A Constraint Programming Based Approach to Detect Ontology Inconsistencies

Moussa Benaissa and Yahia Lebbah
Faculté des Sciences, Université d'Oran Es-Senia, Algeria

Abstract: *This paper proposes a constraint programming based approach to handle ontologies consistency, and more precisely user-defined consistencies. In practice, ontologies consistency is not still well handled in the current software environments. This is due to the algorithmic and language limitations of the tools developed to handle the consistency constraints. The idea of this paper is to tackle this problem by exploiting constraint programming which is proved to be efficient and provides various techniques to handle many types of constraints on different domains.*

Keywords: *Constraint programming, filtering, ontologies, consistency checking, ontology evolution.*

Received January 8, 2008; accepted May 17, 2008

1. Introduction

Ontologies have been widely adopted both in academic and industrial applications. They are entities evolving continuously. One of the main aspects to consider during this evolution process is to detect their consistencies. Consistency can be defined as a set of constraints that should be verified. Below, we distinguish three types of consistencies [6, 14]:

- Structural consistency: this consistency considers constraints required by the underlying model of the ontology and the representational language.
- Logical consistency: this consistency considers logical constraints. Ontology must not contain contradictory information.
- User-defined consistency: this consistency considers constraints defined by the user.

Here, we are concerned with user defined consistency. Ontology development environments such as Protégé [10] include tools based on first order logic. For example Protégé Axiom Language (PAL) for Protégé enables to express consistency axioms as a set of axioms. The work described in [7] consists of studying existing ontologies and proposing a set of axioms templates in order to facilitate composing the user constraints. Nevertheless, algorithmic issues of handling theses constraints are problematic.

Operational research and constraint (logic) programming techniques provide efficient algorithms which can be used to handle this problem. Except Constraint Programming (CP), other techniques are very specific. It is very difficult to process new constraints with them. For example linear programming can process only linear constraints. Logic programming handles clauses in the Herbrand space [5]. However the task of processing constraints holding both on reals and Herbrand space is very difficult since there are no algorithmic connections between logic and linear programming.

Constraint programming overcomes this limitation by allowing several constraints to be expressed on various domains. It uses existing mathematical techniques to process general constraints with generic methods. In other terms, constraint programming is not better than other techniques, but exploits and combines them to process general constraints for which there are no specific techniques.

This is exactly the purpose of this paper: how can we exploit constraint programming to handle user-defined consistency? We propose a constraint programming based framework for processing the constraints presented in [5].

The paper is organized as follows. We begin by motivating this work and illustrating the proposed approach with an example. In section 4, we present user axioms classification developed in an empirical study [5]. Section 5 introduces constraint programming and more particularly the Constraint Satisfaction Problems (CSP) notation and its generic algorithms. In section 6, we present our contribution. It is a constraint programming model for the classification presented in section 4. In section 7, we introduce two approaches to handle consistency. Next, in section 8, we provide two approaches to exploit the model elaborated in section 6. Conclusion and some perspectives conclude the paper.

2. Motivation and Contribution

Ontologies are nowadays ubiquitous. Their number, their complexity and the domains they model are increasing considerably. Ontology management systems should be developed to support the user.

Ontology languages cannot capture all the knowledge embedded in various application domains. So, it is necessary to extend them to make easy the expression and the treatment of other aspects of knowledge such as constraints. In fact, some expressiveness issues have been considered by some works [7], but, to our knowledge, there are no efficient systems for handling large ontologies with constraints in real contexts. This complexity aspect is particularly critical in dynamic industrial applications where system performance is required.

Our contribution consists in adopting constraint programming as the framework to process ontology constraints. This is a natural extension to the current trend which uses logic programming to handle ontology constraints. Actually, constraint programming is recognized as the appropriate framework for studying combinatorial problems. It offers more general and powerful tools than those of logic programming.

3. Tasks Assignment Problem

In order to illustrate our approach, we consider a tasks assignment problem. We have two sets: the tasks *T* and the persons *P*. *T* is defined as *T* = {*A*, *M*, *C*, *O*, *D*}; *A* for Administration, *M* for Maintenance, *C* for Commercial, *O* for Conception, and *D* for Development. The instances of *P* are the set {*Pierre*, *Michel*, *Corinne*, *Julie*, *Bernard*, *Jean*, *Patrice*} which denotes the enterprise employees.

