Joint NCC & IIS Bull., Comp. Science, 16 (2001), 153–170

Subdefinite data types and constraints in knowledge representation language

Yu. A. Zagorulko, Yu. V. Kostov, I. G. Popov

A knowledge representation language with subdefinite data types and constraints is considered. An important feature of this language is the possibility of operating with objects that may have slots (attributes) with imprecisely defined (subdefinite) values. Another important feature of the language is that it allows one to bind any object or relation to a set of constraints defined on the values of the object slots. The constraints defined on the slots with subdefinite values allow one to automatically refine such values. Since the constraints are encapsulated in the objects and relations of the semantic network, they can be added to or deleted from the current set of constraints during the process of computation as a consequence of creation or deletion of these objects or relations. A production rule technique is used to specify the process of inference and to manage the set of constraints, as well as to control a search for precise values (solutions).

Introduction

Knowledge representation languages are traditionally based on semantic networks, frames, and production rules. Usually, these languages allow one to operate on objects whose parameters are either completely known (i.e., have exact or precise values), or completely unknown. This limits their descriptive and functional capabilities and does not allow the use of incomplete or imprecise knowledge about the objects. Extension of knowledge representation languages with the ability to handle objects whose parameters are known approximately, i.e., the ability to represent and process inexact data and incomplete information, would enhance the descriptive power of such a language and expand the class of problems that can be solved.

Another shortcoming of traditional knowledge representation languages is their limited computational power. Unknown values of parameters can be obtained mainly through the inference of some kind. At the same time, there are other approaches and techniques that allow us, on the one hand, to work with incomplete or inexact data and, on the other hand, to support the declarative description of computational problems. In the former case, we refer to the method of subdefinite data types (SD-types) [1], and in the latter case, to constraint programming techniques [2, 3, 4]. Moreover, they are unified into the method of subdefinite computational models [5, 6]: this method is based on SD-types and, at the same time, is one of the most universal techniques in constraint programming.

The apparatus of SD-types can help to represent and process objects whose slots (attributes) can have subdefinite values (or SD-values), i.e., some ranges of possible values (sets of intervals).

The methods developed within the framework of constraint programming paradigm allow us to specify a problem as a set of constraints on the values of object attributes. In this case, a solution of the initial problem is reduced to a constraint satisfaction problem. This approach is particularly convenient when the constraints are represented in the form of ordinary logical expressions.

Since these methods permit one to solve a relatively narrow class of problems reduced to constraint satisfaction, there is a tendency to use them in combination with other means. For example, methods of constraint programming are built into imperative languages [7], as well as into the languages for logic and functional programming [8].

However, the extension of conventional programming languages with constraint programming techniques certainly extends their capabilities but does not make them knowledge representation languages, since their level of knowledge representation remains rather low and they are to be used by knowledge engineers rather than by programmers.

In addition, languages developed within the paradigm of constraint programming offer low-level tools for problem specification. As a rule, they provide the user with the ability to define constraints on "single" variables even when they are parts of complex entities, e.g., parameters of an object. As a result, the constraints on parameters of an object or arguments of a relation are also independent, single elements of knowledge that are connected only in the mind of the developer, even though these constraints either are properties of some objects or describe the semantics of relations between objects.

High-level problem specification tools can be developed by integrating constraint programming techniques with the object-oriented approach [9, 10, 11]. In knowledge representation languages, this can be seen in binding constraints to objects and relations of the semantic network. This approach, in particular, creates the necessary prerequisites for dynamic control of the set of constraints (through adding or deleting objects or relations of the semantic network).

In this paper we consider a knowledge representation language based on the abovementioned approach. This language includes subdefinite data types and constraint programming techniques, as well as some more conventional methods for representation and processing of knowledge and data.

Extending knowledge representation languages with subdefinite data types increases its descriptive power by allowing representation of inexact

and incomplete information. In particular, SD-types can be used to represent the values of parameters (or attributes of objects) for which only an approximate estimate or the type is known. At the same time, the use of the mechanism of constraints in the language allows comprehensive use of subdefinite data types. Once parameters with subdefinite values are linked by constraints, we can refine their values via constraint propagation, and in some situations even obtain their exact values. To resolve situations when the initial data (constraints) are not sufficient to find the exact values of parameters, the language should include the corresponding methods for solution search.

