AN ANALYSIS OF SOME ASPECTS OF COMPONENT-BASED PROGRAMMING FOR SELECTING APPROPRIATE CATEGORICAL STRUCTURES AS THEIR MODELS

William STEINGARTNER*, Davorka RADAKOVIĆ**, Valerie NOVITZKÁ*, Mohamed Ali M. ELDOJALI*
*Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 042 00 Košice, Slovak Republic,
E-mail: william.steingartner@tuke.sk, valerie.novitzka@tuke.sk, eldojalimohamed@gmail.com
**Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia, E-mail: davorkar@dmi.uns.ac.rs

ABSTRACT

Formal methods and formal models are important tools in software engineering. Formal methods provide unambiguous meaning of programs written in some language or constructed from modules. Moreover, they provide the basic mathematical techniques necessary for those who are working with theoretical background in computer science. Categories are interesting mathematical structures which have become important for constructing the models of programs and program systems. In this paper we formulate an introductory analysis of some aspects of component-based programming for selecting appropriate categorical structures as their models.

Keywords: *category theory, component, contract, designs, interface, interactions, software*

1. INTRODUCTION

Component oriented programming has been, for the past few years, an emergent programming paradigm. Component software with its benefits has recently come to the fore of interest not only in information technology but also on markets, with further growth predicted. The ability of computers and customer demands are increasing and therefore it is necessary to increase the productivity of programs. Satisfied customer is the priority therefore the commitment is to follow its needs and requirements. This is the reason why the markets use the concept of component software. An exponential growth in this segment is expected in near future.

Currently, there is no doubt that the formal specifications are necessary to improve the quality of software systems. Formal specification techniques have a formal syntax and formal semantics which are based on a mathematical theory, for example set theory, logic, algebra and category theory [12]. This contribution explains the basic theory of categories, which can be understood as a general theory of structures. The idea is to describe the properties of structures consisting of objects which include morphisms.

Category theory is a relatively young branch of mathematics. Its influence is being felt in many parts of computer science, including the design of functional and imperative programming languages, implementation techniques for functional languages, semantic models of programming languages, type theory, specification languages, etc.

This paper examines category theory for formal description of components in component model. The work depicts two approaches. The first approach is explained on the MVC model. In the second approach contracts play an important role as they are used when components communicate with each other via interfaces. Contracts are the rights and obligations on both sides of the communication. Pre and post conditions are used in Fig. 2. A contract can be compared to the filter, which accepts any input, but the output will show only data that comply with the contract and these are forwarded.

The remainder of the paper is organized as follows: Section 2 gives an overview of some applications of Component oriented programming. In section 3, we present basic notes of Component oriented programming. Section 4 defines interface as main object for communication between components. Finally, some guidelines for future development and conclusions are closing our paper.

2. RELATED WORK

In this section we present various approaches for handling components and their composition. Chen et al. [8] have proposed a model supporting component-based programming using processes to model application programs and glue programs to build new components from existing ones.

ComponentJ [27] is a Java-like programming language with basic idea to be used as a glue language for existing components that are afterwards used in standard Java code. ArchJava [2] is used for expressing software architectural structure thus it provides the confirmation for implementation of the specified architecture at every stage of the software lifecycle.

In [5], Bidinger et al. have presented the Dream framework, i.e. a domain specific type system for messages and components that manage messages. Dream components [7] are standard Fractal components which allow Dream components to exchange messages that are encapsulated Java objects.

COMPO [28] is a language which extends a reflective programming language, building an executable meta-model which allows introspection and intercession. It can be said that COMPO is an operational prototype for developing completely component-based applications.

Rouvoy and Merle have presented an abstract component model as an annotation framework which assembles the basic concepts of the abstract component model [26]. Fabresse et al. stress five requirements that have to be satisfied to support Component-Oriented Programming [10]: decoupling, adaptability, unplanned connections, encapsulation and uniformity, which they have achieved in new Component-Oriented Language, the extension of Simple Component Language (SCL) [11].

Granström in [14] introduces the notion of *world map* and shows that worlds and world maps form a category with arbitrary products. The construction of the category is carried out entirely in type theory and directly implementable in dependently typed programming languages. After replacing the notion of world map with the standard notion of component, he gets a rigorous paradigm for component-based development.

3. COMPONENT ORIENTED PROGRAMMING

Component oriented programming (COP) is a technique of developing software applications by combining pre-existing and new components [14]. Software components are self-contained, self-describing packages of functionality containing definitions of types that expose both behavior and data. There are a number of important reasons why COP is important. It provides a higher level of abstraction. There are an increasingly large number of reusable component libraries that assist in the development of applications for various domains.

