POINTCUT DESIGNATORS IN AN ASPECT ORIENTED LANGUAGE

Ján KOLLÁR
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics,
Technical University of Košice, Letná 9, 042 00 Košice, Slovak Republic, tel. 055/602 2577, E-mail: Jan.Kollar@tuke.sk

SUMMARY

The strength of aspect-oriented languages is given by pointcut designators that pick out join points. In this paper we provide an overview of pointcut designators in AspectJ, classifying them with respect of different kinds of joint points. Our aim in the future is to find a general and flexible way of adding a new aspect to an existing language system. The idea behind our approach is the integration of programming paradigms, such that prevents the occasional insufficiency of a programming language when mapping a problem to a corresponding program. Such integration, as we believe, can be done not excluding neither abstract paradigmatic level nor practical programming language level. From this point of view PFL – a process functional language is a perfect basis for studying the aspect phenomenon in a disciplined way as well as for providing practical experiments. In particular, when aspect approach is considered, the goal is not to provide a complete set of defined primitive pointcut designators – we do not think it is possible, since the world of computation may change in the future. Instead of that, more perspective seems to be the determining the different semantic levels of computation and their relation and hierarchy, their sources and the style in which they can be reflected and affected by a programming language. In this matter, this paper is just a step to this research direction.

Keywords: *programming paradigms, imperative functional programming, aspect oriented programming, implementation principles, programming environments, control driven dataflow, referential transparency, side effects*

1. INTRODUCTION

PFL - an experimental process functional language [5,6,7,8,14,16] integrates the semantics of imperative and functional languages. A programmer is free to decide for using functional programming methodology including monadic approach [12,18], but he can also manipulate the memory cells, if appropriate. Then the imperativeness is reached by application of processes "attached" to the cells by their arguments [8]. "Stateful" evaluation by process application in *PFL* is more relaxed and less restrictive to a programmer than exploiting side effects encapsulated by monad.

Both monadic and process functional approaches are the same if reduced to purely functional methodology. They differ in exploiting imperative methodology, although both hide assignments to a programmer [2].

There are two main differences between them; monadic approach uses visible side effecting functions `unit` and `bind`, hiding memory cells to a programmer. Process functional approach is hiding two functions that perform the `access` and the `update` of memory cells, but all memory cells are visible to a programmer.

The strength of process functional approach is a paradigm that reflects the implementation of both imperative and functional languages bringing it to the source form. It means that each *PFL* program is a highly abstracted expression, which allows to perform source-to-source transformations instead of machine-independent optimization techniques that are well known using directed acyclic graphs and quadruples in imperative languages [9,10,20]. Since semantic information such as binding names to identifiers is not missing using *PFL* expressions, this supports the requirement for source-to-source transformations as desired for the implementation of aspect-oriented languages [1,3,4,19].

On the other hand, less positive is the use of process functional language as a "programming language". Seemingly, its level of abstraction is higher than that of an imperative language, but the methodology of performing side effects by application of processes is still less natural than using assignments. Using *PFL*, much useless control is eliminated, but the integration of just functional and imperative paradigms is evidently insufficient to break the non-conformance of problems in one side and "programs" on the other side. The weakness is that *PFL* flexibility is hardly to exploit practically since of insufficient methodology.

As a possible solution to this problem is an extension of process functional to aspect process functional paradigm. Aspect programming methodology (integrating logic and imperative programming) is more general than object oriented approach [7,16,20] as well as multi-paradigmatic approaches, such as concurrent constraint programming [11,15], imperative functional programming [13], and others.

The crucial role in aspect-oriented programming languages play pointcut designators, which we discuss in this paper, as used in AspectJ programming language. The aim of this paper is to provide a systematic but still informal overview of pointcut designators as a basis to their formal analysis in the future and an extension using process functional paradigm.

2. ASPECT PARADIGM AND ASPECT LANGUAGE

The motivation for aspect-oriented programming is the realization that there are issues or concerns that are not well captured by traditional programming methodologies.

