CSP - There is more than one way to model it

Gerrit Renker, Hatem Ahriz and Ines Arana

School of Computing, The Robert Gordon University
Aberdeen, Scotland, UK.

Abstract

In this paper, we present an approach for conceptual modelling of constraint satisfaction problems (CSP). The main objective is to achieve a similarly high degree of modelling support for constraint problems as it is already available in other disciplines. The approach uses diagrams as operational basis for the development of CSP models. To facilitate a broader scope, the use of available mainstream modelling languages is adapted. In particular, the structural aspects of the problem are visually expressed in UML, complemented by a textual representation of relations and constraints in OCL. A case study illustrates the expositions and deployment of the approach.

1 Introduction

Conceptual models are widely used for problem analysis, abstracting complicated and unwieldly reality into compact and tangible form. The strength of visual models in particular lies in the ability to represent the structure of complex data in a terse and condensed fashion [12]. Conceptual models allow the analysing expert to gain a better understanding of the problem requirements. Visual models are a useful tool to discuss and structure concepts among several developers and to communicate this knowledge to even non-domain experts in an understandable form.

Constraint-based reasoning has successfully been used to solve problems throughout a wide diversity of domains and industrial applications [15]. As a result of more than 30 years of research, this discipline is well defined and understood, and a variety of efficient solving methods does exist. Although constraint problems can already be expressed in a variety of (dedicated) programming languages, coding normally requires specialized expert knowledge. Other computing disciplines like database or software design have been benefiting from conceptual modelling in the form of analysis and design methodologies with a wide range of (visual) tool support for a long while. For applications of constraints however, we do not find the same degree of support for modelling and design in terms of tools, notations and methodologies.

We have been comparing modelling constructs in semantic data modelling, knowledge engineering and formal software specification. Analyzing the commonalities of these approaches with respect to their usefulness for constraint problems has enabled us to formulate a minimal set of requirements for constraint modelling. This has resulted in the modelling approach taken in the RECOP (*REpresenting COnstraint Problems*) project. The main objective of this approach is to achieve a comparable degree of modelling support for the (re-) formulation of CSPs, as it is already available for the analysis and design of databases and general software. As a secondary goal, we aim at a suitable re-use of broadly and publicly available modelling standards, to facilitate a broader scope of applicability.

The outline of the paper is as follows. Section 2 presents the results of our analysis to isolate useful constructs for constraint modelling. In section 3, we present our integrated modelling approach and give reasons for the underlying design decisions. The use of the approach illustrated by a case study in section 4. We relate to existing work in section 5 and conclude in section 6.

2 Requirements of Constraint Modelling

In this section, we analyze the common tasks in setting up a constraint problem and point out the constructs which have shown to be useful for modelling. A discussion of existing modelling approaches for constraints follows in section 5.

2.1 Constraint acquisition

Initially, a problem description may often be ambiguous and appear informally in natural language. Eventually, a machine-interpretable implementation has to be unambiguous and solve precisely what has been stated as initial problem. The modelling process can thus be viewed as working on a concept at various levels of abstraction and detail. The conceptual model helps to record knowledge gained during the analysis. Following this reasoning, the precise extent and definition of *constraints* appearing in a problem may initially be unclear to a developer who solves a CSP, or develops an application with constraint-based reasoning. Constraints have to be identified and defined during the design. This cognitive process is understood as *constraint acquisition* and forms a branch of current constraint research [24].

2.2 Dynamic versus static CSPs

CSPs can be distinguished according to the degree of dynamic change involved. Purely static CSPs remain unchanged over the lifetime, and so are associated models. The other extreme are purely dynamic CSPs, in which all involved constraints are potentially subject to change. Here, a model can at best be used to reflect the changes. All other cases involve at least a core set of static constraints for which a model can be built. Depending on the individual case, work can be done either with the core model, or the model can be updated whenever changes (e.g. constraint addition) become apparent.

