Multiparadigm Approach to Software System Evolution

Jana Bandáková

Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Letná 9, 041 20 Košice, Slovakia
jana.bandakova@tuke.sk

Abstract: Every software system during the life cycle undergoes an evolution in dependence on new requirements such adding a new functionality, improving or removing the existing one, improving the performance of system and so on. Nowadays, in regard of robustness of software systems, new possibilities of an automatic evolution of existing software systems are searched. The human intervention to an evolution process should be as minimal as possible. There are many researcher teams that try to solve this problem, mostly using object oriented approach. In our research project we concentrate on suitable properties of functional, object-oriented and aspect-oriented paradigm. An experimental process functional language (PFL) has been developed to combine and integrate suitable properties of these paradigms. The paper deals with particular paradigms and how these paradigms and their properties will be used for our purpose. Also source to source transformation from PFL to Petri Nets will be shown.

Keywords: software system evolution, aspect paradigm, process functional paradigm, Petri Nets

1 Introduction

Many researchers try to solve the problem of software system evolution especially using object-oriented paradigm. In our research project we concentrate in solving this problem on functional paradigm. Functional languages [10, 11] represent a declarative manner of program development that is based on mathematical understanding of problem. Using mathematical formalism and constructions with a strict defined semantics the problem can be described effectively. On one hand, functional languages can increase the reliability of the system from the correct functionality point of view, but on the other hand, using only functional languages not all real world events such states, input and output functions, error handling etc. can be expressed because of lack of side effects (lack of assignments). Therefore it was necessary to extend pure functional paradigm using imperative and object-oriented properties [14]. At Faculty of Electrical Engineering and Informatics the Process Functional Language (PFL) has been developed. This language combines appropriate properties of functional, imperative and object-oriented paradigms. Additional improvements can be achieved using aspect-oriented techniques which are based on weaving certain parts of code (advices) to an original source code without its modifying. The process of weaving can be static or dynamic. In our research project we concentrate on dynamic weaving process because it can change and control the behavior of the software system during its run time. In Section 2 our motivation will be described, in Section 3 the process functional paradigm will be introduced, in Section 4 an easy example of source to source transformation from PFL to Petri Nets will be shown. Section 5 discuss the aspect-oriented paradigm, in Section 6 related works are mentioned. In conclusion future goals are resumed.

2 Motivation

An existing software system during its life cycle undergoes an evolution in dependences on new requirements. Evolution is an inevitable process when developing any type of software system and belongs to costly stage in its life cycle. Automatization of this process will reduce time costs and resources required to carry out this stage of the life cycle and development. To express large and complex software systems multiple paradigms are required. For our goals we focus on relation between different paradigms (pure functional, object-oriented and aspect-oriented). Based on the idea of integration of positive properties of particular paradigms, the process functional paradigm has been developed. In addition, it was determined that using aspect-oriented approach the behavior of the software system can be controlled and make the evolution process automated. The weaving mechanism should be a mechanism for automatic implementation that is based on process functional paradigm. Pure functional specification, algebraic specification and temporal logic will be used by goal definition of the evolution. It means, the software system can be seen from 3 different points of view – pure functional, algebraic and temporal logic (as subsequent of events).

3 Process Functional Paradigm

As mention about, the main reason for developing the process functional paradigm was to associate positive properties of pure functional, imperative, object-oriented and nowadays also aspect-oriented paradigms. The paradigm comes out from pure functional language Haskell [6, 7, 8]. Primarily, the pure functional language was enhanced to store the values in memory cell through environment variable. The environment variable expects some data value (value of some type T), but may also contains an undefined value ($\perp$). Also unit value () is used and has a function of control value to access the value stored in memory cell. Large systems can communicate through the environment variables exchanging and accessing its values. On the other hand, the environment variable serves as an attribute for arguments of pure function. Such function with attributed argument is called process and environment variable is used only in type definition of the process, not in its definition [6, 7, 12]. There are two basic processes that can handle with values stored in environment variable – data process (*data*) a control process (*control*):

$data :: v\,T \rightarrow T$	$control :: v\,T \rightarrow ()$
$data\ x = x$	$control\ () = ()$
a.) data process	b.) control process

