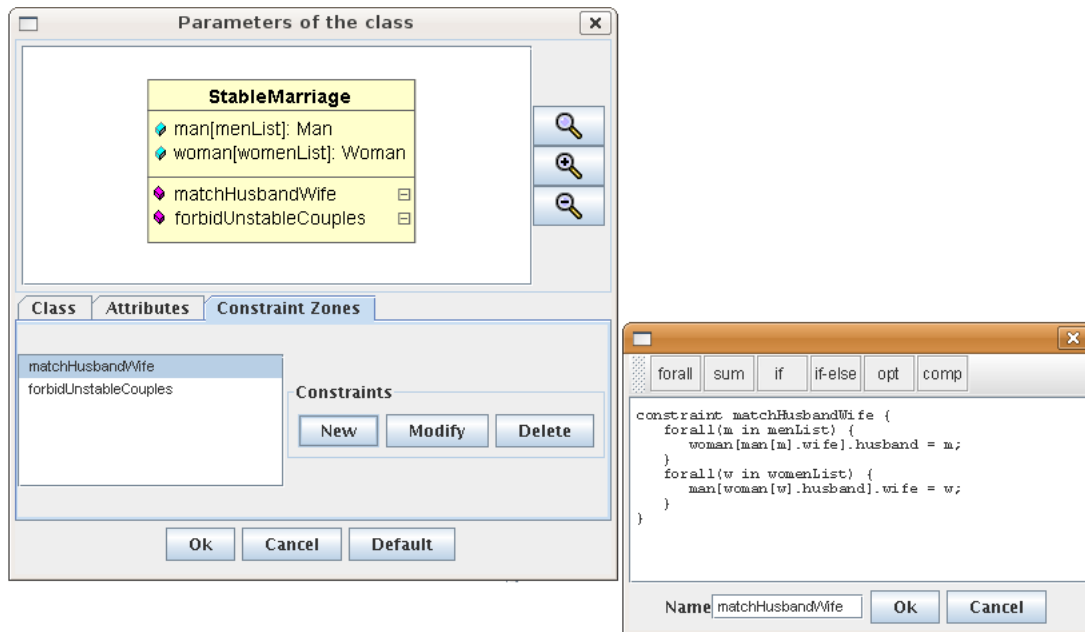


Figure 4.45 – Constraints on the s-COMMA GUI.

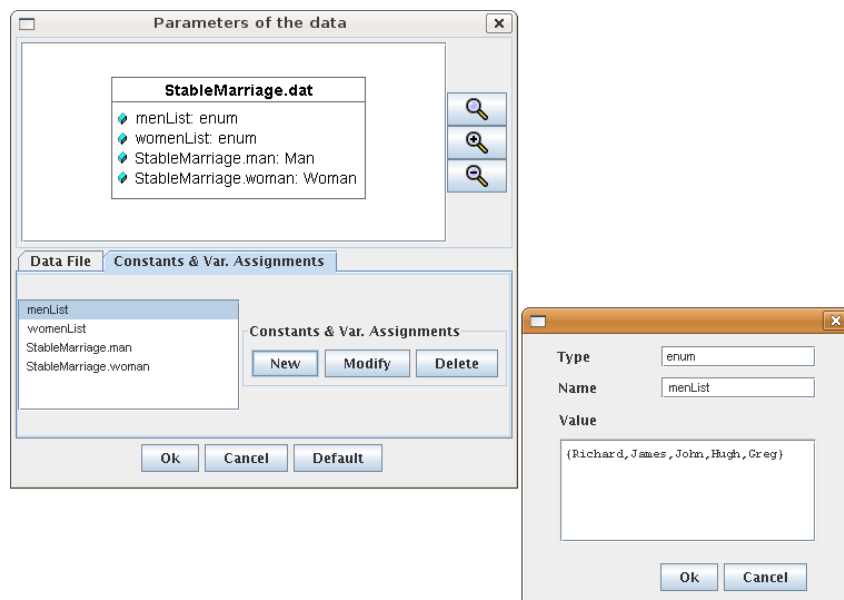


Figure 4.46 – Data files on the s-COMMA GUI.

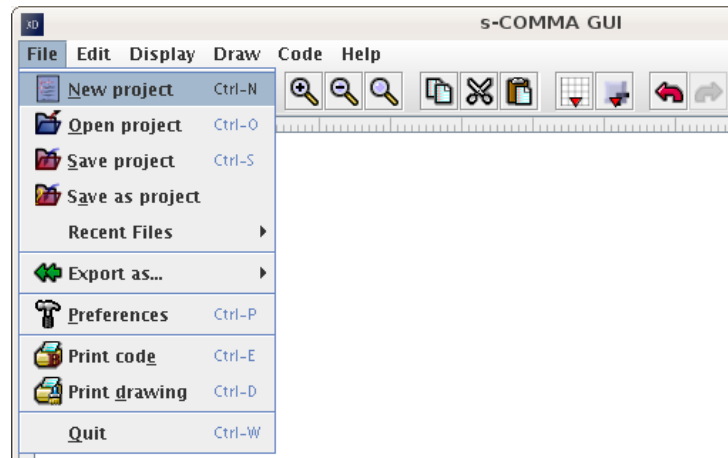


Figure 4.47 – Some shortcuts of the s-COMMA GUI.

4.4 Summary

In this chapter we have presented the s-COMMA language and the s-COMMA GUI. We have illustrated several CP models through the s-COMMA language, showing that the expressiveness offered is suitable for different kinds of problems. The object-oriented style provided is useful for getting elegant and modular models. These models can be tuned with a simple formalism to get efficient solving processes. This formalism permits to define heuristic orderings as well as the consistency level of constraints. The expressiveness of the base language can be extended, an extension file can be defined to add new functions, constraints, and solving options to the language. Finally, the s-COMMA GUI provides a visual and more concise representation of models.

The next chapter focuses on the transformation process from graphical artifacts to solver models. We present the tools and techniques involved in the transformation, and we illustrate several examples of the platform implementation.

Mapping Models to Solvers

The main purpose of our approach is to transform a solver-independent model to different solver-dependent models. That requires (1) to translate languages, from high-level modeling languages to lower level constraint solving languages or computer programming languages, and (2) to modify model structures according to the capabilities of solvers, for instance to unroll loops, or to flatten an object-oriented composition.

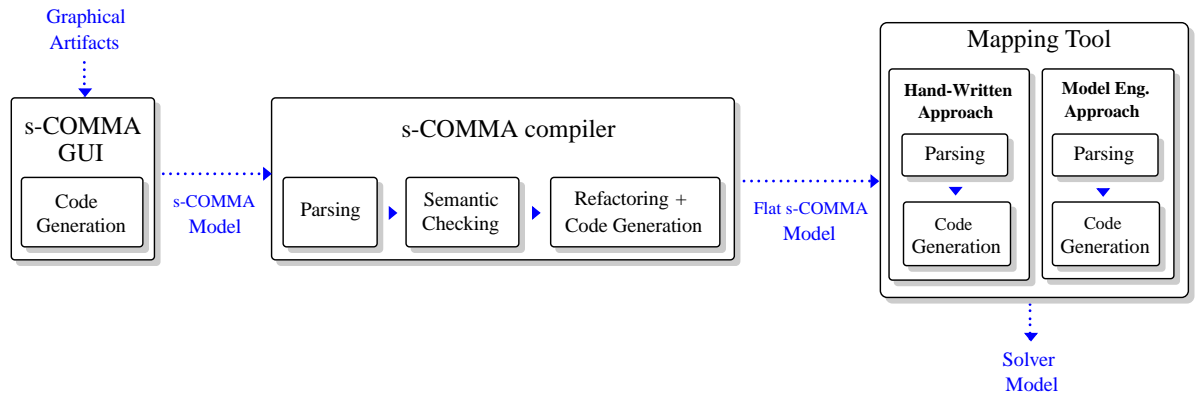


Figure 5.1 – The s-COMMA architecture.

To support these requirements we introduce a new solver-independent architecture able to perform the whole transformation in three main stages (see Figure 5.1). Firstly, the s-COMMA GUI generates the corresponding s-COMMA model by means of a set of Java packages and procedures. In the second stage, the s-COMMA model is parsed, semantically checked and then transformed to the intermediate Flat s-COMMA model. During this translation, several refactoring steps are performed to be closer to the solver level. The idea is to simplify the mapping process to the solver model, and consequently to facilitate the integration of new solvers to the platform. In the third and last stage, the Flat s-COMMA model is parsed and transformed to the solver model. This stage is performed by our so-called mapping tool, in which two transformation approaches can be identified. The first approach has been built using a parsing tool and hand-written Java procedures, and the second approach has been designed and implemented using techniques and tools from the model engineering world.

In this chapter, we present the complete transformation process from graphical artifacts to solver programs. The first section is devoted to the transformation from the s-COMMA GUI to the

s-COMMA model. Some Java classes and procedures are illustrated to provide an overview of that transformation. The following section presents the transformation from s-COMMA to Flat s-COMMA. The technical aspects of the parsing, semantic checking, and refactoring steps are illustrated by means of several examples. We believe this is of interest to designers of further CP languages. The last section targets the design of the mapping tool. The grammar approach and the model-driven approach are illustrated and compared.

5.1 From s-COMMA GUI to s-COMMA

The prototype implementation of the s-COMMA GUI is completely written in Java (about 30000 code lines including the s-COMMA compiler) and the Swing widget library is used to design the graphical interfaces. Three main Java packages can be identified to support the transformation from graphical artifacts to s-COMMA models (see Figure 5.2):

- **dialogBoxes**: contains the dialog boxes that allow users to fill the information of the model.
- **artifacts**: contains the classes that allow users to create, to drag, and to resize the artifacts in the drawing pane of the s-COMMA GUI.
- **modelInformation**: contains the classes that store the information of the model (e.g., constants, classes, attributes and constraint zones).

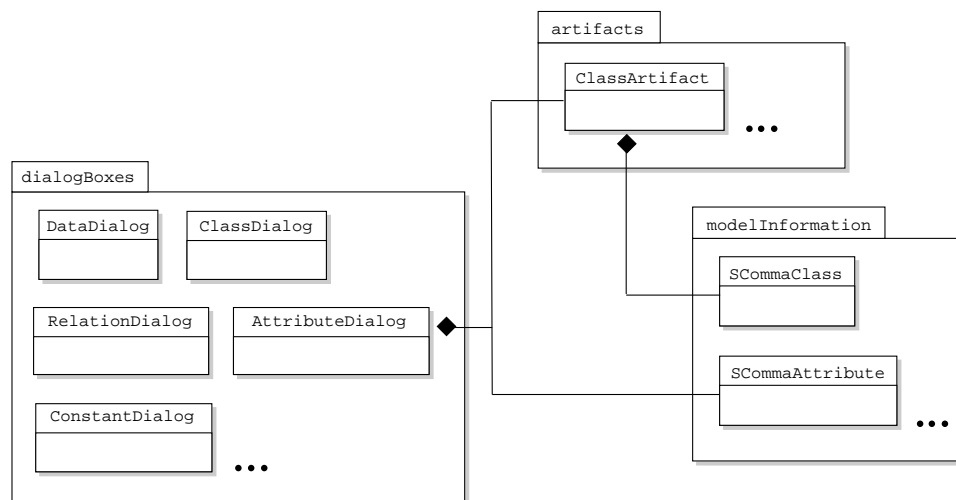


Figure 5.2 – s-COMMA GUI Java packages.

In the **dialogBoxes** package, each graphical element appearing in a model has a dialog box defined by a Java class. For instance, one class for data artifacts, one for class artifacts, and another for relationships. Every element contained in a graphical artifact has a dedicated class as well. For instance, one class to manage the attributes, one to manage the constants and one for the constraint zones. Each of these classes is composed of the common methods to define the position of frames, text fields and buttons in the layout of the dialog box. These buttons trigger actions to add, delete or modify elements of the model. Dialog box classes contain objects, from

both the `artifact` package and the `modelInformation` package. These objects are responsible for gathering the information provided by the user and for storing it in order to generate the corresponding `s-COMMA` textual version.

To show the interaction among these components, let us consider the addition of a `s-COMMA` attribute to a `s-COMMA` class artifact. Four classes participate in this process: `AttributeDialog`, `ClassArtifact`, `SCommaClass`, and `SCommaAttribute`. The packages owning these classes and the relationships among them can be seen in Figure 5.2. The goal is to capture the information of the `s-COMMA` attribute from the corresponding dialog box, and then storing it in the `modelInformation` package.

The process begins when the user fills the properties (type, name, array dimensions, if the variable is defined as a set, domain, and consistency level) of the `s-COMMA` attribute through the dialog box. These properties are captured between lines 6 and 12 of the `AttributeDialog` class (see Figure 5.3). The `getText()` method returns the string provided by the user in the text field, and `isSelected()` returns `true` whether the check box is checked. At the end of the file, `attribute` is added to an instance of a `ClassArtifact` class called `clArtifact`.

```

1.  public class AttributeDialog extends JDialog implements ActionListener {
2.      ...
3.      private ClassArtifact clArtifact;
4.      private SCommaAttribute attribute;
5.      ...
6.      attribute.setType(attType.getText());
7.      attribute.setName(attName.getText());
8.      attribute.setOneDimArray(attOneDimArray.getText());
9.      attribute.setTwoDimArray(attTwoDimArray.getText());
10.     attribute.setIsSet(attIsSet.isSelected());
11.     attribute.setDomain(attDomMin.getText(),attDomMax.getText());
12.     attribute.setConsLevel(attConsLevel.getText());
13.     clArtifact.addAttribute(attribute);
14.     ...

```

Figure 5.3 – The `AttributeDialog` class.

The `ClassArtifact` captures the `attribute` object via the `addAttribute` method (line 5 of Figure 5.4), which then stores it in an instance of a `sCommaClass`.

```

1.  public class ClassArtifact extends ArtifactDrawing {
2.      ...
3.      private SCommaClass sCClass = new SCommaClass();
4.      ...
5.      public void addAttribute(SCommaAttribute attribute) {
6.          sCClass.addAttribute(attribute);
7.      }
8.      ...

```

Figure 5.4 – The `ClassArtifact` class.

The `sCommaClass` is illustrated in Figure 5.5. It is composed of attributes (lines 3 to 9) to store the properties of `s-COMMA` classes (e.g. description, name, etc) and methods to manage these attributes (lines 11 to 20). The process finish when the `s-COMMA` attribute is stored in the

model information package. This is done via the `addAttribute` method, which adds the input parameter `att` (received from the `ClassArtifact` class at line 6) to the `attributes` array (line 12).

```

1.  public class SCommaClass {
2.
3.      private String description;
4.      private String name;
5.      ...
6.      private ArrayList<SCommaAttribute> attributes =
7.          new ArrayList<SCommaAttribute>();
8.      private ArrayList<SCommaConstraintZone> cZones =
9.          new ArrayList<SCommaConstraintZone>();
10.
11.     public void addAttribute(SCommaAttribute att) {
12.         attributes.add(att);
13.     }
14.     ...
15.     public void deleteAttribute(String name) {
16.         for (SCommaAttribute a : attributes)
17.             if (a.getName().equals(name))
18.                 attributes.remove(a);
19.     }
20.     ...

```

Figure 5.5 – The `SCommaClass` class.

Once the information obtained from the dialog boxes is stored in the `modelInformation` package, it can be retrieved to generate the corresponding `s-COMMA` textual model. This process is automatically done when the user closes the dialog box. On the left side of Figure 5.6, the Java method to produce the code of `s-COMMA` classes is illustrated. An example of a generated `s-COMMA` class is shown on the right side of the same figure.

```

1.  public String getCode() {
2.
3.      StringBuffer str = new StringBuffer();
4.
5.      str.append(generateDesc()); //This class represents a Turbo Engine
6.      str.append(generateIsMain()); main class TurboEngine extends Engine [bound] {
7.      str.append("class ");
8.      str.append(sCClass.getName()); ...
9.      str.append(generateSuperClass());
10.     str.append(generateSolvingOpt());
11.     str.append(" {\n");
12.     str.append(generateAttributes()); int diameter in [34, 250];
13.     str.append(generateConstraintZones()); constraint distance { ... }
14.     str.append("}");
15.     return str.toString();
16. }

```

Figure 5.6 – The `getCode` method.

The code is built from a systematic union of strings. The description of the `s-COMMA` class is the first string to be appended. The code of the class follows. The class header consists of an

optional token `main`, a `class` token, a class name, an optional inheritance definition, and solving options. The body of the class is enclosed with curly bracket symbols (`{}`), which are appended at lines 11 and 14. Within the class body, the attributes and constraint zones are added.

5.2 From s-COMMA to Flat s-COMMA

The transformation from s-COMMA to Flat s-COMMA is the most complex part of the whole process. Several transformations must be done so as to facilitate the task of solver-translators, and also to ease the integration of new solvers to the platform. Three main tasks are identified: parsing, semantic checking and refactoring to Flat s-COMMA.

5.2.1 Parsing

The parsing process is responsible for checking the correctness of the syntax of the input string, and for building an abstract syntax tree (AST) to be explored in the following phases. The parsing process consists of two main tasks: the lexical analysis and the syntactic analysis. The lexical analysis must detect tokens from the input string, and the syntactic analysis determines whether these tokens form valid expressions conform to the grammar of the language. The implementation of these two main tasks has been supported by the ANTLR language recognition tool [44]. An ANTLR lexer performs the lexical analysis and an ANTLR parser deals with the syntactic checking.

5.2.1.1 Lexer

The lexer is able to generate the tokens given an input string by means of a set of reserved word definitions and regular expressions (also called rules in ANTLR). Figure 5.7 illustrates a fragment of the lexer file. The reserved words of the language are defined in a specific block called `tokens` (to avoid ambiguities with identifiers). Identifiers are used for giving a name to language constructs that require it, for instance a class name, a variable name, a constraint zone name, etc. The rule to recognize them is stated at line 12. The option `testLiterals=true` is used to explicitly state that identifiers must be checked with respect to the reserved words of the `tokens` block. The `paraphrase` option is used for showing "`an identifier`" in error messages instead of the name of the token (an error message can be seen in Figure 5.14). The `IDENT` rule states that an identifier must firstly be composed of a `LETTER` or an underscore symbol followed by a set of zero or more `LETTER`, `DIGIT` or underscore symbols. The rules to recognize letters and digits are defined next, the double dot operator (`'.'`) is used to consider a range of characters. In the following lines, several other rules are declared, for instance to recognize the punctuation symbols (lines 24 to 26), the brackets (lines 29 to 31) and the operators (lines 34 to 36).

Dealing with rule ambiguities

The rule to recognize numbers (reals and integers) is shown in Figure 5.8. This process is more complex since the number of tokens to check may be undetermined. For instance, to be able to recognize `5.2` as a real (and not as an integer) it should be necessary to detect just two tokens (2-lookahead), `5` as a digit and then the dot as a punctuation symbol.

Note

The lookahead determines the number of tokens to be recognized for matching a rule, it is normally set to 2. Bigger lookaheads may lead to slower parsing processes.

```

1.  tokens
2.  {
3.      RES_IMPORT      = "import" ;
4.      RES_MAIN        = "main" ;
5.      RES_CLASS       = "class" ;
6.      RES_EXTENDS     = "extends" ;
7.      RES_CONSTRAINT  = "constraint" ;
8.      RES_FORALL      = "forall" ;
9.      ...
10. }
11.
12. IDENT
13.     options {testLiterals=true; paraphrase="an identifier";}
14.     : (LETTER|'_' ) (LETTER|DIGIT|'_' ) *
15.     ;
16.
17. LETTER : 'a'..'z'
18.       | 'A'..'Z'
19.       ;
20.
21. DIGIT  : '0'..'9' ;
22. ...
23.
24. PUN_SEMI_COLON : ';' ;
25. PUN_COMMA      : ',' ;
26. PUN_DOT        : '.' ;
27. ...
28.
29. BRA_CURLY_OPEN  : '{' ;
30. BRA_CURLY_CLOSE : '}' ;
31. BRA_ROUND_OPEN  : '(' ;
32. ...
33.
34. OP_PLUS      : '+' ;
35. OP_MINUS     : '-' ;
36. OP_MULTIPLICATION : '*' ;
37. ...

```

Figure 5.7 – Tokens and rules in the ANTLR lexer of *s-COMMA*.

However, a 2-lookahead may not be enough to match different rules sharing more than two initial tokens. For example, a real number with an integer part having two or more digits cannot be recognized since the two initial digits may belong as well to an integer number as to a real number. This kind of ambiguities can be avoided by using a syntactic predicate [PQ94], which is a specific ANTLR feature that permit us to arbitrary extend the lookahead of a determined rule. Syntactic predicates are defined as $(a) \Rightarrow a|b$, where a is the rule to be matched with an extended lookahead, and b is the rule to be recognized if a cannot be matched. For instance, the

rule to define a `NUMBER` is composed of a statement to recognize reals (line 2) and a statement to recognize integers (line 3). The first statement defines that a real is composed of a set of one or more digits followed by a dot and another set of one or more digits. The second statement defines that an integer is composed of one or more digits. The rule first tries to match reals, if this occurs the token is set as a real literal (`LIT_REAL`). Otherwise, the rule recognizes an integer.

```
1.  NUMBER : ((DIGIT)+ PUN_DOT (DIGIT)+) =>
2.          (DIGIT)+ PUN_DOT (DIGIT)+ { $setType (LIT_REAL);}
3.          | (DIGIT)+ { $setType (LIT_INT);}
4.          ;
```

Figure 5.8 – The lexer rule to define numbers.

Note

The use of syntactic predicates generates a grammar called `pred-LL(K)`, where `K` denotes the lookahead.

5.2.1.2 Parser

The parser is able to perform the syntactic analysis by matching a set of rules composed of the tokens stated in the lexer file. These rules are built conform to the grammar of the language and they are responsible for capturing the grammatical structure of the analyzed string by producing an abstract syntactic tree (AST).

In ANTLR, ASTs are built using a Lisp-based notation, ‘#’ being the operator to define tree structures. For instance, `##(a,b,c)` corresponds to a tree where `a` is the root, and `b` and `c` are its child nodes. For example, consider the first rule showed in Figure 5.9, which matches an addition between two integer tokens. The AST for this rule is built using `OP_PLUS` as the root, and the integer tokens as child nodes. A simpler equivalent version of this rule (line 5) can be stated by using the ‘^’ operator. The corresponding AST is shown on the right side of the figure. Non leaf nodes are represented by a folder icon and leaf nodes by a file icon.

```
1.  add_expr : e1:LIT_INT op:OP_PLUS e2:LIT_INT
2.          { ## = ##(op, #e1, #e2);}
3.          ;
4.
5.  add_expr : LIT_INT OP_PLUS^ LIT_INT
6.          ;
```

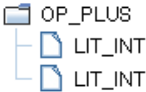


Figure 5.9 – Three parser rules in ANTLR.

