Constraint-Based Programming for Object-Oriented Knowledge Processing

Stefan Trausan-Matu and Gheorghe Ghiculete

Expert Systems Laboratory
Research Institute for Informatics
8-10 Averescu Avenue,
71316 Bucharest
ROMANIA

Abstract: This paper presents a constraint representation and processing system for artificial intelligence applications which can be used in connection with an object-oriented system. Constraints offer powerful knowledge representation and processing facilities. This is the reason why special attention has been paid to providing various constructs such as generic, parameterized and hierarchical constraints, a specialized language for declaring the functionality of constraints, concise connections declaration, as well as flexible processing capabilities. An important part of the paper is reserved to presenting the various possibilities of coupling constraints and objects in the XRL, LISP-based object-oriented programming environment.

1. Introduction

Constraint-based programming is a new programming paradigm, useful for both artificial intelligence (AI) and "traditional" applications (e.g. [1]). AI takes much interest in constraints due to the fact that they are a very suitable representation for describing complex problems involving search [2]. Constraint processing has been used in many artificial intelligence classes of problems: electronic circuits analysis [3], qualitative reasoning in physics [4], job shop scheduling [5], resource allocation [6], planning, design [7, 8], diagnosis [9], and others.

Constraints are relations between variables named the cells of the constraint. Each variable may have values in a particular set, either finite or not. Constraint networks are created by sharing the cells of a set of constraints. Constraint processing consists in the assignment and/or change of values in cells so that all constraints be eventually satisfied. Usually, algorithms for constraint processing use a propagation technique. The main idea would be that changes propagate locally from one constraint to another. These algorithms usually record (in an adequate data structure, for example, a queue) all the constraints to be processed. After considering one constraint, all the constraints affected by the computations are in their turn recorded. Therefore, constraint propagation may manifest as a cycle of selecting a constraint, updating its cells in order to make sure that it holds, and recording the constraints affected (of which cells are common with the modified cells of the current constraint). Propagation terminates when no more constraints are present in the data structure [10].

This paper describes the COPE constraint representation and processing system developed in COMMON LISP. COPE is dedicated to the development of knowledge-based applications. It can be easily connected with LISP-based object-oriented programming (OOP) systems. In the next chapter, some rationales about the design of the COPE constraint representation and processing system are discussed. The declaration of generic, parameterized, and hierarchical constraints as well as the language for connections declaration are also presented in the sections of the second chapter. The third chapter of the paper describes the possibilities of connecting constraints and objects. Finally, some conclusions are drawn.

2. Constraint Representation in Cope

Several constraint systems have been developed in LISP [11, 12, 13, 14, 7, 15]. Each of them has a different manner of representing constraints. For example, in KEE [11], constraints are represented as rules. In other systems (for example, [12]) constraints are relations between object

components. The authors' point of view is that that constraints must have a more complex representation. The rule representation is at a too small granularity level, and the relation one at a too high level. We have come to considering a constraint as an entity (something like a black box) with a number of terminals and specific behaviour (similar to [15]). Behaviour can be described by a set of cases (a kind of rules) in a special language, or by a relation. Consequently, all the rules for a constraint are grouped together and complex behaviours can be described.

Based on our experience in implementing another constraint processing system [19, 20, 25], special attention has been paid to both space and speed efficiency. Following these requirements, one main point was not to implement constraints as objects (which was the case of the former system). This is also justified by some observations about the applications written in the former system:

- Inheritance has not been extensively used for constraints.
- Constraints have a lower rate of change than other objects. The structure of constraints is in most of the cases fixed (for example, a constraint's main components are the cells involved, and the description of behaviour). In fact, in applications involving constraints, things which usually change are not constraints, but constrained entities. Constraints might eventually be relaxed, but relaxation does not imply a change in the structure of the constraint. It usually implies discarding or modifying the behaviour of a constraint.
- The number of uniformity constraint instances is, in general, much larger than the number of the constrained objects. Taking into account that mXRL is a classless OOP language implemented in LISP, for big applications efficiency problems might appear.

