Joint NCC & IIS Bull., Comp. Science, 8 (1998), 81–100
© 1998 NCC Publisher

Knowledge representation language based on the integration of production rules, frames and a subdefinite model*

Yury A. Zagorulko and Ivan G. Popov

A knowledge representation language based on integration of such means as frames, semantic networks, production rules, subdefinite computational models, and methods of constraint programming are considered. An important feature of this language is the possibility of operating with objects which can have slots with imprecisely defined (subdefinite) values. Another important feature of the language is that it allows one to bind to any object a set of constraints defined on the values of the object's slots. These constraints, represented in the form of usual logical expressions, not only serve for control of consistency and correctness of the slots' values but also allow one to refine automatically subdefinite values. Besides, the possibility of including the operation of implication in constraints makes it possible to provide local inference within the object.

Introduction

The main trend in artificial intelligence is not the opposition of different means and methods for knowledge representation and processing, but their rational combination within one system. This trend is based on the understanding of the fact that each of these means executes its function and has its application domain. On the other hand, none of the well-known means for knowledge representation and processing can support comprehensive development of a real application on its own.

New technologies for developing intelligent systems have to unify various complementary means and methods of knowledge representation and processing. In the present paper we describe a language based on an integrated model of knowledge representation unifying such means as frames, semantic networks, production rules, subdefinite computational models, and constraint programming technique. It can be used to create a broad range of intelligent systems, in particular, systems requiring the combination of logical inference and computations over imprecise values and to cope with the problems which have a set of solutions.

*Supported by the Russian Foundation for Basic Research under Grant 96–01–01607.

1. A brief overview of knowledge representation and processing means

Before describing the integrated model, let us consider the main well-known means and methods for knowledge representation and processing.

One of the most popular knowledge representation means is *semantic network* [1]. Due to such properties as high associativity and flexibility for representation of information, semantic networks are considered to be a universal storage for any information (knowledge or data), that can be represented in terms of objects and relations between them. These properties have made semantic networks very popular. However, high flexibility of semantic networks is provided by means of a rather low level of knowledge representation that leads to complicated description of a problem, and thereby makes the work of the knowledge engineer (the user) extremely difficult.

The formalism of *production rules* [2] is regarded as a powerful tool for expressing the operational semantics of the notions of the subject domain and for logical inference. It is characterized by a natural specification of knowledge, simplicity of modification and extension, and natural modularity. But production systems are oriented mostly to symbolic computations, therefore they are very difficult to apply to the solution to problems requiring numerical calculations.

The use of *frames* [3, 4] and *object-oriented approach* [5, 6] allows one to raise greatly the level of the knowledge representation language and the possibility of its customization in a concrete subject domain. The main advantage of such an approach is the possibility of linking locally every frame with its properties. This saves one from the necessity to take care of small details at the global level. Frame-based representation languages have the mechanism of demons and attached procedures which allows one to define functional dependencies for values of the slots. But, as a rule, in frame languages such dependencies are given at a low level, for example, in terms of procedures and functions, which makes the knowledge engineer to turn to skilled programmers for help. It would be better to use for this purpose the mathematical and logical formulae or other means of high level.

In the framework of the paradigm of *constraint programming* [7], which is very popular now, it is possible to specify knowledge in the form of a set of *constraints* over the values of parameters of objects. However, this approach allows one to solve a rather narrow class of problems which are reducible to the constraint satisfaction problem. Consequently, in practice constraints are often used in combination with other, more universal means of knowledge representation and processing.

It should be noted that none of the above techniques has provided the means for operating with imprecisely defined values and objects.

True, the attempts of imbedding such means in AI languages were made. However these languages do not allow one to achieve sufficient efficiency of implemented application systems. The extension of conventional programming languages, for example C++, with the means for operating with imprecisely defined values [8] certainly increases their possibilities but does not make them knowledge representation languages, since their level remains yet low in order to use them by knowledge engineers rather than programmers.

The facilities for working with imprecisely defined values and objects can be found in the method of subdefinite data types [9], proposed in the early eighties, and in the method of *subdefinite computational models* [10] based on it. We consider this approach in a bit more detail, as it is of particular importance.

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:

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

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.

Subdefinite data types can be used to represent uncertain data in a problem which is specified in terms of *constraints* on its parameters.

