TEMPORAL LOGIC FOR POINTCUT DEFINITIONS IN AOP

Ján KOLLÁR, Marcel TÓTH
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, Marcel.Toth@tuke.sk

SUMMARY

The aim of this paper is to propose the use of temporal logic in dynamic weaving of advices in aspect oriented programming (AOP), based on process functional paradigm. Namely, temporal logic appears to be well suited tool to design pointcuts whose task is selection of join points throughout source code (static weaving) or program execution (dynamic weaving). Because it is well known that join point model and techniques used for selecting them (pointcuts) are crucial elements of aspect oriented programming, we tried to propose the way of better (at least we think it is better for describing dynamic join points) pointcut formulation in terms of temporal logic. By now we have developed process functional language (PFL), which is simple but yet powerful abstraction used to illustrate mechanisms of programming languages. PFL includes structures and principles supporting functional, imperative and even object oriented and aspect oriented paradigms (which is still under development), so it is multi paradigmatic language. This paper is presenting reflection as a basic property of PFL used as a tool for weaving semantics of aspect code into original code. Code examples in this paper use PFL, but some examples are written also in AspectJ to clarify informal semantics of its operations (particularly cflow operation, since it is used for selecting dynamic join points).

Keywords: *aspect oriented programming, dynamic join points, static weaving, dynamic weaving, temporal logic, process functional programming*

1. INTRODUCTION

Current implementations of AOP are based on compile-time, load-time and also run-time modification of the application code or control flow of application. As has been observed, this whole spectrum of binding times is needed to cover needs of AOP [9,24,29]. Particularly, dynamic composition (weaving) brings advantages, e.g. adaptation of application to new runtime condition needs without re-compiling, re-deployment and re-starting of application [24,34].

According to characteristic of aspect-oriented programming [5,9], the aim of AOP is clear separation of concerns in program code or in execution flow of the program, thus allowing better software development in all stages of software life cycle (not only in the implementation stage) [13,14,15,16,17,18].

Temporal logic is the term broadly used in the area of computer science, because with use of temporal logic one can easily express temporal relations between instructions and sequences of instructions. However, temporal logic is mostly used for program specification and verification, we can say, for mathematical or formal representation of software systems. Nevertheless, with respect to long-term unification of programming languages and paradigms as well as the convergence of formal specifications and their refinements, our effort is to use temporal logic for pointcuts definition in AOP. Since pointcut's task is to select join points in dependence on execution flow (or control flow), we think that temporal logic offers better apparatus for low-level definition of pointcuts than building of pointcuts from pre-defined blocks (pointcut designators [4,10,23,24,34]).

2. REFLECTION IN PFL

Process functional language (PFL) is an experimental functional programming language with functional syntax and semantics incorporating imperative and functional denotation [6,8,21,24]. We use PFL to formulate our ideas, to show intended and unwanted attributes and behaviors of our language structures propositions (and their semantics). PFL as a functional language from its core is advantageous to imperative languages in that it is very transparent in its execution mechanism. All actions in *PFL* are carried out through an application of process or a function. This uniform execution system facilitates well-arranged interception of join points when used as a root for aspect oriented extension. This advantage what was already tested on profiling of PFL [25].

Let us define function **f** as the small example of PFL code snippet

```
f :: Int -> Int -> Int
f x y = x + y
```

First line of code is type definition, such that both arguments and the value of **f** are of the type Int (integer numbers). Presented definition is purely functional without use of any environment variables, which are important part of *PFL* allowing inner and outer reflection of parameters (`x` and `y`). In addition to pure functional functions, *PFL* offers processes, which can access environment variables indirectly - exploiting the outer reflection property. Let us explain this property using a small example. The process **f** is defined, as follows:

```
f :: v Int -> v Int -> Int
f x y = x + y
```

and its applications are as follows:

```
f 2 3            (I)
f 4 ()           (II)
f () ()          (III)
```

The type **v** `Int` means, that the memory cell for the environment variable **v** is allocated and accessible to be used in a reflective manner. The application (I) updates the environment variable **v** with the value 2 and then with the value 3. The process **f** is evaluated as the function, as follows:

```
f :: Int -> Int -> Int
f x y = x + y
```

The application (II) updates environment variable **v** with the value 4 first, and then it uses the unit value to access this variable, so it is equivalent to the application, as follows:

```
f 4 4
```