We aim to assign tasks to persons so that the following constraints are verified:

- The employee must execute at least one task.
- The employee cannot execute more than one task.
- The number of assignments of some task should be greater than some minimal value and lower than some maximal value. For example *A*[1,2] means that the task *A* is affected to at least one person and to at most two persons. The cardinalities constraints for the different tasks are *A*[1, 2], *M*[1, 3], *C*[1, 1], *O*[1, 2], and *D*[1, 5].
  Let be the following preferences:
- Pierre prefers the tasks *A*, *M* and *C*.
- Michel prefers the tasks *D*, *M* and *C*.
- Corinne prefers the tasks *A* and *M*.
- Julie prefers the tasks *A* and *M*.
- Bernard prefers the tasks *A*, *M* and *C*.
- Jean prefers the tasks *O* and *D*.
- Patrice prefers the tasks *O* and *D*.

A solution that verifies the above constraints is: *Pierre*, *Michel*, *Corinne*, *Julie*, *Bernard*, *Jean*, *Patrice* realize respectively the tasks *A*, *M*, *A*, *M*, *C*, *O* and *D*.

3.1. Ontological Modeling of the Tasks Assignment Problem

The above problem can be modeled with the following ontology as shown in Figure 1. We have two concepts: *T* whose instances are the tasks and *P* whose instances are enterprise employees.

We have two slots: The slot *can-do*: this slot has *P* and *T* as respectively the domain and the range. It captures employee preferences. We represent it as follows:

Can-do ={(*Pierre*, *A*), (*Pierre*, *M*), (*Pierre*, *C*), (*Michel*, *M*), (*Michel*, *C*), (*Michel*, *D*), (*Corinne*, *A*), (*Corinne*, *M*), (*Julie*, *A*),(*Julie*, *M*), (*Bernard*, *A*), (*Bernard*, *M*), (*Bernard*, *C*), (*Jean*, *O*), (*Jean*, *D*), (*Patrice*,*O*) ,(*Patrice*, *D*)}.

The slot *is-realized-by*: This slot has *T* and *P* as respectively the domain and the range. It represents the problem solution. With cardinality constraints, we associate the following user-defined axioms:

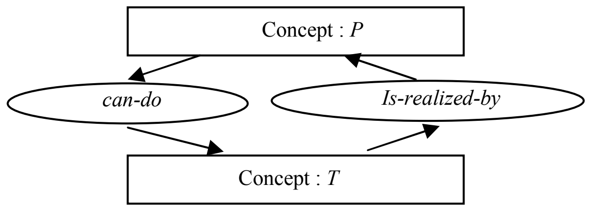

Figure 1. Ontology model.

The value *A*, of the class *T*, has a number of occurrences ranging from 1 to 2 within the slot *is*-realized-by. And with the same manner:

1. The value *M* ranges from 1 to 3.
2. The value *C* ranges from 1 to 1.
3. The value *O* ranges from 1 to 2.
4. The value *D* ranges from 1 to 5.

We can abstract the above axioms by the following template: *the value ___ of class __ has a minimal number of occurrences equal to __ and a maximal number of occurrences equal to __for the slot __.*

3.2. The Problem

Let us suppose that some changes occur in this environment by introducing new data (e.g., new employees, new tasks, or new constraints on employee preferences). It is necessary to maintain ontology consistency after these changes. Obviously, this task is hard to realize manually. The manager may not find easily the solution. The question is: how can we help the manager?

3.3. The Proposed Solution

The solution we propose is based on constraint programming. More precisely, we translate the above ontology to a Constraint Satisfaction Problem (CSP) and use filtering techniques to reduce the search space.

The CSP is defined as follows (for the details see the following sections):

- The persons are denoted as variables.
- The slot *can-do* defines variables domains.
- The axioms template defined above can be modeled with global constraints such as Global Cardinality Constraint (GCC) [12]. The filtering algorithm proposed in [11] determines the set of all inconsistent values with GCC. This reduces the search space and makes easy finding a consistent solution.

4. User Axioms Classification

The constructs of ontology languages such as Ontology Web Language (OWL), though they cover the global definition of ontological knowledge base, do not allow the expression of some constraints between different ontology components.

Every instance of class__ appears at least once in slot __for any instance of class __.
Example: Every student has at least one advisor.
Every instance of Class Student appears at least once in slot advice: Class Professor for any instance of Class Professor.

Figure 2. Example of a template.