In the case of rules having no appropriate token to be used as AST root, it is possible to introduce a root token. In Figure 5.10, the rule `identList` is defined as a set of one or more `IDENT` tokens, and no token is suitable to become the AST root. A new token called `LIST` is introduced, and the tree is formed with the `LIST` token as root and the set of `IDENT` tokens as its child nodes.

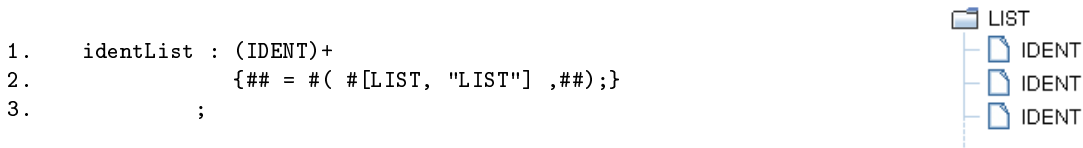
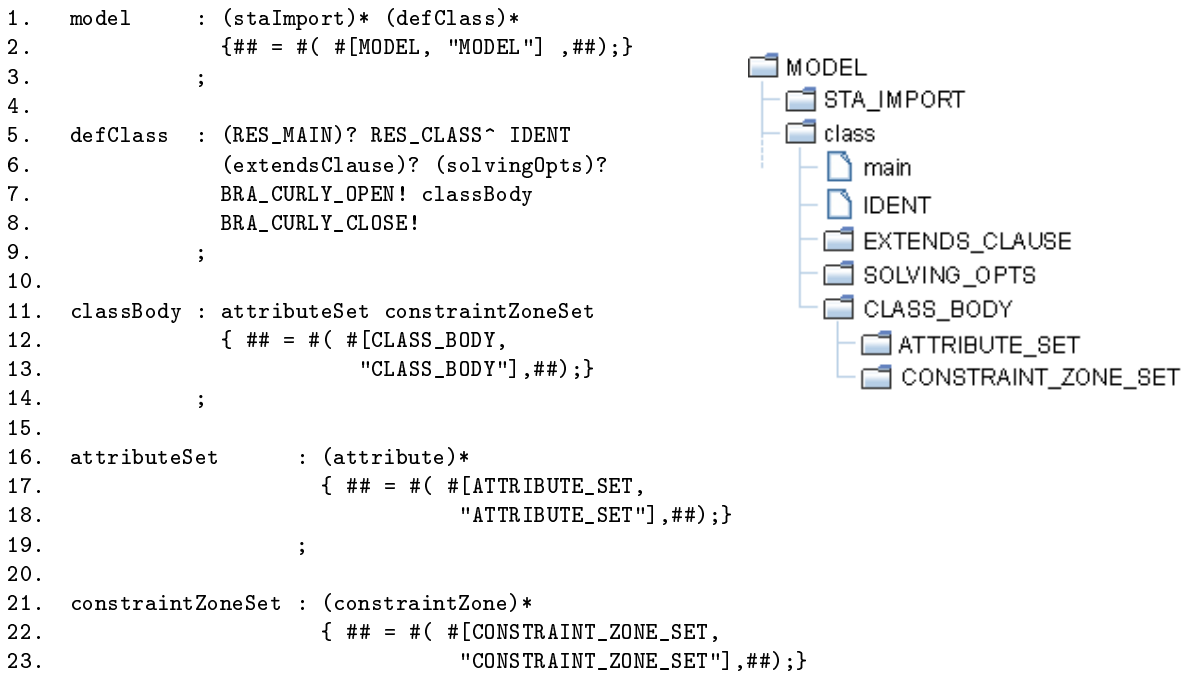


Figure 5.10 – Introducing a proper tree node.

Figure 5.11 illustrates five rules of the parser file of *s-COMMA*. Such rules are composed of tokens, calls to other rules and statements for building ASTs. The first rule consists of two rule calls (*staImport* and *defClass*) and a statement to define the root of the AST (`{## = #(#[MODEL, "MODEL"] ,##);}`). The rule states that a model is composed of a set of zero or more import statements followed by a set of zero or more class definitions. Let us notice that lower case is used to rule names in order to differentiate them from tokens.

Figure 5.11 – Parser rules of *s-COMMA*.

Lines 5 to 8 describe the rule for recognizing *s-COMMA* classes. A class definition begins with the **main** reserved word given by the `RES_MAIN` token. The use of this token is optional, denoted by the ‘?’ symbol. The `RES_MAIN` token is followed by the **class** reserved word and by an identifier corresponding to the name of the class. The `extendsClause` rule is also optional, being called only if the *s-COMMA* class owns a superclass. Then, the `solvingOpts` rule call is used to recognize the solving options stated in the class. The body of the class is defined within curly brackets. Each bracket token is postfixed with a ‘!’ symbol. Such a symbol defines the no inclusion of a token in the ASTs. It is used for tokens giving no relevant information for the parsing process.

The body of a class is defined as a set of attributes and a set of constraint zones. Attributes are recognized by the first rule of Figure 5.12. Such a rule states that the declaration of an attribute begins with its consistency level. This rule call is optional and followed by the type of the attribute. The reserved word **set** is next defined, it is also optional and it is used to state set variables. The name of the variable follows as an **IDENT** token. Then, the optional **array** rule call is used to define arrays. The domain of the variable is defined by the reserved word **in** followed by a call to the **domain** rule. The declaration must be terminated by a semicolon symbol.

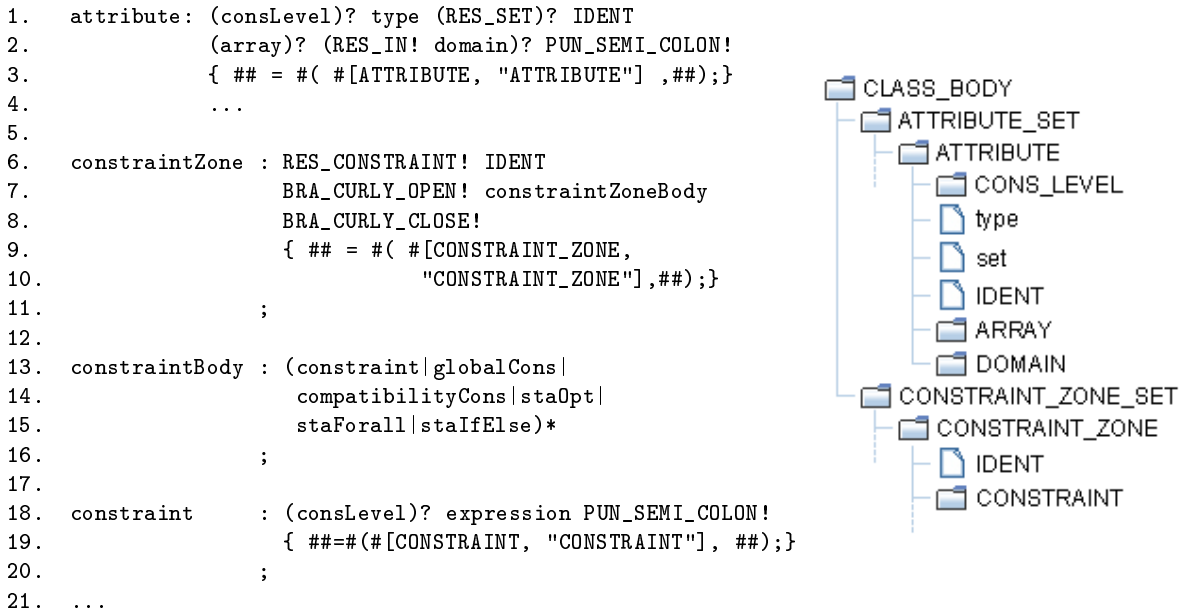


Figure 5.12 – Parser rules of s-COMMA.

A constraint zone declaration (line 6) must begin with the reserved word **constraint** given by the **RES_CONSTRAINT** token. This token is followed by **IDENT**, which represents the constraint zone name. The constraint zone body is defined inside curly brackets. It can be composed of several constructs, i.e. constraints, global constraints, compatibility constraints, an optimization statement, forall loops and conditionals. A constraint is defined as an expression, prefixed by its optional consistency level and finished by a semicolon.

Expressions are recognized using a set of rules (see Figure 5.13), each one including one or more operators having the same priority. The idea is to perform calls from one rule to the next one respecting the priority of these operators (from lower to higher). Each rule is of the form $a : b (op\ b)^*$, where a is the name of the rule, b is a call to the next rule, and op is the operator. The first rule includes the lowest priority operator (the operator priorities can be found in Table 4.1), which corresponds to the equivalence (\leftrightarrow) symbol. The next rule includes the implication (\rightarrow) and reverse-implication (\leftarrow) operators. Several rules follow respecting the operator precedences. The rule stated at line 32 deals with unary arithmetic operators. If a unary minus operator is detected, it is not included in the AST, but the operand is captured in an additional node called **OP_UN_MINUS** (this is done to improve readability of ASTs). In the case of detecting a unary plus operator (which is optional), it is not included in the AST, but no additional node is used since this operator has no relevance within expressions.

The last rule deals with operands. An operand may be a value (integer, real or boolean), an identifier (e.g. a variable, a constant), an access (an access to the attribute of an object or an access to an array), or a function (e.g. a sum loop, the cardinality of a set, etc). Finally, the operand can also be an expression enclosed with parentheses.

1.	<code>expression</code>	: <code>exprIMP</code>	<code>((OP_EQV^ exprIMP)*)</code>
2.			<code>;</code>
3.	<code>exprIMP</code>	: <code>exprOR</code>	<code>((OP_IMP^ OP_RIMP^) exprOR)*</code>
4.			<code>;</code>
5.	<code>exprOR</code>	: <code>exprAND</code>	<code>((RES_XOR^ RES_OR^) exprAND)*</code>
6.			<code>;</code>
7.	<code>exprAND</code>	: <code>exprNot</code>	<code>(RES_AND^ exprNot)*</code>
8.			<code>;</code>
9.	<code>exprNot</code>	: <code>(RES_NOT^)* exprRel</code>	
10.			<code>;</code>
11.	<code>exprRel</code>	: <code>exprSetRel</code>	<code>((OP_EQUAL^ </code>
12.			<code>OP_DISTINCT^ </code>
13.			<code>OP_LESS_THAN^ </code>
14.			<code>OP_GREATER_THAN^ </code>
15.			<code>OP_LESS_THAN_OR_EQUAL^ </code>
16.			<code>OP_GREATER_THAN_OR_EQUAL^)</code>
17.		<code>exprSetRel</code>	<code>)*</code>
18.			<code>;</code>
19.	<code>exprSetRel</code>	: <code>exprSetOp</code>	<code>((OP_SUBSET^ OP_SUPERSET^)</code>
20.		<code>exprSetOp</code>	<code>)*</code>
21.			<code>;</code>
22.	<code>exprSetOp</code>	: <code>exprSum</code>	<code>((OP_UNION^ OP_DIFF^ OP_SYMDIFF^) exprSum)*</code>
23.			<code>;</code>
24.	<code>exprSum</code>	: <code>exprProduct</code>	<code>((OP_PLUS^ OP_MINUS^) exprProduct)*</code>
25.			<code>;</code>
26.	<code>exprProduct</code>	: <code>exprInter</code>	<code>((OP_MULTIPLICATION^ OP_DIVISION^) exprInter)*</code>
27.			<code>;</code>
28.	<code>exprInter</code>	: <code>exprExpon</code>	<code>((OP_INTERSECT^) exprExpon)*</code>
29.			<code>;</code>
30.	<code>exprExpon</code>	: <code>unMinus</code>	<code>(OP_EXPON^ unMinus)*</code>
31.			<code>;</code>
32.	<code>unMinus</code>	: <code>(OP_MINUS! exprUnit)</code>	
33.			<code>{ ##=#([OP_UN_MINUS, "OP_UN_MINUS"], ##) ;}</code>
34.			<code> ((OP_PLUS!)? exprUnit)</code>
35.			<code>;</code>
36.	<code>exprUnit</code>	: <code>value IDENT access function </code>	
37.			<code>(BRA_ROUND_OPEN expression BRA_ROUND_CLOSE)</code>
38.			<code>;</code>

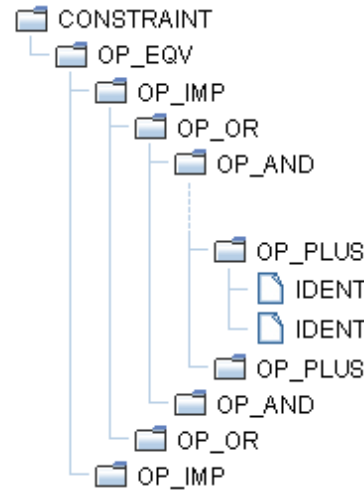


Figure 5.13 – The rule to recognize expressions.

Syntactic Errors

Let us notice that syntactic errors are automatically handled by ANTLR. When the parsing rules are not able to match a given input string, the relevant information of the syntactic error is gathered and displayed to the user. An example is shown in Figure 5.14. The error has been generated from a model file having a class declaration in which the name is missing (`class {`). The error message contains the file name, the line number, and the column number related to the

conflict. The paraphrase "an identifier" defined in the lexer has been used to denote `IDENT` as the missing token.

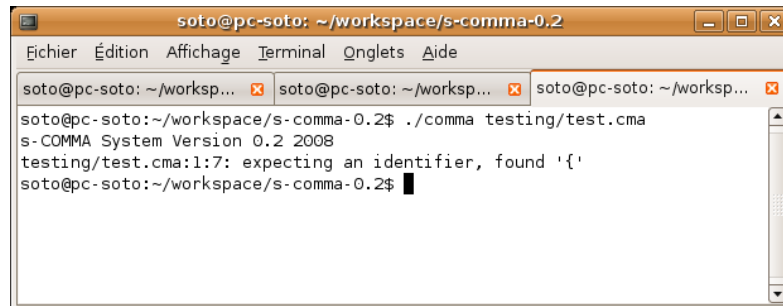


Figure 5.14 – A syntactic error.

5.2.2 Semantic Checking

The lexical and syntactic analysis are unable to detect all the errors appearing in a model. The lexical analysis detects the tokens and the syntactic analysis groups these tokens into grammatical structures. The role of the semantic analysis is to check the “meaning” of these grouped tokens conform to the semantic rules of the language. The semantic checking is performed by exploring the AST and by building a symbol table to store the relevant information for the checking. In the `s-COMMA` architecture, the exploration of the AST is done by ANTLR top-down tree walkers. The notation used to define the AST exploration is analogous to the one used for the AST construction. For instance, a tree composed of a root and two child nodes can be explored by the rule `#(A b c)`, where `A` is the name of the root token and `b` and `c` are calls to the exploring rules of the left and right subtree, respectively.

Performing the whole semantic checking process requires to combine the AST exploration with another routines. For instance to create the table of symbols, to handle the corresponding semantic errors, and to build intermediate representations. These routines are implemented in Java and ANTLR permits calling them, embedded in code blocks, from the exploration rules.

Note

An intermediate representation of the `s-COMMA` model is built during the semantic checking. This intermediate representation is stored in several Java objects, which are then explored to build the Flat `s-COMMA` model. Details about code generation mechanisms can be seen in Section 5.3.1.3.

Figure 5.15 depicts the rule to explore class definitions. The rule states that the first node to be explored must be the reserved word `class`. The first child of that node corresponds to the optional `main` token. Such a node is stored in a local variable called `isMain`, which is then used as input parameter of the Java method call `addClass`. The `checkMainClass` method is called to ensure that models own at most one main class. The next node to be explored corresponds to an `IDENT` token, being also stored in a local variable. This local variable is the input parameter of the `setIdClass` method call, which sets the `id` of the class in a global variable called `idClass`. Such a global variable will be used in further exploration rules. In the following line, two optional rule calls are stated. In the first one, the token of the reserved word `extends` is read, and the

name of the superclass is stored in `idSuperClass`. In the second one, the solving options are explored and stored. At line 4, the `addClass` method adds the class to the symbol table and to the intermediate representation of the model.

```

1.  defClass      : #(RES_CLASS ((isMain:RES_MAIN {this.checkMainClass()}))?
2.                  id:IDENT {this.setIdClass(Id)})
3.                  (RES_EXTENDS idSuperClass:IDENT)? (sOptClass:solvingOpts)?
4.                  {mI.addClass(isMain,id,idSuperClass,sOptClass);}
5.                  (classBody)))
6.                  ;

```

Figure 5.15 – Tree walker of s-COMMA.

Let us note that ANTLR is unable to automatically handle the semantic errors (as it does it for the syntactic errors), being necessary to define specific procedures to handle them. For instance, multiple class name declarations are checked within the `addClass` method (see Figure 5.16). This procedure firstly tests if there is no class previously declared using the same identifier. The `id` variable is a tree node containing the information of the token concerning the name of the class to be added, and `id.getText()` returns the name of the class. If the condition of the procedure is satisfied, the new class is added to model. Otherwise, an error message is triggered. The message is formatted by the `semanticError` method to display the relevant error information. The file name, the line number and the column number of the conflicting token are obtained from `id`. The error message is shown in Figure 5.17.

```

1.  public void addClass(AST isMain, AST id, AST idSuperClass, AST sOptClass) {
2.      if (!model().containsClass(id.getText())) {
3.          model().addClass(isMain,id,idSuperClass,sOptClass);
4.      } else {
5.          Message.semanticError("redeclaration of class '" + id.getText()
6.                                + "'", id);
7.      }
8.  }

```

Figure 5.16 – A Java procedure to check class redeclarations.

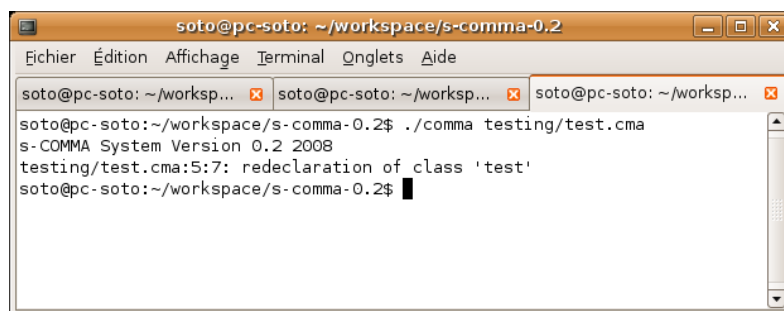


Figure 5.17 – A semantic error.

Handling Semantic Errors in a Second Top-Down Tree Exploration

All the potential semantic errors of a model cannot be detected in one top-down tree exploration. For instance, type checking cannot be performed if the information of all the classes is unavailable in the symbol table. As an example, consider the model shown in Figure 5.18. The tree walker begins by exploring the first class. The attribute `b` is recognized but the tree walker is unable to check its type since the class `B` has not been explored yet. Likewise, the structure of the access `b.a` cannot be checked either.

```
class A {
  B b;
  constraint cz {
    b.a < 2;
  }
}

class B {
  int a in [0,9];
}
```

Figure 5.18 – Two `s-COMMA` classes.

A common way used in object-oriented languages is to perform a second exploration of the AST. Figure 5.19 illustrates the rule of the second tree walker to check the type of attributes. The method call is embedded in the `type` rule, which acts when the type is defined as an `IDENT`. The method checks if the variable is correctly typed. There are two valid possibilities: the variable has been typed with an enumeration or it corresponds to an object instance.

```
1.  attribute : #(ATTRIBUTE ((consLevel)? type (RES_SET)? IDENT
2.             (array)? (domain)?));
3.
4.  type      : (TYPE_INT|TYPE_REAL|TYPE_BOOL
5.             |id:IDENT {vI.checkObjectOrEnumType(id)});
```

Figure 5.19 – The rule to check attributes in the second pass.

The rule to check constraints in the second tree parser is depicted in Figure 5.20. The rule begins by matching the `CONSTRAINT` node, which owns two children: the consistency level of the constraint and an expression. The correct formation of these expressions is validated by the `checkExpression` method (line 2). Finally, the constraint is stored in the intermediate representation. Expressions are read using one big rule (line 5). Every possible operator is explored with its corresponding child nodes, which are defined as expressions. At the end of the rule, the potential operands are explored (value, variable, access and function). Two methods check if the variables and the accesses have been correctly declared.

```

1.  constraint      : #(CONSTRAINT cLevel:consLevel exp:expression
2.                  {vI.checkExpression(exp);}
3.                  {sI.addConstraint(idClass,idConstraintZone,cLevel,exp);});
4.
5.  expression
6.      : #(OP_EQV          expression expression)
7.      | #(OP_IMP          expression expression)
8.      | #(OP_RIMP         expression expression)
9.      | #(RES_OR          expression expression)
10.     | #(RES_XOR          expression expression)
11.     | #(RES_AND          expression expression)
12.     | #(RES_NOT          expression)
13.     | #(OP_EQUAL        expression expression)
14.     | #(OP_DISTINCT     expression expression)
15.     | #(OP_LESS_THAN    expression expression)
16.     | #(OP_GREATER_THAN expression expression)
17.     | #(OP_LESS_THAN_OR_EQUAL expression expression)
18.     | #(OP_GREATER_THAN_OR_EQUAL expression expression)
19.     | #(RES_IN          expression expression)
20.     | #(OP_SUBSET       expression expression)
21.     | #(OP_SUPERSET     expression expression)
22.     | #(OP_UNION        expression expression)
23.     | #(OP_DIFF         expression expression)
24.     | #(OP_SYMDIFF      expression expression)
25.     | #(OP_PLUS         expression expression)
26.     | #(OP_MINUS        expression expression)
27.     | #(OP_MULTIPLICATION expression expression)
28.     | #(OP_DIVISION     expression expression)
29.     | #(OP_INTERSECT    expression expression)
30.     | #(OP_EXPON        expression expression)
31.     | #(OP_UN_MINUS     expression)
32.     | value
33.     | id: IDENT {vI.checkVariable(idClass,id);}
34.     | acc: access {vI.checkAccess(idClass,acc);}
35.     | function
36.     | BRA_ROUND_OPEN expression BRA_ROUND_CLOSE
37.     ;

```

Figure 5.20 – The rule to check constraints in the second pass.

