Journal of ELECTRICAL ENGINEERING, VOL 51, NO 3-4, 2000, 67–74

CONTROL–DRIVEN DATA FLOW

Ján Kollár *

Control-driven data flow is a paradigm for a process functional language in which purely functionally defined processes perform updates on environment variables. In this paper, the relation between $\mathcal{PFL}$ — an experimental process functional language and the control-driven data flow execution is defined. The transition chains are selected to execute the processes by the evaluation of expressions. In addition, the $\mathcal{PFL}$ ability for parallel and non-deterministic execution is presented.

K e y w o r d s: process functional programming, imperative functional programming, stateful computation, side effects, functional nets

1 INTRODUCTION

Using a pure functional language, the programs are concise, composable and reliable [1,2]. The referential transparency of expressions makes it possible to substitute each expression by an expression of the same value. This is the basis for equational reasoning — the program synthesis and the proof of correctness [3]. It has been understood already that the functional paradigm could increase the reliability of the systems, at least from the viewpoint of their correctness.

On the other hand, the lack of side effects has been seen as a weakness of purely functional languages since the state, input/output and exception handling is cumbersome [4,5]. The systems are still developed using imperative elements [6,7,8,9] as the only way in which it is possible to express the stateful computation [4] in which the state plays a significant role.

For example, the problem of separating the functional and structural information in object oriented systems [6] could be easily solved using purely functional specification. Unfortunately, systems are not functional, they are stateful. Therefore a purely functional approach cannot be utilized in this case. Other applications related to stateful algorithms are computer graphics and virtual reality [9]. Ray-tracing algorithms [10] are executed more efficiently when the updates are used. However, performing the updates by assignments in an imperative language, the referential transparency and all the advantages of a purely functional approach are lost. The reliability and the speedup seem to be contradictory requirements.

Using functional paradigm, stateful computation can be expressed in Core ML and Standard ML [11]. In these languages, the assignments are expressions of unit types and they are evaluated on environment variables. Assignments can be used elsewhere in expressions in an undisciplined manner. SML and Core ML approaches may be characterized as rather 'reversed' imperative than functional — statements are used in expressions. Sometimes these languages are not considered functional [2].

The gap between stateful systems and purely functional specifications has been narrowed using linear types in Haskell [12] and Clean [13]. Assignments in these languages are hidden. They are performed by the application of a monad operation in Haskell [14], or by the application of any function of linear type in Clean [13].

The disadvantage, from the viewpoint of software engineering, is that the updatable cells are hidden by linear types — the types comprising single pointed values. Moreover, hiding the variables, the flow of data is hardly to realize and cannot be reasoned about using the methods of analysis of time-critical systems [15,16,17] based on Petri nets. One of the reasons is that the structure of Petri nets is modelled using bipartite graphs while the target structures of purely functional programs are graphs that consider just transitions and omit places [18].

In $\mathcal{PFL}$, the definitions of processes are purely functional, like in Haskell or Clean. At the same time, the variable environment is visible, like in SML.

Fig 1 Static variable-to-process binding

In this paper, we specify the relation between $\mathcal{PFL}$ processes [19] and functional nets [20], in which pro-

* Department of Computers and Informatics, Technical University of Košice, Letná 9, 041 20 Košice, Slovakia, kollarj@ccsun.tuke.sk

ISSN 1335-3632

cess functional programs are executed by control-driven data flow. We also present the $\mathcal{PFL}$ ability for nondeterministic and parallel computation. Finally, we introduce an example of how processes are evaluated performing updates.

The implementation of the selected transition chains can vary depending on the target architecture; even a specialized architecture may be considered. In this paper, we concentrate on the relation between the $\mathcal{PFL}$ structure and the structure of functional nets as a basis for possible implementations.

2 PROCESSES VERSUS FUNCTIONS

Parametrically polymorphic data types in $\mathcal{PFL}$ — a process functional programming language, are primitive types (such as $char$, int, and $float$), algebraic types, and extended types comprising function types and process types.

A process type is an extended type comprising at least one environment variable. A process is a function of a process type.