For the sake of uniformity, we have chosen to describe more efficient structure. We consider that the loss of flexibility is, for constraints, lesser than the gain in efficiency. Nevertheless, we provide a strong coupling facility between objects and constraints, as will be discussed in the third chapter.

COPE is an enhancement of the old system not only in the above described aspects but also in providing newer, stronger abstraction facilities. The design of COPE took into account the fulfilment of the following goals:

- Provision of facilities for declaring generic, parameterized, and hierarchical constraints
- Provision of a specialized language for declaring the functionality of constraints
- Reduction to a minimum of the declarations of connections between constraints
- Possibility of developing different kinds of constraint processing applications (e.g. constraint satisfaction problems, modelling and simulation, second generation expert systems)

2.1 Generic Constraints

Generic constraints are descriptions of classes of constraints which can be instantiated to particular constraints. This dichotomy is similar to the class - instance one in class-based object-oriented languages. Not only does it serve as an abstraction facility but also does it show performances because of a much simpler internal representation of an instance than that of a generic constraint. Another benefit from generic constraints is their reusability.

One of the main ideas on which COPE was built was the provision of a powerful language for the declaration of constraint functionality. We have described the functionality of a constraint as a set of rules which have in the condition and action parts some specific atoms and functions (e.g.: newvalue,: newcell atoms, and: known,: unknown, :exists,: val,: set-cell functions). For example, in the generic multiplier constraint with two terms below, the functionality is described by the production rules given in the slot: PropagationPrologue and those indicated as: PropagationCases.

```
(DeclareConstraint! multiplier
 :cells (p m1 m2)
 :vars (x)
 :PropagationPrologue ((when (:known :newcell)
                        (unless (eql (:val :newcell) :newvalue)
```

```
                (error "Conflicting values ! ....."))
                (:set-cell :newcell :newvalue))
 :PropagationCases (:zero_m1_or_m2 :all_known
:unknown_p
 :unknown_m2 :unknown_m1)
 :zero_m1_or_m2 ((:exists x :in '(m1 m2) :suchas
(and (:known x)(zerop (:val x))))
                (:set-cell 'p 0))
 :all_known ((:known '(m1 m2 p))
      (unless ( = (:val 'p)(* (:val 'm1) (:val 'm2)))
    (Break "Unsatisfied constraint relation .....")))
 :unknown_p ((:known '(m1 m2))
        (:set-cell 'p  (* (:val 'm1) (:val 'm2))))
 :unknown_m2 ((:known '(p m1))
        (:set-cell 'm2 (/ (:val 'p) (:val 'm1))))
 :unknown_m1 ((:known '(p m2))
        (:set-cell 'm1 (/ (:val 'p) (:val 'm2)))))
```

An instance, named instance_mul, of the multiplier is built in the first form and the value 8 for p is propagated using the second form below:

```
(MakeConstraint! multiplier :instance-name
instance_mul)
(constr-propagate 'p 'instance_mul 8)
```

2.2 Parameterized Constraints

Parameterized constraints are further steps towards providing powerful abstraction constructs and for reducing the amount of programs. A parameterized constraint is a generic constraint containing at least one parameter which refers to the declaration of cells in the generic constraint. For example, there might be multipliers with two, three and, in general, n terms. Without a parameterization facility, for each number of terms a generic constraint has to be defined. By declaring n, the number of terms as a parameter, a generic multiplier with n entries can be defined as follows:

```
(DeclareConstraint! pmul
 :parameters (n)
 :cells ((n :of m :at least 2)
        p)
 :vars (x)
 :PropagationPrologue .....
 :PropagationCases (:one_m_zero :all_known
                :one_m_unknown :unknown_p)
 :one_m_zero ((:exists x :in (:param-cells 'm)
                :suchas (and (:known x)(zerop
(:val x))))
                (:set-cell 'p 0))
 :all_known ((and (:known 'p)
                (:known (:param-cells 'm)))
            (unless ( = (:val 'p)
                (apply #'* (:param-cells-val 'm)))
            (Break "Unsatisfied constraint relation ....")))
.....)
```