2.3 Common modelling tasks

The process of formulating a CSP so that the search space is minimized and possible symmetries are broken is similar in concept to the design of a relational database, in which the objective is to minimize the amount of unnecessary links and redundant information stored in tables. In relational database design, this is typically achieved via normalisation [19]. Similar, but less standardised, guidelines for the design of CSPs can be found throughout the literature, e.g. in [20]. This section covers only a selection of the better known modelling techniques. Not specific to the modelling of CSPs is the often used approach of dividing large, complex problems into easier solvable subproblems. More important is the problem of selecting the right representation (model) [32, 20, 16, 22], as using the right model can significantly reduce search effort. By detecting and purposefully breaking *symmetries* in a model, further solver effort can be spared. In some situations it may even pay to choose an alternative model of the problem, which exhibits more symmetry that can be broken. Clusters of individual constraints can in many cases be replaced by global constraints such as `alldifferent()`, `atmost()` and the like [20]. Adding *redundant constraints* (which are entailed by the CSP) to the model can further improve solution convergence. A study in [31] has shown that adding entailed constraints can make a problem path consistent prior to solver execution. Finally, a clever exploitation of inheritance (where available) can group sets of behaviourally equivalent constraints into a much reduced number of *class constraints* [25].

2.4 Relational basis of a CSP

A constraint is a relation that must hold for one or more variables. It can be expressed in *extensiona*l form by explicitly stating the (non-) valid tuples, or it can be represented in *intensional* form using a formula. A CSP is commonly defined as follows.

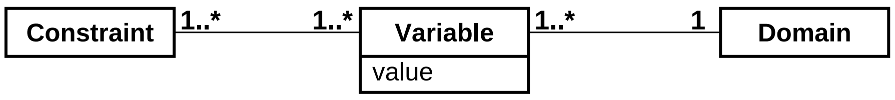

Figure 1: Constraint Satisfaction Problem (CSP)

Definition 1. *A constraint satisfaction problem (CSP) is a 3-tuple $\langle V, D, C\rangle$, in which $V = \{x_1, .., x_n\}$ represents the set of variables x_i of the CSP. $D = \{d_1, .., d_n\}$ represents the set of all domains in the problem; a bijection from D to V associates one domain d_i with each variable x_i. The set C contains all the constraints c_i of the problem, such that c_i yields* `true` *if constraint c_i is satisfied. A solution D_{Sol} to the CSP is a subset of the Cartesian product $D_{Cart} = d_1\ x\ ..\ x\ d_n$ such that $\forall\, c_i \in C :\ c_i = true$.*

Figure 1 illustrates this definition in UML [18]; a variable has exactly one domain, but may be associated with multiple constraints.

The procedure that computes the solution(s) to a CSP can in turn be expressed in terms of a relation, as it assigns the subset $D_{Sol} \subseteq D_{Cart}$ to the elements in V. If the constraints in C are too restrictive, the problem is over-constrained and $D_{Sol} = \{\}$. Solving a CSP can be expressed in relational algebra [19] in terms of applying the *selection operator* σ on D_{Cart}, $D_{Sol} = \sigma_F(D_{Cart})$. The logical formula F of the selection operator σ comprises the conjunction of all constraints $c_i \in C$. This connection of constraint problems and relational algebra has been pointed out in form of discussing the analogies between solving CSPs and relational databases throughout the literature [8, 34, 28].

We therefore argue that it does make sense to base a modelling paradigm for CSPs on a (visual) expression of relations. Semantic data models, for example, offer a rich variety of notations for categorically different forms of relations [29].

2.5 Structural abstraction

The traditional CSP representation (figure 1) is restricted in expressivity and offers limited abstraction capabilities. Considering that a domain is a special form of a unary constraint, the representation in figure 1 can be further reduced. As a result, it leaves this form of CSP representation to comprise only entities with just a single attribute (the variables) and allowing only one type of relationship between the entities (the constraints).