The type definition of process f is as follows:

$$f \;::\; \overline{t_1} \to \overline{t_2} \to \cdots \to \overline{t_n} \to \overline{t}$$

where $n \geq 0$ and at least one of the types $\overline{t_k}$ and $\overline{t}$ is in the forms $V_k\ t_k$ and $V\ t$, respectively, where V_k and V are environment variables and t_k and t are the data types.

Furthermore, if $\overline{t_k} = V_k\, t_k$, then $V_k :: t_k$, and if $\overline{t} = V\, t$, then $V :: t$.

On the other hand, a (pure) function is just a special case of process, where none of $\overline{t_k}$, $\overline{t}$ comprises an environment variable. Then the type definition of function f is as follows:

$$f \;::\; t_1 \to t_2 \to \ldots \to t_n \to t$$

2.1 The specification

An example of pure function `f` defined in $\mathcal{PFL}$ script is as follows:

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

Evaluating the expression `(f 2 3)` yields `5` without a side effect. The type definition (the first equation above) is optional, since it may be derived by the type checker. The definition (the second equation above) is obligatory.

Let us introduce now a script containing processes g and h and an expression to be evaluated.

Fig 2 The translation of functions and processes

```
g :: A int  ->int  ->B int
g x y = x + y

h :: A int  ->B int  ->()          (1)
h u v = ()

h (g 2 3) ()
```

Note that the type definitions for processes are obligatory, but the definitions are optional. The definition of a process having the unit value (such as h) may be derived by the compiler.

In $\mathcal{PFL}$ the unit value represents the control value and any function of arguments and/or value of unit types is a process.

The static binding of processes g and h to variable environment comprising variables A and B is depicted in Fig. 1.

2.2 The translation

Translating the type definition FTDEF of a function f, the association of variable f with the type expression $int \to int \to int$ is added to the extended type environment TENV (see Fig. 2). The definition FDEF is translated into the definition $f = (\lambda x.\lambda y.x+y)$ in the enriched lambda calculus form ELC. For example the definition of function f above is translated into $f = (\lambda x.\lambda y.x + y)$ form.

In contrast to pure functions, processes are translated at two stages. At the first stage, a process type definition PTDEF is processed adding the association of each variable name (by extracting it from PTDEF) with its type to the type environment TENV. At the same time, the association of the process name with the function type being obtained by extracting the environment variables is added to TENV. For example, processing the type definitions of processes g and h, the following associations are added to TENV:

$$A :: int$$
$$B :: int$$
$$g :: int \to int \to int$$
$$h :: int \to int \to ()$$

The definition PDEF of a process is translated into the form in which lambda variables and expressions are bound to environment variables. For example, processes g and h are translated into the following form:

$$\begin{aligned} g\ \overset{A}{x}\ y &= \overbrace{x+y}^{B} \\ h\ \overset{A}{u}\ \overset{B}{v} &= () \end{aligned} \tag{2}$$

Notice, that lambda variables x and u are bound to the same (shared) environment variable A, while lambda variable v and expression $(x+y)$ are bound to the shared environment variable B.

At the second stage, the definitions (2) and the main expression are translated into the extended enriched lambda calculus form ELC as follows:

$$\begin{aligned} &\textbf{LETREC}\ A\ B \\ &\quad g\ =\ \lambda A.\ \lambda y.\ \textbf{LETENV}\ B = A + y\ \textbf{IN}\ B \\ &\quad h\ =\ \lambda A.\ \lambda B.\ () \\ &\textbf{IN}\ h\ (g\ 2\ 3)\ () \end{aligned} \tag{3}$$

where (in addition to purely functional case) LETREC expression above comprises environment variables A and B, that are shared by processes g and h.

3 CONTROL–DRIVEN DATA FLOW

To be able to define the semantics of execution of process functional programs in functional nets, let us consider first the essential properties of LETENV expression which performs stateful computation, evaluating the expressions on the data and the control values.