An instance of this constraint can be defined as:

```
(MakeConstraint! pmul :instance-name
n_instance_mul :n 6)
```

2.3 Hierarchical Constraints

The hierarchical definition of constraints has two targets. The first one is to stick to the ideas of [15] i.e.to give the possibility of defining new, compound constraints from already defined constraints. The second one is to meet the requirement for grouping a set of constraints in a network of interacting constraints.

Here is an example of a hierarchical definition of a constraint describing a resistor, using two adders and a multiplier:

```
(DeclareConstraint resistor
 :cells (u1 u2 i1 i2 r)
 :GroupsOfCells ((t1 (u1 i1))
                (t2 (u2 i2)))
 :InnerConstraints ((mul :isa multiplier)
                (add :isa adder)
                (o :isa minus))
 :Connections ((:map 'r :to 'm2 :of 'mul)
                (:map 'i1 :to 't1 :of 'o)
```

```
(:map 'i2 :to 't2 :of 'o)
(:map 'u1 :to 's :of 'add)
(:map 'u2 :to 't1 :of 'add)
(:connect 't1 :of 'o :to 'm1 :of 'mul)
(:connect 'p :of 'mul :to 't2 :of 'add)))
```

The Resistor constraint can be further used for defining a new constraint, the TwoSeriesResistors constraint as follows:

```
(:Declare TwoSeriesResistors
 :cells (u1 u2 i1 i2 r)
 :GroupsOfCells ((t1 (u1 i1))
                 (t2 (u2 i2)))
 :InnerConstraints ((r1 r2 :isa resistor)
                    (node :isa node2)
                    (add :isa adder))
 :Connections ((:group-map 't1 :to 't1 :of 'r1)
               (:group-connect 't2 :of 'r1 :to 't1 :of 'node)
               (:group-connect 't1 :of 'r2 :to 't2 :of 'node)
               (:group-map 't2 :to 't2 :of 'r2)
               (:connect 'r :of 'r1 :to 't1 :of 'add)
               (:connect 'r :of 'r2 :to 't2 :of 'add)
               (:map 'r :to 's :of 'add)))
```

2.4 The Connections Declaration Language

For complex problems, with a great number of constraints and with many connections, the declaration of the connections between the constraints can become cumbersome and error-prone. For this reason and giving the possibility of using parameterized constraints in the definition of other constraints, a connection language has been devised and the grouping of a number of cells has been enabled. The connections language is used, of course, in a compound constraint. It reduces considerably the number of the declarations of connections. Two examples of the usage of this language are the: connections component of the Resistor and TwoSeriesResistors.

One further example, also involving a parameterized sum constraint is the NSeriesResistors:

```
(:DeclareConstraint NSeriesResistors
 :cells (u1 u2 i1 i2 r)
 :GroupsOfCells ((t1 (u1 i1))
                 (t2 (u2 i2)))
 :parameters (n)
 :InnerConstraints ((sum :isa (n_adder :n n))
                    (res :isa (:set n :of resistor :atleast 2))
                    (node :isa (:set (1- n) :of node2)))
 :connections ((:map 'r :to 's :of sum)
               (:group-map 't1 :to 't1 :of (res 1))
               (:group-map 't2 :to 't2 :of (res n))
               (do ((i 1 (1- i))) ((= i n))
                (:group-connect 't2 :of (resi) :to 't1 :of (node i))
                (:group-connect 't2 :of (node i) :
                 to 't1 :of (res (1+ i)))
                (:connect 'r :of (resi) :to (termi) :of 'sum))
               (:connect 'r :of (res n) :
                to (term n) :of 'sum)))
```

The GroupsOfCells facility offers the possibility of the declaration of a number of related cells as a whole. This is the case of the terminals (t1,t2) in the above resistor examples. One such terminal can be treated in a similar manner as a cell. This facility makes that the number of connection declarations be very much reduced and that another abstraction mechanism be provided (for example, when connecting two electrical components, we refer to the terminals connecting, abstracting the fact that the voltage and current cells of their corresponding constraints are connected). The GroupsOfCells idea is similar to a facility offered in the CONSTRAINTS language [15].