The application (III) just accesses environment variable **v** for both parameters and uses values previously stored in environment variable **v.**

The inner reflection (opposite to outer reflection) is a property of a function/process, which, when defined using a local process, can reflect its arguments in environment variables of this function/process. Let us consider a definition, as follows:

$$\mathbf{f}\ x_1 \ldots x_n = e$$
$$\text{where } g :: T_1` \text{->} \ldots \text{->} T_m` \text{->} T`$$
$$g\ y_1` \ldots y_m` = e`$$

If $T_k` = v_k\ T_k$, where $v_k = x_i$, then the value of x_i is reflected in v_k local to **f**.
For example, the definition of **f**, as follows

```
f x y = x + y
```

is semantically equivalent to the definition:

```
f x y = g () ()
      where
      g :: x Int -> y Int -> Int
      g u v = u + v
```

Since the names for environment variables `x` and `y` are identical to names of lambda variables of the function **f**, the arguments `e1` and `e2` used in an application `(f e1 e2)` are automatically present in environment variables `x` and `y` and they are accessed by application `(g () ())`. In the definition below, just `x` lambda variable is reflected. The environment variable `z` is the environment variable local to **f** and external to `g` in which the value of `y` is reflected by the application `g () y)`. The application `(g () ())` would yield undefined value for `v`, since `z` is not initialized.

```
f x y = g () y
      where
      g :: x Int -> z Int -> Int
      g u v = u + v
```

Notice also that the definition of `g` could be written using `x` an `y` lambda variables, i.e. in the form

```
g x y = x + y
```

since they do not clash with neither environment variables of `g` nor with the lambda variables of `f`. Illustrated reflection property can be utilized for implementation of weaving advice code to original semantics.

3. ASPECT ORIENTED PROGRAMMING

AOP is a programming style providing programmers with clear and transparent separation of concerns that are problematic to isolate in existing programming paradigms. The idea of AOP, although quite new programming methodology, has already demonstrated its true power through its integration to several common imperative programming languages. Anyway, actual AOP implementations have several weaknesses:

- Current AOP frameworks are rather extensions to their respective programming languages than true integrations with them. Examples are AspectJ, AspectC++, Spring, etc. [4,29].
- There is no effective and fully transparent way of selecting dynamic join points and weaving in them. There are some efforts, for example cflow operation in AspectJ [4,5] or PROSE's approach to run-time dynamic weaving [24], but generally the problem is more related to the static nature of whole weaving mechanism than to operations themselves.

4. STATIC VS. DYNAMIC WEAVING

AOP uses two methods of composing semantics of aspects and semantics of original program:

- Static composition (weaving)
- Dynamic weaving

Static composition is simply accomplished by source-to-source code transformation.
The advantages of static weaving comprise

- relatively simple implementation
- clear separating of concerns in isolated source code segments

- insignificant overhead of code execution

On the other hand dynamic composition is about run-time composing of aspect's semantics and original program semantics. Dynamic composition is very useful for debugging, profiling, and fine tuning of existing applications, as well as for adapting of applications to new requirements.

Dynamic weaving unlike static weaving, needs an apparatus to which it can attach advice code (way of identifying of join points within running application). Virtually, some hooks in program execution, since source code is not accessible during program execution. Thus, dynamic composition can be divided into two categories:

- load time approaches
- run-time approaches

It is obvious, that e.g. run-time dynamic weaving module cannot be just simple source-to-source transformer. Run-time dynamic weaving needs the structures in execution flow (not in source code) appropriate to attach an advice code.

For these reasons, the implementation of dynamic weaving is only possible when component code (code the advices will be woven into) has at least one of the following properties:

- Is is written in an interpreted language
- It written is in language translated to an intermediate form
- Program constructions are wrapped to something with "hooks" (classes, etc.)

It is important to distinguish between the notions of dynamic composition and dynamic set of join points. Dynamic join points can be selected (and thus used for weaving) with the use of just static composition, which works with not as strong mechanism as dynamic composition, but the solution is reasonably simpler. Let us show one example of dynamic join points:

```
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
```

The result having been reached in the execution of `main` in the example 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]
[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. This is, why different selecting mechanism is needed.

According to [8], it is possible to select a superset of dynamic join points statically, and then to weave advices dynamically, depending on run-time conditions.