There are three major goals of COP [30, 32]:

- *Conquering complexity:* COP provides an effective way to deal with the complexity of software: divide and conquer.
- *Managing change:* the user requirements change, specifications change, personnel change, budget changes, technology changes, and so on. Components are easy to adapt to new and changing requirements.
- *Reuse:* software reuse allows to design and implement something once and to use it over and over again in different contexts. This will realize large productivity gains, taking advantage of best-in-class solutions, the consequent improved quality, and so forth.

COP supports highest level of software reuse because it allows various kinds of reuse including white box, gray box and black box reuse. White box reuse means that the source of a software component is made available and can be studied, reused, adapted, or modified. Black box reuse is based on the principle of information hiding. The interface specifies the services a client may request from a component. The component provides the implementation of the interface that the clients rely on. As long as the interfaces remain unchanged, components can be changed internally without affecting clients. Gray box reuse is somewhere between white box reuse and black box reuse.

COP provides a manageable solution to deal with the complexity of software, the constant change of systems, and the problems of software reuse. COP is now the paradigm for developing large software systems [30, 32].

3.1. Software Component

The software component can be defined as an independent part, self-deployable computer code with well-defined functionality which can be stacked with other components via the interface. The component is a program or collection of programs that can be compiled and execute. It is independent so as to provide a coherent functionality. It is self-deployable and it can be stacked with other components so they can be reused as a unit in different contexts. Integration, thus joining portion together is done through the interface component, which means that the implementation of the internal component is typically hidden from the user. The component technologies that meet these definitions include Java and Enterprise Java Beans (first mentioned by Sun Microsystems), the COM (Component Object Model), DCOM (Distributed Component Object Model), and .NET components from Microsoft Corporation, CORBA (Common Object Request Broker Architecture) [32].

The component technology field is currently dominated by three players: Microsoft (D)COM, OMG CORBA, and Java. When comparing these technologies with respect to attributes such as distribution, mobility, language and platform independence, that there are many differences as listed in Table 1. First of all, notice that component technology does not automatically mean distribution. For example, JavaBeans and Microsoft COM do not support distribution (denoted by $-$ in Table 1). Secondly, whereas language independence seemed to be of importance in the pre-Java era (denoted by $*$ in Table 1), that is for (D)COM and CORBA, it is not so for the Java-based solutions. Finally, platform independence (denoted by $*$) is hard to achieve. But, fortunately, it is on the agenda of all three technologies, including (D)COM.

It is worth mentioning that the three major technologies have a rather different origin. Microsoft (D)COM is primarily a desktop technology, with Office as its killer application, whereas CORBA originated from the need to have an enterprise-wide solution for distributed objects. Java is a special case. It started as a Web-based language, but rapidly took position in the desktop and enterprise world as well.

Table 1 Technology comparison

	distribution	**mobility**	**language**	**platform**
COM	–	–	*	–
DCOM	+	–	*	+/–
CORBA	+	–	*	*
JAVA/Beans	–	classes	Java	*
Java/RMI	+	classes	Java	*
Voyager	+	objects	Java	*

3.2. Component versus Object

The concepts *component* and *object* are often used interchangeably. Both components and objects are making their services available via well-defined interfaces, have encapsulation properties, are considered to improve the reuse of software, are considered to alleviate the software evolution phase, are thought of being natural abstractions of real-world entities, and a real-world entity can be modeled or implemented using either notion.

Their main conceptual differences according to [24, 6] are presented in Table 2. Software components are units of composition with contractually specified interfaces and context dependencies only. Software component can be deployed independently, is subject to composition by third parties. A system built up from components is more robust, is more flexible, has a shorter development time and the foremost advantage is reuse of software.

An object is abstraction from a real-world entity, with associated items of information and a set of specific operations. An object has a unique and invariant identifier, a class to which it belongs, a state that has a certain value.

Table 2 The conceptual difference

Objects	Components
describe / implement real-world entities (and their hierarchies)	describe / implement services of real-world entities
mathematical modelling approach to software	engineering approach to software
partition the state space	partition the service space

Regarding component oriented programming there is a new component oriented language based on Java which is called BoxScript.

BoxScript supports two main properties of component oriented programming compositionality and flexibility. Design of BoxScript seeks to address the needs of teachers and students for a clean and simple language. The language builds upon an existing programming language with which students are likely to be familiar. A component in BoxScript is called a *box* which is a black box entity, its internal details is strongly encapsulated, its interface is only exposing to the outside. A group of boxes can be composed to form a larger box that provides some higher-level functionality. The units of code needed to build a box are a box description, interfaces and their implementations, configuration information, and box manager code.

