STATIC WEAVING AT DYNAMIC JOIN POINTS

Ján KOLLÁR, Valerie NOVITZKÁ
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics,
Technical University of Košice, Letná 9, 042 00 Košice, Slovak Republic,
tel. +421 55 602 2577, +421 55 602 4182, E-mail: Jan.Kollar@tuke.sk, Valerie.Novitzka@tuke.sk

SUMMARY

Aspect oriented programming is a programming methodology based on separating concerns (aspects) of computation, describing them separately in the form of advices that are applied in clearly selected set of points of a program using pointcut designators. In this paper we will present an approach to the implementation of crosscutting concept of aspect oriented programming using environmental basis of PFL - a process functional programming language. We will concentrate on the advices, defined by pointcut designators with temporal logic operations. We also introduce possible directions in further conceptual solutions based on process functional paradigm.

Keywords: *programming paradigms, process functional programming, aspect oriented programming, dynamic join points, advice implementation*

1. INTRODUCTION

Aspect oriented programming [1, 2, 3, 6, 7, 8, 31] is the desire to make programming statements of the form: *In programs P, whenever condition C arises, perform action A over conventionally coded programs P*. Thus, in aspect oriented programming we want to be able to say, "This code realizes this concern. Execute it whenever these circumstances hold." This breaks completely with local and unitary demands – we can organize our program in the form most appropriate for coding and maintenance. We do not even need the local markings of cooperation. The weaving mechanism of the aspect system can, by itself, take our quantified statements (advice) and the base program and produce the primitive directions to be performed. A characteristic of aspect-oriented programming, as embodied in AspectJ [6, 7], AspectCOOL [2, 3], Aspect BASE [31] and other aspect languages is the use of advice to incrementally modify the behavior of a program. An advice declaration specifies an action to be taken whenever some condition arises during the execution of the program. The events at which advice may be triggered are called join points. Join points are dynamic if they refer to events during execution, otherwise they are static. The process of executing the relevant advice at each join point is called weaving. The condition is specified by a formula called a pointcut designator. In this paper we illustrate crosscutting concept of aspects oriented programming using simple Wand's example taken from [31], expressing it in PFL – a process functional language, having been developed at the Department of Computers and Informatics [9, 10, 11, 12, 13, 14, 21, 22, 25, 26, 27].

Process functional paradigm is based on evaluation of processes that affect the memory cells by their applications. PFL - an experimental process functional language, aimed to von Neumann machines comes from the Haskell concept of pure functional languages [20], including an imperative feature of manipulating programming environments [4, 14], however, neither in monadic manner [19, 30] nor in an assignment based manner. Instead of this, we introduce pure functional form for function definition and type definition of this function including the environment variables to be input memory places for a subset of formal parameters. Such structuring of a program means that the function of computation is free of an undisciplined memory access and unexpected affecting by side effects.

We attend, that separating concerns of memory access/update and function of computation, both concerns – or aspects of computation remain mutually dependent.

In essence, the notion of *functionality* is relative. These days, mechanisms that deal with concurrency and failures are, for instance, considered as *non-functional* aspects of the application. It is tempting to separate these aspects from the other functionalities of the application [8].

On the other hand, aspect programming is based on weaving different aspects, and it would be imposible, if aspects are fully independent. That is why instead of thinking about their independency we must find the simpliest and the most general rule for expressing their dependence.

We mention that this orientation of research is possible and useful for recognizing what aspects in programming can be considered and how to exploit them.

2. PFL CONCEPTION

Let us introduce an example 2.1 of a simple purely functional program comprising a pure function `p` of three arguments, which value is defined by an expression `e`.

* *This work was supported by VEGA Grant No. 1/1065/04: Specification and Implementation of Aspects in Programming.*

Example 2.1 Purely functional program

```
p :: T1 -> T2 -> T3 -> T
p x1 x2 x3 = e

main = print (p e1 e2 e3)
```

For the purpose of simplicity, let `print` is a built-in function of the type `T->()` that prints the value of its argument and `main` is the main function, which starts the evaluation. Evaluating `main`, the application `(p e1 e2 e3)` is evaluated and its value `e[e1/x1,e2/x2,e3/x3]` is printed.

The type definition of *PFL* process comprises environment variables, such as v_1, v_2 and v_3 in the example 2.2.