5. TEMPORAL LOGIC

Introducing PFL language and its reflection property and the basic AOP principles above, let us show how temporal logic can exploited considering process functional paradigm.

Temporal logic is an extension of ordinary logic to include certain kinds of assertions about the time (past or future) or operations order (in computer science). Temporal logic is rooted in the field of exact philosophy and is a variant of modal logic. Modal logic deals with propositions that are interpreted with respect to a set of possible worlds. The truth value of propositions depends on the respective world and basically two operations "necessarily" and "possibly" exist which denote that a proposition is true in all possible worlds respective in some possible worlds. Here is one example of world (or time) dependent expression:

"Some **M**an exists who **S**tood on the Moon"

which is equivalent to

$$\exists x(Mx \& Sx)$$

where M and S are predicates "is Man" and "Stood on the moon"

This is world (time) independent proposition that always holds, which is true in this (present) world, but not in the world one hundred years ago. If this proposition has to consider all possible worlds, we have to use predicate **E** with the reading „actually exists" (exists in this time, or in this world)

$$\exists x(\mathbf{Ex\&}Mx\&Sx)$$

Therefore, as we can see, there is a set of possible worlds (possibly infinite). An ordered set of possible worlds can correspond to a temporal sequence of states in temporal logic. In result, the two basic modal operations "necessarily" and "possibly" become the temporal quantifiers "always (henceforth)" and "eventually". Based on the linearity of time, the additional operations like "next" and "until" as well as "past" operations were introduced [10,11,12]. To explain the notion of linear time we must mention, that there are different ways of viewing future of the present time or state, (or possible world in modal logic), depending upon one's conception of the nature of time. There are two different ways of viewing possible futures: the theories of **branching time** and **linear time.**

In the branching time theory, all of the possible futures are equally real (e.g. in a nondeterministic system, the present does not determine a unique future, but rather a set of possible futures). Thus, in branching time theories, possible futures create the tree of possible futures, in which every branch has equal likelihood of happening.

In the theory of linear time, at each instant there is only one future that will actually occur. All assertions are interpreted as statements about that one real future.

Depending on which theory is used, i.e. how possible futures are considered, the semantics of temporal operations is significantly different [10,11]. According to [11], linear time theory is more suitable for concurrent systems and branching time theory for non-deterministic systems. All the temporal operations listed in this paper are meaningful for linear time temporal logics, as derived in [12], so we will consider this one.

6. TEMPORAL OPERATIONS

In the next table we introduce the set of temporal operations, that formal definition and proof system can be found in [11].

Temporal logic operator	Description (name)
$\Box$	HENCEFORTH
$\Diamond$	*Eventually*
$\boxminus$	*HasAlways*
$\Diamond\!\!\!-$	*Once*
$\bigcirc$	*StrongNext*
$\ominus$	*StrongPrevious*
$\tilde{\bigcirc}$	*WeakNext*
⊜	*WeakPrevious*
$\mathcal{S}$	*StrongSince*
$\mathcal{U}$	*Until*
$\mathcal{B}$	*WeakSince*
$\mathcal{W}$	*WeakUntil (unless)*

Informally, their semantics is as follows (we will use names instead of symbolic form for operators, and prefix form for binary operators):

Henceforth P:
P holds now and will always hold in the future.

Eventually P:
P will be true sometimes in the future.

HasAlways P:
P was always true.

Once P:
P was true once in the past.

StrongNext P:
This assertion holds if there exist a previous step and *P* was true in the previous step (state). *StrongNext* differentiates from weak next in requiring existence of the next state.

StrongPrevious P:
This assertion holds if there exist a next position (state, step) and *P* holds there. This operator is past equivalent of *StrongNext*.

StrongPrevious P
holds if *P* will be true in the next step. If there is no next step, this assertion always holds, that is: last state in the sequence satisfies *StrongPrevious P* for any *P*.

WeakPrevious P
holds if *P* was true in the previous step or if there is no previous step (state).

Since P Q
holds if *P* held continuously since the previous occurrence of Q, which is guaranteed to have happened.

`Until P Q`
holds if `P` holds continuously until the next occurrence of `Q`, which is guaranteed to happen.

`WeakSince P Q`
holds at the state `j` if `P` holds continuously at all states less than or equal to `j`, either to the beginning of the sequence, or at least to the first preceding occurrence of `Q`.

