UDC 004.41,004.51

Olexander Letichevsky, Olexander Letychevskyi, Vladimir Peschanenko, Igor Blynov, Dmitry Klionov

CONSTRAINT PROGRAMMING IN INSERTION MODELING SYSTEM

The paper relates to practical aspects of insertion modeling. Insertion modeling system is an environment for the development of insertion machines, used to represent insertion models of distributed systems. The architecture of insertion machines and insertion modeling system IMS is presented. Insertion machine for constraint programming is specified as an example, and as a starting point of 'verifiable programming' project.

Introduction

Insertion modeling is the approach to modeling complex distributed systems based on the theory of interaction of agents and environments [1–3]. Mathematical foundation of this theory was presented in [4]. During the last decade insertion modeling was applied to the verification of requirements for software systems [5–9]. First time the theory of interaction of agents and environments was proposed as an alternative to well known theories of interaction such as Milner's CCS [10] and pi-calculus [11], Hoare's CSP [12], Cardelli's mobile ambients [13] and so on. The idea of decomposition of a system to a composition of environment and agents inserted into this environment implicitly exists in all theories ofinteraction and for some special case it appears explicitly in the model of mobile ambients.

Another source of ideas for insertion modeling is the search of universal programming paradigms such as Gurevich's ASM [14], Hoare's unified theories of programming [15], rewriting logic of Meseguer [16]. These ideas were taken as a basis for the system of insertion programming [17] developed as the extension of algebraic programming system APS [18]. Now this system initiated the development of insertion modeling system IMS which started in Glushkov Institute of Cybernetics. The development of this system is based on the version of APS enhanced by the former student of the author V.Peschanenko. The first version of IMS and some simple examples of its use are available from [19].

To implement the insertion model in IMS one must develop insertion machine with easily extensible input language, the rules to compute insertion functions and a program of interpretation and analyzing of insertion models. The architecture, input languages and examples of insertion machines and insertion modeling system are considered in the paper.

1. The Architecture of Insertion Modeling System

Insertion modeling system is an environment for the development of insertion machines and performing experiments with them. The notion of insertion machine was first introduced in [17] and it was used as a tool for programming with some special class of insertion functions. Later this notion was extended for more wide area of applications, different levels of abstraction, and multilevel structures.

Insertion model of a system represent this system as a composition of environment and agents inserted into it. Contrariwise the whole system as an agent can be inserted into another environment. In this case we speak about internal and external environment of a system. Agents inserted into the internal environment of a system themselves can be environments with respect to their internal agents. In this case we speak about multilevel structure of agent or environment and about high level and low level environments.

As usually, insertion function is denoted as $E[u]$ were E is the state of environment and u is the state of an agent (agent in a given state). $E[u]$ is a new

environment state after insertion an agent u. So, the expression $E[u[v], F[x, y, z]]$ denotes the state of a two level environment with two agents inserted into it. At the same time E is an external environment of a system $F[x, y, z]$ and F is an internal environment of it. All agents and environments are labeled or attributed transition systems (labeled systems with states labeled by attribute labels [9]). The states of transition systems are considered up to bisimilarity. This means that we should adhere to the following restriction in the definition of states: if $E \sim_B E'$ and $u \sim_B u'$ then $E[u] \sim_B E'[u']$.

The main invariant of bisimilarity is the behavior $beh[E]$ of transition system in the state E (an oriented tree with edges labeled by actions and nodes labeled by attribute labels). Therefore the restriction above can be written as follows:

$$beh(E) = beh(E') \wedge beh(u) + beh(u') \Rightarrow$$

$$\Rightarrow beh(E[u]) = beh(E'[u']).$$

Behaviors themselves can be considered as states of transition systems. If the states are behaviors then the relation above is valid automatically, because in this case beh(E) = E, beh(u) = u. Otherwise the correctness of insertion function must be proved in addition to its definition. In any case we shall identify the states with the corresponding behaviors independently from their representation. To define finite behaviors we use the language of behavior algebra (a kind of process algebra defined in [4]). This algebra has operation of prefixing, nondeterministic choice, termination constants $(\Delta, 0, \perp)$ and approximation relation. For attributed transition systems we introduce the labeling operator for behaviors.To define infinite behaviors we use equations in behavior algebra. Usually these equations have the form of recursive definitions $u_i = F_i(u), i \in I$. Left hand sides of these definitions can depend on parameters $u u_i(x_i) = F_i(u, x), i \in I$. To define the attribute labels we use the set of attributes, symbols taking their values in corresponding data domains. These attributes constitute a part of a state of a system and change their values in time. All attributes are devided to external (observable) and internal (nonobservable). By default the attribute label of a state is the set of values of all observable attributes for this state.