Many real-life problems exhibit a fair degree of texture [30], hence an adequate expression of structure and aggregation is required [25]. Further, entities can have more than a single attribute and relationships (constraints) can be complex and involve several levels of abstraction [7]. Organizing the inherently flat structure of CSPs (fig. 1) into a hierarchy of subtypes and supertypes (ISA hierarchy, [19]) has several advantages. First, it allows a drastic gain in abstraction. Paltrinieri for example uses such a hierarchy for constraint solving and is able to completely express the semantics of 168 constraints of a bridge building CSP by just 7 class constraints [25]. Second, domains with a high inherent degree of structure, such as configuration problems [30, 5], can adequately be modeled. Last, an important benefit of abstraction via ISA-hierarchies lies in the improved facilities for constraint visualisation and debugging. CSPs can conventionally be visualized by constraint graphs, in which variables appear as vertices and constraints as (hyper-) edges. This form of representation does not allow structural abstraction and it does not scale well for problems of a larger size. Current visualisation methods like Goualard's S-boxes [13] are therefore based on a hierarchical restructuring of given source code, effectively arranging the program constraints into an inclusion hierarchy, which is conceptually close to an ISA hierarchy. Rather than re-introducing structure for debugging purposes into a finished program, we argue that it makes more sense to use structure as an integral part of the design process.

3 Modelling approach for constraint problems

This section presents the constructs we have found most useful for modelling CSPs. The basis of the approach (which relates to OO analysis) is subsequently introduced and our methodology is presented. This is illustrated by the case study in section 4.

3.1 Choosing the constructs

Following the reasoning in section 2.4, we have decided to center modelling around the expression of relations between entities. Experiences in semantic data modelling [29] have shown that using a single type construct of relationships leads to semantic overloading, i.e. several categorically different kinds of relationships have to be represented by the same construct. We have therefore chosen to use several different relationship constructs that were successfully used for modelling in knowledge engineering [6], object-oriented analysis [14, 27, 17] and semantic data models [19, 29]. These are attribute access, association, aggregation and ISA[1] relationships. Attribute relations allow to conceptually build complex entities (types) from simple (atomic) ones. Deployment of attribute and ISA relationships allows to achieve the structural abstraction discussed in section 2.5. As a result of employing these constructs, the developer can work with the model at various levels of detail (cf. section 2.1), information about a given type can be isolated from that about its attributes and subtypes. Aggregation allows a similar abstraction, from the constituent parts to a whole. These basic relationship types correspond to an analytical decomposition of the real world; the colour 'attributed' to a car is conceptually different from its 'associated' driver. Traditionally, research into object-oriented analysis has been enthusiastic in that virtually everything could potentially be modelled using objects (or frames, [6]). Quoting [17, p. 235]: *"An OO way of thinking can be used to develop* any *kind of system – whether or not the system is implemented using OO technology*".

A central guideline of the RECOP project has been that simple things should be simple to express, whereas complex things should not be prohibited. We have thus rooted our modelling approach on OO concepts, while giving the developer enough freedom to integrate his own systematics. Modelling is a human activity, and there may be as many different approaches (or even methodologies) as there are different kinds of developers. This is confirmed by the sheer multitude of different modelling approaches in database modelling [19], OO analysis [27, 17] and software modelling [14]. The use of the above relationship types represents the intersection of modelling in disparate disciplines and should be applicable to a range of different methodologies. For the graphical notation, we have further chosen to use UML [18], since it has refined modelling notions of the past decade, captures all the above types and permits even further nuances. For instance, associations can be expressed in five ways; association class, binary, n-ary, qualified and derived association. In addition to this, UML is widely

[1]in this text, we use ISA and specialization/generalization relationship interchangeably.

available. Many developers are already familiar with UML, and CASE tools exist on virtually every platform. We have also looked at the Alloy modelling language [9], but abandoned the idea, as Alloy does not support attribute relationships and there is currently no support for basic types like integers, real numbers or strings. Also, it is less widely supported as UML.

