Towards Generic Implementation of Software Architectures

Jana Bandáková, Ján Kollár

Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Kosice
Letna 9, 041 20 Kosice, Slovakia
Jana.Bandakova@tuke.sk, Jan.Kollar@tuke.sk

Abstract: The software architecture is the structure or the set of structures of the system, which comprise software elements, externally visible properties of those elements and the relationship amongst them. Very important role in developing of software architecture has its modeling. Based on the model, the properties of modeled system can be analyzed, to obtain required behavior. The essence of our idea presented in this paper is integrating modeling and implementation stages using process functional paradigm, and select the propositions for developing a method for automatic software architecture generation based on aspect paradigm. In this paper we present the principles of generic modeling environments, and the mechanisms for weaving process functional programs.

Keywords: Generic modeling, meta-models, process functional paradigm, program transformation, software generation, aspect-oriented programming

1 Introduction

The software architecture of a computing system is the set of structures of the system, which comprise software elements, the externally visible properties of those elements and the relationship amongst them. The architecture defines how elements and components relate to each other and describes the parts that the system consists of and the relation between them. The architecture represents an abstraction of a software system what means that the architecture component can represent anything from small components to large subsystems. Mutual interconnection of the components is also specified at the high level of abstraction. Specifying the software architecture in developing of a system is very important because it represents a high level description of the system being developed [13,16]. Very important role in developing of software architecture has its modeling [5]. In software engineering, models are used for various purposes and helps us to ensure that the functionality is complete and correct, end-user needs

are met and program design supports scalability, security and other characteristics [1,6,8]. Modeling is mostly based on the graphical techniques, but for our purposes it is more substantial that it is not just about description and illustration, but it also adheres to rules and patterns. Domain specific design environments specify and configure, or generate the target application in a particular domain specific field. On the other hand, generic environments are configurable for a wide range of domains. Such environment contains a set of generic concepts.

Our goal is to integrate the generic modeling approaches with automatic software architectures generation with respect of current and future requirements. This paper is focused on mechanisms. We introduce the principles of generic modeling, concentrating on the meta-modeling property as well as the characteristics and ability of process functional paradigm [7,13,15] for the development of systems in aspect-oriented manner.

2 Principles of Generic Modeling

A generic modeling environment (GME) is a programmable and configurable toolkit for domain specific modeling. It means that it can by configured and adapted from meta-level specification (which is called a meta-model or modeling paradigm) that describes the domain. Before any system is built-up, it is necessary to specify what is to be modeled, how the modeling is to be done and what type of analyses must be formalized. Modeling paradigm represents a design choice, i.e. syntactic, semantic and presentation information, which are necessary to create the model of a system. The paradigm may differ based on the application area. Known paradigms are for example UML, SF (finite state machine paradigm), and HFSM (hierarchical finite state machine paradigm). After the paradigm is selected, the meta-model is constructed. Meta-model contains the specification of domain specific environment and represents a formal description of the model construction semantics of the modeling environment. Meta-model contains description of the entities, attributes and relationships that are available in target modeling environment. This meta-model is then used to configure the generic modeling environment and is used to automatically generate the target domain specific environment containing the modeling elements and valid relationships that can be constructed in domain specific. The domain specific environment is used to build domain models, which are usually stored in database.

As mentioned above, generic modeling environment is based on meta-modeling what means that the first step, before we begin work with an environment, is its configuration - modeling of a modeling process. The output of the meta-modeling process is a set of rules. This set of rules represents a paradigm, which configure the environment for a specific application domain. The paradigm contains all

syntactic, semantic and presentation information with respect of the application domain and is represented for example by UML class diagram.

First thing the modeler needs is the specification of the modeling application that is to be implemented. This specification comes in natural language. When we have a specification, we can begin with modeling. In meta-modeling, two basic things are the most important to identify – entities used by the model and relations between entities. Entities and relations have their attributes, which are used to identify and qualify them. In meta-model we specify that this entity or this relation has such attributes but the attributes are not further specified (attributes have no values). As mentioned above, the GME support a set of generic concepts, which is built into GME. The vocabulary of the domain specific languages is based on that set. GME support various concepts for building complex models. GME uses the following basic elements:

Atom

Atom is a basic entity that has no inner structure and cannot contain parts. It has only attributes.