For example, constraints on roles values of the same class instances cannot be expressed. To overcome this deficiency and help the users, ontology development environments such as Protégé [10] include tools, generally based on first order logic such as PAL (Protégé Axiom Language), to capture these axioms.

However, it is pointed out [8] that these tools are not exploited and only few ontologies contain user defined axioms. The reason behind this weakness is the complexity [7] of languages underlying these tools.

To solve this problem, some works [7, 13] suggest specific user interfaces that make easy axioms expression without using these languages. Below we describe briefly the approach proposed in [7] on which we have elaborated our contribution.

The approach described in [7] consists of studying significant existing ontologies and deriving templates from user defined axioms. About twenty templates that cover 85% of user defined axioms have been abstracted in this study.

An environment has been developed as a plug-in of the protégé environment. It consists in an interface that allows the users to compose axioms by "filling in the blanks". In Figure 2, we give an example of such templates [7].

5. Constraint Programming

Constraint Programming (CP) is a problem solving environment exploiting various techniques coming from artificial intelligence, and operational research. It has been successfully applied to many problems in various domains such as planning and scheduling. It is particularly efficient to solve combinatorial problems.

Within CP, a specific framework called Constraint Satisfaction Problem (CSP) is defined. Below we present briefly this framework and we recommend the specialized references for further details [1, 3].

5.1. Definition of CSP

Formally, a CSP P is defined as a triple (X, D, C) where:

- X a set of n variables $x_1,\dots, x_n$.
- D a set of finite domains $D(x_1)$, …, $D(x_n)$ where $D(x_i)$ is the domain of the variable x_i.
- $C = \{C_1, ..., C_m\}$ a set of constraints defined on some variables. A constraint C_i is defined on a subset of variables. A solution is an assignment of values to the variables such that all the constraints are satisfied.
- For example, let be the following CSP:
  $X = \{x, y, z\}$
  $D = \{D(x),\ D(y), D(z)\}$ where
  $D(x) = D(y) = \{1, 2\}, D(z) = \{1, 2, 3, 4\}$.
  $C = \{C_1(x, y), C_2(y, z), C_3(x, z)\}$, where
  $C_1(x, y)$: $x \neq y$,
  $C_2(y, z)$: $y \neq z$,
  $C_3(x, z)$: $x + 1 \geq z$.

5.1.1. Solving CSPs

CSP solving techniques are organized according to the following algorithmic components: filtering, exploration and enumeration. Filtering reduces the search space induced by the CSP. Exploration explores intelligently the search space. Let be some CSP. If filtering cannot reduce the domains to a single solution or cannot prove the non-existence of solutions, then the values of the variables domain are enumerated.

The exponential complexity of this solving process is due essentially to enumeration. Filtering has usually a weak polynomial complexity.

More precisely [12], each constraint has a specific filtering algorithm. It removes, from the domains of constraints variables, all the values for which some constraint cannot be satisfied. These values are called non-locally consistent.

Arc-consistence [9] is the most used general property of filtering algorithms. Below, we give an informal introduction and some illustrations and we refer the reader to specialized references [1, 3] for more details.

When a value v of some variable x_i violates a constraint C_j, this value cannot participate to a solution; so it must be removed. We say that the value v of the variable x_i has no support in the constraint C_j, or more commonly: we say that v is not arc-consistent. In opposite, a value v of x_i, for which we can find values in the other variables of the constraint C_j, is called arc-consistent. Let be the CSP given above. We can notice that the value 1 for x has the support 2 in y in the constraint C_1; in the same way for 2. However, in C_3, the value 4 for z has not a support in C_3, and then it is removed. It is the unique value that must be removed with the arc-consistency property. The arc-consistency can be verified on various types of binary constraints (where each constraint holds on at most two variables) in a time weakly polynomial. There does not exist an efficient global algorithm for n-ary constraints; however algorithms exist for specific constraints such as the constraint of difference $x_1 \neq x_2 \neq \ldots \neq x_n$.

If a value v of some variable x_i leads to a solution for all constraints, we say that this value is globally consistent, otherwise it is removed. Reaching the global consistency of some value is NP-complete. So, filtering with the global consistency is naturally of exponential complexity. For example in the above CSP we see that the values 1 and 2 for z cannot be completed in x and y to form a solution, so they are removed. However the value 3 finds 2 in x and 1 in y.

Hence, the arc-consistency filtering leads to the following domains $D(x) = D(y) = \{1, 2\}$, $D(z) = \{1, 2, 3\}$. And the global consistence filtering leads to the following domains $D(x) = D(y) = \{1, 2\}$, $D(z) = \{1, 2, 3\}$.

