Joint NCC & IIS Bull., Comp. Science, 5 (1996), 69–82
© 1996 NCC Publisher

Agents as constraint objects

I.E. Shvetsov, T.V. Nesterenko

In the paper a multi-agent technology based on integration of the object-oriented approach and constraint programming is proposed. The key notion of this technology is an "active object" that has three specific features. First, an active object has the ability to change its state based on the analysis of the states of other "visible" objects. Second, the interaction of active objects is asynchronous and controlled by events rather than messages. Third, the behavior of an active object, that is, its transitions from one state to another, is described by constraints rather than methods. In addition, we propose a new solution to the problem of integrating imperative programming with constraint programming. In the technology of active objects, it is the constraint paradigm which plays the leading role. It is extended by the capability of simulating sequential processes, using the mechanism of conditional constraints and the semantics of the tick-by-tick computations.

Introduction

The agent paradigm is very popular in modern programming, especially in the field of artificial intelligence [1]. It considers the software system as a collection of independently specified entities that can communicate with each other and the external world. In contrast to the ordinary objects, agents are more intelligent and more independent.

In this paper we propose a multi-agent technology based on integration of the object-oriented approach and constraint programming. The key notion of this technology is an "active object" that has three specific features. First, an active object has the ability to change its state based on the analysis of the states of other "visible" objects. Second, the interaction of active objects is asynchronous and controlled by events rather than messages. Third, the behavior of an active object, that is, its transitions from one state to another, is described by constraints rather than methods, whereas methods are used only to implement the basic machine-dependent functions, for instance, output of the graphical image of the object to the screen, or the object's interaction with external devices (mouse, keyboard, sensors, etc.).

The principal advantage of the constraint-based approach is declarative programming style. However, usually this approach is used to solve a rather narrow class of problems which are reduced to the constraint satisfaction problem (CSP). The solution to this problem consists in the transition from the domains of values of parameters linked by constraints to precise values or subsets of the initial domains which satisfy all of the constraints simultaneously. Thus, by solving the CSP we can find some state of the object with given constraints for its parameters.

At the same time, dynamic change of the states of an object, related to changing the values of its parameters, cannot be described in the framework of the ordinary constraint paradigm. Consequently, in practical problems constraints are often used in a combination with other, more universal programming technologies, for example, object-oriented [2], logic [3] and concurrent [4] ones.

In most cases of integration with other technologies, constraints are regarded as an additional tool enhancing the user-oriented and computational capabilities of the base mechanism. In the technology of active objects (TAO), however, it is the constraint paradigm which plays the leading role.

The main contribution of the present paper is the application of the constraint paradigm to specification of the structure and behavior of multi-agent systems. In addition, TAO proposes a new solution to the problem of integrating imperative programming with constraint programming. TAO essentially uses only one method of computation control — constraint propagation. It is extended by the capability of simulating sequential processes, but the means for imperative flow control are not explicitly provided at the user level.

It was decided to base TAO on the local constraint propagation technique suggested in so called subdefinite models (SD-models) [5,6]. We consider SD-models to be one of the most universal, flexible, and efficient approaches existing within constraint programming. SD-models support joint computations on finite domains and numerical intervals and allow one to solve problems described by arbitrary, in particular, cyclic constraint systems, which may include nonlinear dependencies [7]. An important feature of SD-models, which had an essential influence on their choice, is the mechanism of conditional constraints. This mechanism makes SD-models close to concurrent constraint programming.

The paper is structured as follows. First, we consider the relationship between TAO and other constraint programming technologies. Next, we introduce the notion of an active object, describe its structure and computational semantics. We distinguish objects of three activity levels which are characterized, respectively, by a dependence on the current events ("event-driven"), on the previous state of the environment ("state-driven"), and on time ("time-driven").

In addition, the paper informally describes and illustrates the main components of the specification language for active objects. We consider some important aspects of the functioning of active objects, including graphical visualization and interaction with the outer environment.

1. TAO and other constraint programming technologies

Since TAO is an extension of SD-models, for better understanding we will characterize briefly their main properties and capabilities. First of all, note that SD-models are one example of the constraint approaches, which were classified by E. Davis as propagation with interval labels [8]. This means, in particular, that SD-models work not only with exact data, but also with domains of values of variables, including numerical intervals.

The principal distinction between SD-models and other methods of constraint programming is that the domains of values of the variables are treated as subdefinite data types (SD-types) over which the corresponding multisort algebra of operations is constructed.