Formally, *a constraint* is a boolean expression $C(v_1, \dots, v_n)$, that is required to be true. The variables $v_1, \dots, v_n$ linked by the constraint may get values of any subdefinite data types [2].

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:

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

For example, the constraint

[2]Variables in the programming language sense are meant here.

$$A = B + C \tag{1}$$

can be interpreted by the following three interpretation functions:

$$\begin{aligned} A &= f_1^*(B, C), \\ B &= f_2^*(A, C), \\ C &= f_3^*(A, B), \end{aligned}$$

where

$$\begin{aligned} f_1^*(x, y) &= x + y, \\ f_2^*(x, y) &= x - y, \\ f_3^*(x, y) &= x - y. \end{aligned}$$

Here $'+'$ and $'-'$ denote subdefinite extensions of arithmetical operations 'plus' and 'minus', respectively.

If an expression in a constraint is too complicated, it is possible to simplify it by adding new variables and splitting the complex constraint into several simpler ones.

For example, the constraint

$$A = (B + C)^2 \tag{2}$$

can be divided into two more simple constraints:

$$\begin{aligned} S &= B + C, \\ A &= S^2, \end{aligned}$$

where S is an auxiliary variable.

In most cases, any set of constraints can be transformed in a set of elementary constraints as it is shown in the latter example.

A *subdefinite computational model* (*SD-model*) is represented by a bipartite oriented graph (*subdefinite functional network* or *SD-network*) and a discipline of its processing, or *data-driven computations*. There are two types of vertices in an SD-network: variables and interpretation functions. The incoming edges of a function vertex connect it to the variables whose values are input arguments of the function. Outgoing edges of the function vertex point to variables in which store the results produced by the function.

For example, the subdefinite functional network corresponding to the constraint (1) is illustrated in Figure 1.

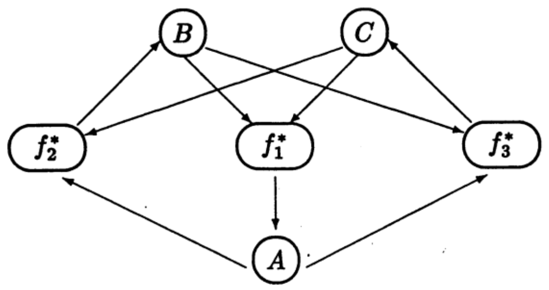

Figure 1. A simple subdefinite functional network

The principle of data-driven computations means that a change of the value of variable vertices activates (causes execution of) the function vertices; execution of the function vertices, in turn, may cause a change of the resulting value of variable vertices, and so on. If at least one variable gets an inconsistent value (empty set), the process will be stopped and the *SD-model* will be considered inconsistent.

During the process of interpretation a new value of the variable is calculated and intersected with the old one. Since it is the result of this intersection that is assigned to the variable, the value of the variable can be only refined, i.e., the new value is a subset of the old one. The value of the variable is considered to be changed only if it is actually refined.

As it was shown in [10], for 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. This threshold determines the maximum possible distance between two values for which they are still considered to be identical.

Presently, subdefinite extensions have been constructed for various data types: *integer*, *real*, *symbolic*, *logical*, *sets*, etc. We emphasize that the same method is used to solve problems for all of these types, that is the method of subdefinite computational models. This proves its universal nature. For this reason, the method of subdefinite computational models can be used to implement the mechanism of constraint propagation.

It should be noted that exclusive use of data-driven computations in this method makes it difficult to apply it to problems requiring logical inference. In addition, existing implementation of SD-models have one drawback that precludes their broad use. They are static, i.e., their structure cannot be changed during computations, for instance, by adding new constraints or modifying (deleting) existing ones. Therefore, they cannot be used, in particular, to construct intelligent systems taking into account the dynamics of processes.

Thus, a brief overview of knowledge representation and processing means and methods accumulated in the field of artificial intelligence shows that, in spite of their remarkable qualities, each of them has shortcomings which limit their possibilities and application domain. Only integration of these means and methods within one knowledge representation model allows one to create a powerful and universal apparatus for development of a broad range of knowledge based systems.

2. The integrated knowledge representation model

Let us consider *the integrated knowledge representation model.* The main notions of this model are objects, relations, constraints, and production rules.