Regarding the graphic notation, the number of visually expressible relations is naturally limited, as research into new diagrammatic notations has shown [11]. Thus, an accompanying textual notation for logical formulae (as proposed in [25]) is a good complement to the diagrams, whose strength is the terse expression of problem structure. Out of these considerations, we have further found it useful to adopt the Object Constraint Language (OCL) [33] of the UML for our purposes. OCL was initially intended as textual addendum to UML, to describe object behaviour at runtime. We are using and extending OCL for the declarative specification of constraints. The fact that it combines a form of first-order predicate logic with a rich variety of additional expressions has made it interesting to test whether the language is sufficiently expressive for constraint problems. We have achieved very promising results in our case studies and include an example problem for evaluation in section 4.

3.2 Constraint Representation

In analogy to the usual definition of a CSP (sec. 2.4), we now define the object-oriented representation that we use in the RECOP project.

Definition 2. *An object - oriented CSP (OOCSP) is a CSP in which the variables in V are represented either as classes or as attributes of classes, the constraints C as associations between the elements of V and the domains in D are represented as unary constraints over the elements of V. A solution to an OOCSP can be represented as a fully instantiated object graph in which the assignments to the elements in V satisfy both the constraints in D and C.*

It is important to point out that the OOCSP representation is primarily used for modelling purposes. It is not imperative to actually implement each entity as an object.

3.3 General methodology

Figure 2 illustrates the different modelling stages. Reading from left to right, the aim is to increase the degree of precision, up to the implementation level. Starting with the problem specification, a *conceptual model* is developed, clarifying the requirements of the problem. The definition of entities in OOCSP representation provides the building blocks for the *structural model*, visualized in UML. Graphical notation is well suited to represent the structural aspects, but less effective for precisely documenting the details of a system specification. To this avail, the *algebraic model* is build from the structural one, using a textual OCL representation to resolve ambiguities of the graphical model and

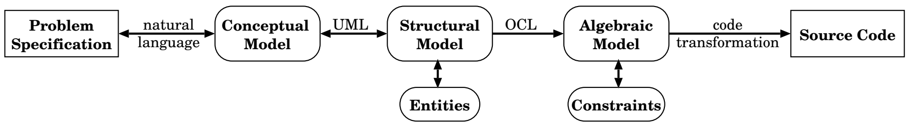

Figure 2: Design Process

to define the constraints. The algebraic model marks the end of the modelling process, having achieved a concise and unambiguous model, whose specification is precise enough to support the implementation.

Summarizing, the general modelling process involves three steps:

1. identification of the main entities in the problem
2. definition of the structural model, visualized in UML
3. definition of the algebraic model, using textual OCL representation

4 Case Study

We now illustrate the use of the approach on a particular CSP, the steel mill slab design problem [2, 3]. This problem belongs to a class of difficult problems, in which the structure of the problem is not fully known at the begin of the solving process. The problem is comparable to the popular warehouse location problem [26, 16]. A brief outline of OCL is provided to clarify the expositions.

4.1 A Sketch of OCL

The Object Constraint Language (OCL) [33] is fully integrated into the UML standard and is used to define the well-formedness of the UML meta-model as well as for other meta-models within the OMG [23]. The description of the various OCL concepts and their use fills an entire book [33]. Thus, we can only introduce the most prominent concepts here.

OCL allows to further specify associations, in annotated form directly in the diagram or via a separate text file. A separate OCL expression always begins with the class context it refers to. From this context, expressions and navigations throughout the entire diagram are possible, using the role names at association ends or class names with lower case first letter. Apart from the built-in types such as `Integer`, `Real`, `Boolean` and `String`, a rich notation for collection types is provided in OCL by the `Sequence`, `Bag` and `Set` types. Expressions on collection types always relate to the (navigation) context they are stated in, so that the notation remains unambiguous. Among a variety of set-theoretic operations, universal (`forAll`) and existential (`exists`) quantification are supported. To distinguish operations on collections from those on objects, the arrow symbol (`->`) is used in place of the usual dot. All navigations, classes and attributes of the UML model are accessible in OCL, thus allowing to post constraints on any element of the diagram. The built-in types allow modelling in both finite and continuous domains. Additionally, if non-standard domains are required, the type extension mechanism of UML can be used [18, p. 484]. The special `OCLType` class permits access to the meta-level of the model, which allows further modifications and provides room for extensions. The fact that OCL is a typed modelling language greatly simplifies the verification of models.