In the following sections, we detail how to use filtering algorithms to detect ontology inconsistencies.

5.2. Dynamic CSPs

The CSP presented above is static. Its different components (e.g., variables, domains and constraints) do not change during solving. Nevertheless, in practice [15], this assumption is not usually verified, particularly when we are confronted to dynamic problems.

The dynamic CSP model [4] has been introduced to manage the changes occurring during solving. Below, we give a formal definition of dynamic CSPs.

Definition [15], a Dynamic CSP (DCSP) is a sequence $(P_1, P_2, \ldots, P_n)$ where for each $i \in 1 .. n$:

- $P_i = (X, D, C^i)$ denotes a CSP.
- Ca_i denotes a set of added constraints.
- Cs_i denotes a set of removed constraints.
- $Cs_i \subseteq C^{i-1}$ and $C^i = C^{i-1} + Ca_i - Cs_i$.

Note that this definition is so general that it can consider any CSP. It is the case when Cs_i are all constraints of P_{i-1}. In fact, we suppose that only a portion of constraints are added or removed at each step i. (The approach described in section 7.2 can be modeled with a dynamic CSP where the changes model is known a priori. Indeed, the CSP P_{i+1} results from the CSP P_i by reducing the domain D_i of the variable x_i to a single value v_i.)

6. CP Model for Ontologies Axioms

We show how the axioms templates elaborated in [7] can be translated to constraints within the CSP framework. This translation allows efficient algorithms developed in constraint programming to be used. The objective is to verify the user-defined constraints efficiently. We propose two schemes to translate axioms to CSP constraints [2].

Below, we give some translations of some axioms presented in [8]. The others are detailed in the appendix section.

Scheme 1: let be the class C. We note:

- $I_1, \ldots, I_n$: The instances of class C.
- $R_1, \ldots, R_m$: The roles associated to the class C.
- $Range(R_i)$: The set of values associated to the role R_i.

For the role R, we consider the following correspondence:

- x_i corresponds to I_i, i=1...n: the instances I_i are considered as the CSP variable x_i.
- $D_i = D(x_i) = Range(R)$: The domain D_i of the variable x_i is equal to the range of R.

The user defined axioms can then be expressed as CSP constraints. We give below an example.

Example

- Axiom: every instance of the class C must have a unique value for the role R.
- Corresponding CSP constraint: The axiom mentioned above can be expressed by the constraint: $Alldiff(x_1, \ldots, x_n)$ [11].

Scheme2

Let be some class C.

- $I_1, \ldots, I_n$: The instances of class C.
- $R_1, \ldots, R_m$: The roles associated to the class C.

  We consider the following correspondence:
- x_i corresponds to R_i, i=1,..., m: the roles R_i are considered as CSP variable x_i.
- $D_i = D(x_i) = Range(R_i)$: the domain $D(x_i)$ of the variable x_i is equal to the range of R_i.

The user defined axioms can then be expressed as the CSP constraints. We give below an example.

Example

- Axiom: for every instance of the class *C* the roles R_i and R_j cannot have the same value.
- Corresponding CSP constraint: The axiom mentioned above can be expressed by the constraint $x_i \neq x_j$.

7. Processing the Constraints

Let be a CSP modeling axioms coming from the scheme of some ontology. In order to detect inconsistent values we propose two approaches:

1. An incomplete approach: in this approach, the filtering algorithms associated to the constraints are executed iteratively until all constraints are locally consistent. This process detects and computes all values of the variables that are locally inconsistent with each constraint separately. Obviously, this procedure does not detect the global inconsistencies, i.e. the inconsistencies concerning all the constraints simultaneously.
2. A complete approach: in this approach we proceed with two phases. The first phase consists in applying a global solving algorithm to find all solutions. The second one examines the solutions set and detects the globally inconsistent values.

Obviously, the complete approach is the ideal one as it detects all the inconsistencies. However, it is computationally very expensive because of its NP-Hard nature. Here, the enumeration is forced by the guarantee that all values lead to solutions.

The incomplete approach, due to its low complexity, can be a reasonable approach as it enables finding inconsistencies as the user fixes the ontology variables without, however, ensuring that these values lead to solutions.

In the following, we propose architectures to integrate CP in inconsistencies detection. In practice, we can use the complete approach if it is not computationally very expensive; otherwise we opt for the incomplete one.