It should be noted here that the use of SD-types and constraints in knowledge representation languages not only improves its ability to infer new information, but also expands its domain of use covering the class of problems solvable by constraint programming.

Thus, such a language can be regarded both as a knowledge representation language, with extended capabilities for knowledge representation and processing, and as a high-level object-oriented constraint programming language.

The paper is structured as follows. Section 1 defines subdefinite data types and subdefinite computational models. In Section 2, we consider data types and knowledge representation facilities and knowledge processing capabilities of the knowledge representation language, and Section 3 gives an example of the use of the language.

1. Subdefinite data types and subdefinite computational models

In the early eighties, A.S.Nariny'ani proposed *the apparatus of subdefinite data types* [1] and method of *subdefinite computational models* [5, 6] as facilities for working with imprecisely defined values and objects.

First we consider subdefinite data types.

Let T be an "ordinary" data type with the set of values A and the corresponding set of operations over A. We denote the set of all subsets of A by A^*. Elements of A^* will be called *subdefinite values* or *SD-values* and denoted by a^*. The values a^* containing only one element of A will be called *exact values*. A special value equal to the entire set A will be said to be *fully indefinite*, and a value equal to the empty set will be called *inconsistent*.

For each operation $P : A^n \to A$ of type T, we can define the correspondent operation $P^* : A^{*n} \to A^*$ as a subdefinite extension of the operation P:

$$P^*(a_1^*, \ldots, a_n^*) = \{P(a_1, \ldots, a_n) | a_1 \in a_1^*, \ldots, a_n \in a_n^*\}$$

These new operations have similar semantics but may be applied to subdefinite values and the result of each of them is, in general, a subdefinite value too. So we can build a *subdefinite data type* T^* on the base of the original 'exact' data type T.

Presently, subdefinite extensions have been constructed for various data types: *integer*, *real*, *symbolic*, *logical*, *sets*, etc.

The method of subdefinite computational models implies the use of subdefinite data types to represent uncertain data in a problem which is specified in terms of *constraints* on its parameters.

Formally, *a constraint* is a boolean expression $C(v_1, \ldots, v_n)$ that is required to be true. The variables $v_1, \ldots, v_n$ linked by the constraint may get values of any subdefinite data type.

In the considered method each constraint must have *functional interpretations*. This means that the constraint can be represented by a set of functions:

$f_i^* : A^{*n-1} \to A^*$,

which are called *interpretation functions*. Each of these functions allows us to calculate the value of one variable from the values of the other ones:

$v_i = f_i^*(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n)$.

The set of such constraints is called *a subdefinite computational model* (*SD-model*) of the task. Interpretation of the constraints of this model, performed by a special data-driven algorithm, allows us to refine subdefinite values or even obtain exact values of the task parameters which satisfy given constraints.

The principle of data-driven computations (algorithm) means that a change of the value of any variable activates (causes execution of) the interpretation functions which depend on this variable; execution of the interpretation functions, in turn, may cause a change of the value of other variables, and so on.

If at least one variable gets an inconsistent value (for example, empty set of values), the process will stop and the *SD-model* will be considered inconsistent.

The important point here is that, during the process of interpretation, a new value a'' of the variable v is calculated and intersected with the old one (a'):

$v = a'' \cap a'$.

Since it is the result of this intersection that is assigned to the variable v, the value of v can be only refined, i.e. a new value is a subset of the old one. Notice that the value of the variable is considered to be changed only if it is actually refined.

Thanks to described above semantics of the assignment operator, as it

was shown in [6], for variables of all data types containing only a finite set of SD-values, this algorithm terminates in a finite number of steps. In the case of infinite sets of SD-values (for instance, intervals of real values), the stopping criterion can be based on the preset threshold of computation accuracy e. This threshold e determines the maximum possible distance between two values for which they are still considered to be identical.