`WeakUntil P Q`
holds at some position (state), if `P` holds continuously at all positions greater or equal to this position (state), either to the end of sequence, or at least to the first occurrence of `Q`.

In the following, we will use just `Once` operator, designating it by ♦, and `StrongPrevious` operator designating it by ●.

It is true, that every operation is derivable from the basic set of them, but this fact is not important for what we want to show. For full-range definitions, theorems, rules and whole proof systems, we recommend [10,11,12,20,21,23,30].

7. TEMPORAL OPERATIONS IN DEFINITION OF POINTCUTS

In the following lines, we are going to reason about what temporal operations (if any) could be useful in pointcut definitions. There is a strong feeling that at least semantics of some temporal operations is suitable for pointcut designators definition. We will show why pointcut designator definition is useful and how it can help on the simplified example of AspectJ's cflow operation [4]:

```
Pointcut matchF():
    call ( * f(..) ) &&
    cflow ( call ( * one (..) ) &&
            call( * two (..) ) );
```

Here we find the `matchF` pointcut, where a match is made against a method **`f()`** (with any parameters, any type and any returning value). Since **`cflow`** is execution flow operation, then by combining calls to both **`one()`** and **`two()`** in the same **`cflow`** operation we tell the system to match only when **`f()`** method occurs in the execution flow of both **`one()`** and **`two()`** methods. The only way the execution can occur in both methods is when the **`one()`** method calls **`two()`** method or the **`two()`** method calls the **`one()`** method. Let us say that the sign => means "calls", then there are two potential options of execution flow that will our pointcut **`matchF`** match:

```
one() => two() => f()
```

or

```
two() => one() => f()
```

The problem is, that if both execution flows occur in the program (and we want to match just one of them), we are not able to tell the system, which of these two options is the right one for us. Since we know the temporal logic is the logic for expressing order in sequences of states, we can use this piece of knowledge. At this point we can see the benefit of the use of temporal operations: clear and transparent selection of operations order in execution flow, that is, which joinpoint should occur first and which later. As we have mentioned already, PFL language that we have developed suits our needs very well. Particularly, in PFL as well as in all functional languages, everything is an application of function (or process). In imperative languages, there are many syntactic construcs requiring many kinds of pointcut designators (for efficient AOP weaving mechanism), with rather different semantics, which is not uniform core, that we need. Then we can exploit uniformity of functional language (as we mentioned before, PFL's reflection offers also imperative semantics, so we are not losing anything) to transparent selection of join points. Our aim is to define dynamic pointcut designators, in the form

```
on event
        event based pattern
```

To make it the simplest possible way, let us suppose that desired events are:

- start of function (process) evaluation
- end of function (process) evaluation

Expressed in way that is more formal:

`start(f)` - event of starting of function *`f`* evaluation
`end(f)` – event of finishing of function *`f`* evaluation

These are the most important events in the execution of PFL program. The question is, which of temporal operations are the expressive enough to solve our problem. As long as the program runs (is evaluated), it is easy to use states or values already evaluated, so past part of temporal logic (operations) will facilitate the task more than future fragment of temporal logic (at least, it is more conceivable to relate our expressions to something that already happened, than to something that is still unsure to happen). We will use two temporal operations now, to propose ideas for later research:

♦ – once in the past
● – strong previous

The use of ♦ operation is without any problems. On the other hand, ● operation is not real time operation, but rather the operation for expressing discrete instants. In program execution, ● could mean an instruction, which was executed right before currently executed instruction. So far, we have considered just events of starting and ending of function evaluation. Now, let us consider every event as a state change. Hence, the state sequence is just the sequence of events `start(f)` and *`end(f)`*. This simplification enables the use of operation ●.

We will show below, how to express the phases of evaluation of function *f* using temporal operation ♦ (once in the past).

$$[\blacklozenge(\mathit{start(f)})] \wedge [\neg\blacklozenge(\mathit{end(f)})] \qquad (1)$$

i.e., the execution flow is in the function ***f*** right now (function ***f*** is being evaluated).

The assertion (1) states this: Once in the past, it was true, that event of starting of function ***f*** occurred. In addition, it is not true that once in the past event of finishing of function ***f*** evaluation occurred. Therefore, we are in evaluation of function ***f*** now.

$$[\blacklozenge\ (\mathit{start(f)})\ ] \wedge [\blacklozenge\ (\mathit{end(f)})\ ] \qquad (2)$$

i.e. the function *f* was evaluated already (in the past).