8. Consistent Handling of Changes

In this section we propose two approaches to support the user in managing consistently the ontology changes. These approaches go beyond the simple detection of inconsistencies. They offer user means ensuring the ontology evolution towards a consistent state. In the first approach, qualified with systematic, we describe the system architecture. For the second approach, qualified with interactive, we propose an interaction model between a user and a CP solver. This interaction leads to the final consistent ontology.

8.1. Systematic Approach

This approach consists in generating, a priori, all inconsistent values with ontology constraints. This set will be exploited to analyze the user input. Below, we describe the system architecture and the inconsistency checking process.

8.1.1. System Architecture

The proposed system contains the following components as shown in Figure 3.

1. The EZPAL interface: this interface is developed by [7] as a plug-in in the Protégé editor. It allows the user introducing his axioms simply by instantiating predefined templates. The axioms are then stored in axioms base.
2. The translator: this component translates the axioms in CSP Constraints. It uses the transformation base and generates the CSP constraints.
3. Inconsistent values generator: this component executes a set of filtering algorithms developed in constraint programming. It generates all the inconsistent values with CSP constraints generated by the translator.
4. Inconsistency checker: this component uses the inconsistent values base generated above and generates all the inconsistent values in the user data flow. A feedback is returned to the user.

8.1.2. The Process of Consistency Checking

The process of consistency checking proposed here can be divided into two phases as shown in Figure 3.

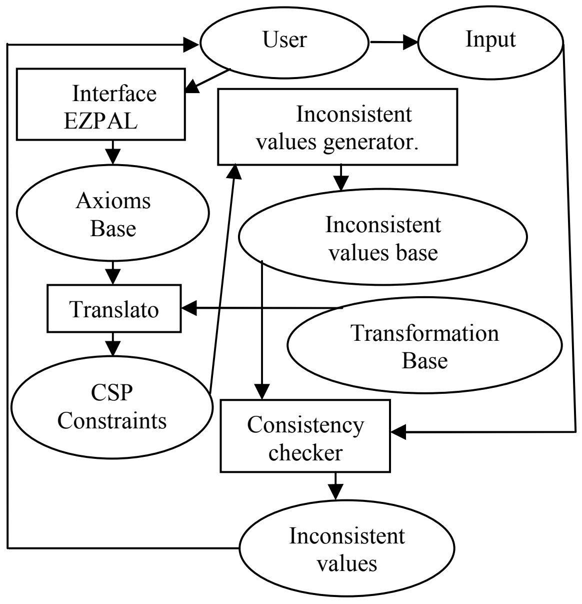

Figure 3. Specification of the environment.

Phase 1: the ultimate objective of this phase is to generate inconsistent values that enable detection of inconsistent values in the user data flow introduced in phase 2. This phase can be divided into two steps:

- Step 1: CSP constraints generation. The user composes his axioms by instantiating appropriate templates. He uses the EZPAL interface [7]. Then, these axioms are transformed into CSP constraints by the translator.
- Step 2*:* inconsistent values generation. The values that violate the CSP constraints, developed in the previous step, are generated. At this level, the filtering algorithms generate efficiently, a priori, all the values inconsistent with CSP constraints generated in step 1 above.

Phase 2: The ultimate objective of this phase is to detect the inconsistent values introduced in the user data flow. This phase can be divided into two steps:

- Step 1: formulation of the user request. The user request is assumed here to be of high complexity, in the sense that it contains a significant amount of data:
- The different classes to be modified.
- For every class, the different instances values for different roles. This information can be new, for instance, when adding instances or updating data.
- Step 2: Inconsistency detection. The inconsistency checker detects all the inconsistent values introduced in the user data flow in step 1 of phase 2 mentioned above. For this purpose, it uses the inconsistent values base.

The inconsistent values set are then returned to the user as a system feedback.

8.2. Interactive Approach

We give the interaction model between the user and a CP solver. This interaction has the particularity of conducting to a final consistent state as shown in Figure 4.