We emphasize that the method of subdefinite computational models may be used to solve the problems that include variables of different types simultaneously. This proves the universal nature of this method. For this reason, we use the method of subdefinite computational models to implement the mechanism of constraint propagation in our knowledge representation language.

2. The knowledge representation language

The language is based on an integrated knowledge representation model that unifies such classic means as semantic networks, production rules, apparatus of subdefinite data types, as well as the methods of constraint programming. This natural integration is based on the object-oriented approach.

2.1. The data types

In the language, there are three kinds of data types: elementary, compound and semantic. *Elementary types* include integer, real, atom, char, and boolean. *Compound types* are string, structure, set and tuple. Classes of objects and relations defined by the user are considered as *semantic data types*.

A value of the type *atom* is a label (indivisible string) which can be only compared with another label.

A value of the type *boolean* can be only *true*, *false* or *undefined*, including both values *true* and *false* simultaneously.

An elementary type can have both definite and subdefinite values. A subdefinite value can be represented by an interval or enumeration. E.g., a subdefinite integer value can be the interval from 10 to 100,

$integer(10..100)$,

or the enumeration of the odd digits,

$integer(1, 3, 5, 7, 9)$.

The real interval from 1.8 to 43.7,

$real(1.8..43.7)$,

and the character ranges from 'a' to 'z' and from 'A' to 'Z',

$char('a'..'z','A'..'Z')$,

are also the examples of subdefinite values.

A subdefinite value of the type *atom* can be represented only as enumeration of labels, e.g.,

$atom(black, white, red, blue)$.

Notice that subdefinite values are basic, whereas the usual "precise" values are considered just as a special kind of subdefinite values.

A full-scale set of arithmetical and logical operations, as well as trigonometric and other mathematical functions are defined over numerical data. In addition, the language includes the built-in functions which return current upper and lower boundaries of a subdefinite value x, (*Low(x)* and *High(x)*), and the logical function detecting whether the value x is definite or subdefinite, (*IsPrecise(x)*).

Compound data types are used to aggregate values of any type except a string that consists of only characters. Elements of a structure, a set and a tuple can be not only values of elementary types but also references to objects of compound and semantic data types.

A structure is a construction which is similar a structure in C or a record in the Pascal language. It can integrate data of different types.

A set is a disordered list of elements with the traditional operations over sets (union, intersection, subtraction). A tuple is an ordered list of elements (which may be duplicated) with the operations of indexation and concatenation. It should be noted that all elements of a set and a tuple must have the same type. Sets and tuples can be nested.

In addtition to conventional operations, expressions with quantifiers (iterators) are defined for tuples and sets:

forall x **in** *Domain* **when** $P(x) : Q(x)$

and

exist x **in** *Domain* **when** $P(x) : Q(x)$.

The first iterator (**forall**) produces the value *true* if, for all elements x of *Domain* which satisfy the predicate P(x), the predicate $Q(x)$ is true. The second iterator (**exist**) produces *true* if at least one such element exists. *Domain* in the iterator can be any expression whose result is a tuple or a set; the predicates $P(x)$ and $Q(x)$ are logical expressions; the predicate $P(x)$ may be omitted.

Both iterators can also be regarded as the conjunction and disjunction, respectively, of the predicate Q(x) for all elements of Domain restricted by the predicate P(x).

The language also includes two additional iterators **sum** and **prod** which are used for arithmetical calculations:

sum x **in** *Domain* **when** $P(x) : F(x)$

and

prod x **in** *Domain* **when** $P(x) : F(x)$.

The first iterator (**sum**) calculates the sum of values of $F(x)$ for all elements x of *Domain* which satisfy the predicate $P(x)$. The second iterator (**prod**) calculates the product in the similar way. Note that the sum and the product may have a subdefinite value and the predicate P(x) may be omitted.

For example, we can check the presence of a positive number in the set $S = \{-3, -1, 0, 2, 5\}$ using the following expression

exist x **in** $S : x > 0$,

and if we want to check whether the set S includes only positive numbers, we have to use the expression

forall x **in** $S : x > 0$.

The result of calculation of the first expression will be **true** and one of the second expression will be **false**.

