UDC 004.2,004.4

Vladimir Peschanenko

PARTIAL EVALUATION IN INSERTION MODELING SYSTEM

The paper relates to practical aspects of insertion modeling. Insertion modeling system is an environment for development of insertion machines, used to represent insertion models of distributed systems. The notions of insertion modeling are stated. The main features of partial evaluation are described in the paper. The conception of partial evaluation in insertion modeling is presented.

Introduction

Insertion modeling is an 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 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 system to composition of environment, and agents inserted into this environment implicitly exists in all theories of interaction 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 was started in Glushkov Institute of Cybernetics. The first version of IMS and some simple examples of its use are available in [19]. IMS has many applications [20-22], that is why a speed of interpretation of the IMS is very important. One of the techniques which helps to speed up interpretation is partial evaluation.

Partial evaluation was the subject of rapidly increasing activity over the past decade of previous century since it provides a unifying paradigm for a broad spectrum of work in program optimization, interpretation, compiling, other forms of program generation, and even the generation of automatic program generators.

Many applications today have concerned compiling and compiler generation from interpretive programming language definitions, but partial evaluation also has important applications in scientific computing, logic programming, meta-programming, and expert systems.

It is distributed a program optimization technique, which is called program specialization. Full automation and the generation of program generators, as well as transforming single programs, are central themes and they have been achieved [23].

Presentation of partial evaluation in IMS is the main goal of the paper. The second section presents the insertion machines, their properties and restrictions that can be met in practice. The main notions about partial evaluations are described in the third section. Partial evaluations for insertion modeling are considered in the last section.

1. Insertion Modeling

Insertion modeling is the development and investigation of distributed concurrent systems by means of representing them as a composition of interacting agents and environments. Both agents and environments are attributed transition systems, considered up to dissimilarity, but environments are additionally provided with insertion function used for the composition and characterizing the behavior of environment with inserted agents. Attributed transition systems are labeled as transition systems and the labels of transitions are called *actions,* they have states labeled by *attribute labels*. If *s* is a state of a system, then its attributed label will be denoted as $al(s)$. Transition system can be also *enriched* by distinguishing in its set of states *S* the set of *initial states* $S_0 \subseteq S$ and the set of *terminal states* $S_\Delta \subseteq S$. For attributed transition system we use the following notation: $a:s \xrightarrow{a} a':s'$ means, there is a transition from the state *s* with attributed label $a \in L$ to the state s' labeled by attributed label $a' \in L$, and this transition is labeled by action $a \in L$. Therefore enriched attributed system *S* can be considered as a tuple

$$< S, A, L, S_0, S_\Delta, T \subseteq S \times A \times S, al : S \to L >.$$

A pair $< A, L >$ of actions and attributed labels is called a signature of system *S*. We also distinguish hidden action τ and hidden attributed label 1. Unlike other actions and attributed labels these hidden labels are not observable.

Behaviors. Each state of transition system is characterized up to bisimilarity by its behavior represented as an element of behavior algebra (special kind of process algebra). The behavior of system in given state for the ordinary (labeled, but not attributed) systems is specified as an element of complete algebra of behaviors $F(A)$ (with prefixing *a.u,* non-deterministic choice *u+v,* constants $0, \Delta, \perp$, the approximation relation $\sqsubseteq$, and the lowest upper bounds of directed sets of behaviors). In the sequel we shall use the term process as a synonym of behavior.

For attributed systems *attributed behaviors* should be considered as invariants of bisimilarity. The algebra $< U, A, L >$ of attributed behaviors consists of three sorted algebra. The main set is a set *U* of attributed behaviors, *A* is a set of actions, *L* is a set of attribute labels. Prefixing and non-deterministic choice are defined as usually (nondeterministic choice is associative, commutative and idempotent). Besides the usual behavior constants 0 (deadlock), Δ (successful termination) and $\perp$ (undefined behavior), the empty action τ is also introduced with the identity

$$\tau.u = u.$$

The operation $(\alpha : u) \in U$ of labeling the behavior $u \in U$ with an attribute label $\alpha \in L$ is added. The empty attribute label 1 is introduced with the identity

$$1 : u = u.$$

The approximation is extended to labeled behaviors, so that

$$(\alpha : u) \sqsubseteq (\beta : v) \Leftrightarrow \alpha = \beta \wedge u \sqsubseteq v$$