The general architecture of insertion machine is represented on the fig. 1.

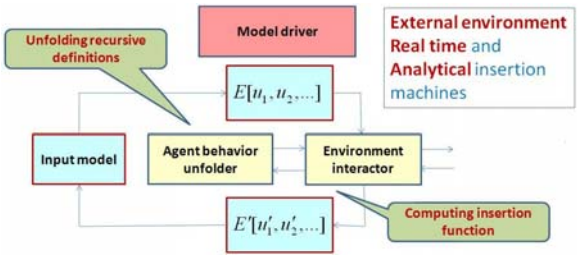

Fig. 1. Architecture of Insertion Machine

The main component of insertion machine is model driver, the component which controls the machine movement along the behavior tree of a model. The state of a model is represented as a text in the input language of insertion machine and is considered as an algebraic expression. The input language include the recursive definitions of agent behaviors, the notation for insertion function, and possibly some compositions for environment states. The state of a system must be reduced to the form $E[u_1, u_2, ...]$. This functionality is performed by the module called agent behavior unfolder. To make the movement, the state of environment must be reduced to the normal form $\sum_{i \in I} a_i \cdot E_i + \varepsilon$ where a_i are actions, E_i are environment states, ε is a termination constant. This functionality is performed by the module environment interactor. It computes the insertion function calling if it is necessary the agent behavior unfolder. If the infinite set I of indices in the normal is allowed, then the weak normal form $a.F + G$ is used, where G is arbitrary expression of input language.

Two kinds of insertion machines are considered: *real type* or *interactive* and *analytical* insertion machines. The first ones exist in the real or virtual environment,

interacting with it in the real or virtual time. Analytical machines intended for model analyses, investigation of its properties, solving problems etc. The drivers for two kinds of machines correspondingly are also divided on *interactive* and *analytical* drivers.

Interactive driver after normalizing the state of environment must select exactly one alternative and perform the action specified as a prefix of this alternative.

Insertion machine with interactive driver operates as an agent inserted into external environment with insertion function defining the laws of functioning of this environment. External environment, for example, can change a behavior prefix of insertion machine according to their insertion function. Interactive driver can be organized in a rather complex way. If it has criteria of successful functioning in external environment intellectual driver can accumulate the information about its past, develop the models of external environment, improve the algorithms of selecting actions to increase the level of successful functioning. In addition it can have specialized tools for exchange the signals with external environment (for example, perception of visual or acoustical information, space movement etc).

Analytical insertion machine as opposed to interactive one can consider different variants of making decision about performed actions, returning to choice points (as in logic programming) and consider different paths in the behavior tree of a model. The model of a system can include the model of external environment of this system, and the driver performance depends on the goals of insertion machine. In the general case analytical machine solves the problems by search of states, having the corresponding properties(goal states) or states in which given safety properties are violated. The external environment for insertion machine can be represented by a user who interacts with insertion machine, sets problems, and controls the activity of insertion machine.

Analytical machine enriched by logic and deductive tools can be used for symbolic modeling. The state of symbolic model is represented by means of properties of the values of attributes rather then their concrete values.

General architecture of insertion modeling system is represented on fig. 2.

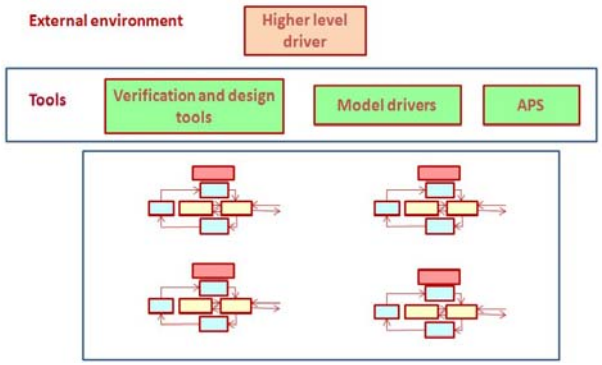

Fig. 2. Architecture of Insertion Modeling System IMS

High level model driver provides the interface between the system and external environment including the users of the system. Design tools based on algebraic programming system APS are used for the development of insertion machines and model drivers for different application domains and modeling technologies. Verification tools are used for the verification of insertion machines, proving their properties statically or dynamically. Dynamic verification uses generating symbolic model traces by means of special kinds of analytical model drivers and deductive components.