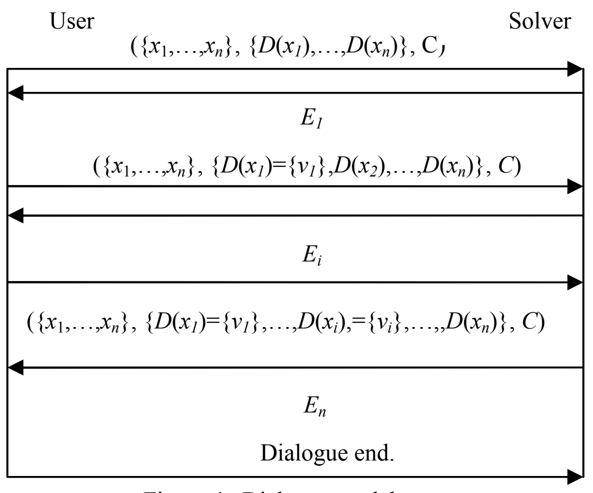
User
Solver
$(\{x_1,\ldots,x_n\}, \{D(x_1),\ldots,D(x_n)\}, C)$
E_1
$(\{x_1,\ldots,x_n\}, \{D(x_1)=\{v_1\},D(x_2),\ldots,D(x_n)\}, C)$
E_i
$(\{x_1,\ldots,x_n\}, \{D(x_1)=\{v_1\},\ldots,D(x_i),=\{v_i\},\ldots,D(x_n)\}, C)$
E_n
Dialogue end.

Figure 4 . Dialogue model.

8.2.1. The Interaction Model

Let be a CSP of n variables, the interaction model consists of a succession of n cycles. Each cycle consists of two successive phases described below.

Phase 1: the user sends a message to the solver. This message consists of a request. The purpose of the request is the computation of all values of x_i domain, D_i, that are consistent with the constraint C.

At this level the request embodies the following information:

- The domains $D_1, \ldots, D_{i-1}$ reduced to values $v_1, \ldots, v_{i-1}$ determined in previous cycles.
- $D_i, \ldots, D_n$ as defined in the initial CSP.

Phase 2: The solver sends a message to the user, which consists in an answer to the request formulated in Phase 1. The solver computes the set E_i (the set of all values of the variable x_i belonging to D_i that are consistent with the constraint C) which is transmitted to the user.

- In Phase 1 of cycle 1, the CSP transmitted is the initial one.
- The end of interaction occurs when the user sends an end message to the solver.
- At the end of each cycle, the user selects a value from the set returned by the solver and restarts the following cycle.

8.3. Comparison

The interactive approach has the following advantages:

1. The performance of filtering algorithms developed in constraint programming enables a real time interaction.
2. Interaction between the user and the solver converges to a consistent ontology.
3. The interaction model makes easy to the user selecting the affected value.

In the other hand, the systematic approach has the advantage of reducing the search space to only consistent values. This allows the user managing consistently the ontology changes. However, this approach requires also user intervention.

9. Conclusions and Perspectives

In this paper we showed that constraint programming can really serve ontologies consistency management. More precisely, filtering algorithms developed in constraint programming can contribute efficiently to the ontology user defined consistency checking. We proposed a translation model of axioms to constraints which is the core of an environment for supporting the user in managing consistently the ontology changes.

Large CSPs (with many variables), however, should be generated by the system. This is the main limit of the approach presented here. We are considering this issue by studying the boundary upon which the dialogue model will be hardly practicable.

The first perspective of this work is to implement our framework as a Protégé plug-in. In fact, the system is under development. The second perspective is to consider more general axioms not mentioned in [7].

The system presented here is built upon axioms which are abstractions of constraints expressed in much significant ontologies belonging to various domains. Hence, the system applications range is large: bioinformatics, management, etc., It can be plugged in every ontology management system.

References

[1] Apt K., *Principles of Constraint Programming*, Cambridge University Press, 2003.

[2] Beldiceanu N., Carlson M., Demassey S., and Petit T., "Global Constraints Catalog: Past, Present and Future," *Computer Journal of Constraints*, vol. 12, no. 1, pp. 21-62, 2007.

[3] Dechter R., *Constraint Processing*, Morgan Kaufmann Press, 2003.

[4] Dechter R. and Dechter A., "Belief Maintenance in Dynamic Constraint Networks," *in Proceedings of 7th National Conference on Artificial Intelligence*, USA, pp. 37-42, 1988.

[5] Fages J., *Programmation Logique et Par Contraintes*, Ellipses Presse, 1996.

[6] Haase P. and Stojanovic L., "Consistent Evolution of OWL Ontologies," *Lecture Notes in Computer Science*, 2005.

[7] Hou J., Musen A., and Noy F., "EZPAL: Environment for Composing Constraint Axioms by Instantiating Templates," *International Journal of Human Computer Studies*, vol. 62, no. 5, pp. 578-596, 2005.