3. The Integration of Constraints and Objects

OOP is now largely viewed as a powerful programming paradigm, able to cope with software change and reuse. For AI applications, there have been developed OOP languages under LISP environments (for example, CLOS [21], KEE [22], and those implemented by the authors of this

paper: XRL [16], mXRL [17], O3). In these languages, objects are strongly related to the frame knowledge representation paradigm [23]. As a consequence, complex structuring of objects is used with a great emphasis put on multiple inheritance with method combination, and other AI knowledge representation and control mechanisms are integrated: rules, logic programming, demons, constraints.

The integration of a constraint-oriented framework into an OOP system enhances the power of representation and processing. Relations among objects or among components of an object, transformations of the state of objects [24], can be very elegantly described. If, in an AI OOP environment, besides constraints, an assumption-based truth maintenance (ATMS) [18] and world mechanisms [11] are integrated, complex search problems can be very naturally described and solved.

We consider that there are two main ways of connecting constraints and objects:

A) The connection of constraints on an existing object network. In this case we have implemented the following possibilities:

- coupling of a constraint on an object
- coupling of constraint on a fixed number of objects (statically determined)
- coupling of a (parameterized) constraint on a dynamically determined number of constraints
- coupling of constraints on tuple of objects of some type (clones or descendants of some object)

B) The connection of implicit constraints the moment an object is created. These constraints refer to slots of the created object. There are the following two possibilities:

- Implicit constraints (eventually parameterized) coupled on a single object
- Implicit constraints coupled on all the combinations of n objects

3.1 The Connection of a Constraint on An Existing Network

This coupling method may be useful in various problem solving regimes. For example, it may be used in a refinement-based system to infer values for some components or for checking the consistency of the object network.

The next example (used throughout the paper) is inspired by the DEXTY civil engineering expert system for the design of industrial halls (which the first author developed some years ago in XRL, without constraints).

```
(unit hall
 self (a unit)
 opening1
  (a opening
     width 12
     column1 (a column h 10 d 0.5 weight 3)
     column2 (a column h 10 d 0.3 weight 2.5)
     roof (a roof-struct
            beam (a transvers-beam weight 5)
            chesson (a chesson l 6 weight 2)
            chessons-no 4))
 opening2
  (a opening
     width 15
     column1 (a column h 10 d 0.8 weight 3.5)
     column2 (a column h 10 d 0.5 weight 3)
     roof (a roof-struct
            beam (a transvers-beam weight 6)
            chesson (a chesson l 6 weight 2)
            chessons-no 4)))

(unit roof-struct
   weight-roof undf
   weight-chessons undf)

(unit chesson l undf weight undf)
(unit transvers-beam l undf weight undf)
```

The coupling of an instance of the mul constraint on the hall object is done by:

```
(obj-c
```

```
'mul
'(hall ((m1 (opening2 roof chesson weight))
       (m2 (opening2 roof chessons-no))
                          (p (opening2 roof
weight-chessons))))
t)
```

The effect of this constraint is the computing of the total weight of the chessons in the second opening.

3.2 The Coupling of a Parameterized Constraint on an Object Network

There are many cases when the number of the components of a relation cannot be known in advance. Therefore, the coupling of a parameterized constraint on a variable number of objects may be very useful. Such a coupling of the parameterized "psum" constraint (a sum with a variable number of terms) on the widths of the openings in a hall is described below:

```
(unit hall1
 self (a unit supers (hall))
 total-width 53
 opening3 (a opening width 8)
 opening4 (a opening width undf))
(defparameter *hall1-openings*
   (let ((*%obj%* 'hall1))
    (allslots-clone-of 'opening)))

(objn-c 'psum *hall1-openings* 'ad 'width 'hall1 's
'total-width)
```

As the total-width (53) and the widths of the first, second, and third openings (12, 15, and 8) are known, the opening is to be computed (18).