To calculate the sum of squares of elements of the set S and product of its positive numbers we can can use the following expressions

sum x **in** $S : x^2$

and

prod x **in** S **when** $x > 0 : x$.

There are also special types *any* and *nil*. The type *any* can be considered as the most base type for all other types. The value of *any* can be any value of the elementary, compound or semantic types. This type can be used in all cases when the actual type of the value is unknown. The actual type of a value will be determined at the run time. The type *nil* has only one value denoted by *'?'* symbol. This value may be used for compound and semantic data type to indicate that "no value" is actually presented.

The semantic data types are described in the next section.

2.2. The knowledge representation means

The basic tool used to represent declarative knowledge in this language is a so-called object-oriented *semantic network*, which consists of objects and relations. An *object* can be any entity of the subject domain defined by the knowledge engineer. *Relations* are used for linking objects in the semantic network.

Each object is characterized by its name (reference) and values of its attributes (*slots*). The values of the slots may be of any type defined in the language. The value of the object slot may be represented by the object as well. Objects may have *constraints* defined on the values of the object slots. These constraints are also considered as the features of the object. A similar set of constraints may be associated with a relation, but in this case constraints are defined on the slots of objects linked by this relation.

The constraints associated with some object or relation constitute its local SD-model.

Since the value of a slot can be an object, as well as a set or a tuple of objects, constraints can link the values of slots from several objects. The set of local SD-models of all objects and relations presented in a semantic network constitutes a *global SD-model* or *functional network*.

Notice that logical expressions from which constraints are composed may contain iterators defined for tuples and sets in addition to arithmetic and logical operations over values of elementary types.

Objects with the same properties are combined into one *class*. The properties of the class define the names and types of the values of object slots, possibly their default values, and the behavior of the objects.

Some classes may inherit the properties of other classes (in this case, the former are called subclasses, and the latter are superclasses), with a possibility of multiple inheritance. Some properties of a superclass can be redefined in subclasses.

An important feature of the language is that object slots may have subdefinite values. Subdefinite values can be defined as intervals or sets of possible values of elementary types. Such subdefinite values can be refined as a result of constraint propagation in the functional network (let us remind that the method of subdefinite computational models is used to implement the mechanism of constraint propagation in such a network).

Notice that the functional network is activated for every modification of the values of object slots it contains; it ensures recalculation and modification of the values of slots of the related objects. Activation and interpretation of the functional network is performed by a data-driven algorithm described in Chapter 1.

There is a special class of relations. These are *binary relations* for which a user can specify, along with the properties described above, the mathematical properties, e.g., *reflexivity, symmetry, transitivity, antireflexivity*, etc.

2.3. The knowledge and data processing means

To specify the knowledge inference processes, the language provides two methods. The first of them is declarative. It lies in the definition of the constraints on the slots of objects given in the specifications of the object classes and relations and determines local processes of computations in the functional network. The second method can be considered as less declarative, but more procedural or imperative. It serves for the specification of global processes of the knowledge inference in the form of the production rule system worked over the semantic network.

The first method was considered above, so let us consider the second one.

A production system consists of *the production rules* which have the traditional form:

$$CONDITION => ACTION,$$

where $CONDITION$ is the conditions necessary for the application of the rule, and $ACTION$ is the actions executed if the conditions are satisfied. Sign $'=>'$ is used to separate the left-hand part of a rule ($CONDITION$) from its right-hand part ($ACTION$).

The left-hand part of the rule contains a global condition and a list of patterns with local conditions.

The global condition is an ordinary logical expression.

The pattern is a template of instance of an object or a relation with the given values of slots. In addition to concrete values, the pattern can also contain local variables which get their values, when the pattern is matched with some object or relation of the semantic network. Each local condition is a usual logical expression defined on local variables of a rule; it specifies additional constraints on their values.

A part of patterns may be labeled by **not** specificator and treated as a negative context. The $CONDITION$ of the rule is satisfied only if the global condition is satisfied, and all ordinary patterns (ones not labeled by **not** specificator) and none of the patterns from the negative context are matched with the semantic network.