[8] Hou J., Noy F., and Musen A., "A Template-Based Approach Toward Acquisition of Logical Sentences," *in Proceedings of the Conference of Intelligent Information Processing Montréal*, Canada, pp. 25-28, 2002.

[9] Mackworth A., "Consistency in Networks of Relations," *Computer Journal of Artificial Intelligence*, vol. 8, no. 1, pp. 99-118, 1997.

[10] The Protégé Project, http://protege.stanford.edu.

[11] Régin C., "A Filtering Algorithm for Constraints of Difference Constraint Satisfaction Problems," *in Proceedings of Association for the Advancement of Artificial Intelligence*, USA, pp. 362-367, 1994.

[12] Régin C., "Modélisation et Contraintes Globales en Programmation Par Contraintes," *Habilitation à Diriger des Recherches*, Université de Nice Sophia Antipolis, 2004.

[13] Staab S. and Maedeche A., "Axioms are Objects too Ontology Engineering Beyond the Modelling of Concepts and Relations," *in Proceedings of The 14th Electronic Cultural Atlas Initiative, Workshop on Ontologies and Problem Solving Methods*, Berlin, pp. 159-163, 2000.

[14] Stojanovic L., "Methods and Tools for Ontology Evolution," *PHD Thesis*, University of Karlshue, 2004.

[15] Verfaillie G. and Jussien N., "Constraint Solving in Uncertain and Dynamic Environment a Survey," *Computer Journal of Constraints*, vol. 10, no. 3, pp. 253-281, 2005.

Moussa Benaissa received his engineering degree and his Master degree in computer science from the University of Es-Senia Oran, Algeria, in 1988 and 1992, respectively. He is actually a lecturer at University of Oran Es-Senia. His research interest includes e-learning, ontologies, and constraint programming.

Yahia Lebbah received his engineering degree in 1995 from the University of Es-Senia Oran, Algeria, and his MS degree in 1996, from the University of Paris 13, and PhD degree in 1999 from Ecole des Mines de Nantes, France. All degrees are in computer science. He is currently a professor in the Computer Science Department at the University of Es-Senia Oran, Algeria.

Appendix

C denotes some class. *I* denotes an instance. *R* denotes a slot.

Axiom 1. Every instance of class *C* must have a unique value in slot *R*.

Constraint: $\forall I_1, I_2 \in C, R(I_1) \neq R(I_2)$.

Axiom 2. For every instance of class *C*, slots R_i and R_j cannot have the same value.

Constraint: $\forall I \in C,\ R_i(I) \neq R_j$.

Axiom 3. At least one instance of class *C* contains value *v* in slot *R*.

Constraint: $\exists I \in C, v \in R(I)$.

Axiom 4. If an instance of class *C* contains value v_1 in slot R_i it must contain value v_2 in slot R_j.

Constraint: $\forall I \in C, v \in R_i(I) \Rightarrow v_2 \in R_j(I)$.

Axiom 5. Every instance of class C_1 appears at least once in slot R of any instance of Class C_2. Constraint: $\forall I_1 \in C_1, I_1 \in R(C_2)$.

Axiom 6. For every instance of class C, value of slot R_i is > (or =, <), than the value of slot R_j. Constraint: $\forall I \in C, R_i(I)\ op\ R_j(I)$ where $op \in \{>, =, <\}$.

Axiom 7. For every instance of class C, slot R_i and slot R_j must have instances of the same class. Constraint: $\forall I \in C, \exists C_1, R_i(I) \in C_1$ and $R_j(I) \in C_1$.

Axiom 8. For every instance I_1 of class C_1, there must be an instance I_2 of class C_2 which contains I_1 in R.

Constraint: $\forall I_1 \in C_1, \exists I_2 \in C_2, I_1 \in R(I_2)$.

Axiom 9. Every instance of class C that shares the same value in slot R_i can not (resp. must) have identical values in R_j. Constraint: $\forall I_1, I_2 \in C, R_i(I_1) = R_i(I_2) \Rightarrow R_j(I_1) \neq R_j(I_2)$ (resp. $\forall I_1, I_2 \in C, R_i(I_1) = R_i(I_2) \Rightarrow R_j(I_1) = R_j(I_2)$).

Axiom 10. For every instance of class C_1, value of slot R_i must also be a value of slot R_j of an instance of class C_2.