The repository of insertion machines collects already developed machines and their components which can be used for the development of new machines as their components or templates for starting. Special library of APLAN functions supports the development and design in new projects. The C++ library for IMS supports APLAN compilers and efficient implementation of insertion machines. Deductive system provides the possibility of verification of insertion models.

2. Input Languages of Insertion Machines

Input language of insertion machine is used to describe the properties of a model and its behavior. This description consists of the

following parts: environment description, behavior description (including the behavior of environment and the behaviors of agents), and insertion function. The behavior description has the following very simple syntax:

<behavior>::= Delta | bot | 0 | < action > | <action> . <behavior> |
<behavior> + <behavior>|
<environment state>[<list of named agent behaviors separated by ,>]|
<functional expression>
<named agent behavior>::=<agent name>:<behavior>

Therefore, the language of behavior algebra (termination constants, prefixing and nondeterministic choice) is extended by functionals expressions and explicit representation of insertion function. The syntax and semantics of actions, environment states, and functional expressions are defined in the environment description. We shall not consider all possibilities and details of environment description language restricting ourselves by making only some necessary comments.

First of all note, that all main components of behavior algebra language (actions, environment states, and functional expressions) are algebraic or logic expressions of base language (terms and formulas). This language is a multisorted (multitype) first order logic language. The signature of this language is defined in the environment description. Functional and predicate symbols can be interpreted and uninterpreted. Interpreted symbols have fixed domains and interpretations given by algorithms of computing values or reducing to canonical forms. All uninterpreted symbols have types and their possible interpretations are restricted by definite domains and ranges. Uninterpreted functional symbols are called attributes. They represent the changing part of the environment. Attributes of arity 0 are called simple attributes, others are called functional ones. Predicates are considered as functions ranging in Boolean type $\{0, 1\}$. If an attribute f has functional type $(\tau_1, \tau_2, ..) \to \tau$ then attribute expressions $f(t_1, t_2, ...)$ are available for all other expressions.

2.1. Examples of Insertion Machines

The simplest insertion machines are machines for parallel and sequential insertion.

Insertion function is called sequential if $E[u, v] = E[u; v]$ where ";" means sequential composition of behaviors. Special case of sequential insertion is a strong sequential composition: $E[u] = (E; u)$. This definition assumes that actions of agents and environment are the same and environment is defined by its behaviors. The sequentiality of this composition follows from associativity of sequential composition of behaviors. Example of insertion machine with strong sequential insertion is represented on fig. 3.

```
Model Sequential(
 interactor rs(P,Q,a)(
    Delta[P+Q]=Delta[P]+Delta[Q],
    Delta[a.P]=a.Delta[P],
    Delta[P]=Delta[unfold P],
    Q[P]=(Q;P)
 );
 unfolder rs(x,y)(
     (x;y)=seq(x,y),
    A=a.A+Delta,
    C=c.C+Delta
 );
 initial(C[A]);
 terminal(Delta[Delta])
)
```

Fig.3. Example of Strong Sequential Insertion

The function seq is a function from IMS library that defines the sequential composition of behaviors:

$$(u;v) = \sum_{\substack{a \\ u \to u'}} a.(u';v) + \sum_{u=u+\varepsilon} (\varepsilon;u),$$

$$(0;v) = 0, (\Delta;v) = v, (\bot;v) = \bot$$

The function unfold reduces the behavior expression to normal form $\sum a_i \cdot u_i + \varepsilon$. This insertion machine generates a word $c^n a^m$ with

nondeterministically chosen $m, n \geq 0$ and successfully terminates. We can define as the condition for the goal state the equality $m = n$ and the driver for this machine will terminate on traces $c^n a^m$.