The base tool used to represent declarative knowledge in this model is *a semantic network*, which consists of objects linked by binary relations. The processes of inference and data processing are defined mostly as *a production rule system* working over the semantic network.

An *object* can be any entity of the subject domain, defined by the knowledge engineer. Each object is characterized by its name and the values of its attributes, or slots.

Objects with the same properties are combined into one *class*. The properties of the class determine the number and types of its slots, their possible values, and the behavior of the object. Classes may inherit 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 superclasses can be redefined in subclasses, which provides polymorphism as one of the main notions of the object-oriented approach.

The values of slots can be characters, strings, atoms (indivisible strings), integer and real numbers, tuples and sets. An object may be the value of a slot too. An important feature of objects is that their slots may be subdefinite, i.e., their values may be subsets of the domain of admissible values. Subdefinite values can be defined as intervals of values for numerical data types and as sets of possible values for other types.

The integrated model allows one to bind with any object a set of constraints defined on the values of its slots. *The constraints* associated with some object constitute its *SD-network* and not only serve for control of consistency and correctness of the slots' values but also make it possible to refine automatically subdefinite values. The method of subdefinite computational models is used to implement the mechanism of constraint propagation in such a network.

Since the value of a slot can be an object (or a set of objects), constraints can link the values of slots from several objects. The set of SD-networks of all objects presented in a semantic network constitutes a *global SD-network*, which is activated for every modification of the values of slots of the objects it contains; it ensures recalculation and modification of the values of slots of the related objects. Let us remind that the activation and execution of a SD-network are a data-driven process.

The possibility of including in the SD-network constraints with the operations of implication makes it possible to model the simplest production rules providing local inference within the object. Since any object may be the value of a slot of another object, the global SD-network can be modified by the execution of such rules. This takes place because the process of setting up new links among objects can lead to involving into the global SD-network new constraints which include slots of these objects.

Thus, the global SD-network may be modified (expanded) during the application system operation as a consequence of both insertion of new objects and relations into the semantic network and the local inference which is provided by constraints which have form of logical implications.

A *binary relation* is treated as a special object with two slots. We can define constraints for relations just as we do it for objects. We have chosen binary relations for the development of the integrated model because of their flexibility and popularity, and also because they have useful properties like *reflexivity*, *symmetry*, *transitivity*, which can be built into the system.

Classes of objects and relations describing the notions of a subject domain serve to represent its model. The model of a problem (a concrete application) is given as a set of production rules, which operates with a semantic network formed by the above-mentioned objects and relations.

3. The knowledge representation and processing language

The knowledge representation and processing language based on the above model includes facilities for describing classes of objects and relations, as well as tools for defining the process of inference and data processing which are required by a particular application.

3.1. Data types

In the language, there are three kinds of data types: elementary, structured and semantic. *Elementary types* include integer, real, atom, char, string, boolean. *Structured types* are 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 which can be only compared with another label. A value of the type *boolean* can be only *true*, *false* or *undefined.*

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 an 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. For example the obvious value 10 and the subdefinite value $integer(10)$ are identical.

A full-scale set of arithmetical, logical and trigonometric operations as well as mathematical functions are defined over numerical data. In addition, the language includes the built-in functions which return current upper and low 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)*).

Structured data types (set and tuple) are used to aggregate values of any type, i.e., their elements can be not only values of elementary types but also references to objects. A set is a disordered list of elements with the traditional operations over sets (union, intersection, substraction). 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.

Besides conventional operations, expressions with quantifiers (iterators) are defined for structured types:

forall x **in** *Domain* : $Q(x)$,
exist x **in** *Domain* : $Q(x)$.

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

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

sum x **in** *Domain* : $F(x)$,

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

The first iterator (**sum**) calculates the sum of expression $F(x)$ for all elements x of *Domain*. The second iterator (**prod**) calculates the production in the similar way.

New data types can be derived from all these types. Every elementary data type T_0 can be specialized to a new more refined type T. In this case the value domain of the type T must be only a subset of the value domain T_0:

$$dom(T) \subset dom(T_0),$$

where $dom(T)$ is the value domain of the type T and $dom(T_0)$ is the value domain of the type T_0. For example, the construction

$$integer(0..100)$$

defines the new type which have the same operations as the integer type but its values are limited by the range from 0 to 100.