As mention about, using these processes the value stored in environment variable can be handled as follows: applying the data process *data* to a data value (of type T), the value in environment variable will be updated to an actual value of process argument. On the other hand, applying the control process *control* to unit value () (of unit type ()), the value in environment variable will be accessed, but remains unchanged. Using environment variables, state can be manipulated like in imperative language. In the following, an example of such data process will be shown and illustrated. Let us have a process of two arguments as follows:

$$proc :: uT_1 \rightarrow vT_2 \rightarrow T$$

$$proc\ \ x\,y = x * y$$

Let us assume that values in both environment variables are undefined $\perp$ (Fig. 1). The process *proc* is applied to arguments 5 and 6 (*proc* 5 6). The result of the application will be 30. The application affects also the value in environment variables as follows: environment variable *u* contains value 5 and environment variable *v* contains value 6 after application. It means, both of these environment variables are updated (Fig. 2). It is necessary to say that the application (*proc* 5 6) is evaluated by subsequent parameter passing.

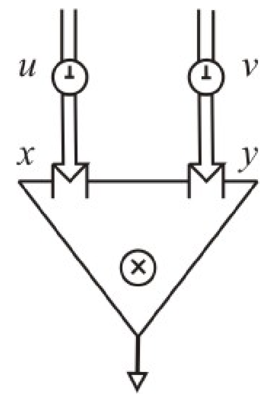

Figure 1
Before application

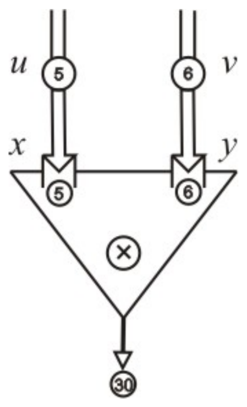

Figure 2
After application

In the example above *u* and *v* in type definition of process *proc* are environment variables that represent also attributes of pure function arguments. On the other hand, variable *x* and *y* in definition of process *proc* represent lambda variables. In process functional paradigm the lambda variable represents not only bound variable used in the body of an anonymous function but also a stack memory cell. Next, an example of using unit type to access the value in environment variable will be shown. Let us have a pure function *function* with a local process *local_proc* (contains an environment variable as an attribute of function's argument T_1) as follows:

$$function :: T_1 \rightarrow T_2 \rightarrow T$$

$$function\ \underline{x}\ y = local\_proc\ \ x\ y + local\_proc\ \ ()\ y + 2$$

$$where$$

$$local\_proc :: \underline{x}T_1 \rightarrow T_2 \rightarrow T$$

$$local\_proc\ \ x\ y = x*y$$

When we apply function *function* to arguments 5 and 6 (*function* 5 6) then the result of this application will be 62. Applying the process *local_proc* to a unit value () the value 5 stored in environment variable, as a result of the previous application of this process, is accessed and used as an argument of process *local_proc*. The variable $\underline{x}$ in function *function* designates the lambda variable and $\underline{x}$ in process *local_proc* designates an environment variable – it means, for both of $\underline{x}$ the same memory cell on the stack is used.

As can be seen, the processes manipulate with a state what can be designated as state transformation that has an important role in source to source transformation from PFL to Petri Nets.

4 Using Petri Nets

Petri net is one of the several mathematical representations of discrete distributed systems. It is also a modeling language that graphically represents the structure of the system [9]. Our goal is to achieve that Petri net can be used to execute the program. The main idea is that we have pure functions (for example add, sub) and environment variables that are separated from each other and can be associated so that pure functions are connected with environment variables (input, output), whereby these variables can be shared by various functions, to achieve a transformation. These transformations are separated and have pure functional core. For our purpose Petri nets can be defined as follows:

$$\mathrm{PN} = (\mathbf{E}, \mathbf{T}, \mathrm{I}, \mathrm{O}), \text{ where}$$

$\mathbf{E} = \{u, v, w \mid u: T_u, v: T_\mathrm{v}, w: T_w\}$ represents set of typed variables (variable contains the value of some type T or undefined value $\perp$ so $T = \mathrm{T} \cup \{\perp\}$),

T	represents the set of transformations from input to output,
I: $T \rightarrow E$	represents an input function,
O: $T \rightarrow E$	represents an output function

In state transformation has an important role state function or transition function that transforms input values to output values and is defined as follows:

$$\delta\colon\ T^n \rightarrow \mathbf{T} \rightarrow\ T^n$$

For example, if there are two inputs values that are transformed to two outputs values the transition function is defined as follows:

$$\delta\colon\ T^2 \rightarrow \mathbf{T} \rightarrow\ T^2$$

$$T^2 = T_u \times\ T_\mathrm{v}$$

A simplified example of such transformation is shown on Fig. 3.