3.3 Constrained Objects

Constrained objects, as opposed to the couplings discussed above, belong to a conceptually different kind of coupling. The difference between the two classes consists in the latter being implicit and contrasting with the explicit coupling discussed until now. This implicit coupling appears as a side effect of the creation of an object which has been declared as a constrained object. The fulfilment of the implicit coupling is possible via "after-create" demons attached to the constrained objects.

We have considered two ways of implicit coupling, depending on the number of objects which are implicitly coupled. For the first case, the single constrained objects must be meta-described as a ConstrainedUnit:

```
(unit ConstrainedUnit
 after-create (couple&activate-constraints))
```

The applicable constraints are listed in the "constraints" slot. Each of them is further described on its own in a slot as clones of one of the following two objects:

```
(unit Constraint-declaration
 precond t
 ctype undf
 pairs undf)

(unit Par-constraint-declaration precond t)
```

As an example we shall redefine the roof-struct object from the previous examples as a constrained object. After the definition of the new roof-struct, all its clones and descendants of roof-struct will be constrained objects.

```
(unit roof-struct
 self (a ConstrainedUnit)
 constraints (ches-w tot-w)
 ches-w
  (a constraint-declaration
    ctype mul
    pairs ((m1 (chesson weight))
           (m2 chessons-no)
```

```
        (p weight-chessons)))
 tot-w
 (a constraint-declaration
   ctype sum
   pairs ((t1 weight-chessons)
          (t2 (beam weight))
          (s weight-roof)))
 weight-chessons undf
 weight-roof undf
 chessons-nr undf
 chesson (a chesson)
 beam (a transvers-beam))
```

As an usage example, consider:

```
(unit hall238
 self (a unit)
 opening1
  (a opening
    roof (a roof-struct
           beam (a transvers-beam weight 5)
           chesson (a chesson l 6 weight 2)
           chessons-no 4))
opening2
 (a opening
   roof (a roof-struct
          weight-roof 15
          beam (a transvers-beam weight undf)
          chesson (a chesson l 6 weight 2)
          chessons-no 4))
opening3
 (a opening
   roof (a roof-struct
          weight-roof 12
          beam (a transvers-beam weight 6)
          chesson (a chesson l 6 weight 2))))
```

In the next example, the tot-w-roof will maintain the sum relation among all the weights of the roofs of all the openings and the total weight. This is an example of definition of a constrained object with a parameterized constraint.

```
(unit hall-with-total-roof-weight
 self (a ConstrainedUnit)
 constraints (tot-w-roof)
 tot-w-roof
   (a par-constraint-declaration
     ctype psum
     par-obj-list (allslots-clone-of 'opening)
     par-cell ad
     par-slot (roof weight-roof)
     cell s
     slot total-weight)
 total-weight undf)
```

A second possible implicit coupling, already implemented, is the connection of couples of objects with a constraint. Each time a new object is created and declared as being in a couple of constrained objects, new instances of the specified constraints are created and connected. For better understanding this way of connecting objects and constraints, we shall give an example: consider that in a factory area there are two kinds of machines (machine-type1 and machine-type2). A technological constraint states that these machines have some successor-predecessor relations. Another constraint states that the distance between two machines must be lesser than 3 meters. The types of machines and their instances are represented as:

```
 (unit machine-type1
 self (a unit)
 x-location undf
 y-location undf
    next-machine-type (machine-type2
machine-type7 machine-type11))
(unit machine-type2
self (a unit)
x-location undf
y-location undf
```

```
previous-machine-type (machine-type1 machine-type6))

(unit machine-type3
 self (a unit supers (machine-type1))
 x-location 22
 y-location 22
 next-machine-type (machine-type9))

(unit machine-type4
 self (a unit supers (machine-type2))
 x-location 7
 y-location 7
 previous-machine-type (machine-type1))

(setq s1
 (a machine-type1
   x-location 10
   y-location 10
   next-machine-type (machine-type2)))

(setq s2
 (a machine-type2
   x-location 22
   y-location 22
   previous-machine-type (machine-type7)))
```