A structured data type can be also derived from another one. One structured data type T (tuple or set) is derived from another structured data type T_0 if the type E of elements of T is derived from the type E_0 of elements T_0, i.e.,

$$dom(E) \subset dom(E_0) \rightarrow dom(T) \subset dom(T_0).$$

For semantic data types, it is allowed to define classes of objects and binary relations and derive any of them from the other ones as it will be described below.

There are also special types *any* and *nil*. The value of *any* can be any value of the elementary, structured or semantic types. This type can be used in all cases when the actual type of the value is unknown. Actual type of value will be determined at run time. Type *nil* has only one value which is denoted by *'?'* symbol. It means the absolute indefiniteness of the domain for any type and may be used instead of any its value. So type *any* can be considered as the most base type for all other types, whereas type *nil* is the most derived of them. The value *'?'* may be also used for any data type to indicate that "no value" is actually presented.

The use of the hierarchy of data types provides one with flexible and powerful means for static and dynamic control of data types. Thus, using only most specialized data types, the user can produce more reliable and efficient code. On the other hand, if the type *any* is mostly used in the program, the code produced will be more universal and more expensive because of runtime type control.

3.2. The means for describing classes of objects and relations

An object class description generally has the following structure:

```
class NAME ( Superclass1, ..., SuperclassN )
  <description of slots>
constraints
  <logical expressions>
end;
```

A class definition specifies a set of slots, their possible values, and constraints restricting them (*SD-network*). If necessary, it includes the names of the superclasses whose properties this class inherits.

The description of a slot defines its type and possibly its default value. The type of the slot's values can be any standard type or its subset given either by enumerating all possible values or by indicating their ranges.

If a subclass inherits from several superclasses containing slots with the same name, then these slots are "glued" into a single slot. The types T_i of these slots specified in the definition of superclasses must be the same or at least they must be derived from the same base type T_0 and have a nonempty common subset of values, i.e.,

$$\cap dom(T_i) \neq \emptyset.$$

In this case the slot in the subclass will have type T derived from T_0 and

$$dom(T) = \cap dom(T_i).$$

In the definition of a subclass we can redefine the type of an inherited slot. The new type T of the slot must refine the old type T_0, i.e.,

$$dom(T) \subset dom(T_0).$$

Each constraint in an objects class definition is a logical expression (sometimes labeled) which binds object slots. Use of labels of constraints makes it possible to redefine or delete this constraint through the process of inheritance. To redefine (or delete) a constraint, one introduces a new (an empty) constraint with the same label when the subclass is defined.

Notice that some values of object slots can be references to other objects which in turn can contain references to other objects, etc. Using such slots in the constraints, one can bind values of different objects and thus construct an integrated SD-network.

To define a binary relation, one must specify the types of arguments of a relation, set of constraints (similar to that of a class of objects), as well as mathematical properties, e.g., reflexivity, symmetry, transitivity, antireflexivity, etc.

```
relation NAME ( ARG1: type1; ARG2 : type2 )
  Property1, Property2, Property3, ... ;
constraints
  <logical expressions>;
end;
```

The mathematical properties defined within the description of a relation influence essentially the behavior of the semantic network when new element of the relation is added. Thus, for example, if a relation R is specified to be symmetric, once an element $R(A, B)$ is inserted into R, the element $R(B, A)$ is also automatically inserted into R. The mathematical properties enable the logical inference of new elements of semantic relations.

3.3. Operations defined over semantic network

To support operating with a semantic network, the language includes operations for creating objects and elements of binary relations (**new**), editing them (**edit**) and deleting them from the network (**delete**). It is also possible to save and restore the content of the semantic network at any time.

The **new** operation creates objects and elements of binary relations. When an object is created, its slots are filled with the values indicated in the operation **new**. The slots whose values are not specified will be filled with default values.

If there is no default value and type of the slot values is elementary, then the slot will be given a subdefinite value which is the entire set (domain) of admissible values. If type of the slot values is not elementary (i.e., it is a tuple, set, or object), then the slot will be filled with the completely indefinite value *'?'*.

Only after all slots are filled with values, the object is inserted into the semantic network. If the description of the object's class includes constraints, then the object's SD-network added to the global SD-network.