Model

Model is an abstract object that represents something in the real world and which has its own state, identity and behavior. The main purpose of GME is to create and manipulate models. Model has inner structure and it can contain a parts. It contains atoms, other models (hierarchy) or other types of the objects. Model can be opened to show its internal structure.

Connection

Connection represents the relation between two objects and has its attributes. It is represented as a line between these objects. Connection can connect only objects which are in the same diagram or when one diagram is the child of the other. When we have a hierarchical model this can be a problem. To overcome this problem, there are two important concepts in GME available – the references and the sets. There are three different types of connections used in modeling – generalization, containment and association. Generalization is associated with the inheritance of entities.

Containment is a line that connects an object to its container. For example a router contains ports, so the router represents a container and ports represent contained objects. The last type of the connection is an association. This type of connection is used mostly and it expresses an association of the object to another object. It is a ternary relation between the association class and two endpoints.

References and Sets

References and sets are another important concepts in GME, which are used mostly in hierarchical models to connect the objects at a different level of

hierarchy. Reference is an object (not real object, just reference) that represents another object somewhere in the hierarchy and is associated with referred object. It can behave like a pointer or alias. The main difference between pointer and alias is that a pointer is separated from referred object and can by reset to another object during the lifetime, while alias represents an object indistinguishable from a referred object and cannot exist without them.

Sets are used when we need to associate an object with a large number of neighboring objects in the diagram. It can be said that a particular object is a member of the set. Sets can be seen as meta-aspects and they can be replaced by the connection.

While creating a meta-model aspects and constraints can be defined. Aspect specifies the set of objects that can be visible if that aspect is active. By other words, aspects represent different points of view. Each model can have more than one aspect. Each aspect shows related information about the meta-model. On the other hand, constraints are used to express the general validity rules. Object constraints language (OCL) is used to express constraints. This language is based on predicates. The predicate can by true or false and in order to satisfy the constraints it has to be true. To make a control of some objects and their relationships, multiplicities can be used. For example, a container entity can contain optional contained entities (e.g. router contains ports).

After the meta-model is created, it should be interpreted and registered. The output of this process is a set of rules, which represents a paradigm. The paradigm can be used to configure the environment for a specific application domain. Now the specific application model based on this new paradigm can be created. The environment for the specific domain contains components with relevant attributes that were determined in meta-model. If any change occurs in meta-model, it must be reinterpreted and registered again. The existing model is then updated according to a new paradigm.

As shown above, the strength of generic modeling is the ability for exploiting different paradigms, coming out from meta-modeling approach.

The weakness of meta-modeling concept is still in that it comprises in fact just two-stage process of defining paradigm and then (based on this paradigm) a model, without direct binding to the implementation. Then, the implementation of software architecture is the separated process.

3 Process Functional Paradigm

To be able to transform the software architecture model to software architecture implementation, we need the language support for performing the required transformations and the mechanisms for these transformations.

In the past, coming out from functional programming languages [11], process functional language (PFL) has been developed, which allows to express the computation by expression evaluation without assignments, as it is in purely functional languages, but with side effects, as in imperative languages.

The advantage of expression form is clear, when program transformations are required.

The side effects without assignments are achieved by the new concept of environment variable in PFL, which is the essence of process functional paradigm. An environment variable is hidden, from the viewpoint that it is never used in expressions. But it is visible in type definitions of a process. It means that omitting type definition of a process, its definition is purely functional. This, at the first sight perhaps confusing concept is achieved by two-fold meaning of environment variable. First, it is a memory cell. Second, it is mutable abstract type defined by two overloaded operations, as follows. Let *env* represents the environment in which the variable is mapped to a value:

$$env = variable \rightarrow value$$

Then

$$\begin{aligned} &v :: T \rightarrow T \\ &v\,x = x \end{aligned} \tag{1}$$

$$\begin{aligned} &v :: () \rightarrow T \\ &v\,() = env\ v \end{aligned} \tag{2}$$

According to (1), if an environment variable is applied to a value of data type T, it updates the cell v by the identity application. According to (2), if an environment variable is applied to the unit value () of unit type (), the value this application is the value of environment variable *env v.*

The access and update of an environment variable are illustrated in Fig. 1.