The distance limiting condition is introduced as a constraint (dist3) telling if two machines are farther than 3 (meters). The next LISP form will have as effect that all the new clones or descendants of the machine-type1 and machine-type2 (successive in the manufacturing process) will be connected to dist3 constraints.

```
(obj2f-c 'dist3
 'machine-type1
      '(member 'machine-type2 (fslot
'next-machine-type *%obj%*))
 '((t1 x-location)(t2 y-location))
 'machine-type2
      '(member 'machine-type1 (fslot
'previous-machine-type *%obj%*))
 '((t3 x-location)(t4 y-location)))
```

For example, after getting the following new machine:

```
(setq s11
 (a machine-type1
   x-location 20
   y-location 20))
```

will signal a farther distance between s11 and machine-type4:

```
****************************************
```

There is a greater distance between points (7,7) and (20,20)

that correspond to

```
((Y-LOCATION . MACHINE-TYPE4)
 (X-LOCATION . MACHINE-TYPE4)
 (Y-LOCATION . MACHINE-TYPE1-210)
 (X-LOCATION . MACHINE-TYPE1-210))
****************************************
(setq s22 (a machine-type2
  x-location 11
  y-location 11))
```

After modifying the s11 machine position:

```
(pslot 'x-location s11 9)
(pslot 'y-location s11 9)
```

all the constraints will be satisfied.

4. Conclusions

We have tried to define a powerful constraint language for providing a new representation dimension to developing knowledge- based

applications. Aiming at providing flexibility, abstraction support, and concise description of programs, we have defined constructs for the declaration of generic, parameterized, and hierarchical constraints. The functionality of constraints and the connections between them are declared by specialized languages.

Not to implement constraints as objects in the context of LISP-based object-programming languages has been our choice. The explanation is that in such languages objects are usually used as frames, involving a lot of processing not required by constraints with more fixed character. Another positive argument for our opinion is that the number of constraints might be significantly larger than the number of the objects involved. The system presented in this paper has been used for developing several applications: a simple second generation expert system, a typical constraint satisfaction problem, an explanation-based learning problem, and some modelling and simulation applications in electronics and ecology. All these applications have demonstrated that constraint-based representation is a powerful abstraction facility, orthogonal with object-oriented representation, which can dramatically reduce the complexity of programs from the above domains. One of the applications put forward thousands of constraints and satisfactory results with regard to both space and speed were obtained.