Example 2.2 *PFL* program illustrating the update

```
p :: v1 T1 -> v2 T2 -> v3 T3 -> T
p x1 x2 x3 = e

main = print (p e1 e2 e3)
```

Evaluating `main`, the printed value is again `e[e1/x1,e2/x2,e3/x3]` and the evaluation performs three side effects – it stores e_1 to v_1, e_2 to v_2, and e_3 to v_3. These side effects are performed before expression `e` is evaluated. The example 2.2 illustrates the update of environment variables not using them in expressions.

Finally, we will show the access of environment variables. Let `seq` is binary sequencing operation of the type `()->()->()`, which guarantees the first argument is evaluated before the second argument. According to the example 2.3 the value of application will be printed twice.

Example 2.3 *PFL* program illustrating the access

```
p :: v1 T1 -> v2 T2 -> v3 T3 -> T
p x1 x2 x3 = e

main = seq (print (p e1 e2 e3))
           (print (p () e4 ()))
```

The first printed value is as before, i.e. it is

$$e[e_1/x_1,e_2/x_2,e_3/x_3]$$

The second printed value is as follows:

$$e[e_1/x_1,e_4/x_2,e_3/x_3]$$

It means that the application `(p () e4 ())` performes the access of the values e_1, and e_3 having been stored already in v_1 and v_3, as a result of preceding application `(p e1 e2 e3)`.

Let us designate our applications by numbers, as follows:

$$(1) \equiv (p\ ()\ e_4\ ())$$
$$(2) \equiv (p\ e_1\ e_2\ e_3)$$

The state change is as follows:

$$\langle v_1[\bot]\ ,v_2[\bot]\ ,v_3[\bot]\rangle$$
$$\overset{(1)}{\Rightarrow} \langle v_1[e_1],v_2[e_2],v_3[e_3]\rangle$$
$$\overset{(2)}{\Rightarrow} \langle v_1[e_1],v_2[e_4],v_3[e_3]\rangle$$

where $v_i[\bot]$ designates undefined value stored in the variable v_i, $v_i[e_k]$ designates value e_k stored in the variable v_i., and $\overset{(a)}{\Rightarrow}$ means the transition caused by application `(a)`.

Of course, it is possible to show state changes on separate variables:

$$\langle v_1[\bot]\rangle \overset{(1)}{\Rightarrow} \langle v_1[e_1]\rangle$$

$$\langle v_2[\bot]\rangle \overset{(1)}{\Rightarrow} \langle v_2[e_2]\rangle \overset{(2)}{\Rightarrow} \langle v_2[e_4]\rangle$$

$$\langle v_3[\bot]\rangle \overset{(1)}{\Rightarrow} \langle v_3[e_3]\rangle$$

Omitting the details on *PFL* semantics, we can summarize the following facts:

1. We never use environment variables in *PFL* expressions, still being able to access and/or to update the variables.
2. We may use an environment variable for any argument type, not however for the value type in type definitions of processes.
3. We may say: Provided that an argument type of a function in its type definition "comprises" an environment variable, then this function becomes the process which application affects the state of computation.

But the last statement is very near to aspect approach, since *each process realises the concern of memory access/update whenever it is applied*.

We will discuss the variety of *PFL* ability for aspect oriented development later.

In the next section we introduce Wand's conception of aspect methodology, as incorporated in Aspect BASE and its relation to our approach.

3. ADVICES IN ASPECT BASE AND *PFL*

In AspectJ model [6, 7], a program consists of a base program and some pieces of advice. The program is executed by an interpreter. When the interpreter reaches certain points, called join points, in its execution, it invokes a weaver, passing to it an abstraction of its internal state (the current join point). Each advice contains a predicate, called a pointcut designator (pcd), describing the join points in which it is interested, and a body representing the action to take at those points. It is the job of the weaver to demultiplex the join points from the interpreter, invoking each piece of advice that is interested in the current join point and executing its

body with the same interpreter. The concept of aspect BASE language [31], related to AspectJ model above, is as follows:

- First, when a piece of advice is run, its body may be evaluated **before**, **after** or instead of the expression that triggered it; this specification is part of the advice. In the last case, called an **around** advice, the advice body may call the primitive **proceed** to invoke the running of any other applicable pieces of advice and the base expression.
- Second, the language of predicates is a temporal logic, with temporal operators such as **cflow**. Hence the current join point may in general be an abstraction of the control stack.
- Each advice body is also interpreted by the same interpreter, so its execution may give rise to additional events and advice executions.
- Last, the set of advice in each program is a global constant.