3.1 The stateful evaluation

An application of lambda abstraction with an environment lambda variable is expressed in terms of LETENV (let environment) expression, as follows:

$$(\lambda V.\ e)\ m\ \equiv\ \textbf{LETENV}\ V = m\ \textbf{IN}\ e \tag{4}$$

in which the argument m is evaluated first. It means that the processes are eager, *ie* they conform with the principle of causality. Informally, the part $V = m$ is the assignment of the value of the expression m to the environment variable V and e is an expression accessing V. Notice that equation (4) is a run-time mechanism. However, for $\overbrace{(x+y)}^{B}$ in (2), where $\overset{A}{x}$, we have $\overbrace{(A+y)}^{B}$. Then the expression **LETENV** $B = A + y$ **IN** B in (3) corresponds to the specific lambda abstraction application $(\lambda B.\ B)\ (A + y)$, evaluated in the compile time. It means, that the right hand side of the definition of process g was translated using rule (4) where e is equal to the environment variable B.

Rule (5) says that the process with the argument of process type may be applied to the unit value $()$, which is of the unit type $()$. Then the process value is evaluated with the currently assigned value in the environment variable V.

$$\begin{aligned} (\lambda V.\ e)\ () &\equiv \textbf{LETENV}\ V = ()\ \textbf{IN}\ e \\ &\equiv e \end{aligned} \tag{5}$$

Rule (6) says that the process may be defined by the unit value $()$ and then, when applied, the argument is just assigned to the environment variable.

$$\begin{aligned} (\lambda V.\ ())\ m\ &\equiv \textbf{LETENV}\ V = m\ \textbf{IN}\ () \\ &\equiv V = m \end{aligned} \tag{6}$$

3.2 The transition chains

Our aim is to select the transitions related to $\mathcal{PFL}$ expressions, like a basis for the definition of operational semantics in terms of functional nets.

Let us designate the control value by $\circ$ and (the same or the different) data values by $\bullet$ and $\star$.

Let us consider just a single argument function and/or process f definition as follows:

$$f\ x\ =\ e$$

where x is a (lambda) variable and e is an expression.

The flow of the argument value is invoked by the function/process application. The expression e is the active element of the computation. The possible transitions are depicted in Fig. 3.

Fig 3 The transitions for the chain e

Let the type definition be $f :: t_1 \to t_2$.

Case 1: If $t_1 \neq () \wedge t_2 \neq ()$ then the application $(f\ m)$, where m is an expression, is performed by the transition DD. In Fig. 3 the value of m is marked by the token $\bullet$ and the value of $(f\ m)$ is marked by the token $\star$. If the function f is the identity $(f\ x = x)$, then $\bullet = \star$. The DD transition is the only one which is evaluated either eagerly or lazily, depending on the strictness analysis.

Case 2: If $t_1 \neq () \wedge t_2 = ()$ then the application $(f\ m)$, is performed by the transition DC yielding the control value that is marked by $\circ$ in Fig. 3. The transition DC is always evaluated eagerly, otherwise the expression e would not be evaluated at all.

Case 3: If $t_1 = () \wedge t_2 \neq ()$ then the application $(f\ ())$ is performed by the transition CD yielding the data value that is marked by $\bullet$ in Fig. 3. The transition CD is is evaluated eagerly, otherwise the evaluation of e would invoke the repeated evaluation of e. In a purely functional language it is impossible to use a free variable in e, hence the process above is a constant. In $\mathcal{PFL}$, however, if e comprises the application of a process then the value of the expression $(f\ ())$ may differ since the evaluation of e is stateful. Although in this case the definition of f is not referentially transparent, this fact can be detected statically using the call dependency analysis.

Case 4: If $t_1 = () \wedge t_2 = ()$ then the application $(f\ ())$, is performed by the transition CC yielding the control value.

Corresponding to the cases above, we may state:

1. If $t_1 = ()$ (for the transitions CD and CC) then the process may be applied just to the control value, *ie* its definition is as follows:

$$f\ () \ = \ e$$

   where $()$ is the pattern, and e is an expression.
2. If $t_2 = ()$ (for the transitions DC and CC) and the definition is omitted then the implicit value of a process is $()$.
3. Provided that $f :: t_1 \to t_2$, the transition DD corresponds to a function which is evaluated eagerly or lazily and transitions DC, CD and CC correspond to processes that are evaluated eagerly. In conclusion, the mapping $t_1 \to t_2$ is performed using the evaluation chain e and the possible transitions are: DD, DC, CD, and CC.