It should be noted that since global and local conditions of the rule may contain subdefinite values or variables with subdefinite values, the result of computation of this logical expressions may be, in general, undefined. In this case the rule is not applied.

The application of the rule is defined as execution of the actions specified in its right-hand part. This part contains the operations over objects and relations of the semantic network, as well as other operations, such as control of production activation, hypotheses and alternatives testing, search for exact values, data input/output and so on.

To support working with semantic networks, the language includes the operators for creation of objects and relations (**new**), editing them (**edit**) and deleting from the network (**delete**). It is also possible to save and restore the content of the semantic network at any time.

When an object is created, its slots are filled with the values indicated in the **new** operator. The slots the value of which are not specified will be filled with the default values. If there is no default value, then the slot will be filled with a subdefinite value — the entire set (domain) of admissible values if the slot is of an elementary type, or the completely indefinite value

'?' if the slot is of a compound or semantic type. Only after all slots are filled with values, the object is inserted into the semantic network. If the description of the object's class contains an SD-model, it is concretized by the values of its slots and added to the functional network.

The **edit** operator makes it possible to change the values of the slots of already existing objects and relations.

After objects or relations are deleted from the network, the semantic and functional networks are corrected. In the semantic network, all references to the deleted objects are replaced by the completely indefinite value *'?'*. The relations which link the deleted objects are removed from the semantic network. At the same time, all constraints related to the slots of the deleted objects and relations are removed from the functional network.

An important feature of the language is that creation, modification, and deletion of any object or relation leads to the immediate activation and interpretation of the functional network.

Since each constraint is linked to some object or relation of the semantic network, the global set of constraints (the functional network) can be modified during the process of computation as a consequence of both insertion of new objects and relations into the semantic network or deletion of the existing ones from it. This process is performed by the production rule system, which is traditional inference tools in the knowledge-based systems.

Thus, subdefinite values of slots may be refined by production rules as a result of direct editing of values of object slots or constraint propagation, that is as a consequence of insertion into the semantic network new relations containing the set of constraints which narrow the ranges of values of the slots.

Other important feature of the language is the availability of means for two-level dynamic control of activation of production rules. These facilities ensure high flexibility of knowledge (or data) processing control. They allow one to structure the set of production rules defined in the application, and then activate or deactivate some its subset depending on the situation.

If there is no enough constraints to completely refine subdefinite values of the object slots, then exact values can be obtained by special **precise** operator:

precise $Object : ClassName(Slot_1, \ldots, Slot_n)$,

where *Object* is a reference to the object, *ClassName* is the name of the object class and $Slot_1, \ldots, Slot_n$ are names of the object slots.

This operator searches for exact values of indicated slots of the object *Object*, which satisfy all constraints of functional network. If no such values are found, the contradiction is arouse.

A search for exact values may be performed together with maximization

or minimization of the value of some expression (target function). In this case the found exact values will satisfy all constraints and provide minimal or maximal value of the target function at the same time:

precise $Object : ClassName(Slot_1, \ldots, Slot_n)$ **minimize** $Expression$;
or

precise $Object : ClassName(Slot_1, \ldots, Slot_n)$ **maximize** $Expression$;

2.4. Hypotheses and alternatives testing

Since modification of the semantic network can lead to a contradiction in the global functional network, it is necessary to have abilities to roll back to the previous state. This mechanism is implemented in the language by *an operator of alternatives* (**try** operator), which allows us to define and execute several alternatives.

try $Alternative_1$
or $Alternative_2$
...
else $DefaultAction$
end

Any sequence of valid operations and statements may serve as an alternative ($Alternative_i$).

If execution of an alternative is successful, no other alternative is concerned. (The execution of the current alternative is successful if it does not lead to a contradiction.)

If execution of an alternative results in inconsistency, rollback is performed with following attempt to process the next alternative.

If all alternatives fail, the actions placed after **else** ($DefaultAction$) are executed.

The contradiction can occur in the following cases:

1) if the functional network is inconsistent (in the case when the value of some slots of objects do not satisfy the given constraints);
2) if the semantic network is inconsistent (i.e., the properties of some binary relation are not satisfied);
3) if **fail** operator is executed (it serves for an explicit generation of failure).