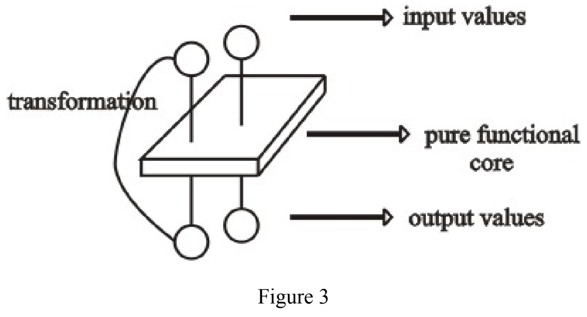

Figure 3
A simple example of transformation

This idea will be shown on a simple example. Let us define two pure functions *function1* and *function2* as follows:

$$function1 :: T_1 \rightarrow T_2 \rightarrow T_3$$

$$function1\ x_1\ x_2 = expr_1$$

$$function2 :: T_1 \rightarrow T_2 \rightarrow T_4$$

$$function2\ y_1\ y_2 = expr2$$

At this moment, the form of expression on right hand side is not important. Next, the input and output functions are defined:

```
input :: $v_1T_1 \rightarrow v_2T_2 \rightarrow ()$
input $z_1$ $z_2$ =  out (function1 () ())
                         (function2 () ())
   where
      function1 :: $z_1T_1 \rightarrow z_2T_2 \rightarrow T_3$
      function2 :: $z_1T_1 \rightarrow z_2T_2 \rightarrow T_4$
output :: $v_2T_2 \rightarrow v_3T_3 \rightarrow ()$
output _ _ = ()
```

Based on this, the transition (from input to output) can be defined as t = (input, output) and t $\in$ **T**. As mention above, pure functions and environment (input, output) variables are separated from each other. Next, pure functions are connected with these variables in order to perform a transformation. These transformations are separated.

5 Towards Aspect-oriented Approach

Aspect-oriented programming languages provide facility to interrupt the flow of control in application at specific points (join points), and insert new computation (advice) at this point without modifying the original source code [3, 13, 15]. Advice represents a piece of code that can manipulate the surrounding state and affect globally running application. In order to triggered advice at specific join points, specified conditions defined by programmer should be met. When conditions are met an advice is woven in join point by weaving mechanism. The weaving mechanism is crucial because it can change and control the behavior of the software system during its run time. Generally, aspect-oriented paradigm has been built as an extension to object-oriented and procedural languages but also functional languages can benefit from positive properties of aspect-oriented approach. In our research project aspects are a subject of research, have been integrated to process functional language and analyzed. The weaving mechanism seems to be an appropriate mechanism for automatic implementation that is based on process functional paradigm and enables the process of evolution to be automated.

6 Related Works

Many researches concentrate on dynamic software system evolution and how to change the behavior of an application during its execution without shutdown it.

In [1] an architecture for ‘closing feedback loop’ over the entire software system evolution process is proposed and enable the construction of self-evolving software systems that are capable of automatically detecting when changing external circumstances or internal conditions can better handled by alternate software modules and enables dynamically swap these modules into place. They introduce a concept of evolution engine that oversees a running application and decides when and how to evolve it based on run time information.

Manuel Oriol in his thesis [4] concentrates also on the dynamic software system evolution based on object-oriented languages. He tries to provide dynamic, unmarshalled and unanticipated evolution of software systems. The main idea of this consists in maximum disconnection of the particular parts of an application and on concepts of the anonymity, associative naming and asynchronism. As mention above, the work concentrates on object-oriented languages where a piece of code is connected to another in such cases as class inheritance, direct references from one object to another, synchronization constrains between different pieces of code, etc. The basic building blocks of an application represent entities that can communicate with each other only through services that are handled with services invocation by communication infrastructure. An entity requests invocation of the services by providing a description of its need to service manager that chooses the suitable service, according to their descriptions without naming them.