An SD-type extends a certain ordinary data type which can be numerical, symbolic, or logical. The domain of its values is constructed as the set of all subsets of the domain of values of the original type. Thus, an exact value is a special case of an SD-value corresponding to a one-element set. A completely indefinite value corresponds to the set coinciding with the entire domain of values for the given type.

On the surface an SD-model looks like a system of constraints (equations, inequalities, logical formulas, relations, etc.) linking data of various types — integer and real numbers, Booleans and sets. Computations in SD-models are an asynchronous data-driven process of performing operations on the domains of values of the variables. The algorithm is close to tolerance propagation proposed by E. Hyvonen [9] for computations on intervals of real numbers. Unlike the last one, however, our algorithm can be applied for computations over data of different types.

Consider a small example demonstrating the use of SD-models to solve a system of numerical constraints.

$$\begin{aligned}
&x^2 + 6 * x = y - 2^k;\\
&k * x + 7.7 * y = 2.4;\\
&(k-1)^2 < 10;\\
&(x < 2.5 * y) \rightarrow (k * y \leq 3) \& (k > y + 1);
\end{aligned}$$

Here x and y are real numbers and k is an integer. In this example the fourth formula is a conditional constraint. Its semantics consists in the following: the constraints on the right-hand side of the implication will be activated only after the constraint on the left-hand side becomes true. In fact, the left- and right-hand sides of the implication correspond to the "ask" and "tell" expressions in the terminology of concurrent constraint programming.

Before the computations, to all the variables we assign intervals with very wide, almost infinite boundaries. The application of the constraint propagation algorithm for the example in question yields the following result:

$$k = [0,4]; \; x = [-6.4132, 0, 4406]; \; y = [0.0828, 3, 6433].$$

Like other algorithms based on local inference, our algorithm generally ensures only local consistency of the solution. This means that the resulting sets of values may contain elements which satisfy only some part of all constraints. To eliminate these "superfluous" values, and to find an exact solution in the set of all feasible ones, SD-models also use an exhaustive search with backtracking (for discrete sets) or bisection (for intervals). Applying this method in the last example, we find two exact solutions:

1) $k = 2$; $x = -0.6586$; $y = 0.4826$;
2) $k = 3$; $x = -1.6078$; $y = 0.9381$.

The above capabilities of SD-models are used in TAO to define relationships between internal slots of active objects. However, they are not sufficient for implementation of multi-agent systems. We tried to find a minimal extension of the computational semantics of SD-models which would allow us to solve the following three main problems staying within the framework of constraint programming. First, we must provide the reaction of active objects to the signals coming in from the outside world. Second, we should organize the interaction of active objects. Third, we must support the main capabilities of imperative programming related to specifying a sequence of actions, as well as recomputing the states of objects on the basis of their previous values.

To solve these problems, TAO includes the notion of an abstract clock. Each tick of the clock represents a computational process, as a result of which all objects get new values which possibly coincide with the old ones. That is, we associate a complete set of objects' values with each tick.

The approaches close to TAO are timed concurrent constraint programming [10] and the Kaleidoscope language [11]. However, as far as we know, the problem of implementing multi-agent systems was not considered in the framework of these approaches. The first approach is mostly concentrated on the development of reactive real-time systems, making use of an appropriate extension of constraint programming language. The main feature of the Kaleidoscope language is the integration of an object-oriented language with constraints.

In our view, the joint use of the imperative programming constructions and constraints, in particular, the availability of two kinds of assignment (destructive and refine) makes the semantics of the programming language too complex. TAO does not have the operators of ordinary imperative programming and uses only one kind of assignment (refine). At the same time, the software technology proposed in the present paper supports the main features of imperative languages using the mechanism of conditional constraints and the semantics of the tick-by-tick computations.

2. Principal properties of active objects

Every software system is built in TAO as a collection of interacting agents of a certain kind which we will call active objects. Numerous approaches may claim the rights to the term "active object," but we propose an original interpretation for it.

An active object is an autonomous entity which is capable of changing its own state independently and asynchronously, using the information about the states of some external objects which are visible from this entity. Aside from its name, an elementary active object includes three components:

1. a set of references to external objects which are visible from this one;
2. a set of internal objects (slots) which are assumed visible from outside;
3. a function mapping the values of external objects into the values of internal ones.

No active object can change the values of external objects. Computation of all visible external objects has been completed, and this is the fact which is the necessary and sufficient condition for its activation.