4.2 Problem specification

The problem involves coordinating steel orders with a given system of steel production. A mill produces steel in units of slabs, which are classified by their weight dimension. Only a finite number of weight classes can be produced. Input orders are characterized by the weight of the requested steel and a required route through the steel mill, indicated by a colour name.

Order	1	2	3	4	5	6	7	8	9
Weight	2	3	1	1	1	1	1	2	1
Colour	Red	Green		Blue	Orange			Brown	

Table 1: Instance data for input orders (taken from [2])

The design problem comprises determining number and dimension of steel slabs such that all orders are fulfilled and the total slab capacity is minimised. The task is thus an optimisation problem with an a priori unknown structure of slabs, having as cost function the sum of the allocated slab weights. The following constraints apply:

C1 orders can not be split between slabs.

C2 the total weight of orders assigned to a slab must not exceed its capacity.

C3 the number of different colours per slab is restricted to `maxColours`.

The problem is introduced in [2]. Three slab sizes *(1,3 and 4)* and five colours *(Red, Green, Blue, Orange and Brown)* are available. Table 1 shows the problem instance data for the input orders.

4.3 Structural model

The structural model is presented in figure 3. The problem centers around the relation between the sets of orders and slabs, this is reflected in by the two classes `Order` and `Slab`. The third class, `SDP`, is mainly a utility class to contain data relevant to problem instances. Informally, the diagram reads as '*the steel mill slab design problem (SDP) is composed of orders and associated steel slabs*'. The constraint C1 is already encoded as multiplicity constraint: each instance of `Order` is associated with exactly one `Slab` instance.[2]

[2]it can redundantly be expressed using **context** `Order` **inv:** `slab->size() = 1`

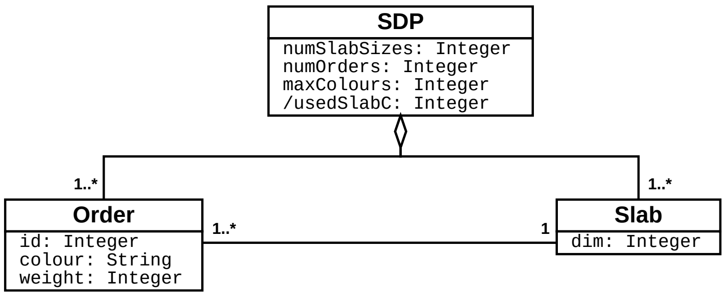

Figure 3: Structural model of the steel mill slab design problem

4.4 Constraints

4.4.1 Multiplicity Constraints

The problem instance data affects the multiplicities of several entities, as follows.

```
context SDP inv:
    numOrders = order->size() and
    numSlabSizes = slab.dim->asSet()->size()
```

Note the stacked operation on `Slab`. Since multiple occurrences of slab sizes can be expected, the navigation to `Slab` forms a bag, whose duplicates are eliminated by the `asSet()` conversion [23, p. 6-43]. The `size()` operator is then applied to the resulting set, yielding the number of different slab sizes in the problem.

4.4.2 Domain constraints

Domains are represented as unary constraints. First, the instantiation value for `maxColours` and the domain for the slab (weight) dimension are stated.

```
context SDP inv:
    maxColours = 2

context Slab inv:
    Set{1,3,4}->includes(dim)
context Order inv:
    Set{'Red', 'Green', 'Blue', 'Orange', 'Brown'}->includes(colour)
```