5.2.3 Refactoring Phase

The translation to Flat **s**-COMMA is carried out by applying several refactoring steps. In fact, it is necessary to transform the modeling constructs provided by **s**-COMMA for which no support exists in the solver layer. To guarantee the independence of solver translators from these complex refactoring steps, the result of the transformation is captured in an intermediate model called Flat **s**-COMMA, from which the solver translator generates the executable solver code. The idea is to reduce the work of the mapping tool and as a consequence to simplify the integration of new solvers to the platform.

Flat **s**-COMMA¹ can be seen as an unrolled version of **s**-COMMA, i.e. the object-oriented style is broken (composition and inheritance relationships are refactored) to state a model just composed of variables and constraints. The syntax to define variables and constraint is equivalent to **s**-COMMA, but the amount of modeling components supported is minor. For instance, control statements such as loops, conditionals are not provided. Enumerations and specific constructs such as compatibility constraints are not supported.

To handle this transformation we define a set of refactoring steps. These steps have been implemented in hand-written Java procedures, which are applied once the semantic checking succeeds. An overview of such steps is given in the following.

Loop unrolling

This phase unrolls the **forall** and the **sum** loops. The process consists in replacing the loop by the whole set of elements that it implicitly contains. Within expressions, the iterator variable used by the loop statement is replaced by an integer corresponding to the current number of loop turns. An example is depicted in Figure 5.21, the loop belonging to the **inside** constraint zone of the packing squares problem is shown on the left column of the figure, the unrolling result is shown on the right one.

<pre>//s-COMMA forall(i in 1..squares) { x[i] <= sideSize - size[i] + 1; y[i] <= sideSize - size[i] + 1; }</pre>	<pre>//Flat s-COMMA x[1] <= sideSize - size[1] + 1; y[1] <= sideSize - size[1] + 1; x[2] <= sideSize - size[2] + 1; y[2] <= sideSize - size[2] + 1; ... </pre>
--	--

Figure 5.21 – Loop unrolling.

Enumeration substitution

In general, solvers do not support non-numeric values. So, the enumerations are replaced by integer values. In Figure 5.22, the enumeration **size** used as type for the attribute **base** of the class **CrankCase** is replaced by the domain **[1,3]**. The value **small** is represented by the integer 1, the value **medium** is replaced by the integer 2, and **large** by the integer 3. Let us note that the original values are stored to give the results in the initial format.

¹The grammar of Flat **s**-COMMA can be found in the appendix.

```
enum size := {small,medium,large};
size base in [1,3];
```

Figure 5.22 – Enumeration substitution.

Data substitution

In this step, every data variable used in the model is replaced by its corresponding value defined in the data file.

Composition flattening

This step eliminates the hierarchy generated by object compositions. The process is done by expanding each object declared in the main class adding its attributes and constraints in the Flat s-COMMA file. The name of each attribute has a prefix corresponding to the concatenation of the names of objects of origin in order to avoid name redundancy. The expansion of objects **cCase** and **cSyst** of the engine problem is shown in Figure 5.23.

```
size cCase_base_;
int cCase_oilVesselVol_;
int cCase_bombePower_;
int cCase_volume_;
int cSyst_quantity_ in [2,12];
int cSyst_distBetCyl_ in [3, 18];
flow cSyst_inj_gasFlow_;
...
volume > cCase_volume_;
```

Figure 5.23 – Composition flattening.

Array containing objects are decomposed into a set of arrays, one for each attribute of the object. If the attribute of the object also corresponds to an object, the array is decomposed again. For instance, in the packing squares problem, the array of objects called **s** is decomposed into three arrays, one for each attribute. The name of each variable is composed of the name of the array (**s**) and the name of the attribute. The value **8** in the size of arrays and the value **5** in the variables' domain come from the data substitution of the constant **squares** and the constant **sideSize**, respectively. The domain of **s_size_[8]** corresponds to the size of squares given by the variable assignment of the model.

```
int s_x_[8] in [1,5];
int s_y_[8] in [1,5];
int s_size_[8] in [1,3];
```

Figure 5.24 – Flattening arrays containing objects.

Conditional removal

Conditional statements are transformed to logical formulas. For instance, `if a then b else c` is replaced by $(a \Rightarrow b) \wedge (a \vee c)$ (see Figure 5.25). If the statement condition is composed of constant values the statement is evaluated and the useless constraint are removed. An example is shown in Figure 5.26.

<pre>//s-COMMA if (quantity = 6) distBetCyl > 6; else distBetCyl > 3;</pre>	<pre>//Flat s-COMMA ((quantity = 6) -> (distBetCyl > 6)) and ((quantity = 6) or (distBetCyl > 3));</pre>
---	---

Figure 5.25 – Conditional removal.

<pre>//Data File n := 1; s := 2; //Model file ... if (2 < 1 + n + s) { x < 1; y < 1; } else { x < 2; y < 2; }</pre>	<pre>//After Data substitution if (2 < 4) { x < 1; y < 1; } else { x < 2; y < 2; } //After evaluation x < 1; y < 1;</pre>
---	---

Figure 5.26 – Conditional evaluation.

Compatibility removal

Compatibility constraints are also translated to a logical formula. We create a conjunctive boolean expression for each n-tuple of allowed values. Then, each constraint of the n-tuple is stated in a disjunctive constraint. The transformed compatibility constraint of the Engine problem is shown in Figure 5.27. Non-numeric values were replaced by the corresponding integer values in the enumeration substitution step.

<pre>//s-COMMA compatibility (gasFlow,admValve,pressure) { ("direct", "small", 80); ("direct", "medium", 90); ("indirect", "medium", 100); ("indirect", "large", 130); }</pre>	<pre>//Flat s-COMMA ((gasFlow=1) and (admValve=1) and (pressure=80)) or ((gasFlow=1) and (admValve=2) and (pressure=90)) or ((gasFlow=2) and (admValve=2) and (pressure=100)) or ((gasFlow=2) and (admValve=3) and (pressure=130));</pre>
--	---

Figure 5.27 – Compatibility removal.

Logic formulas transformation

Some logic operators are not supported by solvers. For example, logical equivalence ($a \Leftrightarrow b$) and reverse implication ($a \Leftarrow b$). We transform logical equivalence expressing it in terms of logical implication ($(a \Rightarrow b) \wedge (b \Rightarrow a)$). Reverse implication is simply inverted ($b \Rightarrow a$).

Expression evaluation

In this step we evaluate expressions composed of constants in order to reduce them and/or to eliminate useless constraints. Figure 5.28 illustrates the evaluation of an expression containing arithmetic and logic operators. Since the resulting value of the expression has no impact on the model, the constraint is removed.

$((1+1) < (1+1))$	and	$((1+1) < (1+1))$	\rightarrow	$((1+1) < (1+1))$	and	$((1+1) < (1+1))$
$((2 < 2)$	and	$(2 < 2))$	\rightarrow	$((2 < 2)$	and	$(2 < 2))$
(false	and	false)	\rightarrow	(false	and	false)
	false		\rightarrow		false	
						true

Figure 5.28 – Expression evaluation.

5.2.3.1 A Flat s-COMMA model

To exemplify some of these refactoring steps, we illustrate the resultant Flat s-COMMA model of the stable marriage problem (see Figure 5.29). The model is composed of four blocks: variables, constraints, enumeration types, and solving options. Within the **variables** block, the whole set of arrays has been generated from the composition flattening step. The array **man_wife_** (line 3) contains the decision variables **wife** of the original array **man**, and the array **woman_husband_** (line 9) contains the decision variables **husband** of the original array **woman**. The size of the array **man_wife_** has been set to 5, this value is given by the enumeration substitution step which sets the size of the array with the size of the enumeration **menList**. The domain $[1, 5]$ has been also produced by this step. The type of both arrays has been maintained to give the solutions in the enumeration format. These values are stored in the block **enum-types**. The arrays stated from lines 4 to 8 and 10 to 14 contain the ranking values for each man and women, respectively.

The constraints posted between lines 18 and 25 come from the loop unrolling phase of the forall statements of the **matchHusbandWife** constraint zone. Likewise, lines 28 to 36 have been generated by the loops of **forbidUnstableCouples**. Within these constraints, the data substitution step has replaced several constants with their corresponding integer values. At the end of the file, the solving options are stated. Since no solving option was defined in the s-COMMA model, the **default** solving option is stated.

```

1.  variables:
2.
3.      womenList man_wife_[5] in [1,5];
4.      int man_1_rank_[5] in [1,5];
5.      int man_2_rank_[5] in [1,5];
6.      int man_3_rank_[5] in [1,5];
7.      int man_4_rank_[5] in [1,5];
8.      int man_5_rank_[5] in [1,5];
9.      menList woman_husband_[5] in [1,5];
10.     int woman_1_rank_[5] in [1,5];
11.     int woman_2_rank_[5] in [1,5];
12.     int woman_3_rank_[5] in [1,5];
13.     int woman_4_rank_[5] in [1,5];
14.     int woman_5_rank_[5] in [1,5];
15.
16.  constraints:
17.
18.      woman_husband_[man_wife_[1]]=1;
19.      woman_husband_[man_wife_[2]]=2;
20.      woman_husband_[man_wife_[3]]=3;
21.      ...
22.
23.      man_wife_[woman_husband_[1]]=1;
24.      man_wife_[woman_husband_[2]]=2;
25.      man_wife_[woman_husband_[3]]=3;
26.      ...
27.
28.      5<man_1_rank_[man_wife_[1]] ->
29.      woman_1_rank_[woman_husband_[1]]<1;
30.      1<woman_1_rank_[woman_husband_[1]] ->
31.      man_1_rank_[man_wife_[1]]<5;
32.
33.      1<man_1_rank_[man_wife_[1]] ->
34.      woman_2_rank_[woman_husband_[2]]<3;
35.      3<woman_2_rank_[woman_husband_[2]] ->
36.      man_1_rank_[man_wife_[1]]<1;
37.      ...
38.
39.  enum-types:
40.
41.      menList := {Richard,James,John,Hugh,Greg};
42.      womenList := {Helen,Tracy,Linda,Sally,Wanda};
43.
44.  solving-opts: default;

```

Figure 5.29 – A Flat s-COMMA model of the stable marriage problem.

5.3 From Flat s-COMMA to solvers

The transformation from Flat s-COMMA toward the solver model is performed via the mapping tool of the platform. Two kinds of translators have been built for this mapping tool (see Figure 5.30). The first ones belong to a previous version of our platform, and they have been written by hand in Java (HW) with the support of the ANTLR tool for parsing the Flat s-COMMA file. The second ones belong to the last implementation of the platform, and they have been implemented using a model-driven (MD) approach. Both kinds of translators are presented and compared in the following sections.

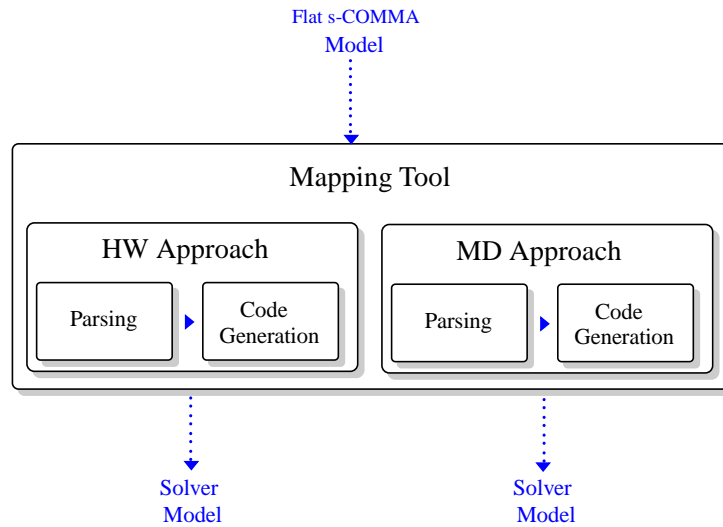


Figure 5.30 – The mapping tool.

5.3.1 Hand-Written Translators

The generation of solver files through our Java hand-written translators requires a prior parsing of the Flat s-COMMA model. We carry out this process using the same tools as in the previous phase. An ANTLR lexer and an ANTLR parser perform the parsing process and produce the corresponding AST. This AST is then explored by an ANTLR tree walker in order to generate the intermediate representation from which the translator builds the target file.

5.3.1.1 Parsing

The lexical analysis is the first phase of the parsing process. A portion of the ANTLR lexer to perform this task is shown in Figure 5.31. Such a file is very similar to the one of s-COMMA. Let us note that the options `testLiterals` and `paraphrase` are not included in the `IDENT` token, as there is no need to check for ambiguities and to show error messages at this stage.

Note

A Flat s-COMMA model is automatically generated from a syntactically and semantically correct s-COMMA model, being unnecessary to re-analyze it.

```

1.  tokens
2.  {
3.      RES_VARIABLES    = "variables"    ;
4.      RES_CONSTRAINTS  = "constraints"  ;
5.      RES_ENUM_TYPES   = "enum-types"   ;
6.      RES_SOLV_OPT     = "solving-opts" ;
7.      RES_AND          = "and"          ;
8.      ...
9.  }
10.
11. IDENT
12.   : (LETTER|'_' ) (LETTER|DIGIT|'_' )*
13.   ;
14.   ...

```

Figure 5.31 – Tokens and the `IDENT` rule in the ANTLR lexer of Flat `s-COMMA`.

Figure 5.32 illustrates the rule to parse a Flat `s-COMMA` model (line 1). Four optional rule calls define the composition of a Flat `s-COMMA` model. The first rule call recognizes the variables, the second one the constraints, the third one the enumeration types, and the final one the solving options of the model. The resulting AST is captured in a root node called `MODEL`. The corresponding rules to parse the set of variables and the set of constraints are depicted below.

```

1.  model      : (variableSet)? (constraintSet)?
2.              (enumSet)? (solvingOpts)?
3.              { ## = #( #[MODEL, "MODEL"] ,##); }
4.              ;
5.
6.  variableSet : RES_VARIABLES! PUN_COLON! (variable)*
7.              { ## = #( #[VARIABLE_SET, "VARIABLE_SET"] ,##); };
8.
9.  constraintSet : RES_CONSTRAINTS! PUN_COLON! constraintSetBody
10.               { ## = #( #[CONSTRAINT_SET, "CONSTRAINT_SET"] ,##); };

```

```

graph TD
    MODEL[MODEL] --- VARIABLE_SET[VARIABLE_SET]
    MODEL --- CONSTRAINT_SET[CONSTRAINT_SET]
    MODEL --- ENUM_SET[ENUM_SET]
    MODEL --- SOLVING_OPTS[SOLVING_OPTS]

```

Figure 5.32 – Parser rules of Flat `s-COMMA`.

The rules to recognize variables and constraints are illustrated in Figure 5.33. The `variable` rule is very similar to the attribute rule defined in `s-COMMA`. The body of a constraint block may be composed of three kinds of model components: a constraint, a global constraint, or an optimization statement.

Note

The optional `consLevel` rule call is absent in the `variable` rule since the consistency level option can only be specified on objects, which do not participate in Flat `s-COMMA`. The composition flattening phase has eliminated them.

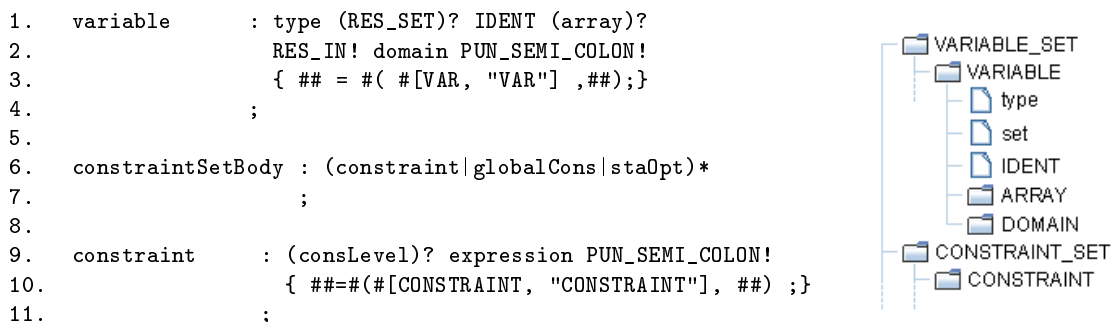


Figure 5.33 – Parser rules of Flat s-COMMA.

5.3.1.2 Exploring the AST

Once the AST has been built, it must be explored to generate the intermediate representation. Figure 5.34 depicts three rules of the tree walker to explore the Flat s-COMMA AST. As we have mentioned, no semantic checking is needed, so Java methods embedded in rules are just used to generate the intermediate representation.

```

1.  model          : #(MODEL (variableSet)? (constraintSet)?
2.                  (enumSet)? (solvingOpts)?)
3.                  ;
4.
5.  variableSet     : #(VARIABLES (variable)*)
6.                  ;
7.
8.  variable        : #(VAR (t:type (set:RES_SET)? idVar:IDENT
9.                  (arr:array)? dom:domain
10.                  {vI.addVar(t,set,idVar,arr,dom);}))
11.                  ;

```

Figure 5.34 – Tree walker of Flat s-COMMA.

5.3.1.3 Code Generation

After the exploration of the AST, the intermediate representation is ready to be examined by the solver translators. The translators are organized in four Java files. One for the code generation of variables, one for the code generation of constraints, one to format variable names and a main file to generate the headers and specific procedures for the solver file. Figure 5.35 shows the initial procedure of the Gecode/J translator main file. This procedure calls each one of the methods required to build the code representing the Gecode/J model: to create the file, to build the headers, to build the constructors, to build the code for showing the results, to build the main method of the file, and finally to close the file.

```

1.  public void buildFile() {
2.      createFile();
3.      buildHeader();
4.      buildConstructor();
5.      buildCopyConstructor();
6.      buildResults();
7.      buildMain();
8.      closeFile();
9.  }

```

Figure 5.35 – The initial procedure of the main Java class of the Gecode/J translator.

The procedure to write the constructor of the Gecode/J model is shown in Figure 5.36. In the constructor, the variables and constraint of the problem are posted. `decVars.translate()` (line 6) generates the variables and `constraints.translate()` (line 7) generates the constraints. At line 8, the solving options for the resolution process are given. The method `println` is used to write strings on the file and `nL` to write a newline character.

```

1.  public void buildConstructor() {
2.      println(" public " + className + "(Options opt) {");
3.      println("     super();");
4.      println("     vars = new VarArray<IntVar>();");
5.      nL();
6.      println(decVars.translate());
7.      println(constraints.translate());
8.      println("     branch(this, vars," + buildSolvingOptions() + ");");
9.      println(" }");
10.     nL();
11. }

```

Figure 5.36 – Code generation of the Gecode/J constructor.

Figure 5.37 illustrates a method for the code generation of a one dimensional array (vector) containing Gecode/J decision variables. The declaration of a vector begins with the type of the Java variable (`VarArray<IntVar>`) followed by its name. The name is obtained from the `decVar` object, which was generated in the intermediate representation. Then, the `initialize` method is used to set four parameters of the vector, e.g. its name (to show the results), its size, and the lower and the upper bounds of its domain. Finally, the new vector is added to a global array for performing the labeling process (`vars.addAll`).

The code generation of constraints is more complicated since they may be composed by several elements. This phase is handled by representing the constraints in the form of a tree. An ANTLR tree walker explores this tree and performs calls to the necessary methods to transform the nodes of the tree into the solver code. Figure 5.38 depicts the ANTLR constraint tree walker.

Constraint are explored in the same way as in the semantic checking of `s-COMMA`. Each operator and operand stated in the rule includes a method call to a code generation procedure. The methods to generate the code of an addition and a distinct relation are depicted in Figure 5.39. The constraints are systematically generated and stored in a data structure called `codeStore`, which is then read by the main translator file to write the constraints in the solver program. For instance, the expression $a + b$ is generated as `new Expr(a).p(b)`, where `p` represents the

```

1.  public StringBuffer integer(FlatVectorDecVar decVar) {
2.
3.      StringBuffer str = new StringBuffer();
4.      str.append("    VarArray<IntVar> "); —————> VarArray<IntVar> man_wife_ =
5.      str.append(decVar.getName()); —————> initialize("man_wife_",5,1,5);
6.      str.append(" = initialize(\""); —————>
7.      str.append(decVar.getName()); —————>
8.      str.append("\",");
9.      str.append(decVar.getSize()); —————>
10.     str.append(",");
11.     str.append(decVar.getIntLowerBound()); —————>
12.     str.append(",");
13.     str.append(decVar.getIntUpperBound()); —————>
14.     str.append(");\n");
15.     str.append("    vars.addAll(");
16.     str.append(decVar.getName());
17.     str.append(");\n");
18.     return str;
19. }

```

Figure 5.37 – Code generation of Gecode/J variables.

```





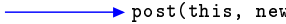



1.  expression
2.      : #(OP_EQV          expression expression) {eT.equivalence();}
3.      | #(OP_IMP         expression expression) {eT.implicance();}
4.      ...
5.      | #(OP_DISTINCT    expression expression) {eT.distinct();}
6.      | #(OP_LESS_THAN   expression expression) {eT.less();}
7.      ...
8.      | #(OP_PLUS        expression expression) {eT.plus();}
9.      | #(OP_MINUS       expression expression) {eT.minus();}
10.     | #(OP_MULTIPLICATION expression expression) {eT.mult();}
11.     | #(OP_DIVISION     expression expression) {eT.div();}
12.     | #(OP_INTERSECT    expression expression) {eT.intersect();}
13.     | #(OP_EXPON        expression expression) {eT.expon();}
14.     | #(OP_UN_MINUS     expression) {eT.unMinus();}
15.     | val:  value        {eT.addValue(val);}
16.     | id:   IDENT        {eT.addIdent(id);}
17.     | acc:  access       {eT.addAccess(acc);}
18.     | f:function        {eT.addFunction(f);}
19.     | BRA_ROUND_OPEN expression BRA_ROUND_CLOSE
20.     ;

```

Figure 5.38 – The tree walker for the code generation of constraints.

‘+’ operator and the operands are obtained from `codeStore`. Relations are generated using the `post` method. For instance, $a <> b$ is generated as `post(this, new Expr(a), IRT_NQ, new Expr(b))`, `IRT_NQ` being the not equal operator.

```

1.  public void plus() {
2.      StringBuffer str = new StringBuffer();
3.      str.append("new Expr(");  new Expr(a).p(b)
4.      str.append(codeStore.getCode()); 
5.      str.append(").p("); 
6.      str.append(codeStore.getCode()); 
7.      str.append(")");
8.      codeStore.add(str);
9.  }
10.
11. public void distinct() {
12.     StringBuffer str = new StringBuffer();
13.     str.append("post(this, new Expr(");  post(this, new Expr(a), IRT_NQ, new Expr(b))
14.     str.append(codeStore.getCode()); 
15.     str.append("), IRT_NQ, new Expr("); 
16.     str.append(codeStore.getCode()); 
17.     str.append(")");
18.     codeStore.add(str);
19. }

```

Figure 5.39 – Two procedures for the code generation of constraints.