Notice that the **fail** operator can be included into the production rule, which traces the appearance of contradictory data in the semantic network. Furthermore, the **fail** operator (in a combination with the **precise** operator) can be used by the knowledge engineer for the forced selection of alternatives with the purpose to obtain all precise solutions.

Thus, the statement of alternatives, on the one hand, allows us to generate hypotheses and reject them if they lead to a contradiction. On the other hand, the use of embedded statements of alternatives allows one to define a case analysis similar to backtracking. In this case a contradiction means that the version under consideration is unsatisfactory and we must test another branch in the analysis tree.

3. Use of the language for developing a robot simulator

As an example, we consider the use of the language for developing a robot simulator.

The robot's world consists of several rooms which may contain some objects. Robot is capable to transfer objects. The classes of objects, such as furniture and equipment, are distinguished. Relations are used to define position of objects with respect to each other.

Since all things in the robot's universe have a certain position in space, they are all represented by geometrical figures.

First, we introduce the class *FIGURE* which includes the most general properties of all geometric objects:

```
class FIGURE
    x, y: integer(0..200 );
    left, right, top, bottom: integer(0..200 );
end;
```

Its parameters x and y denote the center of the geometric object in the Cartesian system of coordinates; *left, right, top*, and *bottom* refer to the objects borders. Basing on the class *FIGURE*, we construct the classes, like *RECTANGLE* or *CIRCLE*:

```
class RECTANGLE ( FIGURE )
    height,width: integer(1..200);
constraints
    height = top - bottom;
    width = right - left;
    left = x - width/2;
    right = x + width/2;
    top = y + height /2;
    bottom = y - height /2;
end;
```

```
class CIRCLE ( FIGURE )
    radius: integer(1..100);
constraints
    left = x - radius;
    right = x + radius;
    top = y +  radius;
    bottom = y - radius;
end;
```

The *RECTANGLE* and *CIRCLE* inherit all the properties of the class *FIGURE*. Besides, the class *CIRCLE* has an additional slot *radius* and the class *RECTANGLE* has two additional attributes *height* and *width*. In addition, these classes have also a set of constraints linking the values of their slots.

The classes *ROOM, TABLE* and *CHAIR* are introduced to represent a room, a table and a chair, respectively.

```
class ROOM( RECTANGLE );
class TABLE( RECTANGLE );
class CHAIR( CIRCLE );
```

The *ROOM* is derived from the *RECTANGLE* and has an additional slot *number* denoting the number of the room. The *TABLE* is a subclass of the class *RECTANGLE*, and the *CHAIR* is a subclass of the class *CIRCLE*.

We can define various relations on these objects, for instance:

```
relation LEFT(what, from: FIGURE)
  antiref, antisym, trans;
constraints
  what.right < from.left;
end;

relation INSIDE(what, into: FIGURE)
constraints
    what.left >= into.left;
    what.right <= into.right;
    what.top <= into.top;
    what.bottom >= into.bottom;
end;

relation IN_THE_CENTER (INSIDE)
    what, into: FIGURE
constraints
```

```
    what.x = into.x;
    what.y = into.y;
end;
```

The relation *LEFT* states that one figure (*what*) is located to the left of other one (*from*). The relation *INSIDE* states that one figure (*what*) lies within the other one (*into*). The relation *IN_THE_CENTER* indicates that one figure is situated in the center of the other. These relations are connected with constraints defining the relationships between the boundaries (*LEFT, INSIDE*) or centers (*IN_THE_CENTER*) of two objects.