This means, that the event `start(f)` occurred as well as the event of *`end(f)`*, what stands for finished function ***f*** execution

$$[\neg\blacklozenge\ (\mathit{start(f)})\ ] \wedge [\neg\blacklozenge\ (\mathit{end(f)})\ ] \qquad (3)$$

The assertion (3) means, that neither the event `start(f)` happened, nor the event *`end(f)`*. The evaluation (execution) of function ***f*** has not started and has not finished too.

All the assertions above recognize where in execution flow of ***f*** is the present time. We can label cases (1), (2), and (3) by temporal predicates

```
1. in    (f)
2. done  (f)
3. prior (f)
```

In the meaning of these predicates and the temporal operation ● (previous) we can state:

$$[\ \mathit{in(f)}\ ] \wedge [\ \bullet \neg \mathit{in(f)}\ ] \qquad (4)$$

According (4) the execution is in the execution flow of ***f*** now, but in the previous step (before some event came) the execution wasn't in the execution flow of **f**. So the result is, that evaluation (execution) just started (in this step).

In comparison with AspectJ **cflow** operation, we are now able to express the same, and even more. In implementation of the event functions (`start(f)` and *end(f)*) in PFL we are able to easily get time of event occurrence, what can be used e.g. for definition of the pointcut as follows

$$[\ \mathit{in(f)} \wedge \mathit{in(g)}\ ] \wedge [\ \mathit{start(f)} < \mathit{start(g)}\ ] \qquad (5)$$

i.e. the execution flow is in ***g*** which is called from ***f***

The (5) assertion (we can say also pointcut) solves the problem of **cflow** (in `matchF` above)**,** where we were able to express that we want to select instructions (or join points) in execution flow of both ***f*** and ***g*** functions, but we were not able to say which is called from which. Certainly we could use **within** pointcut designator, but it comprises only static join points and does not select calls to function from another function.

8. CONCLUSION

In this paper, we have presented briefly the reflection property of PFL language, since it is our aim to exploit it in weaving aspects into original code. The main goal of this paper was to introduce the style in which temporal logic can be exploited for pointcut definitions, without focussing to detailed language constructs coming out from this approach so far. In particular, ww have shown, that temporal logic provide an opportunity for a very expressive definitions of pointcut expressions, because it allows composing of pointcut designators from elementary temporal operations and basic events in process functional (which composes purely functional and imperative) program evaluation. Supporting software engineering approaches based on object paradigm, such as in [1,2,3], by object process functional paradigm [31,32,33] comes out from imperative functional programming [22]. Although the basis of process functional language is environmental [7], this does not destruct a purely functional principle of performing imperative computation by the application, instead of performing statements in less-disciplined imperative manner. This paper is a step to the detailed identification of circumstances in computation, which may occur in visualisation [26], planning parallel tasks [27,28], and many others. Essentially, the reflection property and a uniform structure of process functional language supports the idea of integrating formal and software engineering approaches in AOP.

REFERENCES

[1] Beneš, M., Kružel, M., Vondrák, I.: A Process Description Language. In *Proc. of 34th Spring International Conference ISM 2000*, Ostrava:MARQ, 2000, ACTA MOSIS No. 80, pp. 157-162, ISBN 80-85988-45-3

[2] Beneš, M.: Data Types In Persistent Object Systems. In *Transactions of the VŠB-Technical*

University of Ostrava. Ed. Prof. Ing. Jaromír Polák, CSc., Ostrava: VŠB-TU Ostrava, 2001, Vol. 1, No. 1, Computer Science and Mathematics, pp. 1-10, VŠB-TU Ostrava, ISSN 1213-4279

[3] Beneš, M.: Multiple Inheritance. In *Proceedings of ISM'98 Conference*, Ostrava:MARQ, 1998, 9-14, ISBN 80-85988-24-0

[4] Kiczales, G. et al: An overview of AspectJ. In proceedings European Conference on Object-Oriented Programming, Vol. 2072 of Lecture Notes in Computer Science, 2072:327-355, 2001, pp. 327-353.

[5] 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.

[6] 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.

[7] 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

[8] Kollár, J.: Static weaving at dynamic join points, 2004.