The operation **edit** modifies the values of the slots of already existing objects.

After objects or relations are deleted from the network by the operation **delete**, the semantic and subdefinite functional networks are corrected. In the semantic network, all references to the names of the deleted objects are replaced by the completely indefinite value *'?'*. At the same time, all constraints related to the slots of the deleted objects and relations are removed from the global SD-network.

Note that an important feature of the language is that creation, modification, or deletion of any object or element of a binary relation leads to the immediate activation and execution of the global SD-network until its stability is achieved.

3.4. Production rules

The logical inference and data processing are defined by means of production rules. The production rule in the language has the traditional form:

$$COND => ACT,$$

where $COND$ is the condition which is necessary for the application of the rule, and ACT is the actions executed if the condition is satisfied. Sign $' =>'$ is used to separate the left-hand part of rule ($COND$) from the right-hand part of it (ACT).

The left-hand part of the rule contains a pattern that is a list of objects and relations with the given values of the slots. In addition to concrete values, the pattern can also contain local variables. These variables get their values when the pattern is matched with the semantic network. Along with the pattern, the left-hand part can include a local condition consisting of a logical expression and a so called negative context. Both the logical expression and the negative context specify additional constraints for the values of the local variables. The logical expression can include, in particular, local variables and built-in predicates and functions. The negative context is an extra pattern (or a list of patterns) having at least one common variable with the above-mentioned pattern which is considered as a main one. The condition of the rule is satisfied only if none of the patterns from the negative context is matched with the semantic network.

The main pattern must be preceded by a quantifier (**forall**, **exists** or **for**) which determines the way of the rule application. Thus, a rule with the **forall** quantifier is applied for each network fragment which matches the main pattern and satisfies the local condition. On the other hand, a rule with the **exists** quantifier is applied only for the first such a fragment found. A rule with the **for** quantifier is applied only if all the fragments which match the pattern satisfy the local condition.

The application of a rule is defined as an execution of operations which are indicated in the right-hand part of the rule (ACT). This part contains the above mentioned operations for manipulating objects and relations of the semantic network, as well as other operations and functions permitted by the language.

Finally, the right-hand part of a rule can include certain optional operations which are executed not earlier than the rule itself is completely applied. This facility is useful, for example, to activate other rules (see below).

3.5. Control facilities for activation of production rules

An important feature of the language is the availability of means for two-level dynamic control of the activation of production rules. These facilities ensure high flexibility of controlling knowledge (or data) processing. They allow one to structure the set of production rules defined in the application, and then activate some subset of them depending on the situation. The language provides flexible two-level facilities for the production rules activation. The first level corresponds to the conditions of the rules themselves. The second level is supported by the statements **call** and **activate** which invoke groups of production rules as well as a number of operations and built-in functions.

To make a production rule potentially applicable, one must activate this rule, i.e., to put it into the *Group of Active Rules* (*GAR*). The **call** and **activate** statements replace *GAR* by the group which is indicated as their parameter. In contrast to **activate** which just change *GAR*, **call** statement assumes that when the execution of the newly activated group is completed, the previous *GAR* is restored and the control is returned back to the point of the **call**.

The group of active rules can be not only replaced but also just modified. For this purpose, the language contains *GetActive()* function which returns the contents of the current *GAR*. Combining this function with the operations of concatenation and subtraction of groups, one can produce the modified *GAR* which then must be activated by the **activate** or **call** statements. This enables us to add rules and groups to *GAR* or delete them from it.

The predicates (logical functions) *Applied()* and *Deadlock()* also serve for the control of activation. *Applied()* detects whether at least one rule from the current *GAR* has been already applied. *Deadlock()* returns *true* only if the process of application of the activated rules is completed.

These predicates can be used within conditions of production rules. By definition, the rules which include the deadlock predicate are always executed only when all other rules of this group have been already executed. This means that deadlock-containing rules can be used to organize the transfer of control from one group to another.

3.6. The mechanism of alternatives

The *mechanism of alternatives* is a powerful tool supporting inference and data processing under the condition that both objects and relations between them can be imprecise. This mechanism is implemented in the language by the *statement of alternatives* which allows us to define and execute several alternatives:

```
try Alternative1
or  Alternative2
...
else DefaultActions
end
```

If the processing of an alternative results in inconsistency, we go to the next one. If the processing finishes successfully, no other alternative is concerned. If all tried alternatives fail, the actions placed after **else** are executed.