Constructing a complete algebra $F(A, L)$ of labeled behaviors is similar to the constructing of the algebra $F(A)$. Each behavior *u* in this algebra has a canonical form:

$$u = \sum_{i \in I} \alpha_i : u_i + \sum_{j \in J} a_j . u_j + \varepsilon_u,$$

where $\alpha_i \neq 1, a_j \neq \tau$, ε_u is a termination constant (0, Δ, $\perp$, $\Delta + \perp$), all summands are different and behaviors u_i and u_j are in the same canonical form.

Behaviors, i.e., elements of the algebra $F(A, L)$ can be considered as the states of an attributed transition system. The transition relation of this system is defined as follows:

$$a.u + v \xrightarrow{a} u$$

$$\alpha : u + v = 1 : (\alpha : u + v) \xrightarrow{\tau} \alpha : u$$

$$\alpha : a.u \xrightarrow{a} u$$

$$\alpha : \beta : u \xrightarrow{\tau} \beta : u.$$

Set *E* of behaviors is called transition closed if $u \in E, u \xrightarrow{a} u' \Rightarrow u' \in E$.

Ordinary labeled transition systems are considered as special case of attributed ones with the set of attribute labels equal to $\{1\}$, and the algebra $F(A)$ is identified with $F(A, \{1\})$.

Insertion function. Environment $< E, C, L, A, M, \varphi >$ is defined as a transition closed set of behaviors $E \subseteq F(C,L)$ with insertion function $\varphi : E \times F(A,M) \to E$. The only requirement for insertion function is that it must be continuous w.r.t. approximation relations defined on E and $F(A,M)$. Usually the behaviors of environment are represented by the states of transition system considering them up to bisimilarity. The state $\varphi(e,u)$ of environment resulting after agent insertion (identified with the corresponding behavior) is denoted as $e[u]$ or $e_{\varphi}[u]$ to mention insertion function explicitly and the iteration of insertion function as

$$e[u_1,u_2,...,u_m] = (...(e[u_1])[u_2]...)[u_m].$$

Environments can be considered as agents and therefore can be inserted into higher level environments with other insertion functions. So the state of multilevel environments can be described for example by the following expression: $e_{\varphi}[e^1_{\psi}[u^1_1,u^2_1,...],e^2_{\psi}[u^1_2,u^2_2,...],...]$. The most of insertion functions considered in this paper are one-step or head ones. Typical rules for definition of insertion function are the following (one-step insertion):

$$\frac{e \xrightarrow{a} e', u \xrightarrow{a} u'}{e[u] \xrightarrow{c} e'[u']}, \qquad (1)$$

$$\frac{e \xrightarrow{c} e'}{e[u] \xrightarrow{c} e'[u]}. \qquad (2)$$

The first rule can be treated as follows. Agent u asks for permission to perform an action a, and if there an a-transition exists from state e, performance of a is allowed and both agent and environment come to the next state with observable action c of environment. The second rule describes the move of environment with suspended move of agent. The additivity conditions usually are used:

$$e[u+v] = e[u] + e[v],$$

$$(e+f)[u] = e[u] + f[u].$$

The rules (1-2) can also be written in the form of rewriting rules:

$$(a.e')[a.u'] = c.e'[u'] + f,$$

$$(c.e')[u] = c.e'[u] + g.$$

Two kinds of insertion machines are considered: *real type* or *interactive* and *analytical* insertion machines. The first ones exist in real or virtual environment, interacting with it in 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 into *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 to exchange the signals with external environment (for example, perception of visual or acoustical information, space movement, etc).

Analytical insertion machine 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 system can include the model of external environment of this system, and the driver performance depends on the goals of insertion machine. In 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.