TAO assumes that as a rule the function computing the values of the slots of an object is specified as a system of constraints. However, this function may as well be an arbitrary program written in some base language, for instance, C++.

In TAO, an important role is played by active objects whose set of external objects may be empty. The modification of the states of these objects is controlled by information sources lying outside the environment of active objects. Examples of such sources are various control devices (mouse, keyboard, pen, etc.), sensors, timer, etc. The connection between active objects and external devices is implemented via functions written in the base programming language. Henceforth, we will use the generalized term "sensors" for such active objects.

Computations in the environment of active objects are organized as follows. First, the sensors which do not depend on other active objects are evaluated in arbitrary order or in parallel. Next, the objects that depend on sensors only are evaluated (again, in arbitrary order). Then those objects are evaluated which depend only on the values computed earlier, and so on, until the states of all objects are computed. It is natural to demand that the dependencies between objects constitute an acyclic graph. The sequence of computations for an example is schematically shown in Figure 1.

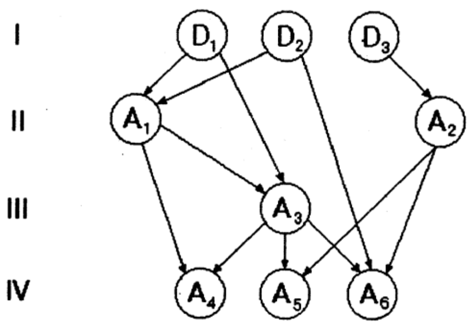

Figure 1

The process after which each active object assumes a new state is called a computation step. An important point is that the state of each object is rigidly linked to the number of the computation step. At the beginning of each step, the values of all objects are set to an indefinite value. The step completes when each object gets some concrete value. Once the value of an object is found, it cannot be changed at the same step.

It is a quite ordinary situation when a new state of an active object coincides with its old state. If, for some object, the values of its external objects did not change in the current step, its new state can be obtained without computations at all, by simple copying of the previous state. Careful tracking of such situations is important if we want to make the computations in the environment of active objects more efficient.

The totality of the states of objects which were computed in some step is called the state of the environment. On the whole, the process of computations in an active environment is a sequence of computation steps. The process stops when either a prescribed number of steps is completed or a special instruction (e.g., STOP) is executed inside an active object.

If inconsistency of data and constraints in a computation step is detected, then all the values computed in this step are ignored. Formally one may think that the step was "idle," and all the objects switched to their new states without changing their values. A special error signal is produced, which may affect the computations in the next step.

Note that the control of all computations in the environment of active objects is based on a single approach which can be characterized as an asynchronous data-driven process. It is used both in organizing the interaction of active objects as well as in the constraint satisfaction algorithm used to compute the states of the objects. There are some distinctions between the two processes, but the principle of control remains the same.

3. Three levels of active objects

We distinguish three groups of objects which differ in their activity level. The first group (the first level) comprises those objects which react only to current events, without accounting for the previous state of the environment. Such objects are, for instance, sensors or those active objects which depend only on the values computed within the current computation step. Active objects of only this kind are necessary for the implementation of systems which are controlled exclusively by sensors. For example, they may be used in the construction of relatively simple interactive graphical interfaces.

Objects of the next group corresponding to the second activity level, take into account the information about the previous state of the environment. In practice this means that the statements of constraints may refer to the values of objects which were computed in the preceding step. In particular, this enables one to express a situation when the new state of an object depends on the previous state of the same object. We can write, for instance, a constraint of the form $A = A' + 1$, where A' denotes the previous value of the object A. Since the previous state of the environment is assumed known at the beginning of the step, and cannot change anymore, those active objects which depend only on this previous state will be evaluated first, independently of each other.

Active objects of the second level extend the class of solvable problems and provide means for modeling dynamics and evolving processes. Furthermore, it can be shown that the second-level objects combined with the mechanism of conditional constraints, which is supported by SD-models, allow us to specify most algorithms expressible by means of the ordinary imperative programming. It is important to note that the principle of asynchronous control of the computations is maintained.

The third group is constituted by the active objects whose behavior depends on time. It is assumed that the computation steps are executed instantly, with equal intervals of time between them. These intervals may vary in different applications. The introduction of time allows us to provide tools for treating the numbers of steps as the values of some special (temporal) data types. For these data types, we define operations and relations which make it possible to express such notions as, for example, "earlier," "later," "every five minutes," "for two hours," etc. Thus, the validity of the constraints which determine the behavior of objects in this group can depend on whether the number of the current step satisfies given temporal constraints. Sensors can also be specified as the objects of the third group — they may produce new values only at prescribed times rather than at every step.