For object-oriented programming languages, the natural unit of modularity is the class. But in object-oriented programming languages, crosscutting concerns are not easily turned into classes precisely because they cut across classes, and so these they aren't reusable, they can't be refined or inherited, they are spread through out the program in an undisciplined way, in short, they are difficult to work with.

Aspect-oriented programming is a way of modularizing crosscutting concerns much like object-oriented programming is a way of modularizing common concerns.

AspectJ is an implementation of aspect-oriented programming for Java. AspectJ adds to Java just one a new concept, a join point, and a few new constructs: pointcuts, advice, introduction and aspects. Pointcuts and advice dynamically affect program flow, and introduction statically affects a program's class hierarchy.

A *join point* is a well-defined point in the program flow. *Pointcuts* select certain join points and values at those points. *Advice* defines code that is executed when a pointcut is reached. These are, then, the dynamic parts of AspectJ.

AspectJ also has a way of affecting a program statically. *Introduction* is how AspectJ modifies a program's static structure, namely, the members of its classes and the relationship between classes.

The last new construct in AspectJ is the *aspect*. Aspects, are AspectJ's unit of modularity for crosscutting concerns They are defined in terms of pointcuts, advice and introduction.

AspectJ advices are expressions (in most cases of unit type) that are advised to be executed before, after or instead other expressions code parts, depending on pointcut designators.

Then AspectJ advice `a` would be expressed in *PFL* style in the next form:

```
advice :: T1 → T2 → … → Tm → T
advice x1 x2 … xm = e[x1, x2, … , xm]
    (before | after | around)
        p[x1, x2, … , xm]
```

which designates the set of constant expressions $e[x_1, x_2, \ldots, x_m]$ selected for join points picked out by pointcut designator

$$p[x_1, x_2, \ldots, x_m]$$

This pointcut uses variables $x_1, x_2, \ldots, x_m$ which are substituted by the values (that usually differ for different join points) and are used by expression $e[x_1, x_2, \ldots, x_m]$ — the advice.

The crucial role of pointcut designators is evident, because after a join point and a set of values

$$[c_1, c_2, \ldots, c_m]$$

are selected, there is no problem to insert before or after a join point or replace the expression forming a join point (in case of around advice) by constant expression which is obtained by the application

$$(\lambda\, x_1\, x_2 \ldots x_m.\ e[x_1, x_2, \ldots, x_m])\ c_1, c_2, \ldots, c_m$$

performed in the compile time.

The detailed analysis of pointcut designators in this paper is informal. We decided for this approach for these reasons: Instead of detailed formal semantics in the whole, AspectJ documentation is oriented to explanation of many examples, which sometimes make more blur than appropriate. Although formal semantics is often available but just for particular constructs, such as in [19], this is insufficient for our purposes. Before we introduce poincut designators (also called poincuts) we will deal with join points as classified in AspectJ system.

3. JOIN POINTS

While aspects do define crosscutting types, the AspectJ system does not allow completely arbitrary crosscutting. Rather, aspects define types that cut across principled points in a program's execution. These principled points are called join points. A join point is a well-defined point in the execution of a program. The join points defined by AspectJ are:

Method call
When a method is called, not including super calls.

Method execution
When the body of code for an actual method executes.

Constructor call
When an object is built and a constructor is called, not including this or super constructor calls.

Constructor execution
When the body of code for an actual constructor executes, after its this or super constructor call.

Initializer execution
When the non-static initializers of a class run.

Static initializer execution
When the static initializer for a class executes.

Object pre-initialization
Before the object initialization code for a particular class runs. This encompasses the time between the start of its first called constructor and the start of its parent's constructor. Thus, the execution of these join points encompass the join points from the code found in `this()` and `super()` constructor calls.

Object initialization
When the object initialization code for a particular class runs. This encompasses the time between the return of its parent's constructor and the return of its first called constructor. It includes all the dynamic initializers and constructors used to create the object.

Field reference
When a non-final field is referenced.

Field assignment
When a field is assigned to.

Handler execution
When an exception handler executes.