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

The main component of insertion machine is model driver, the component which controls the machine movement on the behavior tree of a model. The state of a model is represented as a text in input language of insertion machine and is considered as an algebraic expression. The input language includes recursive definitions of agent behaviors, notation for insertion function, and

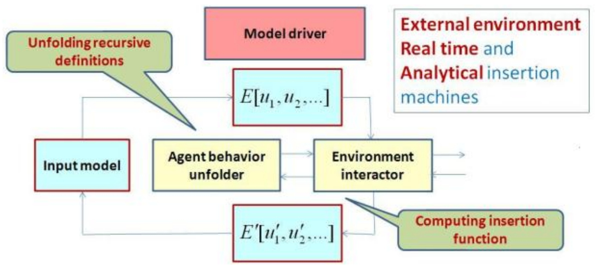

Fig. 1. Architecture of Insertion Machine

possibly some compositions for environment states. The state of a system must be reduced to the form $E[u_1, u_2, \ldots]$. This functionality is performed by the module called agent behavior unfolder. To make the movement, the state of environment must be reduced to 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 the agent behavior unfolder, if it is necessary. If the infinite set I of indices is normally allowed, the weak normal form $a.F + G$ is used, where G is arbitrary expression of input language [9].

2. Partial Evaluations

It is well known that a one-argument function can be obtained from two-argument function by specialization, i.e. by fixing one input to particular value. In analysis it is called *restriction or projection*, and in logic it is called *currying*. Partial evaluation, however, works with *program texts* rather than mathematical functions.

Partial evaluator is an algorithm which produces a so-called residual or specialized program, when a program and some of its input data are given. Running the residual program on remaining input data will yield the same result as running the original program on all of its input data.

The theoretical possibility of partial evaluation was established many years ago in recursive function theory as Kleene's “s-m-n theorem”.

Partial evaluation sheds new light on techniques for program optimization, compilation, interpretation, and generation of program generators. Further, it gives insight into the properties of programming languages themselves.

Partial evaluation can be considered as a special case of program transformation, but emphasizes *full automation* and generation of *program generators* as well as transformation of single programs.

Partial evaluation gives a remarkable approach to compilation and compiler generation. For example, partial evaluation of an interpreter with respect to a source program yields target program. Thus, compilation can be achieved without a compiler, and a target program can be considered as a specialized interpreter.

Moreover, provided partial evaluator is self-applicable, compiler generation is possible: specializing the partial evaluator itself with respect to a fixed interpreter yields a compiler. Thus a compiler can be considered as a specialized partial evaluator, which can specialize only an interpreter for a particular language. Finally, specializing the partial evaluator with respect to itself yields a compiler generator. Thus, compiler generator can be thought of as a specialized partial evaluator, which can specialize itself only.

The application of partial evaluation is not restricted to compiling and compiler generation. If a program takes more than one input, and one of the inputs varies more slowly than the others, then specialization of the program with respect to that input gives a faster specialized program. Moreover, a lot of real-life programs exhibit interpretive behavior. For instance, they may be parameterized with configuration _les, etc., which seldom vary, and therefore they may be profitably specialized.

The range of potential applications is extremely large, as shown by the list of examples in [23]. All examples have been implemented on the computer, by researchers from Copenhagen, MIT, Princeton, and Stanford universities; and INRIA (France) and ECRC (Germany). All have been seen to give significant speedups.

- Pattern recognition.
- Computer graphics by "ray tracing".
- Neural network training.
- Answering database queries.
- Spreadsheet computations.
- Scientific computing.
- Discrete hardware simulation.

In computing partial evaluation is a technique for several different types of program optimization by specialization. The most straightforward application is to produce new programs which run faster than the originals while being guaranteed to behave in the same way. More advanced usages include compiling by partially evaluating an interpreter with the program to be compiled as its input; generating compilers by partially evaluating a partial evaluator with the interpreter for the source language concerned as its input. And finally, generating the compiler-generator by partially evaluating the partial evaluator with itself as its input. It is also true and for interpretation, because partial evaluation makes optimization of the source code of program, which should perform faster. IMS is the interpreter, that is why we will talk about partial evaluation of interpretation.

A computer program, *prog*, is seen as a mapping of input data into output data:

$$prog : I_{static} \times I_{dynamic} \to O .$$

I_{static}, the static data, is the part of the input data, known at interpretation time.

The partial evaluator transforms $\langle prog, I_{static} \rangle$ into $prog^* : I_{dynamic} \to O$ by precomputing of all static input during interpretation time. $prog^*$ is called the *residual program* and should run more efficiently than the original program. The act of partial evaluation is said to *residualize prog to* $prog^*$ [23].