Constraint: $\forall I_1 \in C_1, \exists I_2 \in C_2, R_i(I_1) = R_j(I_2)$.

Axiom 11. For every instances of class C slot R_i cannot contain both instances of class C_2 and class C_3.

Constraint: $\forall I \in C, R(I) \notin C_2 \cap C_3$.

Axiom 12. For every instance I_1 of class C_1 if the value of slot R_i is an instance I_2 of class C_2, they must share the same value in slot R_j for I_1 and slot R_k for I_2.

Constraint: $\forall I_1 \in C_1, R_i(I_1) = I_2 \Rightarrow R_j(I_1) = R_k(I_2)$.

Axiom 13. For every instance I_1 of class C_1 slot R_i must have values that are instances of classes specified by slot R_j of class C_2. Constraint: $\forall I_1 \in C_1, R_i(I_1) \in R_j(C_2)$.

Axiom 14. If an instance I_1 of class C_1 has slot R_i that contains value v then I_2 of class C_2 cannot (resp. must) contain I_1 in slot R_j. Constraint: $\forall I_1 \in C_1, R_i(I_1) = I_2 \Rightarrow I_1 \notin R_j(I_2)$ (resp. $\forall I_1 \in C_1, R_i(I_1) = I_2 \Rightarrow I_1 \in R_j(I_2)$).

Axiom 15. Every instance of class C_1, which has a value in slot R_i > (or =, <) than every (resp. at least one) value of slot Rj of class C_2. Constraint: $\forall I_1 \in C_1, \forall I_2 \in C_2, R_i(I_1)\ op\ R_j(I_2)$ (resp. $\forall I_1 \in C_1, \exists I_2 \in C_2, R_i(I_1)\ op\ R_j(I_2)$) where $op \in \{>, =, <\}$.

Axiom 16. For every instance I_1 of class C_1, if the value of slot R_i has an instance I_2 of class C_2, I_2 must have value v in slot R_j. Constraint: $\forall I_1 \in C_1, R_i(I_1) = I_2 \Rightarrow R_j(I_1) = v$.

Axiom 17. For every instance I_1 of class C_1, if the value of slot R_i has an instance I_2 of class C_2, then there is an instance I_3 of class C_3 that contains I_2 in slot R_j of class C_3. Constraint: $\forall I_1 \in C_1, R_i(I_1) = I_2 \Rightarrow \exists I_3 \in C_3, I_2 \in R_j(I_3)$.

Axiom 18. For every instance I_1 of class C_1, if the value of slot R_i is an instance I_2 of class C_2, then there is an instance I_3 of class C_3 that contain I_1 in slot R_1 and I_2 in slot R_2. Constraint: $\forall I_1 \in C_1, R_i(I_1) = I_2 \Rightarrow \exists I_3 \in C_3, R_j(I_3) = I_1, R_2(I_3) = I_2$.

Axiom 19. For every instance I_1 of class C_1, if the value of slot R_i has an instance I_2 of class C_2, then slot R_1 of I_1 has value > (or =,<) than value of slot R_2 of I_2.

Constraint: $\forall I_1 \in C_1, R_i(I_1) = I_2 \Rightarrow R_1(I_1)$ op $R_2(I_2)$ where $op \in \{>, = >\}$.

Axiom 20. If an instance I_1 of class C_1 has instance I_2 of class C_2 in slot R_1 of I_1 and I_2 of class C_2 has instance I_3 of class C_3 in slot R_2 of I_2 then I_3 must (resp. cannot) have I_1 in slot R_3. Constraint: $R_1(I_1) = I_2$ and $R_2(I_2) = I_3 \Rightarrow R_3(I_3) = I_1$ (resp. $R_2(I_2) = I_3 \Rightarrow R_3(I_3) \neq I_1$).

Axiom 21. For every instance I_1 of class C_1, if the value of slot R_1 has an instance I_2 of class C_2, then if slot R_1 of I_1 has value v_1 slot R_2 of I_2 has value v_2.

Constraint: $\forall I_1 \in C_1, R_i(I_1) = I_2 \Rightarrow (R_1(I_1) = v_1 \Rightarrow R_2(I_2) = v_2)$.

Axiom 22. Every instance I_1 of class C_1 which contains value v in slot R_1 must have values in slot R_2 which are instances of class C_2. Constraint: $\forall I_1 \in C_1, v \in R_1(I_1) \Rightarrow R_2(I_1) \in C_2$.