In [2] Peter Ebraert and Eric Tanter try to update an application dynamically using based-application and meta-level layer that are connected. Through meta-level layer manipulation the behavior or structure of a based-level application can be changed. The application has cleanly separated entities at the based-level and its representation at the meta-level. The application can self-evolve through meta-level manipulation.

In [5] a framework to control software system evolution has been developed. The framework permits the analysis and prediction of indicators of software system evolution such system size or complexity and defines a set of methods to handle the problems like unused object removing, libraries re-modularization using genetic algorithm or restructure the source file directory organization. It is based on diagnostic and predicting system reorganization opportunities and performing some reengineering actions.

Conclusions

Separation of concerns and modularization are essence for software system evolution. By combining transformations, that was described in section 4, we need to make provision for a property of system we would like to obtain through these combinations. The essence problem of software system evolution are system‘s properties because these properties are often expressed not clearly in specification of the software system. If the specification is not clearly then the behavior of the system can not be changed effectively. Our future goal is to express properties of the system as clearly as possible based on algebraic specification and try to predict these properties in the future applying decision rules in process of evolution. The crucial role in software system evolution has weaving process.

Acknowledgement

This work has been supported by the Scientific grant agency project (VEGA) No. 1/1065/04 Specification and Implementation of Aspects in Programming.

References

[1] Ch. Dellarocas, M. Klein, H. Shrobe: An Architecture for Constructing self-evolving software systems. In ISAW ’98: Proceedings of the 3rd International Workshop on Software Architecture, pp. 29-32, New York, NY, USA, 1998, ACM Press

[2] P. Ebraert, E. Tanter: A Concern-based Approach to Dynamic Software Evolution. In Dynamic Aspects Workshop (DAW) proceedings, pp. 51-55, Lancaster UK, March 2004, In conjunction with the conference on Aspect Oriented Software Design (AOSD 2004)

[3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold: An Overview of AspectJ. ECOOP’01, 2001, LNCS, Vol. 2072, pp. 327-355

[4] M. Oriol: An Approach to The Dynamic Evolution of Software Systems. Thesis, pp. 1-211, 2004

[5] M. Di Penta: Evolution Doctor: A Framework to Control Software System Evolution. Ninth European Conference on Software Maintenance and Reengineering (CSMR'05) pp. 280-283, 2005

[6] J. Kollár: Unified Approach to Environments in a Process Functional Programming Language. Volume 25, pp. 439-456, 2003

[7] J. Kollár: Process Functional Programming. Proc. 33rd Spring International Conference MOSIS’99 - ISM’99, Information Systems Modeling, Rožnov

pod Radhoštem, Czech Republic, ACTA MOSIS No. 74, pp. 41-48, April 27-29, 1999

[8] M. Behálek, P. Šalamoun: Parallel Process Functional Language. In WOFEX 2006. Ed. Václav Snášel, Ostrava: FEI-TU Ostrava, 2006, pp. 304-310, ISBN 80-248-1152-9

[9] J. L. Peterson: Petri Net Theory and The Modelling of systems. Prentice Hall, Englewood Cliffs, 1982, pages 288, ISBN: 0136619835

[10] P. Wadler: The Essence of Functional Programming. In 19th Annual Symposium on Principles of Programming Languages, Santa Fe, New Mexico, January 1992, draft, pages 23

[11] J. Peyton: The Implementation of Functional Programming Languages. Prentice-Hall, 1987, pages 445

[12] J. Kollár: Process Functional Programming. Proc. 33rd Spring International Conference MOSIS'99 - ISM'99 Information Systems Modelling, Rožnov pod Radhoštem, Czech Republic, April 27-29, 1999, ACTA MOSIS No. 74, pp. 41-48, ISBN 80-85988-31-3

[13] D. Walker, S. Zdancewic, J. Ligatti: A Theory of Aspects. In Proc. of the 8th ACM SIGPLAN International Conference on Functional Programming, Upsala, Sweden, August 2003

[14] M. Mernik, V. Zumer: Incremental programming language development. Computer languages, Systems and Structures, 2005, Issue 31, pp. 1-16

[15] R. Douence, O. Motelet, M. Sfidholt: A formal definition of crosscuts. In Proceedings of the 3rd International Conference on Reflection. (Kyoto Japan, September 2001), pp. 170-186, ISBN 3-540-42618-3