There are yet three other chains supported by $\mathcal{PFL}$, depending on the type definition of processes: If $f :: V_1\ t_1 \to t_2$ then the evaluation chain is $V{-}e$. If $f :: t_1 \to V_2\ t_2$ then the evaluation chain is $e{-}V$. Finally, if $f :: V_1\ t_1 \to V_2\ t_2$ then the evaluation chain is $V{-}e{-}V$. All the chains for single argument functions/processes are depicted in Fig. 4.

Fig 4 Single argument chains

Fig 5 The data flow

Before we derive the transitions for these chains, let us consider the flow of control and/or data values through the environment variables separately. This flow is depicted in Fig. 5.

The environment variables contain data values, including functions and processes. The possible flows are DD and CD, where DD corresponds to **LETENV** $V = m$ **IN** V, where $m \neq ()$, *ie* it is update, and CD corresponds to **LETENV** $V = ()$ **IN** V, *ie* it is access. The flows DC and CC are excluded, since the control value $()$ cannot be assigned to V.

The variables (in contrast to functions and/or processes) are passive elements of the computation. The data flow is invoked by the application of a process, *ie* evaluating the chains $e{-}V$, $V{-}e$, and $V{-}e{-}V$, respectively.

The transitions for the chains in Fig. 4 are selected by composition which is defined by matching of the control or the data values as follows: Let X, Y, Z be either D (data value) or C (control value). Then it holds:

1 The flow XD matches the transition DZ, producing the transition XZ.

2 The transition XY matches the flow YZ, producing the transition XZ.

The transition for the chain $V{-}e$ is derived using rule 1 above. The transition for the chain $e{-}V$ is derived using rule 2 above. The transition for the chain $V{-}e{-}V$ is derived using both rules in any order since composition is a transitive operation.

The derived transitions for all the chains are summarized in Table 1.

Table 1. The transitions for chains

CHAINS	TRANSITIONS			
	DD	CD	DC	CC
e	$\bullet \to \bullet$	$\circ \to \bullet$	$\bullet \to \circ$	$\circ \to \circ$
$V{-}e$	$\bullet \to \bullet$	$\circ \to \bullet$	$\bullet \to \circ$	$\circ \to \circ$
$e{-}V$	$\bullet \to \bullet$	$\circ \to \bullet$	$\perp$	$\perp$
$V{-}e{-}V$	$\bullet \to \bullet$	$\circ \to \bullet$	$\perp$	$\perp$

It means that the transitions XC are not allowed for the chains $e{-}V$ and $V{-}e{-}V$ since () cannot be assigned in V, hence, it cannot be produced from V.

3.3 Process composition

Control driven data flow is an execution mechanism for the nets that we call functional nets for the following reasons: Firstly, the flow of the data is invoked by the process application, it is not implicit like in dataflow graphs. Secondly, the structure of functional nets is statically defined, but dynamically allocated. The part of a functional net well-defined by the extended LETREC is allocated before the LETREC expression is evaluated and it is deallocated after that.

The order in which the arguments of processes are evaluated determines the run-time allocation requirements.

To be able to apply the methods for modelling and analysis of time-critical systems, it is important in which way functional nets conform with Petri nets. Since we have defined possible transitions and single argument chains, consisting of environment variables and transition, our approach is conforming (from the viewpoint of the structure) with Petri nets if the functional nets are modelled using bipartite graphs, *ie* such in that there is no arc from a transition to a transition and no arc from an environment variable to an environment variable.

Let us propose that functional nets are modelled using bipartite graphs. The chains $e{-}V$ and $V{-}e$ satisfy our proposition. Let us consider the chains $e_1{-}e_2$ and $V_1{-}V_2$.

The chain $e_1 - e_2$ corresponds to the application $f_2\ (f_1\ m)$, where m is an expression and the functions/processes are defined as follows:

$$\begin{array}{l} f_1 :: t_1 \to t \\ f_1\ x = e_1 \end{array} \quad \text{and} \quad \begin{array}{l} f_2 :: t \to t_2 \\ f_2\ x = e_2 \end{array}$$

where t, t_1, and t_2 are either data types or unit types. In this case $f_2\ (f_1\ m) = (f_2 \circ f_1)\ m$, where $\circ$ is composition operation defined sa follows: $(f \circ g)\ x = f\ (g\ x)$. Hence, the chain $e_1{-}e_2$ may be expressed in terms of singleton chain e corresponding to $(f_2 \circ f_1)$.