Any sequence of valid operations and statements may serve as an alternative. The execution of the current alternative is successful if it does not lead to a contradiction. Otherwise, roll-back is performed followed by an attempt to process the next alternative.

The contradiction can occur in the following cases:

1) if the SD-network of some object is inconsistent (in the case when the value of some of its slots do not satisfy given constraints);
2) if the semantic network is inconsistent (i.e., the properties of some binary relation are not satisfied);
3) if the statement **fail** of the explicit generation of a failure (contradiction) is used.

Thus, the statement of alternatives, on the one hand, allows us to generate hypotheses and reject them if they lead to contradiction. With its help, for instance, we can find various choices of the exact values for the slots of a subdefinite object. 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.

Finally, due to the availability of the statement **fail**, we can generate contradictions (failures) explicitly. For instance, a knowledge engineer can use **fail** to force testing of all alternatives in order to obtain all versions of the solution. In addition, this statement can be included in the production rules that monitor the semantic network for appearance of data that the knowledge engineer believes to be contradictory.

4. Example of use of the language

As an example, we consider the use of the language for developing a system supporting flexible planning of production and research under the conditions of an incomplete and imprecise data on time intervals of execution of tasks comprising the plan.

A *task* can be any activity that is continuously performed during some time interval. The system provides facilities to describe plan items, their structure, resources used and sequence of their execution.

Consider the main notions of the system.

The execution of a task is related to the consumption of *resources.* These can be people (person-months), machines, etc. A resource is characterized by its name and the cost per day in some units. To represent this notion, we use the class *RESOURCE* having two slots, *Name* and *Price*:

```
class RESOURCE
  Name: string;
  Price: integer(0..32000);
end;
```

The slot *Price* can have a value from 0 to 32000.

The tasks can describe the planned activity with the varying degree of detail, i.e., represent both the entire activity and its parts. A task can include several subtasks; at the same time, it can be a subtask of another task. In this hierarchy a plan is the top-level task.

Thus, the system has two types of tasks, simple and compound. These notions are represented by two main classes: *SIMPLE_TASK* and *COMPOUND_TASK*. Both classes are based on an auxiliary class *TASK* which includes the characteristics common for both types of tasks:

```
class TASK
  Name: string;
  Start: integer(0..2000);
  Finish: integer(0..2000);
  Duration: integer(0..1000);
  Price: integer(0..100000);
  State: atom( NotStarted, Started, Finished ):= NotStarted;
constraints
  C1: Finish > Start;
  C2: Duration = Finish - Start;
  C3: IsPrecise( Start ) -> State = Started;
  C4: IsPrecise( Start ) and IsPrecise( Finish )
                          -> State = Finished;
end;
```

The slot *Name* serves to name the task.

The interval of execution of a task, which is its basic characteristic, is determined by the *Start* date, the *Finish* date and *Duration*. Note that these characteristics may be defined imprecisely, i.e., they can have subdefinite values.

The parameter *State* (state of a task) reflects the current phase of the execution of a task and can have the following values: *Started*, *NotStarted* and *Finished*. The value of this parameter is set by the user when the operational information on the realization of the plan arrives. The default value is *NotStarted*.

The parameter *Price* indicates an estimate for the total cost of performing a task (in some units). Its value depends on the organizational structure of the task and the cost of the resources it consumes.

The class *TASK* includes four constraints. Two constraints *C1* and *C2* define the relations among the start, end, and duration of a task and allow us to refine one part of these parameters which have subdefinite values using the other part of these parameters with values defined more precisely. The constraints *C3* and *C4* serve to calculate the current state of a task.

The classes *SIMPLE_TASK* and *COMPOUND_TASK* inherit all properties of the auxiliary class *TASK*:

```
class SIMPLE_TASK (TASK)
  Resource: set of RESOURCE;
  Priority: atom( low, high ):= low;
constraints
  C5: Price = sum $r in Resource : $r.Price * Duration;
end;

class COMPOUND_TASK (TASK)
  Structure : set of TASK;
constraints
  C6 : forall $t in Structure :
         $t.Start >= Start and $t.Finish <= Finish;
  C7 : forall $t in Structure :
         $t.Duration <= Duration;
  C8 : Price = sum $t in Structure : $t.Price;
end;
```

Note that the names with the *'$'* sign in the all examples denote variables.