5.3.1.4 A Gecode/J model generated from Flat s-COMMA

Figure 5.40 depicts an extract of the Gecode/J file generated for the stable marriage problem. The initial lines state the headers (package and import statements) of the Gecode/J model. The `man_wife_` array is defined at line 5, being initialized with size 5 and domain $[1, 5]$. At line 6, this array is added to a global array called `vars` in order to perform the labeling process. Lines 11 and 12 illustrate two constraints, which are stated by means of the `post` method. The `get(a, b)` method returns an element of an array, `a` being the array and `b` the position of the element. The `IRT_EQ` parameter represents the equality operator.

```

1.  package comma.solverFiles.gecodej;
2.  import static org.gecode.Gecode.*;
3.  ...
4.
5.      VarArray<IntVar> man_wife_ = initialize("man_wife_", 5, 1, 5);
6.      vars.addAll(man_wife_);
7.
8.      VarArray<IntVar> woman_husband_ = initialize("woman_husband_", 5, 1, 5);
9.      vars.addAll(woman_husband_);
10.
11.     post(this, new Expr(get(woman_husband_, get(man_wife_, 1))), IRT_EQ, new Expr(1));
12.     post(this, new Expr(get(woman_husband_, get(man_wife_, 2))), IRT_EQ, new Expr(2));
13.     ...

```

Figure 5.40 – A Gecode/J model of the stable marriage problem.

5.3.2 Model-Driven Translators

Model-driven translators have been developed using a general model-driven transformation framework. Under this approach, the development of languages is seen from another point of view. A language is not defined by means of grammars and regular expressions. Languages are defined via metamodels and concrete syntax tools. The metamodel specifies the concepts appearing in a language and the concrete syntax tool defines how these concepts appear in the syntax of the language.

A model-driven transformation framework allows us to define a transformation from a source language to a target one using a Model-Driven Architecture (MDA) [20^w] (see Figure 5.41). The level M1 holds the model. The level M2 describes the semantics of the level M1 and thus identifies concepts handled by this model through a metamodel. The level M3 is the specification of the level M2 and it is self-defined. Transformation rules are defined to translate models from a source model to a target one, the semantics of these rules is also defined by a metamodel.

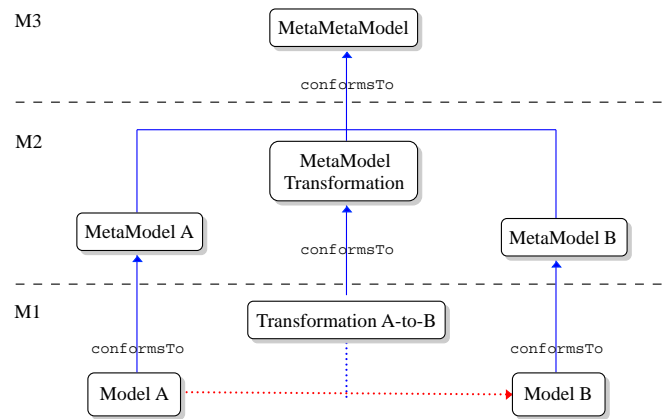


Figure 5.41 – A general MDA for model transformation.

The implementation of this approach in our platform is illustrated in Figure 5.42. The Flat s-COMMA corresponds to the source model and its semantics is defined by its metamodel. The translation to the target language is performed by transformation rules. These rules carry out the transformation process by matching the concepts of the Flat s-COMMA metamodel to the concepts of the solver metamodel.

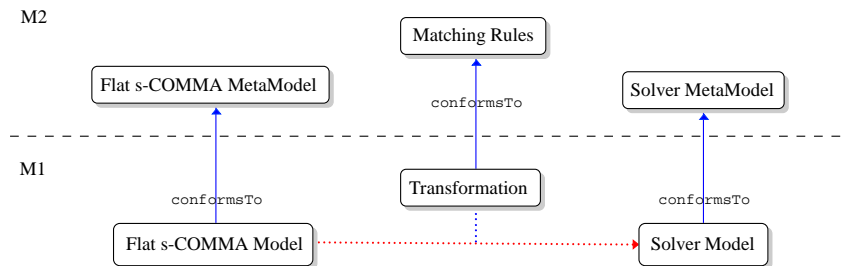


Figure 5.42 – Model-driven translation in s-COMMA.

Remark

A major strength of using this metamodeling approach is that models are concisely represented by metamodels. This allows one to define transformation rules that only operate on the concepts of metamodels (at the M2 level of the MDA approach), not on the concrete syntax of a language. Syntax concerns are defined independently (we illustrate this in Section 5.3.2.4). This separation is a great advantage for a clear definition of transformation rules and syntax descriptions, which are the base of our mapping tool.

5.3.2.1 Metamodeling

The metamodeling phase is carried out by using the KM3 language [JB06] (Kernel Meta Meta Model). Such a language supports most metamodeling standards and it is based on the simple notion of classes to define each one of the concepts of a metamodel. These concepts are needed to define the transformation rules and also to generate the target files. Figure 5.43 illustrates the main concepts of the Flat s-COMMA metamodel. The concepts expressed in KM3 are shown on the left side of the figure and the corresponding metamodel using UML class diagram notation is depicted on the right side.

```

1.  class Model {
2.      attribute name : String;
3.      reference variables [0-∗] container : Variable;
4.      reference constraints [0-∗] container : ConstraintStatement;
5.      reference enumTypes [0-∗] container : EnumType;
6.      reference solvingOpts [0-3] container : SolvingOpt;
7.  }
8.
9.  class Variable {
10.     attribute name : String;
11.     attribute type : String;
12.     attribute isSet : Boolean;
13.     reference array [0-1] container : Array;
14.     reference domain container : Domain;
15. }
16.
17. class Array {
18.     attribute row : Integer;
19.     attribute col [0-1] : Integer;
20. }

```

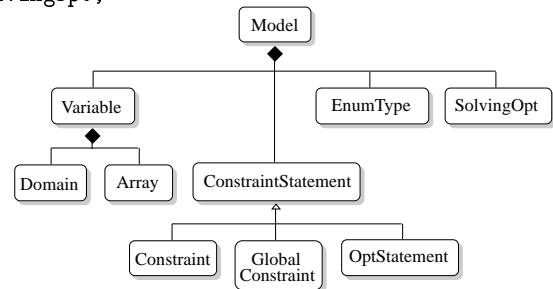


Figure 5.43 – An extract of the KM3 file of Flat s-COMMA.

In the metamodel, a Flat s-COMMA model is defined by the **Model** concept. This concept is composed of one attribute and four references. The attribute **name** at line 2 represents the name of the model and it is declared with the basic type **String**. Line 3 states that the class **Model** is composed of a set of objects from the class **Variable**. The reserved word **reference** is used to define relationships with instances of other classes. The statement **[0-∗]** defines the multiplicity of the relationship. If the multiplicity statement is omitted the relationship is defined as **[1-1]**. Lines 4 to 6 are similar and define that the class **Model** is also composed of **constraints**,

`enumTypes`, and `solvingOpts`. Three solving options can be defined: variable ordering, value ordering and the consistency level used. The class `Variable` is composed of three attributes and two references. The first attribute defines the name of the variable and the following its type. The third attribute is a boolean value used to specify set variables. The reference stated at line 13 is used to define arrays of variables. The declaration of the `Variable` class ends with the reference to state the domain. At line 17, the `Array` class is composed of two attributes. The first one is used to define the array row size, while the second one used to define the array column size.

A constraint statement is specialized in three concepts: `Constraint`, `GlobalConstraint` and `OptStatement`. The KM3 defining the composition of the `Constraint` concept is illustrated in Figure 5.44. It consists of an `Expression` concept and an optional attribute to specify its consistency level. Two kinds of expressions can be identified, binary and unary expressions. The class to define binary expressions is stated at line 12. This class contains two references, `left` corresponds to the left operand and `right` to the right operand of an expression. Both operands are also expressions. At line 17, the class to define unary expressions is defined, just one operand is required. The attribute to define the operator in unary and binary expressions is inherited from the `ExpOperator` class (line 8).

```

1.  class Constraint extends ConstraintStatement {
2.      attribute consLevel [0-1] : String;
3.      reference assertion container : Expression;
4.  }
5.
6.  abstract class Expression {}
7.
8.  abstract class ExpOperator extends Expression {
9.      attribute name : String;
10. }
11.
12. class BinaryExpression extends ExpOperator {
13.     reference left container : Expression;
14.     reference right container : Expression;
15. }
16.
17. class UnaryExpression extends ExpOperator {
18.     reference left container : Expression;
19. }

```

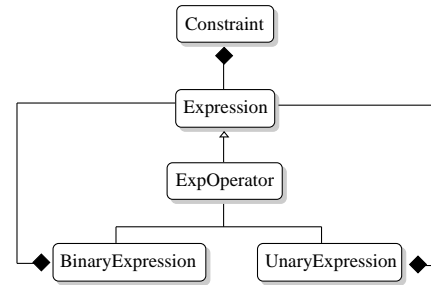


Figure 5.44 – Constraints in the KM3 file of Flat s-COMMA.

An expression may have three kinds of operands: a value, a variable, or a function. In Figure 5.45, the classes to define the values are stated between lines 1 and 9. The class to define variables as operands follows. Such a class is named `VariableOccurrence` and it is composed of one attribute and two references. The `declaration` attribute contains the name of the variable occurrence, and the references are used for array occurrences. The `i` reference is used for the array row index and `j` for the array column index. Both indexes are defined through expressions. At the end, the class to define function calls (e.g. the cardinality of a set) is stated. Its name and its input parameters are given.

```

1.  abstract class Value extends Expression {}
2.
3.  class IntValue extends Value {
4.    attribute value : Integer;
5.  }
6.
7.  class RealValue extends Value {
8.    attribute value : Double;
9.  }
10.
11. class VariableOccurrence extends Expression {
12.   attribute declaration : String;
13.   reference i [0-1] container : Expression;
14.   reference j [0-1] container : Expression;
15. }
16.
17. class FunctionCall extends Expression {
18.   attribute name : String;
19.   reference parameters[*] container : Expression;
20. }

```

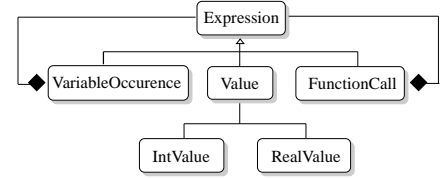


Figure 5.45 – Operands in the KM3 file of Flat s-COMMA.

5.3.2.2 Transformation Rules

The transformation rules to define the mapping between Flat s-COMMA and the solver language are implemented in ATL (Atlas Transformation Language). This language is strongly based on OCL [48], and supports most of its functions and its types. The ATL rules are able to perform a transformation by defining how the concepts are matched from source to target metamodels. Figure 5.46 shows an ATL rule to transform the concepts of the Flat s-COMMA metamodel to the concepts of the Gecode/J metamodel. The Gecode/J metamodel is omitted here since it is very similar to the Flat s-COMMA metamodel.

```

1.  rule ModelToModel {
2.    from
3.      s : FlatsComma!Model (
4.      )
5.    to
6.      t : GecodeJ!Model(
7.        name      <- s.name,
8.        variables  <- s.variables,
9.        constraints <- s.constraints,
10.       enumTypes  <- s.enumTypes,
11.       solvingOpts <- s.solvingOpts
12.      )
13. }

```

Figure 5.46 – ATL rules for the Flat s-COMMA to Gecode/J transformation.

Remark

Flat s-COMMA has been designed to be as close as possible from the solving level. This ensures the Flat s-COMMA metamodel to be very close to solver metamodels. This is a great advantage since translation rules become simple: we mainly need one to one transformations.

The transformation rule is called `ModelToModel` and it defines the matching between the concepts `Model` expressed in Flat s-COMMA and Gecode/J. The source elements are stated with the reserved word `from` (line 2) and the target ones with the reserved word `to` (line 5). These elements are declared like variables with a name (`s,t`) and a type corresponding to a class in a metamodel (`FlatsComma!Model`, `GecodeJ!Model`). In the target part of the rule, the name attribute of the Flat s-COMMA problem is assigned to the Gecode/J name (`name <- s.name`), this matching corresponds to a simple string assignment. The following four matchings are assignments between concepts that are defined as **reference** in the metamodel. Handling these matchings requires to define additional rules. For instance, the Flat s-COMMA KM3 metamodel defines that the reference `variables` corresponds to a set of `Variable` elements. Thus, the statement `variables <- s.variables` implicitly calls the rule `VariableToVariable`, which defines the matching between the elements contained in `Variable` objects. The `VariableToVariable` rule is depicted in Figure 5.47, such a rule matches five elements. The first two statements are string assignments, the third one is a boolean assignment, and the remaining ones are reference assignments. The first reference assignment matches `Array` objects while the second one matches `Domain` objects. The rule to match arrays can be seen on the right side of the figure.

<pre> 1. rule VariableToVariable { 2. from 3. s : FlatsComma!Variable (4.) 5. to 6. t : GecodeJ!Variable (7. name <- s.name, 8. type <- s.type, 9. isSet <- s.isSet, 10. array <- s.array, 11. domain <- s.domain 12.) 13. }</pre>	<pre> 14. rule ArrayToArray { 15. from 16. s : FlatsComma!Array 17. to 18. t : GecodeJ!Array(19. row <- s.row, 20. column <- s.column 21.) 22. }</pre>
--	--

Figure 5.47 – ATL rules for the Flat s-COMMA to Gecode/J transformation.

Although the rules used here are not complex, ATL is able to perform more difficult rules. For instance, the most difficult rule we defined, was the transformation rule from Flat s-COMMA matrices containing sets, which must be unrolled in the ECLⁱPS^e models (since set matrices are not supported). This unroll process is carried out by defining a single set in ECLⁱPS^e for each cell in the matrix. The name of each single variable is composed of the name of the matrix, and the corresponding row and column index. Let us note that this procedure includes calls to ATL helpers, which are used to define specific functions. ATL helpers can be seen as the ATL equivalent to Java methods.

```

1.  rule ModelToModel {
2.    from
3.      s : FlatsComma!Model (
4.        s.hasSetMatrix
5.      )
6.    to
7.      t : ECLiPSe!Model (
8.        name <- s.name,
9.        constraints <- s.constraints,
10.       enumTypes <- s.enumTypes,
11.       solvingOpts <- s.solvingOpts
12.      )
13.    do {
14.      t.variables <- s.variables->collect(e|
15.        if e.isSetMatrix() then
16.          thisModule.getMatrixCells(e)->collect(f|
17.            thisModule.SetMatrixVariableToVariable(f.var,f.i,f.j)
18.          )
19.        else
20.          e
21.        endif
22.      )->flatten();
23.    }
24.  }
25.
26.  rule SetMatrixVariableToVariable(var : FlatsComma!Variable,
27.    i : Integer, j : Integer) {
28.    to
29.      t : ECLiPSe!Variable(
30.        name <- var.name + i.toString() + '_' + j.toString() + '_',
31.        type <- var.type,
32.        domain <- var.domain
33.      )
34.    do {
35.      t;
36.    }
37.  }

```

Figure 5.48 – ATL rules for decomposing matrices containing sets.

Figure 5.48 depicts the rules for handling the matrix transformation. The rule `ModelToModel` is stated at the beginning of the file. It holds a condition (line 4), which calls the helper `hasSetMatrix` to check whether set matrices are defined in the model. If the condition is true, `name`, `constraints`, `enumTypes`, and `solvingOpt` are matched normally, but `variables` has a special procedure to decompose the set matrix. This procedure begins at line 13 with a `do` block. In this block, the `collect` loop iterates over the variables. Then, each of these variables (`e`) is checked to determine whether it has been defined as a set matrix (line 15). If this occurs, the helper `getMatrixCells(e)` calculates the set of tuples corresponding to all the cells of the matrix (`thisModule` is used to explicitly call helpers or rules). Each tuple is composed of the Flat `s-COMMA` variable (`f.var`), a row index (`f.i`) and a column index (`f.j`). Then, the rule `SetMatrixVariableToVariable` is applied to each tuple in order to generate the ECLⁱPS^e variables. This rule has no source block since the source elements are the input parameters. The rule sets to the attribute `name`, the concatenation of the name of the matrix with the respective row (`i.toString()`) and column (`j.toString()`). Attributes `type` and `domain` are also matched. Finally, `flatten()` is an OCL inherited method used to match the generated set of variables with `t.variables`.

5.3.2.3 Code Generation

The code generation process is also performed using the ATL language. An ATL query is defined to create a new target file and to call a set of ATL helpers. These helpers are able to combine the metamodel elements with the syntax of the target language in order to generate the string to be written in the target file. Figure 5.49 depicts the helper for the code generation of a one dimensional Gecode/J array.

```

1. helper context GecodeJ!Variable def: toString2() :
2.   if thisModule.isVector(self) then
3.     'VarArray<IntVar> ' + _____ VarArray<IntVar> man_wife_ =
4.     self.name + _____ initialize("man_wife_",5,1,5);
5.     ' = initialize(\' + _____
6.     self.name + _____
7.     '\",\' + self.array.toString2() + _____
8.     '\',\' + self.domain.toString2()+');\n' + _____
9.     '    vars.addAll(\' + self.name + '\');\n'
10.    ...

```





Figure 5.49 – ATL helper to generate a Gecode/J vector.

The header of the helper is declared at line 1, its name is `toString2` and it is defined for the `GecodeJ!Variable` concept. A condition, at line 2, checks if the current object (`self`) is a one dimensional array. If so, the code of the Gecode/J vector declaration is generated. The `self.name` statement gets the name of the variable, and `self.array.toString2()` calls a helper to get the string representing the array dimensions. Analogously, `self.domain.toString2()` generates the string corresponding to the domain. At the end, the array is added to the global array for performing the labeling process.

The code of constraints is generated in a very similar manner. For instance, the helper to generate the code of an addition in binary expressions is shown in Figure 5.50. The helper appends the left and the right part of the expression with the necessary operators for building the addition expression. Let us notice that `left` and `right` are defined as expressions in the

metamodel. Thus, if the operands of a binary expression are also formed by binary expressions, the ATL engine performs a recursive call to this helper so as to build the whole expression.

```

1.  helper context GecodeJ!BinaryExpression def: toString2() : String=
2.    if (self.name = '+')
3.      'new Expr(' +  new Expr(a).p(b)
4.      self.left.toString2() + ')' + 
5.      '.p' + 
6.      '(' + self.right.toString2() + ')' 
7.    ...







```

Figure 5.50 – ATL helper to generate an addition.

5.3.2.4 Parsing

TCS (Textual Concrete Syntax) [JBK06] is the language used to parse the Flat s-COMMA file. This process is achieved by bridging the Flat s-COMMA metamodel with the Flat s-COMMA syntax. Figure 5.51 shows an extract of the TCS file for Flat s-COMMA. Each class of the Flat s-COMMA metamodel has a dedicated template declared with the same name. Within templates, words between double quotes are tokens in the grammar (e.g. "variables", ":"). Words without double quotes can be seen as template calls, being used to introduce the corresponding list of concepts. For instance, the **Model** template defines the syntactic structure of a Flat s-COMMA model. The four blocks of a Flat s-COMMA model are defined (**variables**, **constraints**, **enum-types**, and **solving-opts**). The **isDefined** function is used to state that the block is optional. For instance, '**isDefined(variables) ?**' is stated to parse the variables block only if the model contains variables. After this conditional statement, the syntactic structure of the variables block is defined. It begins with the reserved word "variables" followed by a colon token and by a call to the **Variable** template. Let us notice that the TCS compiler is able to perform this call since **variables** is defined as a reference to **Variable** objects in the KM3.

```

1.  template Model
2.    : (isDefined(variables) ? "variables" ":" variables)  variables :
3.      (isDefined(constraints) ? "constraints" ":" constraints) ...
4.      (isDefined(enumTypes) ? "enum-types" ":" enumTypes)
5.      (isDefined(solvingOpts) ? "solving-opts" ":" solvingOpts)
6.    ;
7.
8.  template Variable
9.    : type  int set foo[6] in [1,5];
10.   (isSet ? "set") 
11.   name 
12.   (isDefined(array) ? array) 
13.   "in" domain ";" 
14. ;
15.
16. template Array
17.   : "[" row (isDefined(col) ? "," col ) "]"
18. ;

```

Figure 5.51 – Three templates of the TCS file of Flat s-COMMA.

The **Variable** template defines the syntactic structure of a variable declaration, which begins with the type of the variable followed by a conditional structure (`isSet? "set"`). This conditional structure permits the use of an optional token `set` for defining set variables. If the `set` token is encountered in the variable declaration, the `isSet` attribute of the metamodel is set to `true`. Then, the name of the variable is stated. It is followed by another conditional structure, which states that the template **Array** is only called if the variable has been defined as an array. The declaration ends with the definition of the reserved word `in` followed by the domain. The template concerning the **Array** concept is declared at line 16. The array indexes (`row` and `col`) are enclosed with box brackets and separated by a comma token. The `col` attribute is optional, being used only by two-dimensional arrays.

Remark

TCS is not required to add a new translator, as just the TCS for Flat s-COMMA is needed in the platform.

5.3.2.5 Transformation process

TCS and KM3 work together and their compilation generates a Java package (which includes lexers, parsers and code generators) for Flat s-COMMA (FsC), which is then used by the ATL files to generate the target model. Figure 5.52 depicts the complete transformation process. The Flat s-COMMA file is the input of the Java package which generates a XMI (XML Metadata Interchange) for Flat s-COMMA. Over this file, ATL rules act and generate a XMI file for Gecode/J. Finally, this file is transformed into a solver file by means of the ATL query.