The chain $V_1{-}V_2$ comprised in the chain $e_1{-}V_1{-}V_2{-}e_2$ corresponding to the application $f_2\ (f_1\ m)$, where m is an expression and the processes are defined as follows:

$$\begin{array}{l} f_1 :: t_1 \to V_1\ t \\ f_1\ x = e_1 \end{array} \quad \text{and} \quad \begin{array}{l} f_2 :: V_2\ t \to t_2 \\ f_2\ x = e_2 \end{array}$$

where t must be a data type. In this case it holds:

$$f_2\ (f_1\ m) = f_2\ (id\ (f_1\ m)) = (f_2 \circ id \circ f_1)\ m$$

where id is the identity function $id\ x = x$. Hence, the chain $V_1{-}V_2$ is substituted by the chain $V_1{-}e{-}V_2$, where e is equal to x.

We have shown that the structure of functional nets conforms with Petri nets, which are modelled using bipartite graphs. However, control-driven data flow is based not just on the flow of data, extracting and producing the values; it updates memory cells by data values and accesses them by control values again.

3.4 Parallelism

Parallelism is the matter of the control. More precisely, it is the matter of evaluation strategy used when $\mathcal{PFL}$ processes are applied to arguments of unit type, *ie* such that their values are control values.

Let process f be defined as follows:

$$\begin{array}{l} f\ ::\ A_1\ t_1 \to A_2\ t_2 \to \ldots \to A_n\ t_n \to \overline{t} \\ f\ x_1\ x_2\ \ldots\ x_n\ =\ e \end{array}$$

Then the application

$$f\ m_1\ m_2\ \ldots\ m_n$$

is evaluated, evaluating arguments m_1, m_2, ..., m_k sequentially before the expression e is evaluated. The application above corresponds to the following imperative sequence of assignments, followed by the evaluation of expression e.

$$A_1 := m_1;\ A_2 := m_2;\ \ldots\ A_n := m_n;\ e$$

Note that the values of environment variables are used in e above, since $A_1, \ldots, A_n$ occur in e as a result of the translation of f. In the example above, the arguments m_k are of data types t_k, $k = 1 \ldots n$.

Now, let us suppose the arguments m_k of unit type (), $k = 1 \ldots n$. Then the application $f\ m_1\ m_2\ \ldots\ m_n$ corresponds to the imperative sequence as follows.

$$m_1;\ m_2;\ \ldots\ m_n;\ e$$

Moreover, we may support sequential evaluation of expressions by built-in sequence tupling operations, as follows.

$$\begin{array}{ll} (;) & :: () \to () \to () \\ (;;) & :: () \to () \to () \to () \\ (;;;) & :: () \to () \to () \to () \to () \end{array}$$

etc, and parallel evaluation of expressions by built-in parallel tupling operations, as follows.

$$\begin{array}{ll} (\|) & :: () \to () \to () \\ (\|\ \|) & :: () \to () \to () \to () \\ (\|\ \|\ \|) & :: () \to () \to () \to () \to () \end{array}$$

Fig 6 Evaluation – Initial state

etc.

Then $(m_1; m_2; m_3)$ is evaluated sequentially starting with m_1 and finishing with m_3 and $(m_1 \parallel m_2 \parallel m_3)$ is evaluated in parallel, starting with m_1, m_2, and m_3, and finishing if all expressions are evaluated. In both cases the value of evaluation is control value.

3.5 Non-determinism

Let us define the processes f, g and h as follows:

$$f :: A\ int \to int$$
$$f\ x\ =\ x + x$$

$$g :: A\ int \to ()$$
$$g\ x\ =\ ()$$

$$h :: A\ int \to ()$$
$$h\ x\ =\ ()$$

The processes above share the same environment variable A. Hence, the expression $(f\ (g\ 3 \parallel h\ 4))$ is evaluated as follows.

$$(A := 3 \parallel A := 4);\ A + A$$

The result of evaluation is either 6 or 8 depending on the order in which $A := 4\ \ (h\ 4)$ and $A := 3\ \ (g\ 3)$ are evaluated.