An example of sequential (not strong) insertion is shown on fig. 4.

```
Model Imperative(
 insertion rs(P,Q,H,a,x,y,u,v)(
    E[P+Q]=Delta[P]+Delta[Q],
    E[define env H.P]=H[P],
    E[(x:=y).P]=assign proc(E,x,y,P),
    E[check(u,x,y).P]=if(compute
  obj(E,u),E[x;P],E[y;P]),
    E[a.P]=a.Delta[P],
   E[P]=E[unfold P]
 )where(
 assign_proc:=proc(E,x,y,P)(E.x−−>
     compute obj(E,y);return E[P])
 );
 behaviors rs(P,Q,x,y,z,u)(
   (x;y)=seq(x,y),
   (u! else Q)=check(u,P,Q),

while(u,P)=check(u,(P;while(u,P)),Delta),
 for(x,y,z,P)=(x;while(y,(P;z)))
 );
 initial(
   define env obj(i:Nil,x:10,y:Nil,
          fact:Nil);
    y:=1;for(i:=1,i _ x,i:=i+1,y:=y*i);
    fact:=y
  );
  terminal rs(E)(E[Delta]=1,E=0)
 )
```

Fig. 4. Model of Simple Imperative Language

This example is a model of simple imperative language and can be considered as insertion representation of its operational semantics.

Insertion function is called a parallel insertion function if $E[u,v] = E[u \parallel v]$. Special case of parallel composition is a strong parallel insertion: $E[u] = E \parallel u]$.

As in the case of strong sequential composition this definition assumes that actions of environment and agents are the same. Example of a model with strong parallel insertion is presented on the fig. 5.

```
Model Parallel(
interactor rs(P,Q,a)(
  Delta[P+Q]=Delta[P]+Delta[Q],
  Delta[a.P]=a.Delta[P],
  Delta[P]=Delta[unfold P],
  Q[P]=(Q k P )
 );
 unfolder rs(x,y,n)(
  (x;y)=seq(x,y),
    x k y = synchr(x,y)+ lmrg(x,y)+
            +lmrg(y,x)+delta(x,y),
    x |^ 1=x,
    x |^ 2=synchr(x,x)+lmrg(x,x)+
          +delta(x,x),
     x |^ n= x k (x |^(n-1))),
 );
 initial (Delta[((a;b) k (a;b));a+b ]);
 terminal (Delta[Delta])
)
```

Fig. 5. Example of Strong Parallel Insertion

Functions **synchr, lmrg**, and **delta** from IMS library are used for definition of parallel composition. Their meaning can be define by the following formulas:

$$synchr(x,y) = \sum_{\substack{a \\ x \to x'}} \sum_{\substack{b \\ y \to y'}} (a \times b).(x', y'),$$

$$lmrg(x,y) = \sum_{\substack{a \\ x \to a'}} a.(x' \parallel y), delta(x,y) =$$

$$= \sum_{\substack{x = x + \varepsilon \\ y = y + \varepsilon}} \varepsilon \parallel \mu.$$

2.2. Restrictions on Insertion Functions

The most typical restriction is additivity. Insertion function is called additive if $E[u+v]=E[u]+E[v]$, $(E+F)[u]=E[u]+F[u]$. Another restriction, which allow to reduce the number of considered alternatives when behaviors are analyzed is the commutativity of insertion function: $E[u,v]=E[v,u]$. Especially the parallel insertion is a commutative one. Some additional equations:

$0[u] = 0, \Delta[u] = u, \perp[u] = bot \;\; 0.$

The state of environment is called indecomposable if from $E = F[u]$ it follows that $E = F$ and $u = \Delta$. Equality means bisimilarity. The set of all indecomposable states constitutes the kernel of a system. Indecomposable states (if they exist) can be considered as states of environment without inserted agents. For indecomposable states usually the following equations hold:

$$E[0] = 0, E[\Delta] = E, E[\perp] = \perp.$$

In [3] the classification of insertion functions was presented: one-step insertion, head insertion, and look-ahead insertion. Later we shall use insertion functions with the following main rule:

$$\frac{E \xrightarrow{a} E', \alpha : u \xrightarrow{b} \beta : u'}{E[\alpha : u] \xrightarrow{c} E'[\beta : u']}, P(E, a, \alpha, b, \beta, c),$$

where P is a continuous predicate. Continuous means that the value of this predicate depends only on some part of behavior tree in the environment state E, which has a finite height (prefix of the tree E of finite height). Hereby, this rule refers to a head insertion. The rules for indecomposable environment states and for termination constants should be added to the main rule.