Since active objects of each of the three levels are connected with a certain specific version of data-driven control, we introduce the following terminology. We say that the first level consists of objects depending on events, or "event-driven," the second level contains objects depending on the state of the environment, or "state-driven," and the third level contains objects depending on time, or "time-driven" objects. The original term "data-driven" will be applied only to dependences between internal slots of an active object. The relationship between various control types used in TAO is shown in Figure 2.

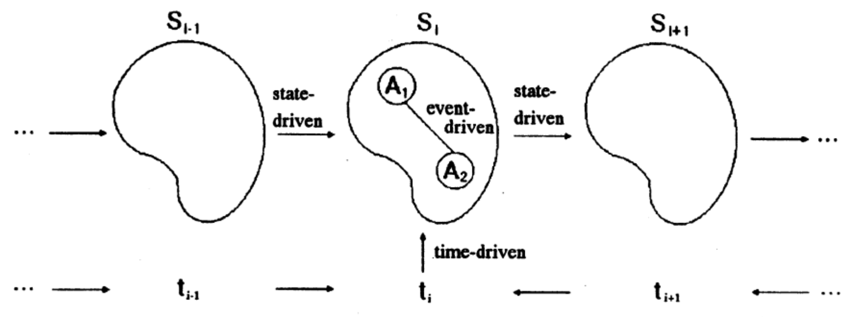

Figure 2

Although control in the environment of active objects is asynchronous, there are at least two possibilities for setting the order of computations indirectly. First, within one step the computation of one object may be made to depend on another object (or other objects). That is, it will be computed only when all the objects it depends on will get their new values. Second, the behavior of an active object may depend on the value it (or some other objects) had in the previous step.

4. Programming in the environment of active objects

In this section, we consider the language of active objects. The description of a typical active object has the following form:

```
object <name> : <prototype>;
      out <set of external objects>;
      in  <set of slots>;
model <constraints>
end.
```

Consider a concrete example. Suppose that an active object is a square with sides parallel to the coordinate axes, which autonomously changes its size depending on the signals arriving from the mouse. Suppose also that in each step it moves two positions along the X axis until it reaches a fixed wall. Then its description is as follows:

Example 1.

```
object act_square : square;
     out   M : mouse; W : wall;
     in    area : number;
     model
           M.right = true -> edge = edge' + 1;
           M.left  = true -> edge = edge' - 1;
           (M.right & M.left) = false -> edge = edge';
           area = edge^2;
           area <= 36;
           x = x' + 2;
           x + edge > W.x -> STOP;
end.
```

Here the object square is the prototype of an active square (act_square), from which the latter inherits the slots "edge" (the size of an edge) and "x" (the coordinate of the top left angle). The object "square" itself is not active, since it does not have external connections. Therefore, it can be used only as a constant object (on the top level) or as a component or prototype of another object. Note that TAO prefers the object-oriented model of the "prototype-instance" kind to the more traditional one, "class-instance." The justification of this choice is beyond the scope of the present paper.

The active object "act_square" has two external objects; one of them (M) is the sensor connected to an external device of the type "mouse," and the other (W) is the constant object, a vertical line simulating a wall. The identifiers M and W are in fact pointers to the values of external objects whose names must be concretized by the moment when this object is to be used. After the colon, we specify the name of the prototype which must belong to the list of the object's ancestors substituted for the corresponding reference.

The object "act_square" specifies one more slot in addition to those inherited from the prototype — "area" which denotes the area of the square and is used to limit its size. Unlike M and W, the identifier "area" is a specific instance of a numerical object rather than a reference.

The implications contained in the model must be treated as the conditional constraints which perform a dynamic modification of the set of constraints. In this case, the dynamics means, firstly, that the set of unconditionally true constraints is extended by those constraints which were found to be true in the current computation step, and, secondly, that the truth of a condition is established, in general, not immediately but only at a certain point of the process of constraint propagation.