In conclusion, functional nets are able to model non-determinism since environment variables may be shared by arguments and/or values of processes.

Fig 7 Evaluation – Step 1

Fig 8 Evaluation – Step 2

4 AN EXAMPLE OF EVALUATION

Let us consider the script (1) consisting of the processes `g` and `h`, and the main expression `(h (g 2 3) ())`. It can be proved that this script is deterministic. That is why we will not consider the order in which the arguments are evaluated. We will concentrate just on the state before the arguments are evaluated, after they are evaluated, and the state after the process body is evaluated.

The initial state of evaluation is depicted in Fig. 6.

When evaluating the expression `(h (g 2 3) ())` the application `(g 2 3)` is evaluated first.

Applying g to the arguments, the value `2` is assigned to A, according to Fig. 7. Then the process g is evaluated, yielding the value `5` which is assigned to B. The result `5` is still accessible like in the case of a pure function. Although no assignment was used by a programmer explicitly, evaluating the expression `(g 2 3)`, the additional side effects on A and B were performed. This situation is depicted in Fig. 8. The state before `(h 5 ())` is evaluated, is depicted in Fig. 9.

Applying h to the arguments `5` and $()$, the value `5` is assigned to A according to Fig. 10. The second argument of the same value 5 is accessed from B. Evaluating the body of h, which is $()$, yields the control value. No value is accessible in the stack evaluating the expression `(h (g 2 3) ())`. The final state is depicted in Fig. 11.

Fig 9 Evaluation – Step 3

Fig 10 Evaluation – Step 4

As obvious, the environment variables are the variables (memory cells) that are shared by processes. The data values are passed through these variables updating them. The control values access the current data values assigned to the environment variables.

5 DISCUSSION

Besides the static binding in the compile time, the dynamic binding is performed while evaluating the processes (compare the state in Fig. 8 and the state in Fig. 9). Although in stack-based implementations this binding is trivial (since evaluating the process application, the result is on the stack), there is a variety of non-trivial implementations that may be considered using the concept of functional nets evaluated by the control-driven data flow.

Fig 11 Evaluation – Final state

The expression (the chain e) in the functional net is related to the transition in the Petri net. Hence, the chain e is not a mapping on the anonymous tokens, but on the data and/or control values.

The variable V is related to the place in the Petri net. However, the value is not extracted from V like a token from the place; it is either updated or accessed.

In this article, we consider the control-driven data flow paradigm as an abstract machine for a process functional programming language rather than an execution model for computer architectures. However, in $\mathcal{PFL}$ a very high abstract level of a pure functional language interacts with a very low level of the state machines in an uniform way.

There are formal methods available for evaluating time-critical systems [15,16,17] that seem to be applicable to the modelling, analysis and profiling of real-time systems that the $\mathcal{PFL}$ language is proposed to. It is important to align the processes to real-time system resources in the way in which the time requirements are satisfied [20]. Type definitions provide possibility to reason about sharing the variables. The definitions provide possibility to reason about the (control and data) flow. This reasoning is one of the subjects of our current research. Of course, the single argument chains presented in this paper have to be extended to multi argument functions/processes, guarded expressions, *etc.*

6 CONCLUSION

In this paper, we have defined the relation between $\mathcal{PFL}$ — an experimental process functional programming language and the structure of transition chains in functional nets.

We have shown the way in which functional nets are mapped to bipartite graphs and how they are evaluated by control-driven data flow performing stateful computation. We have proposed the $\mathcal{PFL}$ language constructs able to explore the parallelism.

A stateful computation by control-driven data flow is performed as efficiently as in any imperative language. At the same time, high abstraction of a purely functional definition of processes is preserved.

From the viewpoint of software engineering, $\mathcal{PFL}$ script is the composition of a purely functional specification and a dataflow design.

Acknowledgement

This work has been supported by the VEGA Grant No.1/5265/98.