Coming out from Wand's "Binding variables with cflow" example as introduced in [31], we will show how *PFL* programs can be aspectized using the environmental conception of *PFL* language.

Let us have the *PFL* program introduced in the example 3.1.

Example 3.1 Source *PFL* program

```
foo n = fac n

fac 0 = 1
fac (n + 1) = (n + 1) * fac n

main = print (fac 6 + foo 4)
```

which, when executed, will print on the screen the number `744`.

Let us require:

For each application fac y, such that it is "called" from foo x, the advice action (print x ; print y) is performed.

It means that we need an advice, which yields the following result on the screen:

```
4 4 4 3 4 2 4 1 4 0 744
```

In terms of Aspect BASE, instead of modifying the source program above, we define this condition separately using pointcut designator as follows:

```
before (fac y) cflow (foo x)
  (print x ; print y)
```

Provided that `(<<)` corresponds to Wand's `before` and `(->)` to `cflow` operation, the corresponding *PFL* form of advice would be as follows:

```
advice x y =
(print x ; print y)<<foo x->fac y
```

So, aspectized source form is introduced in example 3.2, as follows

Example 3.2 Aspectized source *PFL* program

```
-- PFL Language
foo n = fac n

fac 0 = 1
fac (n + 1) = (n + 1) * fac n

main = print (fac 6 + foo 4)

-- Language of advices
advice x y =
  (print x; print y)<<foo x->fac y
```

In the example above the first part is written in *PFL* language, but the second part in a hypothetical language of advices, with a very different semantics. Informally, `advice` is rather a macro than function definition, arguments of `print` are not lambda variables of `advice` but they are arguments of `foo` and `fac`, respectively. Seemingly the action `(print x; print y)` that is applied just if `fac` application is "reachable" by `foo` application needs manipulating control stack in run-time.

Sofar, using current implementation of *PFL* we expressed just advices for positional operations such as `before` that may be evaluated statically, not the advices selected for dynamic join points that require run time application of temporal operations, such as `cflow`, designated by `(->)`. In this paper we will show that static solution for dynamic join points is still possible.

To compare weaving based on stafic and dynamic join points definitions, let us introduce first the advice which determines static position of action applied before `fac` occuring `in foo`, as follows:

```
advice x y =
(print x; print y)<< fac y in foo x
```

This advice, comprising positional operation `in`, can be translated into *PFL* advice process, as follows

```
advice :: u Int -> v Int -> ()
advice x y = print x ; print y
```

which becomes a part of woven form of *PFL* program, according the example 3.3.

Example 3.3 Woven *PFL* program – static join point

```
sfac :: v Int -> ()
sfac x = ()

foo :: u Int -> Int
foo n = fac (sfac n; advice ()())

fac :: v Int -> Int
fac 0 = 1
fac (n + 1) = (n + 1) * fac n

main = print (fac 6 + foo 4)

advice :: u Int -> v Int -> ()
advice x y = print x ; print y
```

Evaluating `main`, we obtain the next output on the screen:

```
4 4 744
```

In example 3.3, we use the environment variable `v` to store argument of `fac` and `u` to store the argument of `foo`. The application `(sfac n)` in `foo` stores the value `n` temporarily and both `u` and `v` values are acccessed by the application

```
(advice ()())
```

in `foo`. Notice also that the value of

```
(sfac n; advice ()())
```

is `()`, hence `fac (sfac n; advice ()())` is equal to `(fac ())`, but, since accessed value is `n`, it is equal to `(fac n)`.

In the next section we will develop the solution for dynamic join points in *PFL*.