The first expression limits the number of different colours per slab to two. The last two expressions state that the value of the respective attribute is contained in the specified set, which is converse to the mathematical notation (value ϵ domain). Next are the instantiation values for the `Order` class, effectively translating table 1 into OCL:

```
context Order inv:
    id = 1 implies (weight = 2 and colour = 'Red') and
    id = 2 implies (weight = 3 and colour = 'Green') and
    id = 3 implies (weight = 1 and colour = 'Green') and
    id = 4 implies (weight = 1 and colour = 'Blue') and
    id = 5 implies (weight = 1 and colour = 'Orange') and
    id = 6 implies (weight = 1 and colour = 'Orange') and
    id = 7 implies (weight = 1 and colour = 'Orange') and
    id = 8 implies (weight = 2 and colour = 'Brown') and
    id = 9 implies (weight = 1 and colour = 'Brown')
```

4.4.3 *Main constraints*

As C1 is already encoded as multiplicity constraint in the diagram (figure 3), the capacity (C2) and colour (C3) constraints remain and are encoded as follows.

```
context Slab inv:
    dim >= order.weight->sum()
    and
    order.colour->asSet()->size() <= sDP.maxColours
```

The first expression represents the weight constraint C2 and asserts that the value for `dim` of every `Slab` instance is never below the sum of all associated order weights. The second expression represents C3 and also uses the conversion into a set before counting the number of distinct colours, which is then related to the constant `maxColours` via the navigation to `SDP`.

4.4.4 *Additional and implied constraints*

After formulating the essential problem constraints, further constraints can be added to prohibit using flawed problem instance data. As an example, we add the constraint that the constant `maxColours` must not exceed the number of colours available in the problem instance.

```
context SDP inv:
    maxColour <= order.colour->asSet()->size()
```

The main objective behind adding implied constraints is improving solution convergence by adding redundant information (cf. section 2.3). As an example, it can be concluded from C1 that the number of slabs will not exceed the number of orders, which is coded as follows.

```
context SDP inv:
    slab->size() <= numOrders
```

For an in-depth treatment of breaking symmetries and using implied constraints in the steel mill slab design problem, see [2] and in particular [3].

4.4.5 *Cost function*

The objective function in this optimisation problem is the consumed total slab capacity, which is stated as derived value `usedSlabC` in figure 3.

```
context SDP inv: -- how to calculate the derived value usedSlabC
    usedSlabC = slab.dim->sum()
```

4.4.6 *Solution*

Figure 4 shows a solution, which was derived in [3]. For perspicuity, the instance of SDP and the otherwise resulting 13 aggregation links are left out in the object diagram. The total cost evaluates to `usedSlabC = 13`, which is optimal in that the total weight of the slabs equals the total weight of the orders (cf. table 1).

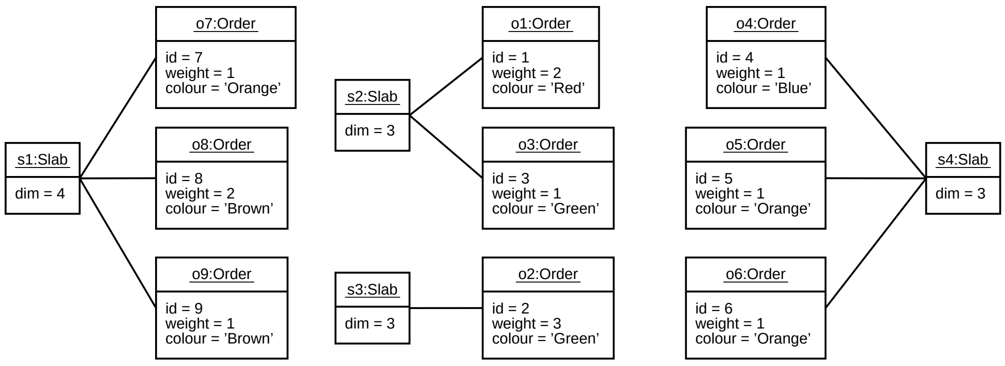

Figure 4: A solution to the example problem (values taken from [3])