References

[1] PEYTON JONES, S.L.: The Implementation of Functional Programming Languages, Prentice–Hall, 1987.

[2] READE, C.: Elements of Functional Programming, Prentice–Hall, 1989.

[3] NOVITZKÁ, V.: Structures of Algebraic Specification Languages, In: Proceedings of the International Scientific Conference ECI '98, Košice–Herľany, Slovakia, October 8–9, pp. 1–6.

[4] 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.

[5] LAUNCHBURY, J.: Lazy Imperative Programming, Computing Science Department, Glasgow University, 1993.

[6] MEYER, B.: Object-oriented Software Construction, Prentice Hall, 1985.

[7] HAVLICE, Z.: Prototyping and Modeling of Large Software Systems, IN: Proceedings of FEI'25 Conference Electronic Computers and Informatics, Košice–Herľany, Slovakia, Sep. 22–23, 1994, pp. 237–242. (in Slovak)

[8] HAVLICE, Z.—PLOŠČICA, O.: Computer Aided Software Engineering Based on Configurable Technology, In: Proceedings of the 2nd International Conference on Renewable Source and

Environmental Electro–Technologies, Oradea, Romania, May 27–30, 1998, pp. 112–118.

[9] SOBOTA, B.—VALIGURSKÝ, M.: Real Time Visualising Kernel for Virtual Reality Applications, In: Proceedings of International Conference Modelling and Simulation of Systems MOSIS'98, Bystřice p. Hostýnem, May 5–7, 1998, pp. 117–122.

[10] SOBOTA, B.—JANOŠO, R.—PARALIČ, M.: The Distributed Raytracing in the Virtual Reality systems, In: Proceeding of the Scientific Conference with International Participation "Informatics and Algorithms", Prešov, Slovakia, September 3–4, 1998, pp. 64–68.

[11] HARPER, R.—MacQUEEN, D.—MILNER, R.: Standard ML, ECS–LFCS–86–2, LFCS Report Series, University of Edinburgh, Department of Computer Science, 1986.

[12] PETERSON, J.—HAMMOND, K. (Ed.): Report on the Programming Language Haskell: A Non-strict, Purely Functional Language Version 1.3. Yale University, May 1996.

[13] ACHTEN, P.—PLASMEIJER, R.: Interactive Functional Objects in Clean, In: Clack et al. (Ed.): IFL'97, LNCS 1467, 1998, pp. 304–321.

[14] WADLER, P.: The Essence of Functional Programming, In: 19th Annual Symposium on Principles of Programming Languages, Santa Fe, New Mexico, January 1992.

[15] HUDÁK, Š.—TELIOPOULOS, K.: TB nets: Properties of Time Interval Profiles, In: Proceedings of International Conference RSEE'97, (T. Maghiar Ed.), Oradea, Romania, May 30–31, 1997, pp. 25–30.

[16] HUDÁK, Š.—TELIOPOULOS, K.: Merging Firing Sequences of P-decomposed Petri Nets, Proceedings of International Conference RSEE'97, (T. Maghiar Ed.), Oradea, Romania, May 30-31, 1997, pp. 31–36.

[17] HUDÁK, Š.—TELIOPOULOS, K.: Loop Spectral Analysis of Time Reachability Problem, In: Proceedings of the 2-nd International Conference RSEE'98, (T. Maghiar Ed.), Oradea, Romania, May 27–30, 1998, pp. 51–61.

[18] LAUNCHBURY, J.—PEYTON JONES, S.L.: Lazy Functional State Threads. Computing Science Department, Glasgow University, 1994.

[19] KOLLÁR, J.: Process Functional Programming, Proceedings of MOSIS'99 Conference, Rožnov pod Radhoštěm, Czech Republic, April 27–29, 1999, pp. 41–48.

[20] KOLLÁR, J.: Functional Nets: A New Opportunity for Realistic Parallel Functional Programming, In: Proceedings of the International Scientific Conference ECI'98, Košice–Herľany, Slovakia, October 8–9, 1998, pp. 35–40..

Received 10 June 1999

Ján Kollár (Doc, Ing, CSc) 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 has been 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 months 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.