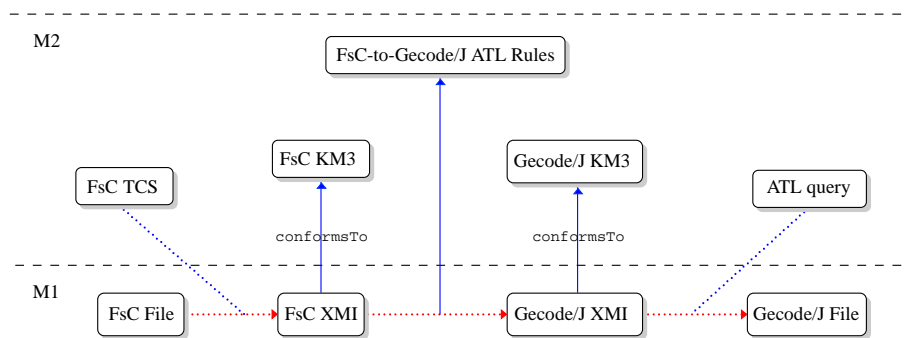


Figure 5.52 – The model-driven transformation process on the example of Flat s-COMMA (FsC) to Gecode/J.

Note

The XMI file used in the transformation includes an organized representation of models in terms of its metamodel concepts in order to facilitate the task of transformation rules.

5.3.2.6 Direct Code Generation

There is another approach to develop translators using the model-driven approach. For instance, if we want to use just the Flat s-COMMA features that are supported by the solver, we can omit the transformation rules and we can apply the ATL query directly on the source metamodel. Figure 5.53 shows the direct code generation process.

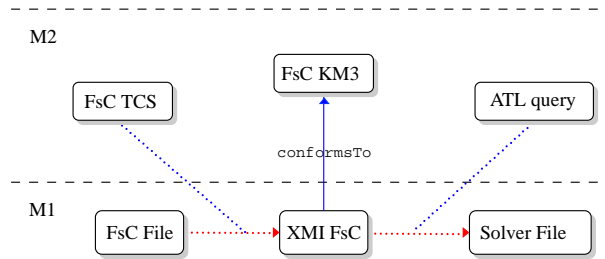


Figure 5.53 – Direct code generation.

Although this approach is simpler, it is less flexible since we lose the possibility of using more elaborated transformations such as the set matrix decomposition presented in Section 5.3.2.2.

5.3.3 Discussion

We have presented two different approaches for building translators in solver-independent architectures. Comparing both approaches, let us make the following concluding remarks.

- The development of hand-written translators is in general a hard task. Their creation, modification and reuse requires to have a deep insight in the code and in the architecture of the platform, even more if they have a specific or complex design. For instance, in our implementation, it is mandatory to master ASTs, Java and intermediate representations to generate the target solver files.
- As we mentioned in Section 5.3.2.2, solver metamodels are similar to the Flat s-COMMA metamodel, and ATL rules correspond mainly to one-to-one transformations. We believe therefore that the development of KM3 and ATL rules for new solver-translators should not be a hard task, and the concrete work for plugging a new solver should be just reduced to the definition of the ATL query for the code generation. This task may also be facilitated with the reuse of existing code generation files.
- The development of hand-written translators requires more code lines. In our implementation, the source files of Java translators are approximately 60% bigger than the model-driven translators source files (ATL + KM3).
- In the model-driven approach, the syntax concerns of a language are divided into the abstract syntax (KM3 metamodel) and the concrete syntax (ATL and TCS). This separation gives us a more organized and modular view of the language, which has simplified the creation and motivated the reuse of our translators. It is also important to contrast this feature with the mapping mechanism used in Cadmium, whose rules operate directly on Zinc expressions (by means of term matching), having no independence between abstract and concrete syntaxes. This property may generate smaller Cadmium programs, but less modular compared to our approach.

5.3.3.1 Experiments

To compare the performance of both kind of translators, in terms of translation time, we have performed a set of tests. The tests have been performed on a 3GHz Pentium 4 with 1GB RAM running Ubuntu 6.06, and the benchmarks used were the following:

- The cryptarithmic puzzle Send + More = Money (Send).
- The stable marriage problem (Stable).
- Two versions of the n-queens problem (10-queens and 18-queens).
- Packing 8 squares into a square of area 25 (Packing).
- The production-optimization problem (Production).
- Solving 20 linear inequalities (Ineq20).
- The assembly of a car engine subject to design constraints (Engine).
- The Sudoku logic-based number placement puzzle (Sudoku).
- The social golfers problem (Golfers).

Table 5.1 shows the translation times for both approaches. The first column gives the problem names. The second and third column depict the translation times using hand-written (HW) and model-driven (MD) translators (using translation rules), from Flat s-COMMA (FsC) to Gecode/J and from Flat s-COMMA to ECLⁱPS^e, respectively. Translation times from s-COMMA (sC) to Flat s-COMMA are given for reference in the last column (this process involves syntactic and semantic checking, and refactoring to Flat s-COMMA). The table exhibits that MD translators are slower than HW translators. This is expected since HW translators have been designed specifically for s-COMMA. They take as input a Flat s-COMMA definition and directly generate the solver file. The transformation process used by MD translators is not direct, it performs intermediate phases (XMI to XMI). However, we believe that translation times using MD translators are reasonable and this loss of performance is an acceptable price to pay for using a generic approach.

Table 5.1 – Translation times (seconds).

Problems	FsC to Gecode/J		FsC to ECL ⁱ PS ^e		sC to FsC
	HW	MD	HW	MD	
Send	0.052	0.688	0.048	0.644	0.237
Stable	0.137	1.371	0.143	1.386	0.514
10-Queens	0.106	1.301	0.115	1.202	0.409
18-Queens	1.122	3.194	0.272	2.889	0.659
Packing	0.172	1.224	0.133	1.246	0.333
Production	0.071	0.887	0.066	0.783	0.288
20 Ineq.	0.072	0.895	0.072	0.891	0.343
Engine	0.071	0.815	0.071	0.844	0.285
Sudoku	1.290	4.924	0.386	4.196	3.503
Golfers	0.098	1.166	0.111	1.136	0.380

We have performed another test to show that the automatic generation of solver files does not lead to a loss of performance in terms of solving time. In Table 5.2 we compare the solver

files generated by MD translators (Generated) with native solver files written by hand (Native). The results show that generated solver files are in general bigger than solver versions written by hand. This is explained by the loop unrolling and composition flattening processes presented in Section 5.3. However, this increase in terms of code size does not cause a negative impact on the solving time. In general, generated solver versions are very competitive with hand-written versions. The data also shows that Gecode/J files are bigger than ECLⁱPS^e files, this is because the Java syntax is more verbose than the ECLⁱPS^e one.

Note

In the comparison, we do not consider solver files generated by HW translators since they have no relevant differences compared to solver files generated by MD translators.

Table 5.2 – Solving times (seconds) and model sizes (number of tokens).

Benchmark	Gecode/J				ECL ⁱ PS ^e			
	Native		Generated		Native		Generated	
	Solv. time	Size	Solv. time	Size	Solv. time	Size	Solv. time	Size
Send	0.002	590	0.002	615	0.01	231	0.01	329
Stable	0.005	1898	0.005	8496	0.01	1028	0.01	4659
10-Queens	0.003	460	0.003	9159	0.01	193	0.01	1958
18-Queens	0.008	460	0.008	30219	0.02	193	0.02	6402
Packing	0.009	663	0.009	12037	0.49	355	0.51	3212
Production	0.026	548	0.028	1537	0.014	342	0.014	703
20 Ineq	13.886	1576	14.652	1964	10.34	720	10.26	751
Engine	0.012	1710	0.012	1818	0.01	920	0.01	1148
Sudoku	0.007	1551	0.007	33192	0.21	797	0.23	11147
Golfers	0.005	1618	0.005	4098	0.21	980	0.23	1147

5.4 Summary

In this chapter we have presented the transformation process from graphical artifacts to solver programs. The architecture supporting this process is composed of three main elements: the s-COMMA GUI, the s-COMMA compiler, and the mapping tool. A complete transformation includes several phases. The s-COMMA GUI transforms its graphical artifacts into the corresponding s-COMMA textual model by means of a set of Java packages. This model is parsed and semantically checked using the ANTLR tool. If the checking process succeeds, the model is transformed to an intermediate language called Flat s-COMMA. In this transformation, several s-COMMA constructs are refactored to facilitate the transformation to the solver language. Finally, the generated Flat s-COMMA model is the input of the mapping tool, which builds the executable solver file. The mapping tool contains two kinds of solver translators: hand-written translators and model-driven translators. The hand-written translators are written in Java, while the model-driven translators

are developed using metamodels and transformation rules. The model-driven approach involves important advantages, which mainly concern implementation tasks.


In the following chapter, we begin the third part of this thesis by giving an overview of the transformation framework for CP. We present the main purpose of this framework and we illustrate a practical example. The second and final chapter of this third part concerns the implementation of the framework.

PART III

The Transformation Framework for CP

CHAPTER 6

Overview

 This chapter gives an overview of the transformation framework for CP. The main improvement of this approach with respect to our previous work and in turn with respect to the state-of-the-art solver-independent architectures is the possibility of choosing different modeling languages as the source of a transformation. This can be achieved by using a pivot model (intermediate model) which is independent from the target model, but also from source languages. The independence of this pivot can be contrasted with current approaches in which the intermediate model is strongly tied (in terms of syntax and constructs supported) to the modeling language, for instance Flat *s*-COMMA to *s*-COMMA, or flatZinc to Zinc and MiniZinc. This new approach is supported by a flexible architecture on which model-driven translators can be plugged to perform the mappings among the different languages. We believe that this new framework involves two important advantages:

- The user will be able to choose his favourite modeling language and the best known solving technology for a given problem provided that the transformation between languages is implemented.
- It may be easy to create a collection of benchmarks for a given language from different source languages. This feature may speed up prototyping of one solver, avoiding the rewriting of problems in its modeling language.

This architecture has been fully implemented using the MDA approach. The implementation is based on the tools presented in the previous chapter (KM3, ATL and TCS). The aim is to take advantage of the MDA benefits to define both clear and concise mapping rules and grammar specifications.

6.1 The Model-Driven Transformation Framework

Figure 6.1 depicts the architecture of our model-driven transformation framework, which is divided in two layers: M1 and M2. M1 holds the models representing constraint problems and M2 defines the semantics of M1 through the metamodels. The transformation rules are defined to perform a complete translation in three main steps: translation from the source model to the pivot model, refactoring/optimization on the pivot model, and translation from the pivot model to the target model. Source and target models may be defined through any CP languages. The pivot model may be refined several times in order to adapt it to the desired target model (see Section 7.2.1.2). These refining phases are similar to the ones performed from *s*-COMMA to Flat *s*-COMMA, but more flexible since it is possible to select the refining steps to be applied in a transformation. For instance, if loops are supported at the target level it is not necessary to unroll them or, if matrices are permitted, there is no need to flatten them. This new feature

Data File

```

1. enum name := {a,b,c,d,e,f,g,h,i};
2. int s := 3; //size of groups
3. int w := 4; //number of weeks
4. int g := 3; //groups per week

```

Model File

```

1.  import SocialGolfers.dat;
2.
3.  class Group {
4.      name set players;
5.      constraint groupSize {
6.          card(players) = s;
7.      }
8.  }
9.
10. class Week {
11.     Group groupSched[g];
12.     constraint playOncePerWeek {
13.         forall(g1 in 1..g, g2 in g1+1..g)
14.             card(groupSched[g1].players intersect
15.                 groupSched[g2].players)= 0;
16.     }
17. }
18.
19. main class SocialGolfers {
20.
21.     Week weekSched[w];
22.
23.     constraint differentGroups {
24.         forall(w1 in 1..w, w2 in w1+1..w)
25.             forall(g1 in 1..g, g2 in 1..g)
26.                 card(weekSched[w1].groupSched[g1].players intersect
27.                     weekSched[w2].groupSched[g2].players) <= 1;
28.     }
29. }

```

Figure 6.2 – A s-COMMA model of the social golfers problem.

```

1. socialGolfers(L):-
2.   S $= 3,
3.   W $= 4,
4.   G $= 3,
5.
6.   intsets(WEEKSCHED_GROUPSCHED_PLAYERS_,12,1,9),
7.   L = WEEKSCHED_GROUPSCHED_PLAYERS_,
8.
9.   (for(I1,1,W),param(L,S,W,G) do
10.    (for(I2,1,G),param(L,S,W,G,I1) do
11.     V1 is G*(I1-1)+I2,nth(V2,V1,L),
12.     #(V2, V3), V3 $= S
13.    )
14.   ),
15.
16.   (for(I1,1,W),param(L,G) do
17.    (for(G1,1,G),param(L,G,I1) do
18.     (for(G2,G1+1,G),param(L,G,I1,G1) do
19.      V4 is G*(I1-1)+G1,nth(V5,V4,L),
20.      V6 is G*(I1-1)+G2,nth(V7,V6,L),
21.      #(V5 /\ V7, 0)
22.     )
23.    )
24.   ),
25.
26.   (for(W1,1,W),param(L,W,G) do
27.    (for(W2,W1+1,W),param(L,G,W1) do
28.     (for(G1,1,G),param(L,G,W1,W2) do
29.      (for(G2,1,G),param(L,G,W1,W2,G1) do
30.       V8 is G*(W1-1)+G1,nth(V9,V8,L),
31.       V10 is G*(W2-1)+G2,nth(V11,V10,L),
32.       #(V9 /\ V11, V12),V12 $=< 1
33.      )
34.     )
35.    )
36.   ),
37.
38.   label_sets(L).

```

Figure 6.3 – The social golfers problem expressed in ECLⁱPS^e.

6.3 Summary

In this chapter, we have presented the transformation framework for CP. An interesting feature of this framework is the possibility of using different modeling languages as the source of a transformation. This can be seen as an improvement of the state-of-the-art solver-independent architectures, whose mapping process is restricted to a unique modeling language. This new architecture performs a transformation in three main steps: translation from source model to the pivot model, refactoring/optimization on the pivot model, and translation from the pivot model to the target model. A practical example has been introduced to show some interesting aspects of a transformation. In the following chapter, we focus on the implementation of this framework. We present the three main phases of the process and the tools used for supporting them.

From Source to Target

In this chapter we present a complete transformation through the framework. We consider the three main parts: from source to pivot, pivot refactoring, and pivot to target. The process is illustrated by using as example the *s-COMMA*-to-ECL^{PS}_e transformation. At the end of the chapter, we discuss some experiments performed on the architecture.

7.1 From source to pivot

The transformation process from the source to the pivot model requires the metamodel (KM3) of the source, the concrete syntax (TCS) of the source, and the transformation rules from the source to the pivot. Figure 7.1 depicts three classes of the *s-COMMA* metamodel in KM3, the corresponding metamodel using UML class diagram notation is illustrated on the right side of the figure.

```

1.  class Model {
2.      attribute name : String;
3.      reference modelElements [0-*] container : ModelElement;
4.  }
5.
6.  abstract class ModelElement {
7.      attribute name : String;
8.  }
9.
10. class Class extends ModelElement {
11.     attribute isMain : Boolean;
12.     reference superClass [0-1] : Class;
13.     reference solvingOpts [0-3] container : SolvingOpt;
14.     reference attributes [0-*] container : Attribute;
15.     reference constraintZones [0-*] container : ConstraintZone;
16. }

```

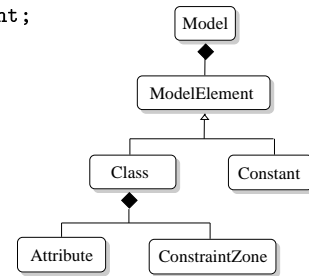


Figure 7.1 – Three classes of the KM3 file of *s-COMMA*.

The metamodel specifies that a *s-COMMA* model is composed of an undetermined number of *ModelElement* objects. The class representing model elements is abstract and it is stated as the superclass of two metamodel concepts: *Class* and *Constant*. The *Class* class represents *s-COMMA* classes and it is composed of attributes and constraint zones. It inherits from the *ModelElement* class its name and it can be defined as the main class of the model using the *isMain* attribute. A *s-COMMA* class can inherit from a superclass and it can also contain solving options.

The class representing attributes is depicted in Figure 7.2. It serves as superclass of variables and objects. The **Variable** class is stated at line 5 and it can be defined as a set using the **isSet** attribute. It also has optional references to the **Array** and to the **Domain** concept.

```

1. class Attribute {
2.   attribute name : String;
3. }
4
5. class Variable extends Attribute {
6.   attribute type : String;
7.   attribute isSet : Boolean;
8.   reference array [0-1] container : Array;
9.   reference domain [0-1] container : Domain;
10. }

```

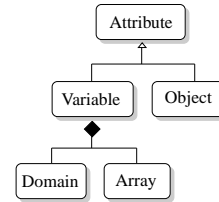


Figure 7.2 – Attributes and variables in the KM3.

The KM3 file concerning the constraint zones is depicted in Figure 7.3. The **ConstraintZone** concept consists of a set of constraint zone elements. Three kinds of constraint zone elements are defined: **IfElse**, **Forall** and **ConstraintStatement**. For instance, the **IfElse** statement is composed of the condition and two set of constraint zone elements. The first set responds to a true condition, and the second one to a false condition. The **Constraint** class is depicted at line 16. It is composed of an **Expression** and of its optional consistency level. The object hierarchy below the **Expression** class can be seen in the Flat s-COMMA metamodel (Figure 5.44).

```

1. class ConstraintZone {
2.   attribute name : String;
3.   reference constraintZoneElements [0-]* container : ConstraintZoneElements;
4. }
5
6. abstract class ConstraintZoneElement {}
7
8. class IfElse extends ConstraintZoneElement {
9.   reference condition container : Expression;
10.  reference trueCtrls [1-]* ordered container :
11.    ConstraintZoneElement;
12.  reference falseCtrls [0-]* ordered container :
13.    ConstraintZoneElement;
14. }
15
16. class Constraint extends ConstraintZoneElement {
17.   attribute consLevel [0-1] : String;
18.   reference assertion container : Expression;
19. }

```

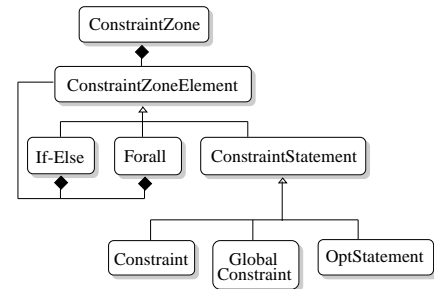


Figure 7.3 – Constraint zones and statements in the KM3.

Figure 7.4 depicts some templates of the s-COMMA TCS file. The first template defines a model, which is composed of a set of model elements. The **main context** keywords are used to create a main symbol table. At line 5, the template for the **ModelElement** concept is stated. It

corresponds to an abstract concept in the metamodel, being necessary to declare it as abstract in the TCS. The `Class` template is defined at line 7. A class declaration is added to the symbol table by means of the `addToContext` keyword. The syntactic structure of a `Class` begins with the optional token `main`, which defines the main class of the model. The reserved word `class` and the class name follow. Then, two optional structures are stated. One is used to define a superclass, while the other one states the solving options. The `refersTo=name` statement is used to get the name of the superclass. Finally, a pair of curly bracket symbols encloses the attributes and constraint zones of a class. The last template defines the syntax of a variable, which is defined with a type, an optional `set` token, and a name. The optional array and domain elements follow, ended by a semicolon token.

```

1.  template Model main context
2.      : modelElements
3.      ;
4.
5.  template ModelElement abstract;
6.
7.  template Class context addToContext
8.      : (isMain ? "main") "class" name
9.        (isDefined(superClass) ? "extends" superClass{refersTo=name})
10.       (isDefined(solvingOpts) ? solvingOpts)
11.       "{"
12.         attributes
13.         constraintZones
14.       "}"
15.       ;
16.
17.  template Attribute abstract;
18.
19.  template Variable addToContext
20.      : type (isSet ? "set")
21.        name (isDefined(array) ? array)
22.        (isDefined(domain) ? "in" domain) ";"
23.      ;

```

Figure 7.4 – Some templates of the TCS file of *s-COMMA*.

Once the KM3 and TCS are defined, the transformation from the source to the pivot is performed by means of ATL rules. Figure 7.5 depicts two transformation rules from *s-COMMA* to the pivot. The rules include only one-to-one transformations since every construct of *s-COMMA* is supported by the pivot.

Remark

The pivot model has been designed to support as much as possible the features of most CP languages, for instance variables of different types, data structures such as arrays and objects, first-order constraints, common global constraints, and control statements. The main idea is to cover a wide range of constructs to facilitate the integration of new translators to the architecture.

```

1.  rule ModelToModel {
2.      from
3.          s : sComma!Model (
4.              )
5.      to
6.          t : Pivot!Model(
7.              modelElements <- s.modelElements
8.          )
9.  }
10.

11. rule VariableToVariable {
12.     from
13.         s : sComma!Variable (
14.             )
15.     to
16.         t : Pivot!Variable(
17.             type    <- s.type,
18.             isSet   <- s.isSet,
19.             name    <- s.name,
20.             array   <- s.array,
21.             domain  <- s.domain
22.         )
23. }

```

Figure 7.5 – Two ATL rules for a transformation from s-COMMA to pivot.

7.2 Pivot refactoring

The pivot only requires a metamodel and the transformation rules to refine it. No TCS file is required. A syntax structure for the pivot is unnecessary since the whole set of transformations is applied only over the concepts defined in its metamodel.

Remark

The pivot metamodel has been designed to be independent from CP languages, i.e. it has no syntax and the constructs supported do not depend on a particular modeling or solver language. This can be contrasted with the state-of-the-art architectures, in which the intermediate language is strongly tied to the syntax and constructs of the modeling language.

Figure 7.6 depicts the main concepts of the pivot metamodel, several concepts are shared with the s-COMMA metamodel. This is due to both metamodels represent CP concepts, e.g. variables, constraints and statements. However, the pivot metamodel is somewhat larger. For instance, it admits classes containing constant declarations. It also provides support for records, which are included in some CP languages, such as OPL and Zinc. Moreover, it includes the predicate concept to handle CLP languages.

7.2.1 Refactoring phase

With the aim of bridging the gap between the source and the target model we have defined several steps of pivot model refactoring. These steps are commonly needed in several transformations from modeling to solver languages. The idea is to refine and to optimize a model to fit as much as possible with the target language concepts. This phase is implemented in several model transformations over the pivot model, and it corresponds to the most complex part of the whole transformation process. The refactoring steps involved have been encapsulated in a set of ATL procedures, which can be reused once a new language is added to the framework.