3. Mixed Computation in Insertion Modeling

There are three known possibilities to realize partial evaluation in insertion modeling:

- Partial behavior evaluation.
- Partial actions evaluation.
- Partial low level language evaluation.

Partial behavior evaluation. The behavior description has the following simple syntax:

<behavior>::= Delta | bot | 0 |
< action > | <action> . <behavior> |
<behavior> + <behavior>|
<behavior>;<behavior>|
<behavior>||<behavior>|
<functional expression>|
<environment state>[<behavior>]|
<agent name>

A set of agent names is considered as a system of equations by the following syntax:

<agent equation system>::=
<list of <agent equations> separated by "," >,
<agent equation>::=
<agent name>=< behavior>.

Therefore, the language of behavior algebra (termination constants, prefixing and nondeterministic choice) is extended by functional expressions and explicit representation of insertion function. We consider extended grammar for behavior with sequential (";") and parallel ("||") compositions [19].

As it was shown in grammar <agent equation> can be recursive in general case. It means that it is possible to substitute not recursive equations in right part of other equations and in behavior.

Let $b=(u,Sys)$ be defined behavior and A_{Sys} is set of agent names (left part of equations), and A_u is set of agent names in behavior *u*. The system of equation is static for concrete example. It means that $Sys \in I_{static}$. In step-by-step insertion the behavior *u* is changed. Speaking generally, $u \in I_{dynamic}$. So, the notion of partial evaluation can be used here for optimization of interpretation.

However, note that equations are used for definition of infinite behavior. So, the partial evaluation here is to eliminate all agent names which define finite behavior in *u* and *Sys*. Let A_f be set of agent names (left part of equations in *Sys*) which defines finite behavior. So, the idea of partial evaluation here is to build $b/A_f=(u/A_f,Sys/A_f)$, where operation "*a/b*" defines an algorithm of elimination in *a* agent names which defines finite behavior A_f is the set of agent names from A_{Sys} which behavior $beh(a)$ and $(a=beh(a)) \in Sys$:

$$A_f=\{a \mid a \in A_{Sys} \wedge a \in A_{beh(a)} \wedge (a= \\ =beh(a)) \in Sys\}.$$

Theorem 1.

$$(b,Sys) \Leftrightarrow (b/A_f,Sys/A_f).$$

Proof. $(b/A_f,Sys/A_f) \Rightarrow (b,Sys)$ is always true, because it is possible to mark some finite behavior b_f inside b and to replace it by a new agent name a and to add new equation $a=b_f$ to the Sys/A_f and so on. If A_f is set of agent names which has finite behavior in Sys then it is possible to eliminate all of them from $beh(A_f)$ (the right parts of equations). It means that $beh(A_f)$ doesn't depend on A_f. ($A_f \cap A_{beh(A_f)}=\varnothing$). Then, it is possible to substitute equations Sys to u ($A_u \cap A_{A_f}=\varnothing$). So, the behavior b/A_f is obtained. Next, if $A_f \cap A_{beh(A_f)}=\varnothing$ and $A_u \cap A_{A_f}=\varnothing$ then all such equations can be removed from Sys. Finally, Sys/A_f was obtained and $(b/A_f,Sys/A_f) \Rightarrow (b,Sys)$. So, the theorem was proved.

Partial actions evaluations. Let split the set of action *A* on two subsets where A_{Cng} is the set of *changing actions* and A_{NCng} is the set of *non-changing actions* $A=A_{Chg} \cup A_{NChg}$. One step of insertion of some action $a \in A_{NChg}$ is inserted in the next way:

$$E[a.u]=a.E[u], a \in A_{NChg}.$$

Here action *a* doesn't change environment state *E* and doesn't add anything to the resulting behavior *u* after its insertion. These actions A_{NChg} are not parameterized.

The one step insertion of some action $a \in A_{Chg}$ is inserted in the next way:

$$E[a.u]=f(a,E,u), a \in A,$$

where $f: A_{Cng} \times E \times F(A) \to E \times F(A)$, A is the set of actions, *E* is a set of the environment states, *F(A)* is an expression in the algebra of behavior. This function *f* could change environments state *E* and could add the new behavior into *u*. The set of actions A_{Cng} and corresponded functions *f* for each of them are static. It means that we could apply here the notion of the partial evaluations.