4. BASIC PRIMITIVE POINTCUTS

Corresponding to join points introduced in preceding section, AspectJ primitive pointcut designators (primitive pointcuts) are classified as follows:

- Method and Constructor-related pointcuts
- Object creation-related pointcuts
- Class initialization-related pointcuts
- Field-related pointcuts
- Exception handler execution-related pointcuts

One very important property of a join point is its signature, which is used by many of AspectJ's pointcut designators to select particular join points.

Method-related pointcuts

AspectJ provides two primitive pointcut designators designed to capture method call and execution join points.

`call(`*Signature*`)`
`execution(`*Signature*`)`

At a method call join point, the *Signature* is composed of the type used to access the method, the name of the method, and the types of the called method's formal parameters and return value (if any).

At a method execution join point, the signature is composed of the type defining the method, the name of the method, and the types of the executing method's formal parameters and return value (if any).

Formally, *Signature* is the method pattern *MethodPat*, in the form:

```
[ModifiersPat] TypePat [TypePat . ]
IdPat ( TypePat | .. , … )
   [ throws ThrowsPat ]
```

ModifiersPat (modifiers pattern) may be a keyword, such as **`private`**, **`public`**, or **`static`**. Another wildcard "**`..`**" is used to designate any number of type patterns, each *TypePat* is one of:

```
IdPat [ + ] [ [] … ]
! TypePat
TypePat && TypePat
TypePat || TypePat
( TypePat )
```

Here "**`+`**" denotes all subtypes and "**`[]`**" denotes array patterns.

Further, operators "**`!`**", "**`&&`**", and "**`||`**" are boolean operators not, and, and or, respectively.

In *IdPat* (the identifier pattern), the "`*`" wildcard matches zero or more characters except for ".".

The second meaning of "**`..`**" wildcard is that it matches any sequence of characters that start and end with a ".", so it can be used to pick out all types in any subpackage, or all inner types.

ThrowsPat is a name of an exception handler being thrown when a method fails its execution yielding an exception.

Both two pointcuts above also pick out constructor call end execution join points.

Object creation-related pointcuts

AspectJ provides three primitive pointcut designators designed to capture the initializer execution join points of objects.

`call(`*Signature*`)`
`execution(`*Signature*`)`
`initialization(`*Signature*`)`

At a constructor call join point, the signature is composed of the type of the object to be constructed and the types of the called constructor's formal parameters.

At a constructor execution join point, the signature is composed of the type defining the constructor and the types of the executing constructor's formal parameters.

At an object initialization join point, the signature is composed of the type being initialized and the types of the formal parameters of the first constructor entered during the initialization of this type.

Formally, *Signature* is the constructor pattern *ConstructorPat*, in the form:

```
[ModifiersPat ] [TypePat . ]
 new ( TypePat | .. , …)
   [ throws ThrowsPat ]
```

Class initialization-related pointcuts

AspectJ provides one primitive pointcut designator to pick out static initializer execution join points.

`staticinitialization(`*TypePat*`)`

Field-related pointcuts

AspectJ provides two primitive pointcut designators designed to capture field reference and assignment join points:

`get(`*Signature*`)`
`set(`*Signature*`)`

At a field reference or assignment join point, the *Signature* is composed of the type used to access or assign to the field, the name of the field, and the type of the field.

Formally, the *Signature* is the field pattern *FieldPat*, in the form:

`[`*ModifiersPat*`] `*TypePat*` [`*TypePat*` . ] `*IdPat*

All set join points are treated as having one argument, the value the field is being set to, so at a set join point, that value can be accessed with an `args` pointcut.

Exception handler execution-related pointcuts

AspectJ provides one primitive pointcut designator to capture execution of exception handlers:

`handler(`*TypePat*`)`

At a handler execution join point, the signature is composed of the exception type that the handler handles.

All handler join points are treated as having one argument, the value of the exception being handled, so at a handler join point, that value can be accessed with an `args` pointcut, introduced in the next section.

Except pointcuts above, other primitive pointcuts are provided, as introduced in the next section.