The next rule

$$\frac{E \xrightarrow{a} E', u \xrightarrow{b} \beta : u'}{E[u] \xrightarrow{c} E'[u']}, P(E, a, c)$$

is the particular case for the head insertion rule in combination with additivity and parallel insertion or commutativity requirements. Such rule will be named permitted rule. It could be interpreted by as follows: agent can execute the action a, and environment permits to execute this action. Predicate E for *permitted rule* will be named *permitted predicate.*

3. Constraint Programming

Constraint programming is a powerful paradigm for solving combinatorial search problems that draws on a wide range of techniques from artificial intelligence, computer science, databases, programming languages, and operations research. Constraint programming is currently applied with success to many domains, such as scheduling, planning, vehicle routing, configuration, networks, and bioinformatics [24].

The Constraint programming paradigm has some resemblance to traditional Operations Research (OR) approach, in that the general path to a solution is:

- analyzing the problem to solve, in order to understand clearly which are its parts;
- determining which conditions (relationships) hold among those parts: these relationships and conditions are key to the solving, for they will be used to model the problem;
- stating such conditions (relationships) as equations; to achieve this step not only the right variables and relationships must be chosen: as we will see, Constraint programming usually offers a series of different constraint systems, some of which are better suited than others for a given task;
- setting up these equations and solving them to produce a solution; this is usually transparent to the user, because the language itself has built-in solvers [25].

There are, however, notable differences with OR, mainly in the possibility of selecting different domains of constraints, and in the dynamic, generation of those constraints. This seamless combination of programming and equation solving accounts for some of the unique components of Constraint Programming:

- the use of sound mathematical methods: well-known and proved algorithms are provided as intrinsic, builtin components of Constraint programming languages and tools;
- the provision of means to perform programmed search, especially in Constraint programming (were search is implicit in language itself);
- the possibility of developing modular, hybrid models, when necessary: many Constraint programming systems offer different constraint systems, which can be combined to model the various parts of the

problem using the tool more adequate for them;

- the flexibility provided by the programming language used, which allows the programmer to create the equations to be solved dynamically, possibly depending on the input data.

As with any other computational approach, all problems are amenable to be tackled with Constraint programming; notwithstanding, there are some types of problems which can be solved with comparatively little effort using Constraint programming based tools. Those applications share some general characteristics:

- No general, efficient algorithms exist (NP-completeness): pecifictechniques (heuristics) must be used. These are usually problems with a heavy combinatorial part, and enumerating solutions is often impractical altogether. A fast program using usual programming paradigms is often too hard and complicated to produce, and normally it is so tied to the particular problem that adapting it to a related problem is not easy.
- The problem specification has a dynamic component: it should be easy to change programs rapidly to adapt. This has points in common with the previous item: Constraint programming tools have builtin algorithms which have been tuned to show good behavior in a variety of scenarios, so updating the program to new conditions amounts to changing the setting up of the equations.
- Decision support required: either automatically in the program or in cooperation with the user. Many decisions can be encoded in mathematical formulae, which appear as rules and which are handled by the internal solvers, so (although, of course, not always) there is no need to program explicit decision trees [26].

Among the applications with these characteristics, the following may be cited: planning, scheduling, resource allocation, logistics, circuit design and verification, finite state machines, financial decision making, transportation, spatial databases, etc.

4. Insertion Machine for Constraint Programming

Some example of insertion machines and restrictions for insertion function are in [3, 9, 23]. In this section we try to show how to use insertion modeling for constraint programming [26]. The problems of constraint programming, where a main goal is behavior of the system, is the closest to the insertion modeling. For example, the problem of wolf-goat-cabbage [27] (A farmer wishes to transfer (by boat) a wolf, a goat, and a cabbage from the left bank of a river to the right bank. If left unsupervised, the wolf will eat the goat and the goat will eat the cabbage, but nothing will happen as long as the farmer is near. Beside the farmer there is only a place for one item in the boat).

Let's consider a formalization of this problem in insertion modeling. Let E be the next environment:

```
obj(
constraints : rs(x, y, z)(
  obj(Wolf : left,Goat : left, x,
        Ferryman : right) = 0,
  obj(Wolf : right, Goat : right, x,
        Ferryman : left) = 0,
  obj(x, Goat : left,Cabbage : left,
        Ferryman : right) = 0,
  obj(x, Goat : right,Cabbage : right,
        Ferryman : left) = 0,
  obj(Wolf : z, x, y, Ferryman : z) = 1,
  obj(x, Goat : z, y, Ferryman : z) = 1,
  obj(x, y,Cabbage : z,
        Ferryman : z) = 1
);
initial : obj(
  Wolf : left,
  Goat : left,
  Cabbage : left,
  Ferryman : left
 )
)
```

where initial is initial state where all creatures are in left bank of the river, constraits are constraint equations with right part of 0 define non possible cases (0 is the neutral element of non-deterministic choice + of insertion modeling) and with 1 if ferryman could transport those creatures in that case. These

both values are covered by two different states of ferry: 1 is just before ferry and 0 - after.