4. DYNAMIC JOIN POINTS

The result having been reached in execution of `main` in the example 3.3, is more visible considering the state of call stack, which is as follows:

```
[main]
-- evaluation of fac 6
[main,fac]
 ........
[main,fac,fac,fac,fac,fac,fac]
[main,fac,fac,fac,fac,fac,fac,fac]
[main,fac,fac,fac,fac,fac,fac]
........
[main,fac]
[main]
-- evaluation of foo 4
[main,foo]
[main,foo,• fac]
[main,foo,fac,°1 fac]
[main,foo,fac,fac,°2 fac]
[main,foo,fac,fac,fac,°3 fac]
[main,foo,fac,fac,fac,fac,°4 fac]
[main,foo,fac,fac,fac,fac]
[main,foo,fac,fac,fac]
[main,foo,fac,fac]
[main,foo,fac]
[main,foo]
[main]
```

The action (`print x; print y`) performed in the join point marked by • yields `4 4`, on the screen.

The same action in join point °1 would yield 4 3 on the screen, in °2 would yields 4 2, etc., and finally the action °4 yields 4 0. Unfortunately, join points, marked by `°1`, `°2`, `°3`, and `°4` cannot be selected using static positional operations such as `in`.

To explain this more precisely, let us consider the advice

```
advice x y =
  (print x; print y)<<foo x->fac y
```

again and prove that program in the example 4.1 is woven incorrectly for this advice.

Example 4.1 Incorrectly woven *PFL* program

```
sfac :: v Int -> ()
sfac x = ()

foo :: u Int -> Int
foo n = fac (sfac n; advice ()())

fac :: v Int -> Int
fac 0 = 1
fac (n + 1) =
 (n + 1)*fac(sfac n; advice ()())

main = print (fac 6 + foo 4)

advice :: u Int -> v Int -> ()
advice x y = print x ; print y
```

Then the result does not conform to our advice, since it is as follows:

```
⊥ 5 ⊥ 4 ⊥ 3 ⊥ 2 ⊥ 1 ⊥ 0
4 4 4 3 4 2 4 1 4 0
744
```

where the first line of numbers comes from the application `fac 6`, the second line from `foo 4`, and the number `744` from `print` application in `main`.

This incorrect weaving would mean that the state of the call stack during evaluation is as follows:

```
[main]
-- evaluation of fac 6
[main,fac]
```

```
[main,fac,¤fac]
[main,fac,fac,¤fac]
[main,fac,fac,fac,¤fac]
[main,fac,fac,fac,fac,¤fac]
[main,fac,fac,fac,fac,fac,¤fac]
[main,fac,fac,fac,fac,fac,fac,¤fac]
[main,fac,fac,fac,fac,fac,fac]
........
[main,fac]
[main]
-- evaluation of foo 4
[main,foo]
[main,foo,•fac]
[main,foo,fac,•fac]
[main,foo,fac,fac,•fac]
[main,foo,fac,fac,fac,•fac]
[main,foo,fac,fac,fac,fac,•fac]
[main,foo,fac,fac,fac,fac]
[main,foo,fac,fac,fac]
[main,foo,fac,fac]
[main,foo,fac]
[main,foo]
[main]
```

Correct join points are marked by •, and incorrect by ¤. Incorrectly selected join points cause the output

```
⊥ 5 ⊥ 4 ⊥ 3 ⊥ 2 ⊥ 1 ⊥ 0
```

attempting even to print yet undefined value of argument of `foo`.

Correct solution for dynamic join points can be still found based on tracing control flow statically. This tracing leads to a new aspectized copy `ffac` derived from `fac` and applied in `foo`, as shown in the example 4.2.

Example 4.2 Correctly woven *PFL* program

```
sfac :: v Int -> ()
sfac x = ()

foo :: u Int -> Int
foo n = ffac (sfac n;advice ()())

ffac :: v Int -> Int
ffac 0       = 1
ffac (n + 1) = (n + 1)*
  ffac (sfac n; advice ()())

fac :: Int -> Int
fac 0       = 1
fac (n + 1) = (n + 1) * fac n

main = print (fac 6 + foo 4)

advice :: u Int -> v Int -> ()
advice x y = print x ; print y
```

The woven *PFL* program in example 4.2 is the result of weaving *PFL* program introduced in the example 3.2. In this way, temporal logic `cflow` may be implemented using current implementation of *PFL* language.

5. DISCUSSION

Although one of the dynamic properties of programs - control flow – can be implemented by the transformation of corresponding advice into *PFL* language, many questions arise in association with the definition of advices in a separate language for them.