So, let $b/A_f=(u/A_f,Sys/A_f)$ and $\varphi: E \times F(A) \to E$, where *E* is a set of environment states, $F(A)$ is an expression in the algebra of behavior, the function φ called *insertion function*. One of main properties of such function is continuity. It means that for each action $a \in A_{Cng}$ the function $f_a(a,E,u)$ is defined by insertion function φ. The set of such functions is marked by F_φ. Usually insertion function is defined as a system of rewriting rules. For one step insertion it is necessary to build behavior in the normal form:

$$\sum_{i \in I} a_i \cdot \mathrm{u}_i + \varepsilon, a_i \in A$$

which is defined by the only way. Where ε is termination constant, u_i is behavior. Then, it is made or we make non-deterministic insertion: $E[x+y]=E[x]+E[y]$, where E is environment state, x and y are behaviors.

So, the main idea of partial evaluation here is to build $\varphi^* : E \times F(A, F_\varphi) \to E$. Let $a \in A_{Cng}$, $u \in F(A)$, $u' \in F(A, F_\varphi)$. $F(A, F_\varphi)$ is made from $F(A)$, by substitution of an action $a \in A_{Cng}$ in $F(A)$ to function $f_a(a, E, u)$.

Theorem 2. $\varphi(E, a.u) = \varphi^*(E, a.u')$.

Proof. Let's collect the set of equations $\{a = f_a\}, f_a \in F_\varphi$, where $a \in A_{Cng}$, f_a corresponding interpretation of action. Corresponded function f_a will always exist because of continuity of insertion function. After that obtained set $\{a = f_a\}$ is substituted into behavior u and the result is $u' \in F(A, F_\varphi)$. Finally, all $f_a \in F_\varphi, a \in A_{Cng}$ are replaced in the insertion function φ by the following condition:

$$E[f_a.u] = f_a(a, E, u).$$

From the other side $E[a.u] = f_a(a, E, u)$ for $a \in A_{Cng}$ and the theorem is proved.

From the other point of view what happens with the program if the both algorithms of partial evaluations are done?

Theorem 3. $A_f \cap A_{Cng} = \varnothing$.

Proof. A_f is the set of agent names, but agent names are not the actions because they are defined by the equation in unfolding. So, the theorem is proved.

From practical point of view this theorem means that these two partial evaluations are independent and could be realized in any combinations.

4. Partial low level language evaluation

The Algebraic Programming System *APS* and Insertion Modeling System *IMS* [24] are used for prototyping of the algorithms first, then for research of the properties and behavior of such algorithms, and finally for realization of a final version for such algorithms. These systems have two languages for realizations of this idea: *APLAN* (*A*lgebraic *P*rogramming *LAN*guage) and *C++* (language of such systems realization). The process of automatically conversion of code from *APLAN* to C++ was described in [25]. So, if some algorithm was researched and realized in final version of it then it is possible to consider it as a static data of the programs. It means that the notion of partial evaluation could be used here. This idea can be used for realization of functions f_a from the previous section and for final realization of the Model Driver module (fig. 1). However, note that the idea spreads for all part of such algorithm. A user should choose what parts of the algorithm are considered as static. And then, our partial evaluation for that case should support that. For realization of partial evaluation here the notion of *APLAN interpreter* is used.

APLAN Interpreters are programs designed for the interpretation of the programs written in *APLAN* language. They are developing in *C++* language on the base of libraries of functions and data structures to work with internal representation of system data structures. Each interpreter is connected with the distinct algebra $T_\Omega(X, A)$, where Ω is signature (the set of marks with arity), *X* is set of *names* in *APLAN*, and *A* is set of *atoms*. *Names* and *atoms* are *APLAN* notions. The easer way to make partial evaluation is to use here the translator of source code which was developed early [25]. The Translator transfers realization of such codes from the set *X* to the *A*. The function names are considered as atoms. If such codes depend on other *APLAN* code then such conversion obtains the internal call of sub-programs only. It means that if some APLAN sub-program is left on *APLAN* language then the resulting codes are called *C++* realization. So, the problem of using C++ procedures in *APLAN* language is solved. If the system has both realizations *APLAN* and *C++* with the same name then after removing of *APLAN* definition the system uses *C++*. However, the problem of replacing of some C++ procedure to *APLAN* procedure is still actual.