REFERENCES

1. BORNING, A., DUISBERG, R., FREEMAN- BENSON, B., KRAMER, A.and WOOLF, M., **Constraint Hierarchies**, OOPSLA '87 Proceedings, 1987, pp. 48-60.
2. L Kanal and V. Kumar (Eds.) **Search in Artificial Intelligence**, SPRINGER-VERLAG, 1988, pp. 287-342.
3. DE KLEER, J., **How Circuits Work**, ARTIFICIAL INTELLIGENCE, 24, 1984, pp. 205-280.
4. DE KLEER, J. and BROWN, J. S., **Theories of Causal Ordering**, ARTIFICIAL INTELLIGENCE, 29, 1986, pp. 33-61.
5. FOX, M., **Constraint Directed Search: A Case Study of Job-shop Scheduling**, Research Report CMU-RI-TR-83-22, Carnegie- Mellon University, 1983.
6. MOTT, D. H., CUNNINGHAM, J., KELLEHER, G. and GADSDEN, J. A., **Constraint-Based Reasoning for Generating Naval Flying Programs**, EXPERT SYSTEMS, Vol. 5, No. 3, August 1988, pp. 226- 246.
7. STEFIK, M., **Planning with Constraints (Molgen: Part1)**, ARTIFICIAL INTELLIGENCE, 16, 1981, pp. 111-140.
8. MURTAGH, N. and SHIMURO, M., **Parametric Engineering Design Using Constraint-Based Reasoning**, Proceedings of AAAI90, 1990, pp. 505-510.
9. DE KLEER, J. and WILLIAMS, B.C., **Diagnosing Multiple Faults**, ARTIFICIAL INTELLIGENCE, 32, 1987, pp. 97-129.
10. DAVIS, E., **Constraint Propagation with Interval Labels**, ARTIFICIAL INTELLIGENCE, 32, 1987, pp. 281-331.
11. FILMAN, R., **Reasoning with Worlds and Truth Maintenance in a Knowledge-Based Programming Environment**, COMMUNICATIONS OF THE ACM, Vol. 31, No.4, April 1988, pp. 382-401.
12. GIUSE, D., **KR: Constraint-Based Knowledge Representation**, Research Report CMU-CS-89-142, Carnegie Mellon University, 1989.
13. GUESGEN, H. W., JUNKER, U. and VOSS, A., **Constraints in a Hybrid Knowledge Representation System**, Proceedings of IJCAI-87, pp. 30-33.
14. MOTTA, E., EISENSTADT, M., PITMAN, K. and WEST, M., **Support for Knowledge Acquisition in the Knowledge Engineer's Assistant (KEATS)**, EXPERT SYSTEMS, Vol. 5, No. 1, February 1988, pp. 6-27.
15. SUSSMAN, G. J. and STEELE, G. L., **CONSTRAINTS - A Language for Expressing Almost-Hierarchical Descriptions**, ARTIFICIAL INTELLIGENGE, 14, 1980, pp. 1-39.
16. BARBUCEANU, M. and TRAUSAN -MATU, S., **The XRL2 Manual**, ITCI Bucharest, 1988.

17. TRAUSAN-MATU, S., **Micro-XRL: An Object-Oriented Programming Language for Microcomputers**, Research Report, Institute for Technical Cybernetics, Slovak Academy of Sciences, Bratislava, 1989.

18. DE KLEERl, J., **An Assumption-Based TMS**, ARTIFICIAL INTELLIGENCE, 28, 1986, pp. 127-162.

19. TRAUSAN-MATU, S., **Constraint Processing in the mXRL Object-Oriented Language**, Research Report, Institute for Technical Cybernetics, Slovak Academy of Sciences, Bratislava, 1989.

20. TRAUSAN-MATU, S., **The Development of Constraint Processing Applications in the mXRL Object-Oriented Language,** Research Report, Institute for Technical Cybernetics, Slovak Academy of Sciences, Bratislava, 1989.

21. KEENE, S., **Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS**, ADDISON-WESLEY PUBLISHING COMPANY, Reading, MA., 1989.

22. FIKES, R. and KEHLER, T., **The Role of Frame-Based Representation in Reasoning,** COMMUNICATIONS OF THE ACM, Vol.28, No.9, September 1985, pp. 904-920.

23. MINSKY, M., **A Framework for Representing Knowledge**, in P.Winston (Ed.) The Psychology of Computer Vision, MCGRAW HILL, New York, 1975, pp. 211-277.

24. KNUDSEN, J.L., **Object-Orientation as an Integrating Perspective on Programming,** Proceedings of EastEurOOPe'91, Bratislava, 1991.

25. TRAUSAN-MATU, S., BARBUCEANU,M. and GHICULETE, GH., **The Integration of Powerful and Flexible Constraint Representation and Processing into an Object-Oriented Programming Environment,** Proceedings of "Representations Par Objets", La Grande Motte, France, June, 1992.