The class *SIMPLE_TASK* serving to represent simple tasks includes additional slots to denote the resources used by a task (*Resource*) and its priority (*Priority*) for the use of resources. The default value of the slot *Priority* is *low*. A task with a high priority of the use of some resources cannot share them simultaneously with other tasks.

The class *SIMPLE_TASK* contains also the constraint *C5* which allows us to calculate the cost of a task from the price of the resources it uses.

The class *COMPOUND_TASK* contains an additional slot *Structure* enumerating all subtasks of a task. The constraint *C6* sets the time limits for subtasks of the task. The constraints *C7* and *C8* relate the duration and price of a task, respectively, with the duration and price of its subtasks.

The user can set the order in which the tasks of a plan are carried out. To this end, the system has binary transitive relations *AFTER* and *BEFORE*:

```
relation AFTER(Task1, Task2: TASK)
  transitive;
constraints
  C9: Task1.Start <= Task2.Finish;
end;

relation BEFORE(Task1, Task2: TASK)
  transitive;
constraints
  C10: Task1.Finish <= Task2.Start;
end;
```

The constraints *C9* and *C10* included in these relations can refine values of start and finishe of the tasks which are ordered. The constraint *C9* defines the dependency between the start of the first task (*Task1*) and the finish of the second task (*Task2*). Constraint *C10* does the same for the finish of the first task and the start of the second task.

The process of operation with a plan in the system is divided into three stages: entering initial data, composing the plan and monitoring it.

At the first stage the user creates the list of planned tasks, list of resources, organizational structure of the plan, sequence of the executions of tasks, and distribution of resources over tasks. The user interface of the system is used to enter this data.

At the second stage, the precise plan is created. Due to the global SD-network, consisting of the constraints of the objects of the classes *RESOURCE*, *COMPOUND_TASK*, *SIMPLE_TASKS* and the elements of the relations *AFTER* and *BEFORE*, the system automatically refines the execution times of tasks. The refined times can be edited by the user again, and can then be used as initial data for further recomputations.

Thus, the process of the plan creation consists of the consecutive refinement of some temporary intervals, introduction of additional links between tasks and resources, and automatic refinement of the plan. Incorrect user actions (from the standpoint of consistency of the plan) can result in inconsistent data. The system will find them and employ the mechanism of alternatives to return the user to the preceding step, offering him to choose a different refinement of times. Thus, the user can play different variants of the plan.

In the same stage the system can find conflicts between tasks using common resources. In this case it will offer the user to resolve the conflict either by redistributing the resources or by correcting the execution times of tasks.

Note that whereas the process of automatic plan refinement is ensured by the global SD-network all other processes are supported by production rules. In particular, the rule *HighPriority* serves to discover conflicts between tasks using common resources:

```
rule HighPriority
forall
  SIMPLE_TASK (Name: $t1, Start: $s1, Finish: $f1,
               Resource: $r1, Priority: high),
  SIMPLE_TASK (Name: $t2, Start: $s2, Finish: $f2,
               Resource: $r2, Priority: $p2)
when
  not( $f2 < $s1 or $f1 < $s2 ) and #($r1*$r2) != 0
=>
  message("Task ", $t1, " with high priority cannot share",
          "resources ", $r1*$r2, " with the task ", $t2,
          "It is necessary to redistribute the resources!");
end;
```

This rule is activated when the intersection (#) of execution intervals of two tasks is not empty, i.e., the condition ($\$f2 < \$s1$ or $\$f1 < \$s2$) is not met, and the two tasks use common resources (i.e., intersection of the sets $\$r1$ and $\$r2$ is not empty).

The third stage of working with the system (monitoring) refers to the time when the plan is executed. Its main function is to monitor the current state of tasks. After the user enters current data (which tasks are started or finished and when) the system corrects the times of start/finish for the tasks that have not been started or finished, as well as finds the tasks that are late for start or finish and issues messages about them.