Let us introduce now an example of how the side effect is achieved when process *add* defined (with environment variables *u,v* in its type definition) as follows

$$\begin{aligned} &add :: u\ Int \rightarrow v\ Int \rightarrow Int \\ &add\ x\ y = x+y \end{aligned}$$

is applied to data values 1 and 2 by the application (*add* 1 2).

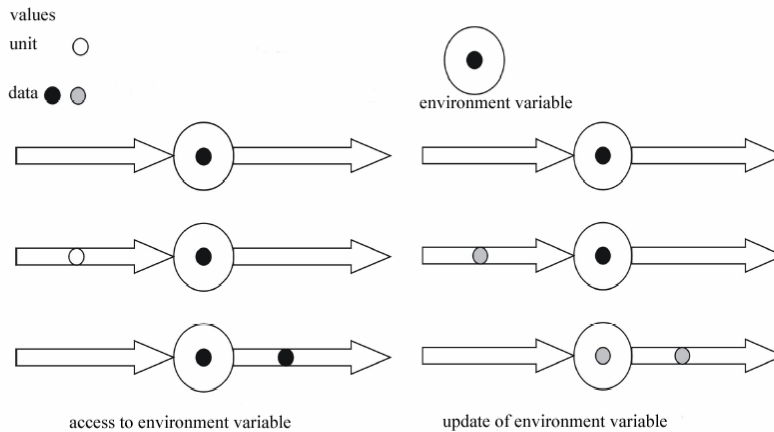

Figure 1
The access and update of an environment variable

The environment variable u is updated to value 1 and variable v to value 2 by the side effects. The new environment is as follows:

$$env = oenv\ [u \mapsto 1]\ [v \mapsto 2] \tag{3}$$

The value of the application will be 3.

Provided that the environment is defined by (3), the application (*add* () ()) does not affect it at all and the value is again 3, since arguments 1 and 2 are accessed from u and v, respectively.

It may be noticed that the same effect we will obtained by pure function *add* (omitting u and v in type definition) and using the application in the form

(*add* (*u* 1) (*v* 2))

In fact, this is an intermediate form for PFL program, since, as mentioned above, we never use environment variables in the source program expressions. On the other hand, we are still able to proceed in backward direction, because we may left function *add* purely functional, we may generate new processes p and q

$p :: u$ Int → Int
$p\ x = x$

$q :: v$ Int → Int
$q\ x = x$

and reflect the values of arguments 1 and 2 to environment variables u and v respectively, by the transformation of application (*add* 1 2) to the form

(*add* (*p* 1) (*q* 2))

But notice that then it is just simple stage to the change of the semantics. For example, substituting *q x = x* by *q x = x+3* above, the value of (*add* (*p* 1) (*q* 2)) will be 6, not 3. Hence, it is possible to add a new aspect of the system preserving the purely functional form of expressions, from the viewpoint that all environment variables occur in types. In this way, memory cells in which values are either reflected or accessed, are systematically separated from code, they are visible, they may be shared or non-shared, they may be the subjects of monitoring, mutual binding by processes, etc. In the next section, we will show, how environment variables of local processes that are aligned to formal parameters of a function, allows implementing aspect paradigm in PFL.

4 Aspect Paradigm in PFL

Aspect paradigm [6,9] has evolved from object paradigm, as a result of consideration that there are programming problems that are insufficiently captured by object paradigm. For example, some parts of the source code are repeated (e.g. logging to a database) what can lead to a tangled code. Aspect-oriented paradigm allows programmers to modularize crosscutting concerns and to weave them in join points, selected by pointcut designators, by weaving. The goal of weaving is to add new advices (parts of code) to many join points, being possibly located in different modules. In AspectJ, there may be three types of advices – after advise, before advice and instead advice.