5 Related Work

A survey on industrial applications of constraints in [15] points out a need for a modelling paradigm in constraint programming and notes that mastering current constraint techniques involves a non-trivial learning period. The evaluation of a questionnaire poll in [1] regarding the needs of constraint programmers shows "*a strong demand for graphical tools*". The recent introduction of the constraint modelling languages EaCL [21] and OPL [26] (with extensions in [10, 28]) have presented a simplified and possibly more user-friendly way of formulating constraint problems. Both are however textual constraint programming languages. Felfernig [5, 4] uses UML diagrams for the construction of configuration knowledge bases. Configuration problems often have highly structured domains and employ (in comparison to CSPs in general) relatively simple constraints. The results in [4] displayed a successful solution for knowledge acquisition and automated generation of the configuration knowledge bases. The principle of the approach lies in the use of a UML extension mechanism, which allows to encode domain-specific knowledge as special instances of UML constructs. These results are very encouraging to continue our work.

To the best of our knowledge, we are currently not aware of any other approach or case studies that adapt available modelling standards like UML and OCL to support work on conceptual modelling in the set-up of general constraint problems.

6 Conclusion

In this paper we have presented our UML-based modelling approach for the conception of problems that use constraint-based reasoning. We presented an example problem to study and evaluate its use. We see the main benefits of our approach in the introduction of a more user-friendly paradigm for constraint problems with visual support. It further simplifies the incorporation of constraints into mainstream software engineering, since many developers are already familiar with UML approaches. It can help to exploit constraint-based reasoning in non-AI applications and to avoid the paradigm shift between traditional programming languages and special-purpose constraint languages. Non-constraint programmers can use the models independently of the host language. Models can be shared between CASE tools and over the network using the (XML Metadata Interchange) standard. Last, the concept is helpful for knowledge reuse, as (i) recurring tasks can be banned into libraries or design patterns [14] and (ii) the format of the models is understandable by people other than the initial developer.

References

[1] A. Aggoun, F. Bueno, M. Carro, and et al. CP Debugging Needs and Tools. In Mariam Kamkar, editor, *Proceedings of AADEBUG '97*. Linköping University Electronic Press, 1997.

[2] Alan M. Frisch, Ian Miguel, and Toby Walsh. Modelling a Steel Mill Slab Design Problem. In Christian Bessiere, editor, *Proceedings of the IJCAI-01 Workshop on Modelling and Solving Problems with Constraints*, pages 39–45, 2001.

[3] Alan M. Frisch, Ian Miguel, and Toby Walsh. Symmetry and Implied Constraints in the Steel Mill Slab Design Problem. In *Proceedings of the CP'01 Workshop on Modelling and Problem Formulation*, pages 8–15, 2001.

[4] Alexander Felfernig, Gerhard Friedrich, and Dietmar Jannach. Generating Product Configuration Knowledge Bases from Precise Domain Extended UML Models. In *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE'2000), Chicago, Illinois, USA*, pages 284–293, 2000.

[5] Alexander Felfernig, Gerhard Friedrich, and Dietmar Jannach. Conceptual modeling for configuration of mass-customizable products. *Artificial Intelligence in Engineering*, 15(2):165–176, April 2001.

[6] Avelino J. Gonzalez and Douglas D. Dankel. *Engineering of Knowledge-Based Systems*. Prentice-Hall, 1993.

[7] Pierre Berlandier. The Use and Interpretation of Meta Level Constraints. In Miguel Filgueiras and Luis Damas, editors, *Proceedings of the 6th Portuguese Conference on Artificial Intelligence (EPIA '93)*, volume 727 of *Lecture Notes in Computer Science*, pages 271–280. Springer, 1993.

[8] Frans Coenen, Barry Eaglestone, and Mick Ridley. Verification, Validation and Integrity Issues in Expert and Database Systems: Two Perspectives. *Expert Update*, 3(3):26–42, 2000.

[9] Daniel Jackson. Alloy: A Lightweight Object Modelling Notation. Technical report, MIT Laboratory for Computer Science, 2001.

[10] Pierre Flener and Brahim Hnich. The Syntax and Semantics of ESRA. Technical report, Department of Information Science, Uppsala University, Sweden, March 2001.