The corresponded input data could be defined in the following way:

(*ferry Wolf* || *ferry Goat* || *ferry Cabbage*).*assetion_constraints*

So, the transition relation of the system is defined in fig. 6.

$$\varphi[p] \xrightarrow{ferryx} \varphi_i[p], constraint\ s(\varphi) = 1$$
$$\varphi[p] \xrightarrow{ferryx} 0, constraint\ s(\varphi_i) = 0 \vee$$
$$\vee \neg constraint\ s(\varphi) = 1$$
$$\varphi[p] \xrightarrow{assertion\_constra\ int\ s} \varphi[p],$$
$$\neg constraint\ s(\varphi) = 0$$
$$\varphi[p] \xrightarrow{assertion\_constra\ int\ s} 0,$$
$$constraint\ s(\varphi) = 0$$

Fig. 6. Relations of System's Transitions

where φ_i a new environment state without acception of constration equation.

Insertion modeling system has found 1 goal trace - all creatures are in other coast and 10 visited traces - those traces cover all possible behaviors of such system.

Typically IMS generated trace is defined by user. It could look like sequence of actions or environment states etc. For this example, to simplify the view of the traces we propose to use one uninterpreted action *transport*:

$$transport(\neg(c = \varphi_i.Ferryman), Ferryman, x),$$

where operation «.» returns state of the ferryman and «¬» returns other coast. So, goal state trace has the next view:

init
transport(right, Ferryman, Shegoat)
transport(left, Ferryman,Nil)
transport(right, Ferryman,Wolf)
transport(left, Ferryman, Shegoat)
transport(right, Ferryman,Cabbage)
transport(left, Ferryman,Nil)
transport(right, Ferryman, Shegoat)

Fig. 7. Example of Goal State Trace

where init is the initial state and Nil means that ferryman is ferried along. In general case, insertion machine for constraint programming should use:

- assetion_constraints agents action in agent behavior and initial state.
- environment description should have non empty section constraints.

Conclusion

The main concepts of insertion modeling system has been considered in the present paper. The system was successfully used for the development of prototypes of the tools for industrial VRS (Verification of Requirement Specification) system and research projects in Glushkov Institute of Cybernetics. Now it is used for the development of program verification tool and 'verifiable programming' project, and for constraint programming. The system continues its enhancement and new features are added while developing new projects. The far goal in the developing of IMS consists of getting of sufficiently rich cognitive architecture to its basis, which could be used in the artificial intelligence research.

1. *LeticShevsky A.A., Gilbert D.R.* A universal interpreter for nondeterministic concurrent programming languages// Fifth Compulog network area meeting on language design and semantic analysis methods, 1996.
2. *Letichevsky A., Gilbert D.* A general theory of action languages//Cybernetics and System Analyses.– 1998. – Vol. 1.– P. 16 –36.
3. *Letichevsky A., Gilbert D.* A Model for Interaction of Agents and Environments. // [In D. Bert, C. Choppy, P. Moses, (eds.)] Recent Trends in Algebraic Development Techniques.-Springer 1999(LNCS). – Vol. 1827. – P. 311–328.
4. *Letichevsky A.* Algebra of behavior transformations and its applications // [In V.B.Kudryavtsev and I.G.Rosenberg (eds)] Structural theory of Automata, Semigroups, and Universal Algebra, NATO Science Series II. Mathematics, Physics and Chemistry.- Springer 2005. – Vol 207. – P. 241–272.
5. *Baranov S., Jervis C., Kotlyarov V., Letichevsky A., and Weigert T.* Leveraging UML to Deliver Correct Telecom Applications// [In L. Lavagno, G. Martin, and B.Selic, (eds.)] UML for Real: Design of Embedded Real-Time Systems. Kluwer, Amsterdam: Academic Publishers, 2003.
6. *Letichevsky A., Kapitonova J., Letichevsky A. jr., Volkov V., Baranov S., Kotlyarov V., Weigert T.* Basic Protocols, Message Sequence