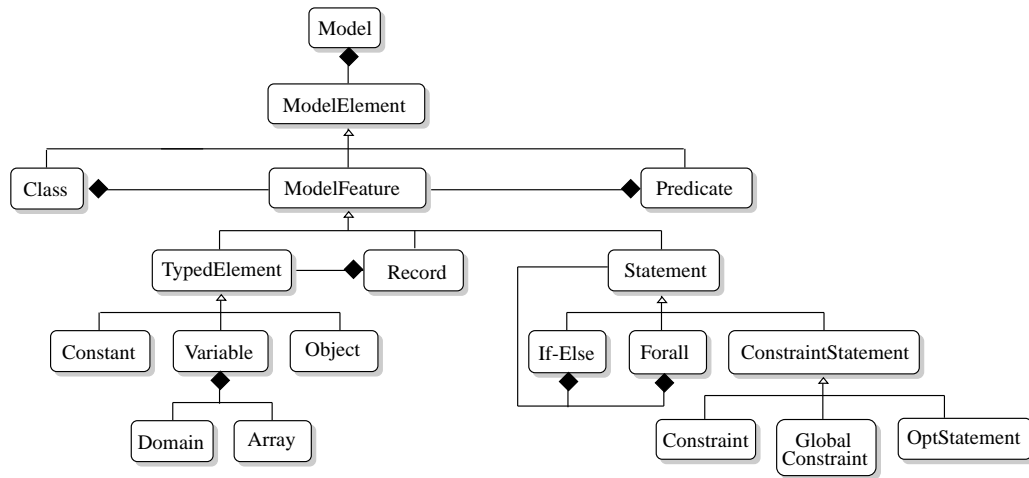


Figure 7.6 – A fragment of the pivot metamodel.

Remark

Since the complex refining work is always done on the pivot, the rules from/to pivot become simpler, and as a consequence the integration of new translators is facilitated.

To simplify the explanations of complex transformations we have defined a pseudo-code language based on ATL. The notations of this language are defined in the following.

7.2.1.1 Rule notations

Figure 7.7 depicts a simple transformation rule from a language called **Source** to a language called **Target**. The rule is called **AToA**, and the type of both concepts to be mapped is denoted by **A**. The rule matches four attributes, from **attribute1** to **attribute4**. The same rule can be expressed in the pseudo-code language as ‘**s**: **A** => **t**: **A**’.

```

1. rule AToA {
2.   from
3.     s : Source!A (
4.     )
5.   to
6.     t : Target!A(
7.       attribute1 <- s.attribute1,
8.       attribute2 <- s.attribute2,
9.       attribute3 <- s.attribute3,
10.      attribute4 <- s.attribute4
11.    )
12. }
  
```

Figure 7.7 – An example of transformation rule.

The left part of the pseudo-code rule (**s** :) corresponds to the source, and the right part (**t** :) to the target. The type of both **s** and **t** is denoted by **A**. Since the matching is performed between concepts having the same type, we assume that every attribute held by the source is implicitly matched to its corresponding one on the target. In the example, the four attributes of the source are matched to the four attributes of the target.

This same rule can be filtered using the **where** keyword followed by a boolean expression. For instance, the following rule allows the matching only if the **name** attribute is defined in **s**. The **isDefined** statement is a function call representing the corresponding call to an ATL helper.

```
s: A where isDefined(s.name) => t: A
```

It is also possible to customize a matching for an attribute. For instance, we explicitly state below that the **name** attribute of **t** must be generated as the concatenation of the strings represented by the **name** and **surname** attributes of **s**. The other attributes of **s** are simply duplicated.

```
s: A where isDefined(s.name) => t: A { name <- s.name + s.surname }
```

Additionally, if the types or structures of entities involved in a transformation are not the same, only the shared attributes (having compatible types) and explicit matchings (**id** <- **s.name** in the next example) are performed.

```
s: A where isDefined(s.name) => t: B { id <- s.name }
```

A collection of entities can be created from one source entity by specifying a sequence type for the target entity, as follows,

```
s: A => t: Sequence of B(s.elements)
```

where **elements** is an attribute of **s** corresponding to a sequence of entities. There are two cases: either the entities of **s.elements** match the **B** type, or other rules must describe how to transform these entities to some entities conformly to **B**.

7.2.1.2 Pivot refactoring rules

The refactoring steps applied on the pivot are very similar to the ones performed on the s-COMMA-to-Flat s-COMMA transformation. For instance: composition flattening, loop unrolling, enumeration substitution, data substitution, conditional removal, auxiliary variables insertion and expression evaluation. In the following paragraphs we give an overview of this process by presenting four refactoring phases. We use the pseudo-code language introduced to illustrate the transformation rules.

Composition flattening

This refactoring step replaces objects by their attributes and constraints. To prevent name conflicts, the names of attributes are prefixed with the name of objects. In Figure 7.8, the first rule (lines 1 and 2) generates a sequence of model features (e.g. variables and constraints), which correspond to the elements encapsulated in the object. If this generated model element corresponds to a variable, the second rule acts (lines 3 and 4). The **parentIsObject** function is

used to test whether the variable is contained in an object. Then, the rule explicitly assign a new value to the **name** attribute of the generated variable by concatenating four strings. The result of this transformation on the s-COMMA object entities of the social golfers model is depicted in Figure 7.9.

```

1.  s: Object =>
2.    t: Sequence of ModelFeature (s.modelFeatures)
3.  s: Variable where parentIsObject(s) =>
4.    t: Variable { name <- s.parent.name + '_' + s.name + '_' }
```

Figure 7.8 – The composition flattening transformation rule.

<pre> //Before flattening main class SocialGolfers { Week weekSched[w]; ... } class Week { Group groupSched[g]; ... } class Group { name set players; ... }</pre>	<pre> //After flattening name set weekSched_groupSched_players_[g*w];</pre>
---	---

Figure 7.9 – Composition flattening on the social golfers problem.

The name of the new array is generated from the concatenation of the names in the objects hierarchy. Since the **weekSched** array is composed of **Week** objects, the prefix of the new name is **weekSched** followed by **groupSched** and **players**. The size of the array is given by $g \times w$. Finally, as we mentioned at the end of Section 6.2, when transforming an array of objects containing constraints, the set of constraints is encapsulated in a **forall** statement. The loop variable of this statement iterates from 1 to the size of the array.

Note

This process differs from the composition flattening in Flat s-COMMA. The use of loops in this implementation allows us to encapsulate constraints (resulting from the flattening) within **forall** statements, instead of unrolling them.

Enumeration substitution

This rule substitutes enumerations by integer values (see Figure 7.10). In the rule, three **Variable** elements are matched, **domain** is matched to **d**, which is computed by the rule stated at line 3. The size of the domain is given by the **getSize** function, which returns the number of elements contained in the enumeration. The result on the social golfers problem is shown in Figure 7.11.

```

1.  s: Variable where isEnum(s.type) =>
2.      t: Variable {name <- s.name, type <- "int", domain <- d} and
3.      d: Domain {lower <- 1, upper <- getSize(s.type)}

```

Figure 7.10 – The enumeration substitution transformation rule.

<pre> //Before enumeration substitution enum name := {a,b,c,d,e,f,g,h,i}; name set players; </pre>	<pre> //After enumeration substitution name set players in [1,9]; </pre>
--	--

Figure 7.11 – Enumeration substitution on the social golfers problem.

Forall unrolling

This step unrolls **forall** loops, i.e. the loop is replaced by the whole set of constraint entities that it implicitly contains. In the rule depicted in Figure 7.12, **foreach** is a function taking as first parameter an iterator definition and as second parameter the statement to repeat. The function **replace** takes three parameters: the entity to replace, the entity to put instead and the entities to process. Thus, the sequence of constraint is initialized with all the constraints returned by the **foreach** function, which generates **s.start** - **s.end** times the set of constraints within loop entities.

```

1.  s: Forall =>
2.      t: Sequence of Constraint(foreach(it in s.start .. s.end,
3.          replace(s.loopVar,it,s.statements)))

```

Figure 7.12 – The forall unrolling transformation rule.

Auxiliary variable insertion

In some CLP languages, it is not possible to use the bracket operator (`'[]'`) to access lists, being necessary to introduce local variables and **nth** predicate calls (as we have shown in Figure 6.3). Figure 7.14 depicts the transformation rules of this phase, and a result is shown in Figure 7.13. This rule acts over one-dimensional arrays stated as operand in expressions. The **VariableOccurrence** concept represents a variable stated as operand in an expression¹. At lines 3 and 4, a new auxiliary variable is created with its corresponding variable occurrence (**V1** in the example). The function **getNextAuxVarName()** returns the name of the next auxiliary variable. The following statement builds the **nth** function call. Its parameters are matched with a sequence of expression objects composed of the variable occurrence corresponding to the new auxiliary variable, the row index of the array (**X** in the example), and a variable occurrence corresponding to the array **L**. The variable **V1** will be then used to represent **L[X]** within expressions.

¹The **Expression** and the **VariableOccurrence** concepts can be seen in Figure 5.44.

```

1.  s: VariableOccurrence where (isDefined(s.array.row)
2.                                and isUndefined(s.array.col)) =>
3.    t: Variable{name <- getNextAuxVarName()} and
4.    u: VariableOccurrence{declaration <- t} and
5.    v: FunctionCall {name <- "nth",
6.                      parameters <- Sequence of Expression(u,s.array.row,w)} and
7.    w: VariableOccurrence{declaration <- s.declaration}

```

Figure 7.13 – Auxiliary variable insertion transformation rules.

```

//Before rule
L[X]

//After rule
nth(V1,X,L)

```

Figure 7.14 – Auxiliary variable insertion process.

Remark

Let us note that the pivot metamodel can be extended. For instance, if a new language is plugged to the framework and no support exists for some of its features, e.g. a global constraint. It suffices to add to the pivot the concept representing such a global constraint or to add the corresponding refactoring phase to transform the global constraint in a representation (if exists) supported by the target language.

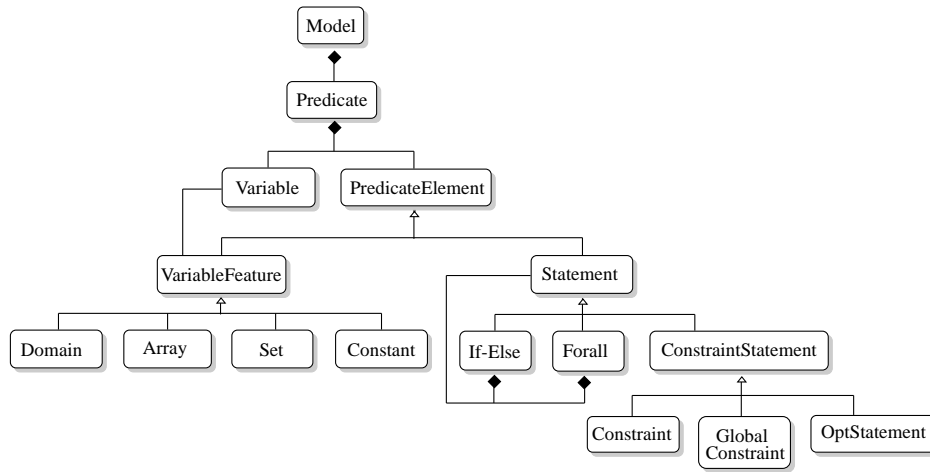
7.3 From pivot to target

The transformation from pivot to target is similar to the source-to-pivot transformation. Mainly one-to-one transformation rules are performed. Like the first step, this phase requires the KM3, the TCS of the target language, and the transformation rules to match with the pivot metamodel.

Note

A same TCS file can be used for parsing a source language and for generating target files in that language. This avoid us to create an ATL query for the code generation tasks.

Figure 7.15 depicts the main concepts of the ECL^iPS^e metamodel. An ECL^iPS^e model can be seen as a set of Prolog-like predicates. Each predicate is composed of variables, and predicate features. A predicate feature is specialized in two classes: **VariableFeature** and **Statement**. Four classes inherits from **VariableFeature**: **Domain**, **Array**, **Set** and **Constant**. Let us note that the structure of this class hierarchy differs from previous metamodels. It has been defined in this manner to correctly handle the different variable declarations provided by ECL^iPS^e . Finally, the

Figure 7.15 – A fragment of the ECLⁱPS^e metamodel.

sub-elements of the **Statement** concept are very similar to previous metamodels.

Figure 7.16 depicts three templates of the ECLⁱPS^e TCS file. The first template defines the model, which is composed of a set of predicates. Predicates are defined with a name and a set of input parameters separated by a comma.

```

1.  template Model main context
2.      : predicates
3.      ;
4.
5.  template Predicate context addToContext
6.      : name _____ → queens(N, Board) :-
7.      "(" parameters{separator=","} " " _____ →
8.      ":-" _____ →
9.      predicateElements{separator=","} solvingOpts "." ...
10.     ;
11. template PredicateElement abstract;
12. template VariableFeature abstract;
13.
14. template Array
15.     "dim" _____ → dim(Board, [N]),
16.     "(" varName{refersTo=name} _____ →
17.     "," "[" row (isDefined(col) ? "," col) "]" " " _____ →
18.     ;

```

Figure 7.16 – Five templates of the TCS file of ECLⁱPS^e.

A parameter corresponds to a **Variable** object. The parameters are enclosed by a pair of round bracket tokens and followed by the `:-` Prolog symbol. A set of predicate elements follows, which are also separated by a comma token. The predicate declaration ends with the solving options followed by a dot symbol. The **PredicateElement** and the **VariableFeature** are abstract templates. The **Array** template is defined at line 18. The `dim` reserved word begins the array

declaration. The name of the variable and the dimensions of the array are then included. The `refersTo=name` statement is used to get the name of the variable, which is defined within the `Variable` concept. The `col` attribute is optional, being only used for two-dimensional arrays.

7.4 Transformation process

As presented in Section 5.3.2.5, the compilation of the TCS file with the corresponding KM3 metamodel generates the necessary lexers, parsers and code generators. The complete transformation process is shown in Figure 7.17. The model file of the source language (the `s-COMMA` file) is the input of the system. This file is transformed to the corresponding `s-COMMA` XMI file (injection phase). The `s-COMMA` XMI is transformed through the ATL rules to the pivot XMI file. Over this XMI file, the whole set of refactoring steps is performed. The refined XMI pivot file is mapped to the XMI file of the target language (ECLⁱPS^e). Finally, the model of the target language (the ECLⁱPS^e file) is generated (extraction phase).

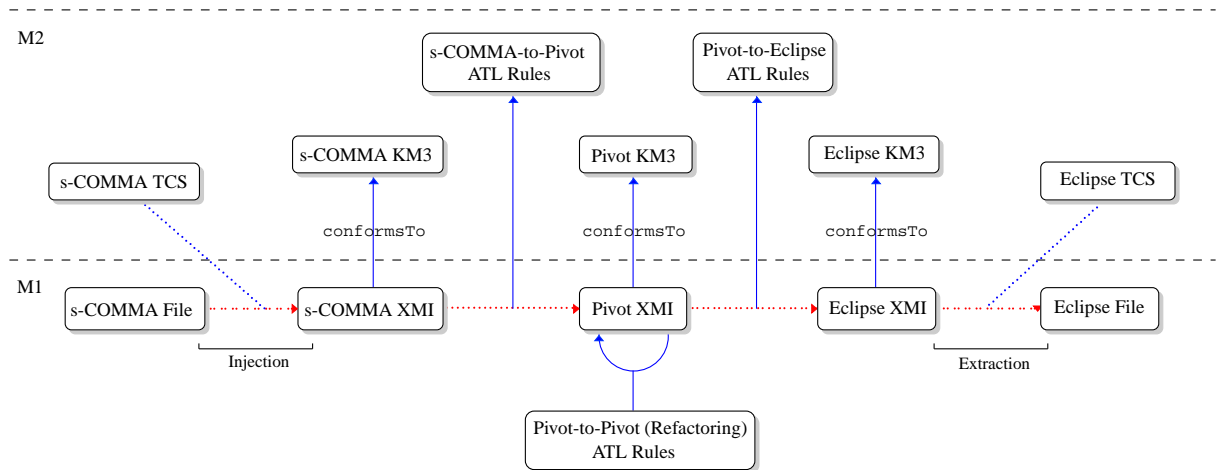


Figure 7.17 – The transformation process on the example of `s-COMMA` to ECLⁱPS^e.

7.4.1 Selecting the refactoring steps.

Applying the whole set of refactoring steps presented in Section 7.2 is not necessary in every transformation chain. Indeed, it clearly depends on the modeling structures of the source and target languages. The idea is to use most of constructs supported by the target language to have a target model close, in terms of constructs, to our source model. For instance, in a `s-COMMA` to ECLⁱPS^e translation, we should transform the objects using the composition flattening step. We also may need the enumeration substitution and other refactoring steps such as the use of local variables and `nth` predicates. Optionally, we may select the expression simplification step.

Remark

This feature may be contrasted with previous approaches (e.g. Zinc, s-COMMA), where the refactoring steps are always applied. This normally breaks the original structure of the model (e.g. the unrolling loop phase generates a model completely different compared to one with no unrolled loops). The possibility of customizing the steps to be applied on the transformation allows one to transfer the source modeling features to the target model. We believe this may enable readability and understanding on the target model.

The set of refining steps to be applied in a transformation can be chosen by means of Ant scripts [w6w]. Figure 7.18 depicts an Ant script specifying a transformation. The first block (lines 1 to 7) states the transformation from s-COMMA to the pivot and the second block (lines 9 to 14) selects the enumeration substitution refactoring step. Lines 3, 5 and 11 define which metamodels to use and lines 4 and 12 specify which models to process. Lines 6 and 13 correspond to the produced models.

```

1.  <!--s-COMMA to Pivot-->
2.  <am3.atl path="/sCOMMAtoPivot/sCOMMAtoPivot.atl">
3.    <inmodel name="sCOMMA" model="sCOMMA"/>
4.    <inmodel name="IN" model="mysCOMMA"/>
5.    <inmodel name="Pivot" model="Pivot"/>
6.    <outmodel name="OUT" model="myPivot" metamodel="Pivot"/>
7.  </am3.atl>
8.
9.  <!--Enumeration Substitution-->
10. <am3.atl path="/PivotRefining/enumerationSubstitution.atl">
11.   <inmodel name="Pivot" model="Pivot"/>
12.   <inmodel name="IN" model="myPivot"/>
13.   <outmodel name="OUT" model="myPivot" metamodel="Pivot"/>
14. </am3.atl>

```

Figure 7.18 – An Ant script for selecting transformations.

7.5 Experiments

To highlight the performance of this new approach, in terms of translation time, we have tested the s-COMMA to ECLⁱPS^e translation on five CP problems. Table 7.1 depicts the results of this first experiment. The first column gives the problem names. The second column depicts the size (in number of lines) of the s-COMMA source files. The following columns correspond to the time of atomic steps (in seconds): model injection (Inject), transformations from s-COMMA to pivot (s-to-p), composition flattening (Comp), enumeration substitution (Enum), transformations from pivot to ECLⁱPS^e (p-to-E), and target file extraction (Extract). The next column details the total time of the complete transformation, and the last column shows the number of lines of the generated ECLⁱPS^e files.

The results show that the text processing phases (injection and extraction) are efficient, but we may remark that the given problems are concisely stated (maximum of 112 lines). The transformation s-COMMA to pivot is slower than the transformation pivot to ECLⁱPS^e. This is explained by the refactoring phases performed on the pivot that reduce the number of elements

Problems	Size	Inject	s-to-p	Comp	Enum	p-to-E	Extract	Total	Size
Golfers	42	0.107	0.169	0.340	0.080	0.025	0.050	0.771	38
Engine	112	0.106	0.186	0.641	0.146	0.031	0.056	1.166	78
Send	16	0.129	0.160	0.273	-	0.021	0.068	0.651	21
Stable	46	0.128	0.202	0.469	0.085	0.027	0.040	0.951	26
10-queens	14	0.132	0.147	0.252	-	0.017	0.016	0.564	16

Table 7.1 – Times of complete transformation chains.

to handle on the pivot to ECLⁱPS^e step. The composition flattening step is the more expensive. In particular, the Engine problem exhibits the slowest running time since it contains a bigger number of object compositions. In summary, considering the whole set of phases involved, the results show reasonable translation times.

The second test we performed aims at analyzing scaling our approach. To this end we have applied the loop unrolling step to six versions (from n=50 to n=100) of the n-queens problem. Table 7.2 depicts the results of this second test. Columns two to eight show the atomic steps of the transformation (in seconds). Column nine contains the sizes (in number of lines) of generated ECLⁱPS^e files, which have been heavily impacted by the loop unrolling step (since the size of the unrolled loops depends on n). At the final column, a ratio exhibits the efficiency of a transformation chain considering the execution time per generated lines. Considering the significant differences of model sizes (from 7505 to 30005 lines) the values indicate this ratio slowly increases, showing that the approach can be used for large models.

Problems	Inject	s-to-P	Comp	Forall	P-to-E	Extract	Total	Size	Total/Size
50-queens	0.132	0.147	0.252	32.773	16.21	1.059	50.573	7505	≈0.0067
60-queens	0.132	0.147	0.252	49.247	28.577	1.509	79.864	10805	≈0.0074
70-queens	0.132	0.147	0.252	68.283	47.951	2.033	118.798	14705	≈0.0080
80-queens	0.132	0.147	0.252	92.693	81.401	2.689	177.314	19205	≈0.0092
90-queens	0.132	0.147	0.252	126.338	123.743	3.390	254.002	24305	≈0.0104
100-queens	0.132	0.147	0.252	165.395	182.871	4.193	352.990	30005	≈0.0117

Table 7.2 – Time of complete transformation chains of the n-queens problem.

7.6 Summary


In this chapter, we have presented the complete transformation process performed by the framework. The implementation of the three main phases has been explained. The first and the last phase concern the source and the target language, respectively. The implementation of both phases requires the definition of a metamodel, a TCS file, and a set of transformation rules to match with the pivot. The middle phase is responsible for applying a set of refactoring steps on the pivot. This model is a key component of the architecture since the most complex transformations are performed on it. This allows us to simplify the transformation from/to the pivot and consequently to facilitate the addition of new translators to the platform. The pivot model is also independent from modeling and solver languages, i.e. it has no syntax and

the constructs supported do not depend on a particular modeling or solver language. Another interesting feature of the architecture is that the set of single steps included in a transformation can be customized. This allows us to obtain a target model closer, in terms of constructs, to our source model.

The development of this framework corresponds to the current work of the author and it is in a preliminary stage. Only three languages have been plugged to the framework (*s*-COMMA, ECLⁱPS^e and RealPaver). Thus, at the moment, it is not possible to completely ensure that the pivot is able to support all the constructs provided by every existing modeling language. However, we believe that it represents a considerable basis to support a large list of common constructs. Another limitation of the framework is that only the declarative parts of models can be processed since it is not possible to partially execute a computer program that builds the constraint store. In the following chapter we conclude the thesis and we propose some future research perspectives.