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

[10] Lamport, Leslie: The temporal logic of actions, ACM Transactions on Programming Languages and Systems, ACM Press, 1994, pp. 872-923.

[11] Lamport, Leslie: “SOMETIME” is sometimes “NOT NEVER”, Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press, 1980, pp. 174 – 185.

[12] Manna, Z., Pnueli, A.: The Anchored Version of the Temporal Framework, Lecture Notes in Computer Science vol. 354, Sringer-Verlag, London, 1988, pp. 201-284.

[13] Mernik Marjan, Korbar Nikolaj, Zumer Viljem: LISA: A Tool for Automatic Language Implementation. ACM SIGPLAN Notices, Vol. 30, No. 4, 1995, pp. 71 - 79, ISSN 0362-1340

[14] Mernik Marjan, Lenic Mitja, Avdicausevic Enis, Zumer Viljem. Multiple attribute grammar inheritance. Informatica, 2000, Vol. 24, No. 3, pp. 319-328.

[15] Mernik Marjan, Zumer Viljem, Lenic Mitja, Avdicausevic Enis. Implementation of multiple attribute grammar inheritance in the tool LISA. ACM SIGPLAN not., June 1999, Vol. 34, No. 6, pp. 68-75.

[16] Mernik Marjan, Zumer Viljem: Implementation of Denotational Semantics. INFORMATICA 1/91, Vol. 15, No. 1, 1991, pp. 48 - 53, ISSN 0350-5596

[17] Mernik Marjan, Zumer Viljem: Operational Semantics of LISP Subset. AMSE transactions, Tassin-la-Demi-Lune, Vol. 17, No. 4, 1993, pp 49 - 55, ISSN 0761-2532

[18] Mernik, Marjan, Lenic Mitja, Avdicausevic Enis, Zumer Viljem. A reusable object-oriented approach to formal specifications of programming languages. L'Objet, 1998, Vol. 4, No. 3, pp. 273-306.

[19] Mostýn, V., Skařupa, J.: Improving mechanical model accuracy for simulation purposes. Journal Mechatronics, Volume 14, Issue 7, September 2004, GB, Oxford: Elsevier Ltd., 2004, s. 777-787; ISSN 0957-4158

[20] Owicki, S., Lamport, L.: Proving liveness properties of concurrent programs. ACM Transactions on Programming Languages and Systems, 4 (3): 455–495, July 1982.

[21] Ostroff, S. J.: Temporal logic for real time systems, John Wiley & Sons, Inc, ISBN 0-471-92402-4, 1989.

[22] 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.

[23] Pnueli, Amir: The temporal logic of programs. In Proceedings of the 18th Symposium on the Foundations of Computer Science, pages 46–57. ACM, November 1977.

[24] Popovici, A., Gross, T., Alonso, G.: Dynamic Weaving for Aspect-Oriented Programming, ACM, ISBN: 1-58113-469-X, 2002, pp. 141-147.

[25] 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

[26] Šaloun, P.: Simulation and Visualization of Parsing, In proceedings of MOSIS ´97, 31th International conference Modelling and Simulation of Systems, part 2, Czech Republic, Hradec nad Moravicí, 1997, pp. 25-29.

[27] Škrinárová, J., Siládi, V.: The design of planning algorithm for multiprocessor system MUPRO. DIDINFO 2003, Banská Bystrica, 2003. ISBN 80-8055-786-1, pp. 118-120. (in Slovak)

[28] Škrinárová, J.: Parallel programming. DIDINFO 2004. Banská Bystrica, 2004. ISBN 80-8055-908-2, pp. 143-146. (in Slovak)

[29] Sullivan, G. T.: Aspect-oriented Programming

using Reflection, Workshop of Advanced Separation of Concerns in Object-Oriented systems, MIT Press, 2001.

[30] Tuzhilin Alexander: Templar: A Knowledge-Based Language for Software Specifications Using Temporal Logic, ACM Transactions on Information Systems (TOIS), 1995, pp. 269-304.

[31] 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, pp. 92-96, ISBN 80-7099-879-2

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

[33] 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

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

BIOGRAPHY

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.

Marcel Tóth was born in 1981. He graduated at Technical university of Košice, Slovakia. He is working on his PhD. degree at the Department of Computers and Informatics FEII Technical university of Košice, Slovakia. His scientific research is focusing on the aspect oriented programming paradigm, especially on dynamic composition in aspect-oriented programming.