[11] J. Gil, J. Howse, and S. Kent. Constraint Diagrams: A Step Beyond UML. In *Proceedings of TOOLS USA '99*. IEEE Computer Society Press, 1999.

[12] J. Gil, J. Howse, and S. Kent. Formalizing Spider Diagrams. In *Proceedings of IEEE Symposium on Visual Languages (VL-99)*, pages 130–137. IEEE Computer Society Press, 1999.

[13] Frédéric Goualard and Frédéric Benhamou. A Visualization Tool for Constraint Program Debugging. In *Proceedings of The 14th IEEE International Conference on Automated Software Engineering (ASE-99)*, pages 110–118. IEEE Computer Society, 1999.

[14] Hans van Vliet. *Software Engineering: Principles and Practice.* John Wiley and Sons, 2nd edition, 30 August 2000.

[15] Helmut Simonis. Building Industrial Applications with Constraint Programming. In H. Comon, C. Marché, and R. Treinen, editors, *Constraints in Computational Logics: Theory and Applications*, volume 2002 of *LNCS*, chapter 6, pages 271–309. Springer-Verlag, 2001.

[16] ILOG, France. *Ilog solver 4.4, User's Manual*, 1999.

[17] James J. Odell. *Advanced Object-Oriented Analysis and Design Using UML.* Cambridge University Press, sigs reference library edition, 1998.

[18] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Object Technology Series. Addison-Wesley, 1999.

[19] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems Vol. 1.* Computer Science Press, 1988.

[20] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction.* The MIT Press, 1998.

[21] P. Mills, E. Tsang, R. Williams, J. Ford, and J. Borrett. EaCL 1.5: An Easy abstract Constraint optimisation Programming Language. Technical Report CSM-324, University of Essex, 1999.

[22] B. A. Nadel. Representation selection for constraint satisfaction: A case study using n-queens. *IEEE Expert*, 5(3):16–23, June 1990.

[23] OMG. Object Constraint Language Specification. In *OMG Unified Modeling Language Specification, Version 1.4, September 2001*, chapter 6. Object Management Group, Inc., Needham, MA, Internet: http://www.omg.org, 2001.

[24] Barry O'Sullivan, Eugene C. Freuder, and Sarah O'Connell. Interactive Constraint Acquisition. In *Working Notes of the First International Workshop on User-Interaction in Constraint Satisfaction at CP-01*, 2001.

[25] Massimo Paltrinieri. Some Remarks on the Design of Constraint Satisfaction Problems. In Alan Borning, editor, *Second International Workshop on Principles and Practice of Constraint Programming (PPCP-94)*, volume 874 of *LNCS*, pages 299–311. Springer, 1994.

[26] Pascal Van Hentenryck. *The OPL Optimization Programming Language.* The MIT Press, January 1999.

[27] Peter Coad and Edward Yourdon. *Object Oriented Analysis.* Prentice-Hall, 2nd edition, 1991.

[28] Pierre Flener. Towards Relational Modelling of Combinatorial Optimisation Problems. In Christian Bessière, editor, *Proceedings of the IJCAI'01 Workshop on Modelling and Solving Problems with Constraints*, 2001.

[29] Richard Hull and Roger King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys (CSUR)*, 19(3):201 – 260, September 1987.

[30] Daniel Sabin and Eugene C. Freuder. Configuration as Composite Constraint Satisfaction. In George F. Luger, editor, *Proceedings of the (1st) Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161. AAAI Press, 1996.

[31] Barbara M. Smith. How to Solve the Zebra Problem, or Path Consistency the Easy Way. In Bernd Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 92)*, pages 36–37. John Wiley and Sons, Ltd, 1992.

[32] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[33] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison Wesley, 1999.

[34] Kit ying Hui and Peter M. D. Gray. Developing Finite Domain Constraints - A Data Model Approach. In John W. Lloyd and et al., editors, *Proceedings of the First International Conference on Computational Logic (CL-00)*, volume 1861 of *LNAI*, pages 448–462. Springer, 2000.