The solution of this problem is to add the set H of pairs (x_n, f_n), where x_n is *APLAN* name of such procedure or *Nil* if corresponded name was not found, f_n is pointer to *C++* realization of such procedure or *Nil* if corresponded procedure was not realized yet. This set H can be obtained after loading of initial model, because that process builds the algebra $T_\Omega(X, A)$ according to the current *APLAN Interpreter*. The function $pc: H \times T_\Omega(X, A) \to T_\Omega(X, A)$ is defined in *Interpreter*. This function is used in *C++* instead of direct call of function f_n. It finds pointer for the current realization of the *APLAN* procedure. This function works by the following way:

- If $x_n \neq Nil$ then it calls corresponded *APLAN* procedure.
- If $(x_n = Nil) \wedge (f_n \neq Nil)$ then it calls corresponded *C++* procedure.
- If $(x_n = Nil) \wedge (f_n = Nil)$ then it prints error message and returns *Nil*.

The most important feature of realization of such function *pc* is strategies calling [25]. For this case the function $pc^2: H \times H \times T_\Omega(X, A) \to T_\Omega(X, A)$ is defined. The case, when system of rewriting rules (s.r.r.) can be considered as internal function on *C++*, is added to all internal strategies. Let pairs $(x_n^1, f_n^1), (x_n^2, f_n^2) \in H$ be the first and the second arguments of pc^2 function respectively. Then this function works in the following way:

- If $(x_n^1 \neq Nil) \wedge (x_n^2 \neq Nil)$ then it calls corresponded *APLAN* strategy with *APLAN* s.r.r.
- If $(x_n^1 = Nil) \wedge (f_n^1 \neq Nil) \wedge (x_n^2 \neq Nil)$ then it calls corresponded *C++* strategy with *APLAN* s.r.r.
- If $(x_n^1 \neq Nil) \wedge (x_n^2 = Nil) \wedge (f_n^2 \neq Nil)$ then it calls corresponded *APLAN* strategy with *C++* s.r.r.
- If $(x_n^1 = Nil) \wedge (x_n^1 \neq Nil) \wedge (x_n^2 = Nil) \wedge$ $\wedge (f_n^2 \neq Nil)$ then it calls corresponded *C++* strategy with *C++* s.r.r.
- If $(x_n^1 = Nil) \wedge (x_n^1 = Nil) \vee (x_n^2 = Nil) \wedge$ $\wedge (f_n^2 = Nil)$ then it prints error message and returns *Nil*.

This partial evaluation for low level realization gives possibilities to research subprograms of final *C++* program on *APLAN* language. For example, if it is required to research one procedure in large system then we could realize it on *APLAN* only and run it without appreciable loss of performance. It gives us possibilities to research any subprogram of large system that was realized in *APS* and *IMS* systems.

Conclusion

So, the notion of the partial evaluations is applicable to the insertion modeling and could be used in practice. These approaches were realized in *APS* and *IMS*, that makes them more applicable for industrial projects.

1. *Letichevsky 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. *Milner R.* 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. *Letichevsky A.A., Letychevskyi O.A., Peschanenko V.S.* Insertion Modeling System // PSI 2011, Lecture Notes in Computer Science, Vol. 7162, Springer, 2011. – P. 262–274.
20. *VRS* Tool [http://www.issukraine.com/ISS_VRS_tool.htm].
21. *Methodical* Program Complex TerM 7-9 [http://riit.ksu.ks.ua/index.php?q=en/node/228].
22. *Kobets V.M.* Introduction of information technologies knowledge control from economical disciplines // Informatin Technologies in Education. – 2019. – Vol. 3, – P. 123–127.
23. *Neil D. Jones*, *Carsten K.* Gomard, and Peter Sestoft: Partial Evaluation and Automatic Program Generation, Prentice Hall International, 1993.
24. *APS&IMS* system [http://apsystem.org.ua].
25. *Letichevsky A., Letichevsky A. Jr, V. Peschanenko.* Translation Algorithm of APLAN code // Control's Machines and Systems. – 2010. – Vol. 6. – P. 40–46.

Data received 09.04.2012

About author:

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