For example, the rules *AttentionStart* and *AttentionFinish* find the tasks that are behind schedule with respect to their start and finish dates, respectively:

```
rule AttentionStart
forall
  COMPOUND_TASK(Name: $t, Start: $s, State: NotStarted)
when
  $s < $current_date
=>
```

```
  message("Task", $t, " is behind the starting date!");
end;

rule AttentionFinish
forall
  COMPOUND_TASK(Name: $t, State: Started, Start: $s,
                                         Duration: $d)
when
  $s + $d < $current_date
=>
  message("Task ", $t, " is behind the finishing date!");
end;
```

Note that in both rules *$current_date* is a global constant denoting the current date.

Thus, from the above example we see that the language suggested allows us to define in a clear and natural manner the main notions of a planning system and to set the rules supporting the main phases of its use.

Conclusion

In contrast to other knowledge representation languages which use a limited set of means and methods, the language described in this paper is based on the integration of various complementary tools of knowledge representation and processing. The use of the object-oriented approach makes it possible to unify various means and methods, such as frames, semantic networks, production rules, subdefinite computational models, and constraint programming technique, in the framework of one language.

This language provides a user with powerful and flexible means for the development of an intellectual system. It makes it possible to form a knowledge base in a rather natural, high-level manner. Besides providing additional functional and descriptive capabilities, it significantly increases the productivity of the knowledge engineer designing an application. After defining the necessary system of notions in this language, we can manipulate objects in terms of production rules, without worrying about their internal semantics.

An important feature of the language is the possibility of operating with objects which can have slots with imprecisely defined values.

Another important feature of the language is that it allows one to bind to any object a set of constraints defined on the values of the object's slots. These constraints not only serve for control of consistency and correctness of the slots' values but also allow their values to be automatically refined. With the help of the mechanism of alternatives we can find various choices of the exact values for the object's slots.

Due to the object-oriented approach and the use of the particular class of constraints which allows one to simulate the simplest productions, it is possible to localize inside the object not only the computations but also the logical inference.

Because the global SD-network can be modified during the process of computation, this language can also be used to create intelligent systems which are capable of taking into account the dynamics of processes.

Thus, the knowledge representation language described in this paper can be used to create a broad class of intelligent systems which require a combination of logical inferences and subdefinite computations and enable us to deal with dynamic processes.

Acknowledgements. The authors wish to thank A.V. Zamulin whose valuable remarks and suggestions have significantly improved the paper.

References

[1] A. Deliyanni, R.A. Kowalski, *Logic and semantic networks*, Comm. ACM, 1979, **22**, No. 3, 184–192.

[2] R. Davis, J. King, *An overview of production systems*, Computer Science Department Stanford University, Report STAN–CS–75–524, 1975.

[3] R.B. Roberts, I.P. Goldstein, *The FRL manual*, Cambridge, 1977, (Rep./MIT; A.I. Memo N 409), 1446–1452.

[4] D.G. Bobrow, T. Winograd, *An overview of KRL, a knowledge representation lanuage*, Cognitive Science, 1977, **1**, No. 1, 3–46.

[5] A. Goldberg, D. Robson, *Smalltalk-80: the language and its implementation*, Reading a.o.: Addison Wesley, 1983.

[6] *Object Oriented Development*. Ed. by D.Tsichritzis, Geneve, Centre Universitaire d'Informatique, Universite de Geneve, 1989.

[7] B. Mayoh, *Constraint Programming and Artificial Intelligence*, Constraint Programming, Springer–Verlag, 1993, NATO ASI Series F: Computer and Systems Sciences, **131**, 17–50.

[8] E. Hyvonen, S. De Pascale, A. Lehtola, *Interval Constraint Programming in C++*, Constraint Programming, NATO ASI Series F: Computer and Systems Sciences, Springer–Verlag, 1993, **131**, 350–366.

[9] A.S. Narin'yani, *Sub-definiteness and basic means of knowledge representation*, Computers and Artificial Intelligence, 1983, **2**, No. 5, 443–452.

[10] V. Telerman, D. Ushakov, *Data types in subdefinite models*, Jacques Calmet and others (eds.), Artificial Intelligence and Symbolic Mathematical Computation, Lecture Notes in Computer Science, **1138**, Springer, 1996, 305–319.