CHAPTER 8

Conclusion

 In this thesis, we have presented two main works: the `s`-COMMA platform and a model-driven transformation framework for CP languages. In this chapter, we recall the most important aspects of these two approaches, we discuss their limitations and we give the corresponding concluding remarks. We finish the chapter by presenting some future research directions.

8.1 `s`-COMMA

`s`-COMMA is the first work we presented in this thesis. Such a system involves an object-oriented language for modeling CP problems and a solver-independent architecture. This approach is the result of an investigation of several important concerns in the development of modern CP architectures. Several innovations and advantages can be found:

- The object-oriented style provided allows us to elegantly capture the inherent structure of problems. The problem can be divided in subproblems to be captured in single classes. The result is in general a more modular model, which motivates the reuse and facilitates the management of constraint models.
- The `s`-COMMA language can be naturally represented through graphical components. The `s`-COMMA GUI is the graphical interface of the platform, allowing users to obtain a visual and a more concise representation of models.
- The `s`-COMMA language can be extended. An extension mechanism is able to adapt the modeling language to further updates of the solving layer. Such a mechanism works by defining extension files on which the rules of the translation between the new functionalities and `s`-COMMA are defined.
- The search process is a main phase of the problem resolution. Accordingly, a simple parameter formalism is provided. This formalism permits to define ordering heuristics over classes, and consistency levels over objects, classes and constraints.
- `s`-COMMA is supported by a flexible and extensible solver-independent architecture. This architecture enables users to process one model with different solvers in order to facilitate experimentation tasks. Additionally, the platform is open to be connected with new solvers. This task can be carried out via powerful model transformation techniques.

We believe `s`-COMMA is a complete approach for modeling a wide range of CP problems, its expressiveness is considerable and it can even be increased by extension mechanisms. The object-

oriented style is the basis to get concise and elegant models. Such models can also be tuned to obtain efficient search processes. The graphical tool is a useful option for users looking for a visual modeling perspective, and the solver-independent architecture is an excellent support for experimentation tasks.

Finally, it is necessary to mention some limitations, which are mainly related to the capabilities of the underlying solvers. For instance, the language features of *s-COMMA* not supported by solvers cannot always be successfully mapped nor transformed. A common example is the use of real numbers in *s-COMMA*, which are not supported by finite domain solvers (e.g. Gecode). Another example concerns the use of interval solvers (e.g. RealPaver), in which is not possible to check the equality of values, allowing only the use of some relation operators (\leq , \geq , $=$). The same problem occurs with the heuristic ordering and consistency level parameters, just the options provided by the chosen solver can be used at the modeling phase. The current implemented solution is to inform the user with warning messages.

8.2 Transformation framework for CP languages

We have presented a new framework for CP model transformations as the second work of this thesis. This framework is supported by a set of MDE tools and by an independent pivot model to which different languages can be mapped. In this framework, a transformation chain is made of three main steps: from the source to the pivot model, refining of the pivot model, and from the pivot model to the target. This new approach follows important advantages.

- Modelers are able to use their favorite language and to solve the problem by means of the best known solving technology. Experimentation of new solvers may also be easier, as a collection of benchmarks in this new language can be built from different sources.
- Refactoring and optimization steps are always implemented over the pivot. In this way, the translation from/to the pivot becomes simpler, facilitating the addition of new translators. Additionally, the refactoring phases to be applied in a transformation can be selected to get a target model closer, in terms of modeling constructs, to the source model.

The work done on this framework can be seen as an improvement of the architecture implemented in *s-COMMA*. The framework is in preliminary stage and the main limitation is that only the modeling fragments of languages (i.e. the declarative part) can be processed since it is not possible to partially execute a computer program that builds the constraint store.

8.3 Future research directions

Solver-independent architectures and model transformation in constraint programming is a recent trend. Just a few platforms involving both concerns have been developed. We believe that extension or improvement of such platforms may lead to a wide future work. For instance, *s-COMMA* can be extended in several ways, the more visible way is to increase the number of underlying solvers, which may belong to the CP field as well as to the mathematical field (e.g. AMPL, GAMS). The use of solvers using local search techniques will be interesting too. This may imply facing up to several new challenges in terms of model transformation concerns.

We are also interested in extending \mathfrak{s} -COMMA to be used in the dynamic CSP framework [GF03, MF90]. We currently support the definition of activity and compatibility constraints, but we do not support activity objects (the creation of an object is subject to constraints) and the dynamic definition of object attributes (the definition of attributes is subject to constraints). This will allow us to state dynamic CP models in a more elegant way.

The transformation framework we presented can be improved as well. As in \mathfrak{s} -COMMA the most visible direction to follow is to extend the list of translators supported. To study and implement new refactoring/optimization pivot phases such as the automatic transformation of global constraints is another aspect to be considered. We also want to better manage complex CP models transformation chains. Models could be qualified to determine their level of structure and to automatically choose the required refactoring steps according to the target language.

Our last future goal is related to the MD-transformation tools. We have used ATL as the transformation language over the entire framework and sometimes the implementation of some complex transformations on the pivot was quite difficult to carry out. We believe it may be interesting to extend ATL with some built-ins to perform complex tasks (e.g. composition flattening, loop unrolling, etc.). Such an extension may probably lead to the definition of a new language completely aimed at CP model transformation.

Appendixes

APPENDIX A

Grammars

A.1 s-COMMA Grammar

In this appendix we describe the grammar of s-COMMA and Flat s-COMMA. The description is done by means of EBNF using the following conventions: Angle brackets are used to denote non-terminals (e.g. $\langle \textit{Class-Body} \rangle$). Bold font and underlined bold font are used to denote terminals (e.g. **class**, ;). Square brackets denotes optional items (e.g. $[\langle \textit{Array} \rangle]$). Square brackets with a plus symbol defines sequences of one or more items (e.g. $[\langle \textit{Class} \rangle]^+$). Square brackets with a star symbol are used for sequences of zero or more items (e.g. $[\langle \textit{Import} \rangle]^*$), and square brackets with a range $\{a, b\}$ defines sequences from a to b items (e.g. $[\langle \textit{Solving-Option} \rangle]^{\{0,2\}}$)

Model

$$\begin{aligned} \langle \textit{Model} \rangle &::= [\langle \textit{Import} \rangle]^* [\langle \textit{Class} \rangle]^* \\ \langle \textit{Import} \rangle &::= \textbf{import} \langle \textit{Path} \rangle \\ \langle \textit{Class} \rangle &::= \textbf{[main] class} \langle \textit{Identifier} \rangle [\textbf{extends} \langle \textit{Identifier} \rangle] [\underline{[\langle \textit{Solving-Options} \rangle]}] \\ &\quad \{ \langle \textit{Class-Body} \rangle \} \\ \langle \textit{Class-Body} \rangle &::= [\langle \textit{Attribute} \rangle]^* [\langle \textit{Constraint-Zone} \rangle]^* \\ \langle \textit{Path} \rangle &::= [\langle \textit{Identifier} \rangle \underline{,}]^* \langle \textit{Identifier} \rangle \underline{;} \end{aligned}$$

Attributes

$$\begin{aligned} \langle \textit{Attribute} \rangle &::= \langle \textit{Variable} \rangle \mid \langle \textit{Object} \rangle \\ \langle \textit{Variable} \rangle &::= \langle \textit{Var-Type} \rangle [\textbf{set}] \langle \textit{Mult-Id-Def} \rangle [\textbf{in} \langle \textit{Domain} \rangle]; \\ \langle \textit{Mult-Id-Def} \rangle &::= \langle \textit{Identifier} \rangle [\langle \textit{Array} \rangle] [\underline{,} \langle \textit{Identifier} \rangle [\langle \textit{Array} \rangle]]^* \\ \langle \textit{Object} \rangle &::= [\underline{[\langle \textit{Cons-Level} \rangle]}] \langle \textit{Mult-Id-Def} \rangle; \\ \langle \textit{Var-Type} \rangle &::= \langle \textit{Basic-Type} \rangle \mid \langle \textit{Identifier} \rangle \\ \langle \textit{Array} \rangle &::= [\underline{[\langle \textit{Array-Size} \rangle]}] [\underline{,} \langle \textit{Array-Size} \rangle] \\ \langle \textit{Array-Size} \rangle &::= \langle \textit{Int-Expr} \rangle \mid \langle \textit{Identifier} \rangle \\ \langle \textit{Basic-Type} \rangle &::= \textbf{int} \mid \textbf{real} \mid \textbf{bool} \\ \langle \textit{Domain} \rangle &::= [\underline{[\langle \textit{Bound} \rangle]}] [\underline{,} \langle \textit{Bound} \rangle] \\ \langle \textit{Bound} \rangle &::= \langle \textit{Num-Expr} \rangle \mid \langle \textit{Identifier} \rangle \end{aligned}$$

Constraints

$$\begin{aligned} \langle \textit{Constraint-Zone} \rangle &::= \textbf{constraint} \langle \textit{Identifier} \rangle \{ \underline{\langle \textit{Constraint-Body} \rangle} \} \\ \langle \textit{Constraint-Body} \rangle &::= [\langle \textit{Constraint} \rangle \mid \langle \textit{Global-Constraint} \rangle \mid \langle \textit{Compatibility-Constraint} \rangle \mid \\ &\quad \langle \textit{Forall} \rangle \mid \langle \textit{If-Else} \rangle]^* [\langle \textit{Optimization} \rangle] \\ \langle \textit{Constraint} \rangle &::= [\underline{[\langle \textit{Cons-Level} \rangle]}] \langle \textit{Expr} \rangle \underline{;} \end{aligned}$$

$\langle \text{Compatibility-Constraint} \rangle ::= \text{compatibility} (\langle \text{Access} \rangle [_ \langle \text{Access} \rangle]^*) \{ [\langle \text{Valid-Tuples} \rangle]^+ \}$
 $\langle \text{Valid-Tuples} \rangle ::= (\langle \text{Literal} \rangle [_ \langle \text{Literal} \rangle]^*)$
 $\langle \text{Literal} \rangle ::= \langle \text{Value} \rangle \mid \langle \text{String} \rangle$
 $\langle \text{Global-Constraint} \rangle ::= \langle \text{Identifier} \rangle (\langle \text{Param} \rangle [_ \langle \text{Param} \rangle]^*)$
 $\langle \text{Param} \rangle ::= \langle \text{Access} \rangle \mid \langle \text{Literal} \rangle$

Expressions

$\langle \text{Expr} \rangle ::= \langle \text{Expr-Imp} \rangle [< - > \langle \text{Expr-Imp} \rangle]^*$
 $\langle \text{Expr-Imp} \rangle ::= \langle \text{Expr-Or} \rangle [\langle \text{Op-Imp} \rangle \langle \text{Expr-Or} \rangle]^*$
 $\langle \text{Op-Imp} \rangle ::= - > \mid < -$
 $\langle \text{Expr-Or} \rangle ::= \langle \text{Expr-And} \rangle [\langle \text{Op-Or} \rangle \langle \text{Expr-And} \rangle]^*$
 $\langle \text{Op-Or} \rangle ::= \text{xor} \mid \text{or}$
 $\langle \text{Expr-And} \rangle ::= \langle \text{Expr-Not} \rangle [\text{and} \langle \text{Expr-Not} \rangle]^*$
 $\langle \text{Expr-Not} \rangle ::= [\text{not}]^* \langle \text{Expr-Rel} \rangle$
 $\langle \text{Expr-Rel} \rangle ::= \langle \text{Expr-Set-Rel} \rangle [\langle \text{Op-Rel} \rangle \langle \text{Expr-Set-Rel} \rangle]^*$
 $\langle \text{Op-Rel} \rangle ::= < > \mid != \mid = \mid == \mid < \mid > \mid < = \mid > =$
 $\langle \text{Expr-Set-Rel} \rangle ::= \langle \text{Expr-Set-Op} \rangle [\langle \text{Op-Set-Op} \rangle \langle \text{Expr-Set-Op} \rangle]^*$
 $\langle \text{Op-Set-Rel} \rangle ::= \text{subset} \mid \text{superset}$
 $\langle \text{Expr-Set-Op} \rangle ::= \langle \text{Expr-Sum} \rangle [\langle \text{Op-Set-Rel} \rangle \langle \text{Expr-Sum} \rangle]^*$
 $\langle \text{Op-Set-Op} \rangle ::= \text{union} \mid \text{diff} \mid \text{symdiff}$
 $\langle \text{Expr-Sum} \rangle ::= \langle \text{Expr-Prod} \rangle [\langle \text{Op-Sum} \rangle \langle \text{Expr-Prod} \rangle]^*$
 $\langle \text{Op-Sum} \rangle ::= - \mid +$
 $\langle \text{Expr-Prod} \rangle ::= \langle \text{Expr-Int} \rangle [\langle \text{Op-Prod} \rangle \langle \text{Expr-Int} \rangle]^*$
 $\langle \text{Op-Prod} \rangle ::= * \mid /$
 $\langle \text{Expr-Int} \rangle ::= \langle \text{Expr-Expon} \rangle [\text{intersect} \langle \text{Expr-Expon} \rangle]^*$
 $\langle \text{Expr-Expon} \rangle ::= \langle \text{Un-Expr-Min} \rangle [^ \langle \text{Un-Expr-Min} \rangle]^*$
 $\langle \text{Un-Expr-Min} \rangle ::= - \langle \text{Expr-Unit} \rangle \mid [+] \langle \text{Expr-Unit} \rangle$
 $\langle \text{Expr-Unit} \rangle ::= \langle \text{Value} \rangle \mid \langle \text{Access} \rangle \mid \langle \text{Function-Call} \rangle \mid (\langle \text{Expr} \rangle)$
 $\langle \text{Num-Expr} \rangle ::= \langle \text{Num-Expr-Prod} \rangle [\langle \text{Op-Sum} \rangle \langle \text{Num-Expr-Prod} \rangle]^*$
 $\langle \text{Num-Expr-Prod} \rangle ::= \langle \text{Num-Un-Expr-Min} \rangle [\langle \text{Op-Prod} \rangle \langle \text{Num-Un-Expr-Min} \rangle]^*$
 $\langle \text{Op-Prod} \rangle ::= * \mid /$
 $\langle \text{Num-Un-Expr-Min} \rangle ::= - \langle \text{Num-Expr-Unit} \rangle \mid [+] \langle \text{Num-Expr-Unit} \rangle$
 $\langle \text{Num-Expr-Unit} \rangle ::= \langle \text{Integer} \rangle \mid \langle \text{Float} \rangle \mid \langle \text{Identifier} \rangle \mid \langle \text{Function-Call} \rangle \mid (\langle \text{Num-Expr} \rangle)$
 $\langle \text{Int-Expr} \rangle ::= \langle \text{Int-Expr-Prod} \rangle [\langle \text{Op-Sum} \rangle \langle \text{Int-Expr-Prod} \rangle]^*$
 $\langle \text{Int-Expr-Prod} \rangle ::= \langle \text{Int-Un-Expr-Min} \rangle [\langle \text{Op-Prod} \rangle \langle \text{Int-Un-Expr-Min} \rangle]^*$
 $\langle \text{Int-Un-Expr-Min} \rangle ::= - \langle \text{Int-Expr-Unit} \rangle \mid [+] \langle \text{Int-Expr-Unit} \rangle$
 $\langle \text{Int-Expr-Unit} \rangle ::= \langle \text{Integer} \rangle \mid \langle \text{Identifier} \rangle \mid \langle \text{Function-Call} \rangle \mid (\langle \text{Int-Expr} \rangle)$
 $\langle \text{Value} \rangle ::= \langle \text{Integer} \rangle \mid \langle \text{Float} \rangle \mid \langle \text{Boolean} \rangle$
 $\langle \text{Access} \rangle ::= [\langle \text{Identifier} \rangle [\langle \text{Array} \rangle]^* \langle \text{Identifier} \rangle [\langle \text{Array} \rangle]$
 $\langle \text{Function-Call} \rangle ::= \langle \text{Identifier} \rangle (\langle \text{Param} \rangle [_ \langle \text{Param} \rangle]^*)$

Statements

$\langle \text{Forall} \rangle ::= \text{forall} (\langle \text{Loop-Header} \rangle [_ \langle \text{Loop-Header} \rangle]^*) \{ \langle \text{Forall-Body} \rangle \}$
 $\langle \text{Loop-Header} \rangle ::= \langle \text{Identifier} \rangle \text{ in } \langle \text{Value-Set} \rangle$

A.2 Flat s-COMMA Grammar

Model

$\langle \text{Model} \rangle ::= [\langle \text{Variable-Block} \rangle][\langle \text{Constraint-Block} \rangle][\langle \text{Enum-Block} \rangle][\langle \text{Solving-Block} \rangle]$
 $\langle \text{Variable-Block} \rangle ::= \text{variables: } [\langle \text{Variable} \rangle]^*$
 $\langle \text{Constraint-Block} \rangle ::= \text{constraints: } \langle \text{Constraint-Statement} \rangle$
 $\langle \text{Enum-Block} \rangle ::= \text{enum-types: } [\langle \text{Enum-Type} \rangle]^*$
 $\langle \text{Solving-Block} \rangle ::= \text{solving-opts: } \langle \text{Solving-Options} \rangle$

Variables

$\langle \text{Variable} \rangle ::= \langle \text{Var-Type} \rangle [\text{set}] \langle \text{Identifier} \rangle [\langle \text{Array} \rangle] \text{ in } \langle \text{Domain} \rangle;$
 $\langle \text{Var-Type} \rangle ::= \langle \text{Basic-Type} \rangle \mid \langle \text{Identifier} \rangle$
 $\langle \text{Array} \rangle ::= [\langle \text{Integer} \rangle _ \langle \text{Integer} \rangle]$
 $\langle \text{Basic-Type} \rangle ::= \text{int} \mid \text{real} \mid \text{bool}$
 $\langle \text{Domain} \rangle ::= [\langle \text{Bound} \rangle _ \langle \text{Bound} \rangle]$
 $\langle \text{Bound} \rangle ::= \langle \text{Integer} \rangle \mid \langle \text{Float} \rangle$

Constraints

$\langle \text{Constraint-Statement} \rangle ::= [\langle \text{Constraint} \rangle \mid \langle \text{Global-Constraint} \rangle]^* [\langle \text{Optimization} \rangle]$
 $\langle \text{Constraint} \rangle ::= [\langle \text{Cons-Level} \rangle] \langle \text{Expr} \rangle ;$
 $\langle \text{Global-Constraint} \rangle ::= \langle \text{Identifier} \rangle (\langle \text{Param} \rangle [_ \langle \text{Param} \rangle]^*) ;$
 $\langle \text{Param} \rangle ::= \langle \text{Identifier} \rangle \mid \langle \text{Literal} \rangle$
 $\langle \text{Literal} \rangle ::= \langle \text{Value} \rangle \mid \langle \text{String} \rangle$
 $\langle \text{Optimization} \rangle ::= \langle \text{Opt-Value} \rangle \langle \text{Expr} \rangle ;$
 $\langle \text{Opt-Value} \rangle ::= \text{maximize} \mid \text{minimize}$

Expressions

$\langle \text{Expr} \rangle ::= \langle \text{Expr-Imp} \rangle [\langle - \rangle \langle \text{Expr-Imp} \rangle]^*$
 $\langle \text{Expr-Imp} \rangle ::= \langle \text{Expr-Or} \rangle [\langle \text{Op-Imp} \rangle \langle \text{Expr-Or} \rangle]^*$
 $\langle \text{Op-Imp} \rangle ::= -> \mid <-$
 $\langle \text{Expr-Or} \rangle ::= \langle \text{Expr-And} \rangle [\langle \text{Op-Or} \rangle \langle \text{Expr-And} \rangle]^*$
 $\langle \text{Op-Or} \rangle ::= \text{xor} \mid \text{or}$
 $\langle \text{Expr-And} \rangle ::= \langle \text{Expr-Not} \rangle [\text{and} \langle \text{Expr-Not} \rangle]^*$
 $\langle \text{Expr-Not} \rangle ::= [\text{not}]^* \langle \text{Expr-Rel} \rangle$
 $\langle \text{Expr-Rel} \rangle ::= \langle \text{Expr-Set-Rel} \rangle [\langle \text{Op-Rel} \rangle \langle \text{Expr-Set-Rel} \rangle]^*$
 $\langle \text{Op-Rel} \rangle ::= <> \mid != \mid = \mid == \mid < \mid > \mid <= \mid >=$
 $\langle \text{Expr-Set-Rel} \rangle ::= \langle \text{Expr-Set-Op} \rangle [\langle \text{Op-Set-Op} \rangle \langle \text{Expr-Set-Op} \rangle]^*$
 $\langle \text{Op-Set-Op} \rangle ::= \text{subset} \mid \text{superset}$
 $\langle \text{Expr-Set-Op} \rangle ::= \langle \text{Expr-Sum} \rangle [\langle \text{Op-Set-Rel} \rangle \langle \text{Expr-Sum} \rangle]^*$
 $\langle \text{Op-Set-Op} \rangle ::= \text{union} \mid \text{diff} \mid \text{symdiff}$
 $\langle \text{Expr-Sum} \rangle ::= \langle \text{Expr-Prod} \rangle [\langle \text{Op-Sum} \rangle \langle \text{Expr-Prod} \rangle]^*$
 $\langle \text{Op-Sum} \rangle ::= - \mid +$

$$\begin{aligned}
\langle \text{Expr-Prod} \rangle &::= \langle \text{Expr-Int} \rangle [\langle \text{Op-Prod} \rangle \langle \text{Expr-Int} \rangle]^* \\
\langle \text{Op-Prod} \rangle &::= * \mid / \\
\langle \text{Expr-Int} \rangle &::= \langle \text{Expr-Expon} \rangle [\text{intersect } \langle \text{Expr-Expon} \rangle]^* \\
\langle \text{Expr-Expon} \rangle &::= \langle \text{Un-Expr-Min} \rangle [\wedge \langle \text{Un-Expr-Min} \rangle]^* \\
\langle \text{Un-Expr-Min} \rangle &::= - \langle \text{Expr-Unit} \rangle \mid [+] \langle \text{Expr-Unit} \rangle \\
\langle \text{Expr-Unit} \rangle &::= \langle \text{Value} \rangle \mid \langle \text{Identifier} \rangle \mid \langle \text{Function-Call} \rangle \mid \underline{(\langle \text{Expr} \rangle)} \\
\langle \text{Value} \rangle &::= \langle \text{Integer} \rangle \mid \langle \text{Float} \rangle \mid \langle \text{Boolean} \rangle \\
\langle \text{Function-Call} \rangle &::= \langle \text{Identifier} \rangle (\langle \text{Param} \rangle [_ \langle \text{Param} \rangle]^*) \\
\langle \text{Optimization} \rangle &::= \langle \text{Opt-Value} \rangle \underline{\langle \text{Expr} \rangle} ;
\end{aligned}$$