In this section we adopt the mechanism for the changing the semantics in PFL, introduced above, to the more complex task, how to use it for expressing before advice. Instead of formal description we will introduce the example, which illustrates the ability of PFL for weaving.

Suppose we define advice in the form

elsewhere
  (*f* {*})

advise (*ad* {()}) **where**
  ad :: *x** Int→ *x** Int → Int
  ad u v = *adf* (print *u*) (print *v*)
  adf _ _= *f* {*}

where *x** is used for arguments of picked – out functions that match formal parameter names starting with *x.* The aim of pointcut designator elsewhere (*f* {*}) is to select all applications of the function *f*, regardless of their locations in the program.

Let the function *f* is defined as follows:

f :: Int → Int → Int
f x y = *x* + *y*

and there are just two its applications (f $(xa + xb)$ xc) and (f $(xa * y)$ xb) that occur in the definitions of functions g and h that let be defined as follows:

g :: Int → Int → Int → Int
$g\ xa\ xb\ xc = f\,(xa + xb)\ xc$

h :: Int → Int → Int → Int
$h\ xa\ y\ xb = f\,(xa * y)\ xb$

Based on poincut designator in the advice above, functions g and h are picked out, and for advising there are selected xc and xb formal parameters in g, and xa and xb formal parameters in h.

Based on the advice, the woven form of functions g and h is as follows:

g :: Int → Int → Int → Int
$g\ xc\ xb\ xa = ad$ () ()
  where
    ad :: $\underline{xc}$ Int → $\underline{xb}$ Int → Int
    $ad\ x\ y = adf$ (print x) (print y)

    adf _ _= $f\,(xa + xb)\ xc$

h :: Int → Int → Int → Int
$h\ xa\ y\ xb = ad$ () ()
  where
    ad :: $\underline{xa}$ Int → $\underline{xb}$ Int → Int
    $ad\ x\ y = adf$ (print x) (print y)
    adf _ _ = f $(xa * y)$ xb

which means that before applications of f are performed in picked-out functions g and h, the its arguments $(xa + xb)$ and xc (in g), and $(xa * y)$ and xb (in h) are printed. We underline the environment variables of process ad just for explanation purposes. Let us consider the woven form of g above. First, formal parameter xc and environment variable $\underline{xc}$, reside at the same memory location (on the stack). (The same holds for xb and $\underline{xb}$). Second, the function adf is defined by the application($f\,(xa + xb)$) xc, i.e. in terms of formal parameters xc, xb (and also xa) of g, not in terms of environment variables $\underline{xc}$ and $\underline{xb}$, because environment variables are never used in expressions, as an essential principle of process functional paradigm.

Conclusion

In this paper, we have present the present state of generic modeling, as a software engineering approach, concluding with its main positive property – the ability of forming meta-models for different paradigms. We identify a gap between modeling and implementation approved from the viewpoint of both functional and behavioral aspects of systems using mathematical methods. We have introduced the essence of process functional paradigm, illustrating how it can be exploited in

backward manner to add side effect into side effect–free program, using mathematical transformation into a software system. More complex example in Section 4 illustrates the ability of process functional paradigm (namely its inner reflection property) for the development of a system with aspect paradigm.

Since our grains of computation that are the subject of aspect development may very fine the application areas may affect not just software architectures but the results may be applied in a very specific areas, such grid architectures [2], information environments [3], neural systems [14], communication protocols [12], real-time systems [17].

At the present time, it is well known to us the mechanism for providing source-to-source transformations with or without semantics change as well as the singleton execution mechanism for systems – the application of processes. Our next step will be the detailed analysis of automatic generation principles not just from structure viewpoint [4], but considering the semantics [10].

References

[1] Jimmy Borowski: *Software Architecture Simulation - Performance evaluation during the design phase*, thesis 2004, Inst. för Programvaruteknik och datavetenskap Dept. of Software Engineering and Computer Science, 28 pp.