First, exploiting manyfold aspects there is no proof, that this approach is sufficiently open for adding new aspects of computation in the future. Second, from the viewpoint of aspect language, we need multi-paradigmatic language, which means programming using at least two languages. Third, separating different concerns of computation without uniform language basis may decrease the reliability. Moreover, it seems that any sophisticated software engineering methodology, not supported by mathematical reasoning about the correctness and further properties of the large software systems is not sufficient to guarantee their correct function and behavior required by a user [15, 16, 17, 18, 23, 24]. Hence, providing a uniform and an open aspect language basis is the task of high importance.

At the present time, we have implemented *PFL* as a language integrating functional and imperative properties of current programming languages. Except modularity, which we plan add in a future, this language exploits both parametric type polymorphism and abstract typing, and, since it is environment based, it is appropriate to object programming. The work goes on profiling process functional programs. The significant property is a uniform handling of objects and algebraic data structures, as well as arrays. Since of manipulating environment variables implicitly, stateful computation is performed using just expression evaluation, with well-defined side effects. Although we develop a code generator into Java and Haskell languages, we think about *PFL* as a programming paradigm, rather then programming language. Our aim is to exploit this paradigm to integrate the specification and the implementation of complex software systems. One of inspirative specification methodology is also aspect programming.

Coming back to the process definition in example 2.2, which is as follows

```
p :: v1 T1 -> v2 T2 -> v3 T3 -> T
p x1 x2 x3 = e
```

it can be noticed that “the application of environment variable to a type”, such as (`vi Ti`) defines separately side-effect action on the variable

v_i, for all applications of process p. There is another form for this side effect action available, as introduced below:

```
q :: v {1..10} T' -> T
q x  = e
```

Then, for example, the application

```
q ({2} ())
```

means the access of the value of the second element of the array v using this value as x in expression e, and the application

```
q ({2} 5)
```

means the update of the second element of the array v by the value 5 using this value as x in expression e. In this case, v {1..10} T' defines range checking – in terms of aspect programming it is a kind of action. These actions are defined in $\mathcal{PFL}$ in type definitions, separated from definitions. Hence we may think about the extension of this approach using temporal logic adding a set of additional advices to a function, a process, a class, an instance, an object, etc.

Also type definition is formally just a very restricted and operationally just a very released advice, which allows the use of argument values from restricted domain, checking the type of application of a function or process. The type of application is defined redundantly, since it may be derived. It is a question, whether such redundancy is necessary, if we know that potential occurrence of an environment variable with the process type would be clearly redundant. By other words, need we really to define post-conditions in a program, others than a single one for the main?

Another inappropriate property in programming languages is mixing multiple concepts into a single syntactic construct, i.e. such that are semantically quite different. For example, type definition such as

```
p :: v1 T1 -> v2 T2 -> v3 T3 -> T
```

implies conditions for type checking as well as for subsequent evaluation of arguments. In general, there are two ways how to make this evaluation parallel. First one is to implement n-nary parallel operation applied in expressions. Second, we may define parallel evaluation of arguments for all process applications, in the type expression, for example, as follows

```
p :: v1 T1 || v2 T2 || v3 T3 -> T
```

Both solutions are correct but not flexible enough to handle the ordering when dynamic load balancing would improve the run-time efficiency. The flexibility would increase rapidly if we define a general ordering function able to change its definition during computation. This can be performed, for example by the definition of ordering advice in the form as follows:

```
p :: order T1 T2 T3 -> T
```

Ideally, instead of some mixing concepts we need the definition of function of computation and one or more advices, dealing with computational time, computational space, restrictions on values, allocation of resources, etc.

Incorporating such advice, or a set of advices into the language is our current research.

6. CONCLUSION

In this paper we present the ability of $\mathcal{PFL}$ – a process functional language to express source-to-source transformation, as needed for weaving different aspects of computation. We have shown that using environmental basis of process functional paradigm is sufficient to express woven program aspectized by an advice comprising pointcut designator with control flow operation, belonging to temporal logic.

We also discussed the weakness of current implementation of $\mathcal{PFL}$ and possible directions of the research in the area of aspect languages in the future.