Charts, and the Verification of Requirements Specifications // Computer Networks. –2005. –Vol. 47. – P. 662–675.
7. *Kapitonova J., Letichevsky A., Volkov V., and Weigert T.* Validation of Embedded Systems // [In R. Zurawski, (eds.)] The Embedded Systems Handbook. Miami: CRC Press, 2005.
8. *Letichevsky A., Kapitonova J., Volkov V., Letichevsky A. jr., Baranov S., Kotlyarov V., and Weigert T.* System Specification with Basic Protocols // Cybernetics and System Analyses. –2005. – Vol. 4. – P. 479–493.
9. *Letichevsky A., Kapitonova J., Kotlyarov V., Letichevsky A. jr., Nikitchenko N., Volkov V., and Weigert T.* Insertion modeling in distributed system design // Problems of Programming.–2008. – Vol. 4. – P. 13–39.
10. *Milner R.* Communication and Concurrency // Prentice Hall, 1989.
11. *R. Milner.* Communicating and Mobile Systems: the Pi Calculus. / R. Milner Cambridge University Press 1999.
12. *Hoare C.A.R.* Communicating Sequential Processes // Prentice Hall, 1985.
13. *Cardelli L.* Mobile Ambients. In Foundations of Software Science and Computational Structures // [Gordon Maurice Nivat (eds.)].– Springer 1998(LNCS). – Vol. 1378 – P. 140–155.
14. *Gurevich Y.* Evolving Algebras 1993: Lipari Guide // [In E. Borger (eds.)] Specificationand Validation Methods.– Oxford University Press.–1995. – P. 9–36.
15. *Hoare C.A.R.* Unifying Theories of Programming // He Jifeng Prentice Hall International Series in Computer Science, 1998.
16. *Meseguer J.* Conditional rewriting logic as a unified model of concurrency // Theoretical Computer Science. –1992. – P. 73–155.
17. *Letichevsky A., Kapitonova J., Volkov V., Vyshemirsky V., Letichevsky A.* jr. Insertion programming // Cybernetics and System Analyses. – 2003. – Vol. 1. – P. 19–32.
18. *Kapitonova J.V., Letichevsky A.A., and Konozenko S.V.* Computations in APS // Theoretical Computer Science. – 1993. – P. 145–171.
19. *Insertion* Modeling System.– http://apsystem.org.ua.
20. *Kozen D.* Dynamic Logic / David Harel and Jerzy Tiuryn, 2000.
21. *Hoare C.A.R.* An axiomatic basis for computer programming // Communications of the ACM.–1969. – Vol. 12(10). – P. 576–580.
22. *Floyd R.W.* Assigning meanings to programs // Proceedings of the American Mathematical Society Symposia on Applied Mathematics , 1967. – Vol. 19. – P. 19–31.
23. *Letichevsky A.A., Kapitonova J.V., Volkov V.A., Vyshemirsky V.V.* Insertion Modelling // Cybernetics ans System Anilises .–2003. – Vol. 1. – P. 19–23.
24. *Rossi F., van Beek P., Walsh T.* Elsevier Handbook of Constraint Programming // New York, NY, USA: Science Inc, 2006.
25. *Bartak R.* Tutorial on Filtering Techniques in Planning and Scheduling // The English Lake District, Cumbria, UK., 2006.
26. *Marriott K., Stuckey P.* Programming with Constraints: An Introduction // MIT Press, 1998.
27. *Farmer,* Wolf, Goat and Cabbage Problem.– http://wiki.visualprolog.com/index.php?title=Farmer, Wolf, Goat and Cabbage.

Registration date 03.06.2011

About authors:

O*lexander Letichevsky,*
the Head of Digital Automata Theory Department of Glushkov Institute of Cybernetics of NAS of Ukraine, Academician of the National Academy of Sciences, Doctor of Physics and Mathematics, Professor,

Olexander Letychevskyi,
Research Engineer of Digital Automata Theory Department of Glushkov Institute of Cybernetics of NAS of Ukraine, Candidate of Physics and Mathematics,

Vladimir Peschanenko,
Associate Professor of Informatics Department of Kherson State University, Candidate of Physics and Mathematics, Associate Professor,

Igor Blynov,
Assistant of Informatics Department of Kherson State University,

Dmitry Klionov,
Assistant of Informatics Department of Kherson State University.