5. OTHER PRIMITIVE POINTCUTS

Other primitive pointcuts are as follows:

- State-based pointcuts
- Program text-based pointcuts
- Dynamic property-based pointcuts

State-based pointcuts

Many concerns cut across the dynamic times when an object of a particular type is executing, being operated on, or being passed around. AspectJ provides primitive pointcuts that capture join points at these times. These pointcuts use the dynamic types of their objects to discriminate, or pick out, join points. They may also be used to expose to advice the objects used for discrimination.

`this(`*TypePat* `or` *Id*`)`
`target(`*TypePat* `or` *Id*`)`

The this pointcut picks out all join points where the currently executing object (the object bound to `this`) is an instance of a particular type. The target pointcut picks out all join points where the target object (the object on which a method is called or a field is accessed) is an instance of a particular type.

`args(`*TypePat* `or` *Id* `or "..", ...`)

The args pointcut picks out all join points where the arguments are instances of some types. Each element in the comma-separated list is one of three things. If it is a type pattern, then the argument in that position must be an instance of a type of the type name. If it is an identifier, then the argument in that position must be an instance of the type of the identifier (or of any type if the identifier is typed to Object). If it is the special wildcard `".."`, then any number of arguments will match, just like in signatures. So the pointcut

```
args(int, .., String)
```

will pick out all join points where the first argument is an `int` and the last is a `String`.

Control flow-based pointcuts

Some concerns cut across the control flow of the program. The cflow and cflowbelow primitive pointcut designators capture join points based on control flow.

`cflow(`*Pointcut*`)`

The cflow pointcut picks out all join points that occur between the start and the end of each of the pointcut's join points.

`cflowbelow(`*Pointcut*`)`

The cflowbelow pointcut picks out all join points that occur between the start and the end of each of the pointcut's join points, but not including the initial join point of the control flow itself.

Program text-based pointcuts

While many concerns cut across the runtime structure of the program, some must deal with the actual lexical structure. AspectJ allows aspects to pick out join points based on where their associated code is defined.

`within(`*TypePat*`)`

The within pointcut picks out all join points where the code executing is defined in the declaration of one of the types in *TypePat*. This includes the class initialization, object initialization, and method and constructor execution join points for the type, as well as any join points associated with the statements and expressions of the type. It also includes any join points that are associated with code within any of the type's inner types.

`withincode(`*Signature*`)`

The withincode pointcut picks out all join points where the code executing is defined in the declaration of a particular method or constructor. This includes the method or constructor execution join point as well as any join points associated with the statements and expressions of the method or constructor. It also includes any join points that are associated with code within any of the method or constructor's local or anonymous types.

Dynamic property-based pointcuts

`if(`*BooleanExpression*`)`

The if pointcut picks out join points based on a dynamic property. It's syntax takes an expression, which must evaluate to a boolean true or false. Within this expression, the `thisJoinPoint` object is available. So one (extremely inefficient) way of picking out all call join points would be to use the pointcut

```
if(thisJoinPoint.getKind().equals("
call"))
```

6. FORMULAS ON POINTCUTS

Primitive (and also non-primitive pointcuts) are combined using logical formulas, in the form as follows.

`! Pointcut`
picks out all join points that are not picked out by the pointcut.

`Pointcut0 && Pointcut1`
picks out all join points that are picked out by both of the pointcuts.

`Pointcut0 || Pointcut1`
picks out all join points that are picked out by either of the pointcuts.

`( Pointcut )`
picks out all join points that are picked out by the parenthesized pointcut.

It can be noticed that boolean operators are used to combined pointcuts, not type patterns, as it is in type patterns.