Considering different aspects of computation, such that either simplify software design or define behavior of the systems, or both, and developing mechanisms needed for extended $\mathcal{PFL}$ as aspect $\mathcal{PFL}$, this may contribute to a more general approach to aspect oriented paradigm, that may be applied to different problem areas and different target architectures [5, 28, 29]. Among others, our goal is to provide open language system, based on aspect $\mathcal{PFL}$ which will serve for adaptive definition of new aspects of computation from one side, and will allow to implement woven programs to any programming language. Currently, there exists a bridge from $\mathcal{PFL}$ to Java and Haskell, not however aspect oriented.

The advantage of this approach is that the experiences from different areas, such as Petri nets and process algebras [23, 24], imperative functional programming [19], process functional programming [9, 10, 11, 12, 13, 14, 21, 22, 25, 26, 27], reasoning about the correctness of programs [15, 16], etc. may be studied and they may contribute to aspect oriented programming using uniform language basis.

The aim is to provide the software development methodology, which not just increases the reliability of the systems as a result of better software engineering methodology, but guarantees this reliability rigorously, corresponding to the requirements of a user.

REFERENCES

[1] Andrews, J.: Process-algebraic foundations of aspect oriented programming. http://citeseer.nj.nec.com/andrews01processalgebraic.html, 2001.

[2] Avdicausevic, E., Lenic, M., Mernik, M., Zumer, V.: AspectCOOL: An experiment in design and implementation of aspect-oriented language. ACM SIGPLAN not., December 2001, Vol. **36**, No.12, pp. 84-94.

[3] Avdicausevic Enis, Mernik Marjan, Lenic Mitja, Zumer Viljem. Experimental aspect-oriented language - AspectCOOL. Proceedings of 17th ACM symposium on applied computing, SAC 2002, pp. 943-947.

[4] Hudak, P.: Mutable abstract datatypes - or - How to have your state and munge it too. Yale University, Department of Computer Science, Research Report YALEU/DCS/RR-914, December 1992, revised May 1993

[5] Jelšina, M., Vokorokos, L., Sobota, B.: Parallel Computer Architecture of the MIMD Paradigm, Proc. of the III. Internal Scientific Conference of the Faculty of Electrical Engineering and Informatics, May 2003, Košice, pp. 35-36, ISBN 80-89066-65-8

[6] Kiczales, G. et al: An overview of AspectJ. Lecture Notes in Computer Science, 2072:327-355, 2001.

[7] Kiczales, G. et al: Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, 11th Europeen Conf. Object-Oriented Programming, volume 1241 of LNCS, pp. 220-242, 1997.

[8] Kienzle, J. and Guerraoui, R.: Aspect oriented software development AOP: Does it make sense? The case of concurrency and failures. In B. Magnusson, editor, Proc. ECOOP 2002, pages 37-61. Springer Verlag, June 2002.

[9] Kollár, J.: Process Functional Programming, Proc. ISM'99, Rožnov pod Radhoštěm, Czech Republic, April 27-29, 1999, pp. 41-48.

[10] Kollár, J.: PFL Expressions for Imperative Control Structures, Proc. Scient. Conf. CEI'99, October 14-15, 1999, Herľany, Slovakia, pp.23-28

[11] Kollár, J.: Control-driven Data Flow, Journal of Electrical Engineering, **51**(2000), No.3-4, pp.67-74

[12] Kollár, J.: Comprehending Loops in a Process Functional Programming Language, Computers and Artificial Intelligence, **19** (2000), 373–388

[13] Kollár, J.: Object Modelling using Process Functional Paradigm, Proc. ISM'2000, Rožnov pod Radhoštěm, Czech Republic, May 2-4, 2000, pp.203-208

[14] Kollár, J., Václavík, P., Porubän, J.: The Classification of Programming Environments, Acta Universitatis Matthiae Belii, 10, 2003, pp. 51-64, ISBN 80-8055-662-8

[15] Novitzká, V.: Computer Programming and Mathematics, Fifth International Scientific Conference „Electronics Computers and Informatics´2002“, 10.-11.10.2002, Košice-Herľany, Technická univerzita v Košiciach, 2002, 5, pp. 31-36, ISBN 80-7099-879-2

[16] Novitzká, V.: About the theory of correct programming. February 2003, Elfa s.r.o, Košice, 117pp. (in Slovak)

[17] Novitzká, V.: Mathematical language in programming, Acta Electrotechnica et Informatica, 3, 3, 2003, pp. 31-35, ISSN 1335-8243

[18] Novitzká, V., Kollár, J.: From requirements specification to design specification, Journal of Information, Control and Management Systems, **1**, 2, 2003, pp. 55-64, ISSN 1336-1716

[19] Peyton Jones, S. L., Wadler, P.: Imperative functional programming, In 20th Annual Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993, pp.71-84.