Suppose there is a room that has a number (*number=5*) and a definite size (*width=200, height=200*) and coordinates of the center (*x=100, y=100*), and a table of a definite size (*width=80, height=40*) and indefinite coordinates x and y (by default, each their value will be the interval (*0..200*)). This situation is simulated by addition of two objects *Room* and *Table* to the semantic network:

```
Room: ROOM (number: 5, x: 100, y: 100,
            width: 200, heigth: 200),
Table: TABLE (width: 80, heigth: 40).
```

Now, let the robot get a command to move the table to the room number 5. Linking of the objects *Table* and *Room* by the relation *INSIDE* corresponds to this action:

```
INSIDE ( what: Table, into: Room).
```

After addition of this relation to the semantic network and constraint propagation, the object *Table* gets the following form:

```
Table:TABLE (width: 80, heigth: 40, x: 40..160, y: 20..180,
             left: 0..120, right: 80..200,
             top: 40..200, bottom: 0..160).
```

Note that, though the coordinates of the object *Table* are refined, they remain still subdefinite. So, the center of the *Table* may be at any point of the square with a diagonal from (*40,20*) to (*160,180*).

The robot cannot yet execute the command and apply to his knowledge base. Let the knowledge base of the robot contains a rule according to which a table is situated in the center of the room if the center is not occupied:

```
rule PlaceTableInCenter
forall
```

```
  T: TABLE ( x:X, y:Y ),
  R: ROOM(),
  INISIDE( what:T, into:R )
when
    not IsPrecise( X ) and
    not IsPrecise( Y ) and
    not IN_THE_CENTER ( what:F, into:R )
=>
 new
    IN_THE_CENTER ( what:T, into:R );
end;
```

The application of this rule results in adding the relation *IN_THE_CENTER*, linking the object *Table* with *Room*. Addition of this relation to the semantic network will lead to activation of the functional network. The process of refinement will be realized by the common mechanism of the constraint propagation.

The slots of the object *Table* will be fully refined and have the form:

```
Table: TABLE (width: 80, heigth: 40, x: 100, y: 100,
              left: 60, right: 140, top: 120, bottom: 80).
```

Thus, by linking the subdefinite object *Table* with the definite object *Room* via the relation *IN_THE_CENTER*, we have fully refined the coordinates of the former. It is possible now due to the process of satisfying constraints which describes the internal semantics of the relation *IN_THE_CENTER*.

Assume now that the robot should move the chair of a definite size (*radius = 30*) to the room number 5 and place it to the left or to the right of the table and at the same line with it. First of all, this action implies insertion of the following relation into the network:

```
new Inside( Chair, Room );
```

Since up to now there are no relations between the chair and other objects in the room, the coordinates of the chair are imprecise:

```
Chair: CHAIR( radius: 30, x: 30..170, y: 30..170, left: 0..140,
              right: 60..200, top: 60..200, bottom: 0..140).
```

Assume that at first the robot decides to place the chair to the left of the table. This decision may be realized by the following operator:

```
try
  new LEFT(Chair, Table );
or
  new RIGHT(Chair, Table );
else
  message ("I can't execute this command");
end;
```

After execution of the first alternative and creation of the relation *LEFT (Chair, Table)*, x coordinate of the chair becomes precise, while y coordinate is still imprecise:

```
Chair: CHAIR( radius: 30, x: 30, y: 30..170, left: 0,
              right: 60, top: 60..200, bottom: 0..140).
```

Now with help of following operator the robot can find the precise values of coordinates of the chair, such that the chair is at the same line with the table:

```
precise Chair( y )  minimize Chair.y - Table.y;
```

After finding the precise values satisfying all constraints, both coordinates of the chair become precise:

```
Chair: CHAIR( radius: 30, x: 30, y: 100,
              left: 0, right: 60, top: 130, bottom: 70).
```

If the robot could not place the chair to the left of the table (setting the *LEFT* relation would lead to inconsistency of the constraint system) then it could explore other alternatives, i.e. it would place the chair to the right of the table. If this action would be impossible, then the robot would send a message *"I can't execute this command"*.

Thus, arrangement of objects in the rooms results in adding new relations, linking the objects with rooms. After creation of a new relation, parameters of objects are refined according to new constraints. The process of refinement in all cases is realized by common mechanism of the constraint propagation. This process can be controlled by special facilities similar to **precise** and **try** operators.

4. Conclusion

The presented language is the development of the knowledge representation language [11, 12]. The use of the object-oriented approach makes it possible to unify various means of knowledge representation, such as frames,

semantic networks, and production rules, as well as methods of constraint programming within the framework of one language.