[2] Marián Babík and Ladislav Hluchý: *Towards a Scalable Grid Ontology*, Proc. of Informatics 2005, 20-21.6, 2005, Bratislava, pp. 159-164, ISBN 80-869243-3-8

[3] Július Baráth, Marcel Harakaľ: *Consolidation and modernization of information environment in praxis*, Proc. of Informatics 2005, 20-21.6, 2005, Bratislava, pp. 355-357, ISBN 80-869243-3-8 (in Slovak)

[4] Matej Črepinšek, Marjan Mernik: *Inferring Context-Free Grammars for Domain–Specific Languages*, Conf. on Language Descriptions, Tools and Applications, LDTA 2005, April 3, 2005, Edinburgh, Scotland, UK, pp.

[5] Marc L. McKelvin, Jonathan Sprinkle, Claudio Pinello, Alberto Sangiovanni Vincentelli: *Fault Tolerant Data Flow Modeling Using the Generic Modeling Environment*: Proc. of 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, pp. 229-235, April, 2005

[6] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, John Irwin: *Aspect - Oriented Programming*: Proc. of European Conference on Object-Oriented Programming, 1997, pp. 220-226

[7] Ján Kollár: *Unified Approach to Environments in a Process Functional Language*, Computing and Informatics, Vol. 22, 2003, pp. 439-456, ISSN 1335-91

[8] Ákos Ledeczi, Miklós Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, Peter Volgyesi: *The Generic Modeling Environment,* Proc. of WISP'2001, May 2001, Budapest

[9] Marjan Mernik, Tomaž Kosar, Viljem Žumer: *A Note on Aspect, Aspect-Oriented and Domain-Specific Languages*, Acta Electrotechnica et Informatica, Vol. 5, No. 1, 2005, pp. 5-12, ISSN 1335-8243

[10] Marjan Mernik, Matej Črepinšek: *Prolog and Automatic Language Implementation Systems*, Acta Electrotechnica et Informatica, Vol. 5, No. 3, 2005, pp. 42-49, ISSN 1335-8243

[11] S. L. Peyton Jones: *The implementation of functional programming languages, University College London*, 1987, pp. 439, ISBN 0-13-453333-X

[12] Daniela E. Popescu, Daniel Filipas, Cristian Tiurbe: *Comparative analyses for a hardware and software implementation of the TCP protocol*, Analele Universitatii din Oradea, Proc. 8th International Conference on Engineering of Modern Electric Systems, Felix Spa-Oradea, May 26-28, University of Oradea, Romania, 2005, pp. 104-112, ISSN 1223-21

[13] Jaroslav Porubän: *Time and space profiling for process functional language*, Proceeding of the 7th Scientific Conference with International Participation Engineering of Modern Electric Systems '03, Felix Spa, Oradea, May 29-31, 2003, Felix Spa - Oradea, University of Oradea, 2003, pp. 167-172, ISSN 1223-2106 *Functional Programs Profilation*, Phd. Thesis, 2004

[14] B. Sivák, J. Škrinárová: *The role of border conditions in optimal approximation real functions with neural nets*, Proc. of Informatics 2005, 20-21.6, **2005**, Bratislava, pp. 97-101, ISBN 80-869243-3-8 (in Slovak)

[15] Václavík Peter, Kollár Ján, Porubän Jaroslav: *Object-oriented Programming with Functional Language*, 8th International Conference ISIM'05, Hradec nad Moravicí, Czech Republic, April 19-20, 2005, pp. 167-174, ISBN 80-86840-09-3

[16] W. Zhao, J. G. Gray, B. R. Bryant, C. C. Burt, R. R. Raje, A. M Olson, M. Auguston: *A Generative and Model Driven Framework for Automated Software Product Generation*: Proc. of the 6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction, pp. 103-108. Portland, Oregon, May 2003

[17] Doina Zmaranda, Gianina Gabor, Claudia Rusu: *Evaluation method algorithm used to improve real-time control systems stability*, Analele Universitatii din Oradea, Proc. 8th International Conference on Engineering of Modern Electric Systems, Felix Spa-Oradea, May 26-28, University of Oradea, Romania, 2005, pp. 170-175, ISSN 1223-2106