The "Main" part of a program in the TAO language contains creating, initializing and linking active objects. For Example 1, this part has the following form:

```
main (Example1)
  initialization
        mouse;
     const W1: wall(x = 60, y = 0, l = 50);
        A1: act_square (edge = 1, x = 1, const y = 5);
  linking
        A1 (mouse, W1);
end.
```

When creating an object at the computation step 0, we can set the initial values of some of its slots. The slots and objects whose values do not change in the entire computation process are labeled as constants. The proper computations start at step 1. Undefined slots initially have default or completely indefinite values.

The "Linking" section specifies the relations between objects. A concrete name is assigned to each reference to a visible external object. Here we consider the simplest (static) case, when all objects and relations are initially defined and preserved until the end of the computations. The dynamic case, in which the creation/destruction of objects and the modification of relations between them are possible in any computation step, demands separate investigation.

Consider the interaction of active objects which depend on each other. Suppose that the wall in Example 1 is not fixed but moves towards the square. The process stops when the two objects meet. This leads to the following changes in the program. The description of a new active object "act_wall" is added; the object "act_square" becomes external for this object. The largest changes are made to the "Main" part of the program.

Example 2.

```
object act_wall : wall;
   out   R : act_square;
   model
         x = x' - 1;
         R.x > x -> STOP;
end;
main (Example2);
  initialization
```

```
        mouse;
        W1: act_wall(x = 60, const y = 0, l = 50);
        A1: act_square (edge = 1, x = 1, const y = 5);
  linking
        A1 (mouse, W1);
        W1 (A1');
end.
```

In this program, object $A1$ reacts to a change of the state of object $W1$ within the current computation step, whereas object $W1$ detects a change of the state of $A1$ only in the next step. That is, the dependence of the type $A \longleftrightarrow B$ is described in the language of active objects via two links $A \longleftarrow B$ and $B \longleftarrow A'$ or, conversely, $B \longleftarrow A$ and $A \longleftarrow B'$.

5. Specification of sensors and graphical objects

Sensors in TAO are objects which can change their state even without contact with other objects of the active environment. This means that inside the sensors there is a software component connected with the outside world. In fact, sensors play the role of event generators for the active environment.

At the same time, some aspects of the operation of sensors may depend on the states of other active objects. For example, the shape of the cursor may depend on the object it presently points to. As mentioned above, the activation of a sensor may depend on time. In general, a sensor is a complete active object which is extended with a program providing the communication with the outside world. For the program, the lists of its input and output parameters must be specified, as well as the condition for its activation. It is also necessary that all of its input parameters have concrete values at the time the program is activated.

Example 3.

```
object mouse;
   out  A : hor_line;
   in
        x, y  : integer;
        left, right : boolean;
        mp: mouse_picture;
   model
        y > A.y -> mp.form = 1;
        y <= A.y -> mp.form = 2;
   program mouse_prog;
           input  :
           output : x, y, left, right;
           cond   :   NOW > 10;
end.
```

In this example we describe a mouse which begins to work only after the tenth computation step and is capable of changing the shape of its cursor. NOW is a built-in global object whose value is the number of the current computation step. The shape of the mouse cursor changes depending on its position relative to a given horizontal line. To draw the cursor, the specification of the mouse includes the graphical object "mouse_picture."

Since TAO's minimal representation unit for any kind of information is an object, it is natural to introduce the notion of a graphical object. A graphical object, as any other one, may be active, have a built-in model of behavior, react to sensors, etc. The only distinction of a graphical object is that it is associated with a screen image.

All graphical objects specified by the user are eventually built with a limited set of basic objects, such as point, segment, arc, circle, rectangle, etc. The basic graphical objects have the same status in TAO as the other basic notions, for instance, number, string, or logical value. The only function of a basic graphical object is to draw an image of itself on the screen (the image depends on the specific values of its slots). Thus, it has only one associated method, image drawing, and does not contain any constraints.

The introduction of basic graphical objects makes it possible to localize a low-level, machine-dependent component related to drawing them. All other aspects of the behavior of graphical objects are described within TAO with the help of systems of constraints. For example, consider again the user-defined graphical object "square." Suppose that its prototype is the elementary graphical object "rectangle" with four slots: x, y (the coordinates of the top left corner), "wide" and "high." In this case, the description of the square may be as follows:

```
object square : rectangle;
   in  edge : number;
   model
       wide = high = edge;
       edge > 0;
end.
```

Note that the first constraint in this example performs two functions. On one hand, it states the obvious truth that the sides of a square must be equal. On the other hand, it relates the value of the new slot "edge" with the slots of the basic graphical object, that is, with the parameters of the corresponding drawing method.