[20] Peyton Jones, S.L., Hughes, J. **[**editors**]**: Report on the Programming Language Haskell 98 - A Non-strict, Purely Functional Language. February 1999, 163 p.

[21] Porubän, J.: Profiling process functional programs. Research report DCI FEII TU Košice, 2002, 51.pp, (in Slovak)

[22] Porubän, J.: Time and space profiling for process functional language, Proceeding of the 7th Scientific Conference with International Participation: Engineering of Modern Electric '03 Systems, May 29-31, 2003, Felix Spa - Oradea, University of Oradea, 2003, pp. 167-172, ISSN-1223-2106

[23] Šimoňák, S., Hudák, Š.: Using Petri Nets and Process Algebra in FDT Interfacing, the Fifth International Scientific Conference „Electronic Computers and Informatics´2002“, October 2002, Košice - Herľany, 2002, pp. 8-13, 80-7099-879-2

[24] Šimoňák, S., Hudák, Š.: APC - Algebra of Process Components, EMES '03, May 29.-31. 2003., Felix Spa, Oradea, 2003, pp. 57-63, ISSN 1223 – 2106

[25] Václavík, P.: Abstract types and their implementation in a processs functional programming language. Research report DCI FEII TU Košice, 2002, 48.pp, (in Slovak)

[26] Václavík, P., Porubän, J.: Object Oriented Approach in Process Functional Language, Proceedings of the Fifth International Scientific Conference „Electronic Computers and Informatics´2002“, October 10.-11. 2002, Košice - Herľany, 2002, pp. 92-96, 80-7099-879-2

[27] Václavík, P.: The Fundamentals of a Process Functional Abstract Type Translation, Proceeding of the 7th Scientific Conference with International Participation: Engineering of Modern Electric '03 Systems, May 29-31, 2003, Felix Spa - Oradea, University of Oradea, 2003, pp. 193-198, ISSN-1223-2106

[28] Vokorokos, L.: Data flow computing model: Application for parallel computer systems diagnosis, Computing and Informatics, **20**, (2001), 411-428

[29] Vokorokos, L.: Faults Diagnosis of Transport Machineries Using the Observer, Transport &

Logistics Journal, **4**, 2003, pp. 23-29, YU, ISSN 1451-107X

[30] Wadler, P.: The essence of functional programming, In 19th Annual Symposium on Principles of Programming Languages, Santa Fe, New Mexico, January 1992, draft, 23 pp.

[31] Wand, M.: A semantics for advice and dynamic join points in aspect-oriented programming. Lecture Notes in Computer Science, **2196**:45-57, 2001.

BIOGRAPHIES

Ján Kollár was born in 1954. He received his MSc. summa cum laude in 1978 and his PhD. in Computing Science in 1991. In 1978-1981 he was with the Institute of Electrical Machines in Košice. In 1982-1991 he was with the Institute of Computer Science at the University of P.J. Šafárik in Košice. Since 1992 he is with the Department of Computers and Informatics at the Technical University of Košice. In 1985 he spent 3 months in the Joint Institute of Nuclear Research in Dubna, Soviet Union. In 1990 he spent 2 month at the Department of Computer Science at Reading University, Great Britain. He was involved in the research projects dealing with the real-time systems, the design of (micro) programming languages, image processing and remote sensing, the dataflow systems, the educational systems, and the implementation of functional programming languages. Currently the subject of his research is process functional paradigm and its application in high performance computing and aspect programming.

Valerie Novitzká defended her PhD. Thesis “On formal semantics of Anna” in 1989 at Hungarian Academy of Sciences in Budapest. She is working as lecturer at the Department of Computers and Informatics FEII Technical university of Košice, Slovakia. Her scientific research is focusing on the theoretical foundations of programming related to the specification, type theory, program correctness and the semantics of programming languages.