Enumerations

$$\begin{aligned}
\langle \text{Enum-Type} \rangle &::= \langle \text{Identifier} \rangle \underline{::=} \langle \text{Enum-Data} \rangle ; \\
\langle \text{Enum-Data} \rangle &::= \underline{\{ \langle \text{Literal} \rangle [_ \langle \text{Literal} \rangle]^* \}}
\end{aligned}$$

Solving Options

$$\begin{aligned}
\langle \text{Solving-Options} \rangle &::= \langle \text{Solving-Option} \rangle [_ \langle \text{Solving-Option} \rangle]^{\{0,2\}} \\
\langle \text{Solving-Option} \rangle &::= \langle \text{Var-Ordering} \rangle \mid \langle \text{Val-Ordering} \rangle \mid \langle \text{Cons-Level} \rangle \mid \text{default} \\
\langle \text{Var-Ordering} \rangle &::= \text{min-dom-size} \mid \text{max-dom-size} \mid \text{min-dom-val} \mid \text{max-dom-val} \mid \\
&\quad \text{min-regret-min-dif} \mid \text{min-regret-max-dif} \mid \\
&\quad \text{max-regret-min-dif} \mid \text{max-regret-max-dif} \\
\langle \text{Val-Ordering} \rangle &::= \text{min-val} \mid \text{med-val} \mid \text{max-val} \\
\langle \text{Cons-Level} \rangle &::= \text{bound} \mid \text{domain}
\end{aligned}$$

Bibliography

- [ABPS98] K.R. Apt, J. Brunekreef, V. Partington, and A. Schaerf. Alma-0: An Imperative Language that Supports Declarative Programming. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 20(5):1014–1066, 1998.
- [AP94] A. Aamodt and E. Plaza. Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *Artificial Intelligence Communications*, 7(1):39–59, 1994.
- [Apt03] K.R. Apt. *Principles of Constraint Programming*. Cambridge Press, 2003.
- [BDPS08] S. Brand, G. J. Duck, J. Puchinger, and P. J. Stuckey. Flexible, Rule-Based Constraint Model Linearisation. In *Proceedings of the 10th International Symposium on Practical Aspects of Declarative Languages (PADL)*, volume 4902 of *Lecture Notes in Computer Science*, pages 68–83. Springer, 2008.
- [BKM92] A. Brooke, D. Kendrick, and A. Meeraus. *GAMS: A User's Guide*. The Scientific Press, 1992.
- [BMV94] F. Benhamou, D. Mc Allester, and P. Van Hentenryck. CLP(Intervals) Revisited. In *Proceedings of the 1994 International Symposium on Logic programming (ILPS)*, pages 124–138. MIT Press Cambridge, MA, USA, 1994.
- [BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge Univ. Press, 1998.
- [BN06] A. Brodsky and H. Nash. CoJava: Optimization Modeling by Nondeterministic Simulation. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP 2006)*, volume 4204 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2006.
- [BO97] F Benhamou and W. Older. Applying Interval Arithmetic to Real, Integer, and Boolean Constraints. *Journal of Logic Programming*, 32(1):1–24, 1997.
- [Bor81] A.H. Borning. The Programming Languages Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 3(4):353–387, 1981.
- [Bur69] R. M. Burstall. A Program for Solving Word Sum Puzzles. *Computer Journal*, 12(1):48–51, 1969.
- [CDR99] H. Collavizza, F. Delobel, and M. Rueher. Comparing Partial Consistencies. *Reliable Computing*, 5(3):213–228, 1999.
- [CGLR96] J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-Breaking Predicates for Search Problems. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR 96)*, pages 148–159, 1996.
- [CGS08] R. Chenouard, L. Granvilliers, and R. Soto. Model-Driven Constraint Programming. In *Proceedings of the 10th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 236–246, 2008.

- [CGS09] R. Chenouard, L. Granvilliers, and R. Soto. Rewriting Constraint Models with Metamodels. To Appear In *Proceedings of the 8th Symposium on Abstraction, Reformulation and Approximation (SARA)*. 2009.
- [CIP⁺00] M. Cadoli, G. Ianni, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC: an executable specification language for solving all problems in NP. *Computer Languages*, 26(2–4):165–195, 2000.
- [COC97] M. Carlsson, G. Ottosson, and B. Carlson. An Open-Ended Finite Domain Constraint Solver. In *Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP)*, volume 1292 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 1997.
- [Col82] A. Colmerauer. Prolog II Reference Manual and Theoretical Model. Technical report, Groupe d’Intelligence Artificielle, Université d’Aix-Marseille II, Luminy, 1982.
- [Col90] A. Colmerauer. An Introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.
- [Col96] A. Colmerauer. Les Bases de Prolog IV. Technical report, Laboratoire d’Informatique de Marseille, 1996.
- [DC00] D. Diaz and P. Codognet. The GNU Prolog System and its Implementation. In *Proceedings of the 2000 ACM Symposium on Applied Computing (SAC)*, pages 728–732, 2000.
- [Elc90] E. W. Elcock. Absys: The First Logic Programming Language - A Retrospective and a Commentary. *Journal of Logic Programming*, 9(1):1–17, 1990.
- [FGJ⁺07] A. M. Frisch, M. Grum, C. Jefferson, B. Martínez-Hernández, and I. Miguel. The Design of ESSENCE: A Constraint Language for Specifying Combinatorial Problems. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 80–87, 2007.
- [FGK90] R. Fourer, D.M. Gay, and B.W. Kernighan. A Modeling Language for Mathematical Programming. *Management Science*, 36:519–554, 1990.
- [FHK⁺02] A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global Constraints for Lexicographic Orderings. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 2470 of *Lecture Notes in Computer Science*, pages 93–108. Springer, 2002.
- [FJMHM05] A.M. Frisch, C. Jefferson, B. Martínez-Hernández, and I. Miguel. The Rules of Constraint Modelling. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 109–116, 2005.
- [FM92] E. C. Freuder and A. K. Mackworth. Introduction to the Special Volume on Constraint-Based Reasoning. *Artificial Intelligence*, 58:1–2, 1992.
- [FM06] E. C. Freuder and A. K. Mackworth. *Handbook of Constraint Programming*, chapter 2 - Constraint Satisfaction: An Emerging Paradigm. Elsevier, 2006.
- [FM08] F. Fages and J. Martin. Des Règles aux Contraintes avec le Langage de Modélisation Rules2CP. In *Proceedings of the Quatrièmes Journées Francophones de Programmation par Contraintes (JFPC)*, pages 361–371, 2008.

- [FPA04] P. Flener, J. Pearson, and M. Ågren. Introducing ESRA, a Relational Language for Modelling Combinatorial Problems. In *Proceedings of the 13th International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR)*, volume 3018 of *Lecture Notes in Computer Science*, pages 214–232. Springer, 2004.
- [Fre78] E. C. Freuder. Synthesizing constraint expressions. *Commun. ACM*, 21(11):958–966, 1978.
- [Frü98] T. Frühwirth. Theory and Practice of Constraint Handling Rules. *JLP*, 37(1–3):95–138, 1998.
- [Gas74] J. Gaschnig. A Constraint Satisfaction Method for Inference Making. In *Proceedings 12th Annual Allerton Conference on Circuit and System Theory*, pages 866–874, 1974.
- [GB65] S. W. Golomb and L. D. Baumert. Backtrack Programming. *Journal of the ACM*, 12(4):516–524, 1965.
- [GB06] L. Granvilliers and F. Benhamou. Algorithm 852: RealPaver: An Interval Solver Using Constraint Satisfaction Techniques. *ACM Transactions on Mathematical Software (ACM TOMS)*, 32(1):138–156, 2006.
- [GF03] E. Gelle and B. Faltings. Solving Mixed and Conditional Constraint Satisfaction Problems. *Constraints*, 8(2):107–141, 2003.
- [GJM06] I. P. Gent, C. Jefferson, and I. Miguel. Minion: A Fast Scalable Constraint Solver. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI)*, pages 98–102. IOS Press, 2006.
- [Gou00] F. Goualard. *Langages et Environnements en Programmation par Contraintes d’Intervalles*. PhD thesis, IRIN, Université de Nantes, 2000.
- [GS00] I. P. Gent and B. M. Smith. Symmetry Breaking in Constraint Programming. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI)*, pages 599–603. IOS Press, 2000.
- [HLP⁺04] T. Hinrichs, N. Love, C. J. Petrie, L. Ramshaw, A. Sahai, and S. Singhal. Using Object-Oriented Constraint Satisfaction for Automated Configuration Generation. In *Proceedings of the 15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM)*, volume 3278 of *Lecture Notes in Computer Science*, pages 159–170. Springer, 2004.
- [Hni03] B. Hnich. *Function Variables for Constraint Programming*. PhD thesis, Department of Information Science, Uppsala University, 2003.
- [JB06] F. Jouault and J. Bézivin. KM3: A DSL for Metamodel Specification. In *Proceedings of the 8th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 4037 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2006.
- [JBK06] F. Jouault, J. Bézivin, and I. Kurtev. TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 249–254, 2006.
- [JL87] J. Jaffar and J.L. Lassez. Constraint Logic Programming. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 111–119, 1987.

- [JMSY92] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. Yap. The CLP(\mathcal{R}) Language and System. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 14(3):339–395, 1992.
- [JT02] B. Jayaraman and P.Y. Tamber. Modeling Engineering Structures with Constrained Objects. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL)*, volume 2257 of *Lecture Notes in Computer Science*, pages 28–46. Springer, 2002.
- [KvJ07] I. Kurtev, K. van Den Berg, and F. Jouault. Rule-based Modularization in Model Transformation Languages Illustrated with ATL. *Science of Computer Programming*, 68(3):138–154, 2007.
- [Lau78] J.L. Laurière. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10(1):29–127, 1978.
- [Lho93] O. Lhomme. Consistency Techniques for Numeric CSPs. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 232–238, 1993.
- [Luc91] E. Lucas. *Récréations Mathématiques*. Gauthier Villard, Paris, 2nd edition, 1891.
- [Lv93] J. H. Lee and M. H. van Emden. Interval Computation as Deduction in CHIP. *Journal of Logic Programming*, 16(3):255–276, 1993.
- [Mac77] A. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.
- [McG79] J. J. McGregor. Relational Consistency Algorithms and Their Application in Finding Subgraph and Graph Isomorphisms. *Information Science*, 19(3):229–250, 1979.
- [MF90] S. Mittal and B. Falkenhainer. Dynamic Constraint Satisfaction Problems. In *Proceedings of the 8th National Conference on Artificial Intelligence (AAAI)*, pages 25–32. The MIT Press, 1990.
- [MH86] R. Mohr and T. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28(2):225–233, 1986.
- [Mon74] U. Montanari. Networks of Constraints: Fundamental Properties and Applications to Picture Processing. *Information Sciences*, 7:95–132, 1974.
- [Moo66] R.E. Moore. *Interval Analysis*. Prentice-Hall, 1966.
- [MV02] L. Michel and P. Van Hentenryck. A Constraint-Based Architecture for Local Search. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 83–100, 2002.
- [Neu90] A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge University Press, 1990.
- [NSB⁺07] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc: Towards A Standard CP Modelling Language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007.

- [Pal95] M. Paltrinieri. A Visual Constraint-Programming Environment. In *Proceedings of the 1st International Conference on Principles and Practice of Constraint Programming (CP)*, volume 976 of *Lecture Notes in Computer Science*, pages 499–514. Springer, 1995.
- [PQ94] T. Parr and R. Quong. Adding Semantic and Syntactic Predicates To LL(k): pred-LL(k). In *Proceedings of the 5th International Conference on Compiler Construction (CC)*, volume 786 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 1994.
- [Pug93] J.F. Puget. On the Satisfiability of Symmetrical Constrained Satisfaction Problems. In *Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems (ISMIS)*, volume 689 of *Lecture Notes in Computer Science*, pages 350–361. Springer, 1993.
- [Pug94] J.F. Puget. A C++ Implementation of CLP. In *Proceedings of the 2nd Singapore International Conference on Intelligent Systems (SPICIS)*, 1994.
- [Pug04] J.F. Puget. Constraint Programming Next Challenge: Simplicity of Use. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 3258 of *Lecture Notes in Computer Science*, pages 5–8. Springer, 2004.
- [Rég94] J.-C. Régim. A Filtering Algorithm for Constraints of Difference in CSPs. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI)*, pages 362–367, 1994.
- [RGMW07] R. Rafeh, M. J. García de la Banda, K. Marriott, and M. Wallace. From Zinc to Design Model. In *Proceedings of the 9th Symposium on Practical Aspects of Declarative Languages (PADL)*, volume 4354 of *Lecture Notes in Computer Science*, pages 215–229, 2007.
- [Ric68] D. Richardson. Some Unsolvable Problems Involving Elementary Functions of a Real Variable. *Journal of Symbolic Logic*, 33:514–520, 1968.
- [SF94] D. Sabin and E. C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI)*, pages 125–129, 1994.
- [SG07a] R. Soto and L. Granvilliers. Dynamic parser cooperation for extending a constrained object-based modeling language. In *Proceedings of the 21st Workshop on (Constraint) Logic Programming (WLP 2007)*, pages 70–78, Wuerzburg, Germany, 2007. Technical Report 434, University of Wuerzburg, Germany.
- [SG07b] R. Soto and L. Granvilliers. The Design of COMMA: An Extensible Framework for Mapping Constrained Objects to Native Solver Models. In *Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 243–250, 2007.
- [SG08a] R. Soto and L. Granvilliers. On the Pursuit of a Standard Language for Object-Oriented Constraint Modeling. In *New Challenges in Applied Intelligence Technologies*, volume 134 of *Studies in Computational Intelligence*, pages 123–133. Springer, 2008.

- [SG08b] R. Soto and L. Granvilliers. Tuning Constrained Objects. In *Proceedings of the 21st International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE)*, volume 5027 of *Lecture Notes in Artificial Intelligence*, pages 408–414. Springer, 2008.
- [SGM⁺05] P. J. Stuckey, M. J. García de la Banda, M. Maher, K. Marriott, J. Slaney, Z. Somogyi, M. Wallace, and T. Walsh. The G12 Project: Mapping Solver Independent Models to Efficient Solutions. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 3709 of *Lecture Notes in Computer Science*, pages 13–16. Springer, 2005.
- [SHC96] Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.
- [SSW94] C. Schulte, G. Smolka, and J. Würtz. Encapsulated Search and Constraint Programming in Oz. In *Proceedings of the 2nd International Workshop on Principles and Practice of Constraint Programming (PPCP)*, volume 874 of *Lecture Notes in Computer Science*, pages 134–150. Springer, 1994.
- [ST06] C. Schulte and G. Tack. Views and Iterators for Generic Constraint Implementations. In *Recent Advances in Constraints (2005)*, volume 3978 of *Lecture Notes in Artificial Intelligence*, pages 118–132. Springer, 2006.
- [Sut63] I. E. Sutherland. *Sketchpad, A Man-Machine Graphical Communication System*. Outstanding Dissertations in the Computer Sciences. Garland Publishing, 1963.
- [TC07] G. Trombettoni and G. Chabert. Constructive interval disjunction. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 4741 of *Lecture Notes in Computer Science*, pages 635–650. Springer, 2007.
- [Van89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
- [Van99] P. Van Hentenryck. *The OPL Language*. The MIT Press, 1999.
- [van06] P. van Beek. *Handbook of Constraint Programming*, chapter 4 - Backtracking Search Algorithms. Elsevier, 2006.
- [VDT92] P. Van Hentenryck, Y. Deville, and C.-M. Teng. A Generic Arc-Consistency Algorithm and its Specializations. *Artificial Intelligence*, 57(2-3):291–321, 1992.
- [vK06] W.-J. van Hoeve and I. Katriel. *Handbook of Constraint Programming*, chapter 6 - Global Constraints. Elsevier, 2006.
- [VMD97] P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica: a Modeling Language for Global Optimization*. MIT Press, 1997.
- [Wal75] D. Waltz. Understanding Line Drawings of Scenes with Shadows. In *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.
- [WNS97] M. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A Platform for Constraint Logic Programming. Technical report, IC-Parc, Imperial College, London, 1997.

Hypertext References

[[w1w](#)] *Gecode System (Visited 9/2008).*

.....
<http://www.gecode.org>

[[w3w](#)] *ANSYS, Inc. Software Products (Visited 9/2008).*

.....
<http://www.ansys.com>

[[w4w](#)] *Roman Bartak's On-Line Guide to Constraint Programming (Visited 9/2008).*

.....
<http://ktiml.mff.cuni.cz/~bartak/constraints/>

[[w5w](#)] *Wolfram Mathematica (Visited 9/2008).*

.....
<http://www.wolfram.com/products/mathematica/index.html>

[[w6w](#)] *The Apache Ant Project (Visited 1/2009).*

.....
<http://ant.apache.org/>

[[w7w](#)] *Koalog System (Visited 9/2008).*

.....
<http://www.koalog.com>

[[w8w](#)] *ILOG CPLEX (Visited 10/2008).*

.....
<http://www.ilog.com/products/cplex/>

[[w9w](#)] *CIM (Common Information Model) (Visited 10/2008).*

.....
<http://www.dmtf.org/standards/cim/>

[[w10w](#)] *Eclipse Model-to-model transformation, (Visited 12/2008).*

.....
<http://www.eclipse.org/m2m/>

- [[w11](#)] *ANTLR Reference Manual (Visited 11/2008).*

<http://www.antlr.org>
- [[w12](#)] *Choco Solver (Visited 9/2008).*

<http://www.emn.fr/x-info/choco-solver/doku.php>
- [[w13](#)] *LINGO - Optimization Modeling Software for Linear, Nonlinear, and Integer Programming (Visited 9/2008).*

<http://www.lindo.com/>
- [[w14](#)] *MINOS Solver (Visited 9/2008).*

<http://www.aimms.com/aimms/product/solvers/minos.html>
- [[w15](#)] *The MOSEK Optimization Software (Visited 9/2008).*

<http://www.mosek.com/>
- [[w16](#)] *MProbe 5.0 an assistant for mathematical programming (Visited 9/2008).*

<http://www.sce.carleton.ca/faculty/chinneck/mprobe.html>
- [[w17](#)] *MATLAB - The Language of Technical Computing (Visited 9/2008).*

<http://www.mathworks.com/products/matlab/>
- [[w18](#)] *OMG - Object Constraint Language (OCL) 2.0, 2006 (Visited 11/2008).*

<http://www.omg.org/cgi-bin/doc?formal/2006-05-01>
- [[w19](#)] *OMG - The Unified Modeling Language (UML) 2.1.1 Infrastructure Specification, 2007 (Visited 11/2008).*

<http://www.omg.org/spec/UML/2.1.2/>
- [[w20](#)] *OMG - Model Driven Architecture (MDA) Guide V1.0.1, 2003 (Visited 11/2008).*

<http://www.omg.org/cgi-bin/doc?omg/03-06-01>

- [21^w] *OMG - Systems Modeling Language (SysML) v1.0, 2007 (Visited 11/2008).*
.....
<http://www.omg.org/cgi-bin/doc?formal/2007-09-01>
- [23^w] *SNOPT Solver (Visited 9/2008).*
.....
<http://www.aimms.com/aimms/product/solvers/snopt.html>
- [24^w] *Xpress-MP (Visited 8/2008).*
.....
<http://www.dashoptimization.com/>

Langages et transformation de modèles en programmation par contraintes

Ricardo SOTO

Résumé

La programmation par contraintes est une technologie pour l'optimisation qui associe des langages de modélisation riches avec des moteurs de résolution efficaces. Elle combine des techniques de plusieurs domaines tels que l'intelligence artificielle, la programmation mathématique et la théorie des graphes. Un défi majeur dans ce domaine concerne la définition de langages de haut-niveau pour faciliter la phase de modélisation des problèmes. Un autre aspect important est de concevoir des architectures robustes pour transformer des modèles de haut-niveau et obtenir des modèles exécutables efficaces, tout en visant plusieurs moteurs de résolution. Répondre à ces deux préoccupations est très difficile, car de nombreux aspects doivent être pris en compte, comme par exemple, l'expressivité et le niveau d'abstraction du langage ainsi que les techniques utilisées pour traduire le modèle de haut-niveau dans chacun des langages de résolution. Dans cette thèse, nous proposons une nouvelle perspective pour faire face à ces défis. Nous introduisons une nouvelle architecture pour la programmation par contraintes dans laquelle le problème est défini comme un ensemble d'objets contraints dans un nouveau langage de modélisation haut-niveau. La transformation des modèles est réalisée à l'aide de l'ingénierie des modèles. Les éléments des langages sont alors considérés comme des concepts définis dans un modèle de modèles appelé métamodèle. Cette nouvelle architecture permet d'aborder les phases de modélisation et de transformation de modèles en raisonnant à un niveau d'abstraction supérieur et, par conséquent, de réduire la complexité inhérente à ces deux phases.

Mots-clés: Programmation par contraintes, Langages de modélisation par contraintes, Transformation de modèles

Languages and Model Transformation in Constraint Programming

Abstract

Constraint Programming is an optimization technology that associates rich modeling languages with efficient solving engines. It combines methods from different domains such as artificial intelligence, mathematical programming, and graph theory. A main challenge in this field is to provide high-level languages for facilitating the problem modeling phase. Another important concern is to design robust architectures to map high-level input models to different and efficient solving models. Handling these two concerns is remarkably hard since many aspects have to be investigated, for instance, the expressiveness and the abstraction level of the language as well as the techniques used to transform the high-level model into each of the solver's languages. In this thesis, we propose a new perspective to face those challenges. We introduce a novel constraint programming architecture in which the problem is seen as a set of high-level constrained objects defined through a new modeling language. The model transformation is performed by a model-driven process in which the elements of languages are defined as concepts of a model of models called metamodel. This new architecture allows one to tackle the modeling and the model transformation phases in a higher-level of abstraction and consequently to reduce the inherent complexity behind them.

Keywords: Constraint Programming, Constraint Modeling Languages, Model Transformation

ACM Classification

Categories and Subject Descriptors : D.3.2 [Programming Languages]: Language Classifications—*Constraint and logic languages*; D.3.3 [Programming Languages]: Language Constructs and Features—*Classes and objects, Constraints*; D.2.2 [Software Engineering]: Design Tools and Techniques—*User interfaces*.