For the convenience of working with graphical objects as data of a special kind, one usually defines specialized spatial relations and operations, for instance, inclusion, intersection, distance, to the right of, touches, etc. In TAO such notions are described as parameterized systems of constraints.

For example, the relation of inclusion for two squares may be described as follows:

```
relation inside (A, B : square);
    A.x > B.x;
    A.y < B.y;
    A.x + A.edge < B.x + B.edge;
    A.y - A.edge > B.y - B.edge;
end.
```

Here, as in the preceding examples, we consider the simplest case in which the square's sides are parallel to the coordinate axes, and the square lies completely in the positive quadrant.

The objects are redrawn after each computation step is completed. To this end, the system uses a built-in graphical processor whose main function is implementation of the drawing methods. The work of the graphical processor can be made more efficient, if we redraw not all objects in each step, but only those whose state changed, and take into account possible overlaps and intersections.

Conclusion

In the present paper we propose a technology for creating multi-agent systems which is based on integrating constraint programming with object-oriented programming. The technology makes it possible to describe naturally and laconically complex application systems which are organized as sets of active objects interacting in a dynamically changing environment. Examples of such applications are various complicated mechanisms (e.g., robots), manufacturing processes, transport simulations, or warfare.

At the same time, TAO has good chances of efficient implementation. Firstly, this is due to the efficiency of SD-models used as the basic computational mechanism. Secondly, TAO provides a multi-level asynchronous computation control process which is easily parallelized. Thus, the efficiency of this technology can be manifold improved by using computers with parallel architectures. At present, implementation of the first version of the TAO software is almost completed. It is designed for use in the development of interactive graphical interface. The basic computing engine of this system is the corresponding module of the UniCalc constraint solver [12].

We considered only some basic principles of TAO. A more detailed description of this technology can be found in [13]. Further studies are related to the development of time modeling features and means for dynamic modification of the environment of active objects. However, the most important goal at present is to perform convincing experiments on applying TAO to the solution of real-world problems.

References

[1] M. Wooldridge, N.R. Jennings, *Agent theories, architectures and languages: a survey*, ECAI-94 Workshop on Agent Theories, Architectures and Languages, Amsterdam, Netherlands, 1994, 1–39.

[2] J.-F. Puget, M. Leconte, *Beyond the glass box: constraints as objects*, Proc. of International Logic Programming Symposium, Portland, Oregon, 1995, 513–527.

[3] J. Jaffar, J-L. Lazzer, *Constraint logic programming*, Proc. of POPL-87, Munich, Germany, 1987, 111–119.

[4] V.A. Saraswat, *Concurrent Constraint Programming*, Logic Programming and Doctoral Dissertation Award Series, MIT Press, March 1993.

[5] A.S. Narin'yani, *Subdefiniteness in knowledge representation and processing*, Transactions of USSR Acad. of Sciences, Technical cybernetics, Moscow, No. 5, 1986, 3–28 (in Russian).

[6] I. Shvetsov, A. Semenov, V. Telerman, *Constraint programming based on subdefinite models and its applications*, Proc. of International Logic Programming Symposium, Workshop on Interval Constraints, Portland, Oregon, 1995, 15 p.

[7] A. Semenov, *Solving Integer/Real Nonlinear Equations by Constraint Propagation*, Technical report, Institute of Mathematical Modelling, Lyngby, Denmark, 1994, 22 p.

[8] E. Davis, *Constraint propagation with interval labels*, Artificial Intelligence, **32**, No. 3, 1987, 281–331.

[9] E. Hyvonen, *Constraint reasoning based on interval arithmetic: the tolerance propagation approach*, Artificial Intelligence, **58**, 1992, 71–112.

[10] V.A. Saraswat, R. Jagadeesan, V. Gupta, *Foundations of Timed Concurrent Constraint Programming*, Proceedings of LICS, 1994.

[11] B. N. Freeman-Benson, A. Borning, *Integrating constraints with an object-oriented language*, Proc. of the European Conference on Object-Oriented Programming, 1992, 268–286.

[12] A.B. Babichev, et al., *UniCalc, a novel approach to solving systems of algebraic equations*, Proc. of the International Conference on Numerical Analysis with Automatic Result Verifications, Lafayette, Louisiana, 1993.

[13] I. Shvetsov, *Basic principles of the technology of active objects*, Russian Research Institute of Artificial Intelligence, Preprint No. 3, Novosibirsk, 1995, 26 p. (in Russian).