The use of SD-types and constraints in the knowledge representation language not only improves its ability to infer new information, but also expands its application domain covering the class of problems solvable by constraint programming.

Encapsulation of constraints into objects and relations increases the level and convenience of constraints specification and use. It allows us to represent functional and computational links between attributes of objects or parameters of task by declarative relations.

Encapsulation makes it possible also to control the set of constraints during the process of computation. The constraints can be added to or deleted from the current set of the constraints as a consequence of creation or deletion of objects or relations of the semantic network. The production systems technique included in the language allows one to perform this control using the logical inference and heuristic analysis.

Joint use of logical inference and constraint propagation gives certain advantages. The former stimulates the latter and vice versa. So, refinement of a value of a variable (object slot) obtained as a result of constraint propagation provides conditions for activation of a rule, and in turn, application of a rule can lead to refinement of values of variables or addition new constraints which also can be used for refinement of values of variables and so on.

Thus, the proposed language can be regarded both as a knowledge representation language with extended capabilities for knowledge representation and processing, and as a high-level object-oriented constraint programming language.

The language can be used to create a broad class of intelligent application systems which require the combination of logical inferences, constraint propagation and computations over imprecise values. In addition, since the set of active constraints can be modified during the process of computation, this language can also be used for developing systems which can regard the dynamics of processes, in particular, hybrid expert systems and systems for intelligent robot control.

References

[1] Narin'yani A. S. *Sub-definiteness and Basic Means of Knowledge Representation*, Computers and Artificial Intelligence, 1983, Vol. 2, No. 5, P. 443–452.

[2] Henz M., Smolka G., Wurtz J. *Object-Oriented Concurrent Constraint Programming in Oz*, DFKI Research Report RR-93–16, April 1993.

[3] Freeman-Benson B., Malony J., Borning. *An Incremental Constraint Solver*, Comm. of the ACM, 1990, Vol.33, No. 1, P. 54–63.

[4] Borning A., Freeman-Benson B., Willson M. *Constraint Hierarchies*, Lisp and Symbolic Computation, 1992, No. 5. P. 223–270.

[5] Narin'yani A. S. *Subdefinite Models: A big jump in Knowledge Processing Technology*, Proceedeings East-West Conference on AI: from theory to practice, EWAIC'93, September 7–9, Moscow, 1993, P. 227–231.

[6] Telerman V., Ushakov D. *Data Types in Subdefinite Models.* In: Jacques Calmet and others (eds.), Art. Intell. and Symbolic Mathematical Computation, Lecture Notes in Computer Science; Vol. 1138, Springer, (1996), P. 305–319.

[7] Apt K. R., Brunekreef J., Schaerf A. and Partington V., *Alma-0: An Imperative Language that Supports Declarative Programming*, ACM Toplas, 20(5), P. 1014–1066.

[8] Cohenen J. *Constraint Logic Programming Languages*, Comm. of the ACM, 1990, Vol. 33, No. 7. P. 52–68.

[9] Freeman-Benson B. N., Borning A. *Integrating Constraints with an Object-Oriented Language*, Proc. of the Conf on Object-Oriented Programming, Marseille, 1992. P. 248–266.

[10] Telerman V. V., Sidorov V. A., Ushakov D. M. *Problem Solving in the Object–Oriented Technological Environment NeMo+*, Perspectives of System Informatics (PSI–96), Berlin: Springer, 1996, P. 91–100, (Lect. Notes Comput. Sci.; Vol.1181).

[11] Zagorulko Yu. A., Popov I. G. *Object-Oriented Language for Knowledge Representation Using Dynamic Set of Constraints*, Knowledge-Based Software Engineering, P.Navrat, H.Ueno (eds), (Proc. 3rd Joint Conf., Smolenice, Slovakia), Amsterdam: IOSPess, 1998, P.124–131.

[12] Zagorulko Yu. A., Popov I. G. *Knowledge representation language based on the integration of production rules, frames and a subdefinite model*, Joint Bulletin of the Novosibirsk Computing Center and Institute of Informatics Systems, Series: Computer Science, 8 (1998), NCC Publisher, Novosibirsk, 1998, P.81–100.