7. POINTCUT NAMING AND USING

Pointcut naming

A named pointcut is defined with the `pointcut` declaration.

```
pointcut PointcutId(Type Id, …):
   Pointcut;
```

A named pointcut may be defined in either a class or aspect, and is treated as a member of the class or aspect where it is found. As a member, it may have an access modifier such as `public` or `private`.

```
class C {
   pointcut publicCall(int i):
       call(public * *(int)) &&
       args(i);
}

class D {
   pointcut myPublicCall(int i):
       C.publicCall(i) &&
       within(SomeType);
}
```

Pointcuts that are not final may be declared abstract, and defined without a body. Abstract pointcuts may only be declared within abstract aspects.

```
abstract aspect A {
   abstract pointcut
       publicCall(int i);
}
```

In such a case, an extending aspect may override the abstract pointcut.

```
aspect B extends A {
   pointcut publicCall(int i):
       call(public Foo.m(int)) &&
       args(i);
}
```

For completeness, a pointcut with a declaration may be declared `final`.

Though named pointcut declarations appear somewhat like method declarations, and can be overridden in subaspects, they cannot be overloaded. It is an error for two pointcuts to be named with the same name in the same class or aspect declaration.
The scope of a named pointcut is the enclosing class declaration. This is different than the scope of other members; the scope of other members is the enclosing class *body*.

Context exposure

Pointcuts have an interface; they expose some parts of the execution context of the join points they pick out. In this case formula `Pointcut` in

pointcut declaration above exposes the arguments Id. This context is exposed by providing typed formal parameters to named pointcuts and advice, like the formal parameters of a Java method. These formal parameters are bound by name matching. On the right-hand side of advice or pointcut declarations, a regular Java identifier is allowed in certain pointcut designators in place of a type or collection of types. There are primitive pointcut designators available, where this is allowed: `this`, `target`, and `args`. In all such cases, using an identifier rather than a type is as if the type selected was the type of the formal parameter, so that the pointcut

```
pointcut intArg(int i): args(i);
```

picks out join points where an `int` is being passed as an argument, but furthermore allows advice access to that argument. Values can be exposed from named pointcuts as well, so

```
pointcut publicCall(int x):
call(public *.*(int)) && intArg(x);

pointcut intArg(int i): args(i);
```

is a legal way to pick out all calls to public methods accepting an int argument, and exposing that argument.

There is one special case for this kind of exposure. Exposing an argument of type Object will also match primitive typed arguments, and expose a "boxed" version of the primitive. So,

```
pointcut publicCall(): call(public
  *.*(..)) && args(Object);
```

will pick out all unary methods that take, as their only argument, subtypes of Object (i.e., not primitive types like `int`), but

```
pointcut publicCall(Object o):
  call(public *.*(..)) && args(o);
```

will pick out all unary methods that take any argument: And if the argument was an `int`, then the value passed to advice will be of type `java.lang.Integer`.

Pointcut using

```
PointcutId(TypePattern or Id, ...)
```

picks out all join points that are picked out by the user-defined pointcut designator named by *PointcutId*.

8. EXAMPLES

The difference between call and execution join points is as follows: Firstly, the lexical pointcut declarations within and withincode match differently. At a call join point, the enclosing code is that of the call site. This means that

```
call(void m()) &&
withincode(void m())
```

will only capture directly recursive calls, for example. At an execution join point, however, the program is already executing the method, so the enclosing code is the method itself:

```
execution(void m()) &&
withincode(void m())
```

is the same as

```
execution(void m())
```

Secondly, the call join point does not capture super calls to non-static methods. This is because such super calls are different in Java, since they don't behave via dynamic dispatch like other calls to non-static methods.

Next example illustrate the use of wildcard * and modifiers.

```
call(public final void *.*() throws
  ArrayOutOfBoundsException)
```

picks out all call join points to methods, regardless of their name name or which class they are defined on, so long as they take no arguments, return no value, are both `public` and `final`, and are declared to throw `ArrayOutOfBounds` exceptions.

The defining type name, if not present, defaults to *, so another way of writing that pointcut would be

```
call(public final void *() throws
  ArrayOutOfBoundsException)
```

Formal parameter lists can use the wildcard `..` to indicate zero or more arguments, so

```
execution(void m(..))
```

picks out execution join points for void methods named `m`, of any number of arguments, while

```
execution(void m(.., int))
```

picks out execution join points for void methods named `m` whose last parameter is of type `int`.

```
withincode(!public void foo())
```

picks out all join points associated with code in null non-public void methods named `foo`, while

```
withincode(void foo())
```

picks out all join points associated with code in null void methods named `foo`, regardless of access modifier.

`call(int *())`
picks out all call join points to `int` methods regardless of name.

`call(int get*())`
picks out all call join points to `int` methods where the method name starts with the characters "get".

`call(Foo.new())`
picks out all constructor call join points where an instance of exactly type Foo is constructed,

`call(Foo+.new())`
picks out all constructor call join points where an instance of any subtype of `Foo` (including `Foo` itself) is constructed, and the unlikely

`call(*Handler+.new())`
picks out all constructor call join points where an instance of any subtype of any type whose name ends in "Handler" is constructed.

`Object[]` is an array type pattern, and so is `com.xerox..*[][]`, and so is `Object+[]`.

`staticinitialization(Foo || Bar)`
picks out the static initializer execution join points of either Foo or Bar, and

`call((Foo+ && ! Foo).new(..))`
picks out the constructor call join points when a subtype of Foo, but not Foo itself, is constructed.

9. CONCLUSION

Except some inaccuracies in AspectJ definition, such as the ability for use multiple modifiers such as "`public final`" which does not correspond to syntax in section 4, we may notice the ambiguity of boolean operators (operands may be either type patterns or pointcuts) the ambiguity of wildcard "`..`" which designate any number of arguments but also any sequence of qualifiers (`A..` designate `A.B. A.B.C.` etc.)

Using PFL we can exclude each initialization, provided that we initialize object by default.

We are able to exclude field manipulation poincuts `set` and `get`, because we manipulate environment variables indirectly.

Instead of `call` and `execution` it would be probably better to thing about an `application` as a common pointcut.

Great simplification is omitting all modifiers, such as public, private static, final, etc. that come out from imperative organizing a memory cells. In fact, static cells are just those associated with architecture resources, but then static without exact memory positions is still not sufficient.

Then, of course, it is substantial to deal with not just user organization of his application but also with time and space resources of computation. Hence, defining physical time and space aspects of computation may affect building embedded systems in the future significantly. In particular, it is clear that control flow poincuts are insufficient since of the existence of the second mirroring principle in computation which is data flow [17].

We are not sure, if it is possible to make the combining of different pointcuts more clear. We just recognize experimental basis as wrong. It was the reason why we decided to give attention to pointcuts in AspectJ as a basis for further detailed analysis and extension, based however on process functional language. Its uniform concept of modules, polymorphic classes with multiple superclasses, instances, objects as an application of classes to expressions provide us with simple basis for performing such a task. This however is the future.

REFERENCES

[1] 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.

[2] 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.

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

[4] 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. Springer Verlag, 1997.

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

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

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

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

[9] Mernik, M., Zumer, V.: Incremental language design. IEE Proc. Soft. Eng., April-June 1998, 145, pp. 85-91.

[10] Mernik, M., Lenic, M., Avdicausevic, E., Zumer, V.: A reusable object-oriented approach to formal specification of programming languages. L'Objet, 1998, Vol.4, No.3, pp. 273-306.

[11] Paralič, M.: Mobile Agents Based on Concurrent Constraint Programming, Joint Modular Languages Conference, JMLC 2000, September 6-8, 2000, Zurich, Switzerland. In: Lecture Notes in Computer Science, 1897, pp. 62-75.

[12] 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.

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

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

[15] Smolka, G.: The Oz programming model, In Jan van Leeuwen, editor, Computer Science Today, Lecture Notes in Computer Science 1000, Springer-Verlag, Berlin, 1995, pp. 324-343.

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

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

[18] 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.

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

[20] Zumer, V., Korbar, N., Mernik, M.: Automatic Implementation of Programming Languages using Object Oriented Approach. Journal of System Architecture, 1997, Vol.43, No.1-5, pp. 203-210.

BIOGRAPHY

Ján Kollár (Assoc. Prof.) 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 extension to aspect paradigm.