There are two types of interfaces in BoxScript, a provided interface and a required interface. A provided interface describes the operations that a box implements and other boxes may use. A required interface describes the operations that the box requires and that must be implemented by another box. A box has to have at least one provided interface. Required interface is not necessary. BoxScript uses Java classes to implement the interfaces. BoxScript defines two types of boxes. An abstract box serves as an abstract type for which no implementation is provided. It describes the provided and required interfaces but does not implement the provided interfaces. A concrete box provides implementation that delivers concrete functions. Concrete box can be atomic or compound. A basic element in BoxScript is an atomic box that doesn't contain any other boxes. A compound box is composed from atomic boxes or other compound boxes [19, 20].

3.3. Categorical Approach

Category theory is the mathematical theory of structures used also in theoretical computer science. The advantage of categories is, that they enable to model components for different program systems. Category theory is the mathematical basis for a unified description of component-based techniques for the various modeling techniques because it allows the formulation of basic concepts independently of the particular formalism.

A software-engineering approach to components is well-known and nowadays also widely used. But the formal model of component systems has not been defined uniquely. One of the fruitful approaches is to define a formal model within categorical structures. The first use of category theory for the purpose of large scale system construction seems to be Goguen [13]. However, in that work, components are objects of the category and morphisms of the category represent communication between components. Moreover, the idea of considering a component as a morphism between interfaces can be found in a PhD thesis due to Barbosa [3].

Category theory enables efficient way of demonstrating results. The main evidence can be performed at an abstract level, the specification of a large number of results obtained in the small effort [17, 23]. Category theory provides more possibilities for modeling component base program systems [23, 16]. If we are interesting in component composition then it is suitable to model component interfaces as category objects. In the case of modeling interactions as category morphisms we have two possibilities: either we can construct category morphisms as mappings expressing functionality of interactions, but then we can use only quasi-category where the composition of two morphisms need not be uniquely defined. Or we can model them as relations which lead to relational categories. The latter approach is not obvious and it requires deeper analysis.

If we are interesting in modeling observable behavior of component program systems, it is suitable to use coalgebras [3], where polynomial endofuctor is constructed to model behavior of a system step by step. This endofunctor is constructed over category of states. In this category we consider morphisms as transitions (transition relations) among states.

4. INTERFACE

A component is often characterized as a black box with only visible part called interface. Components in COP communicate with each other via interfaces. A component has multiple interfaces which are sets of operations, called methods. Sometimes interfaces are grouped to make ports which become points of interactions.

Each component will provide and require pre-specified services from other components; hence, the notion of component interfaces becomes an important issue of concern. Interfaces are the mechanisms by which information is passed between two communicating components. A component may either directly provide an interface or implement objects that, if made available to clients, provide interfaces. The interface is used as the contract between the component and the client. It provides the services that the component is ready to provide them to clients. Interface specification therefore describes information needed for the important elements of the component model. The component model defines the components that make up the interface as well as semantics, the importance of these elements. Usually the part of the interface include names of methods, parameters of methods and valid types of parameters [30].

4.1. Model-View-Controller

Model-View-Controller (MVC) is the name of a methodology or design pattern for successfully and efficiently relating the user interface to underlying data models.

- Model: represents the structure of data in the application, as well as application-specific operations on those data.
- View: renders the contents of a model. It specifies exactly how the model data should be presented.
- Controller: translates user actions and user input into application function calls on the model and selects the appropriate view based on user preferences and model state.

MVC is used to create a component model of some product (picture). Basic classes are ProductModel, ProductView and ProductController. ProductModel defines a product, it means product name, product ID and product count. ProductController connects model and view. ProductView provides product information. There are two interfaces: ViewControllerInterface and ModelControllerInterface.
We define interface $I =< T, M >$ where T is name of type and M is name of method.

$ModelControllerInterface =< T, M >$
$T = string, int, void$
$M = getName : string$
$\quad getProductID : string$
$\quad getProductCount : int$
$\quad setName(String) : void$
$\quad setProductID(String) : void$
$\quad setProductCount(Int) : void$
$IgetName = Istring+$
$T : string$
$M : name : ProductModel \longrightarrow string$
$IsetName = Istring+$
$T : string$
$M : name : ProductModel \longrightarrow 1$

In the same way we defined other methods and types. In Fig. 1 is specific MVC model with categories ProductView, ProductModel and ProductController. Interfaces ViewControllerInterface and ModelControllerInterface are arrows f, g (methods).

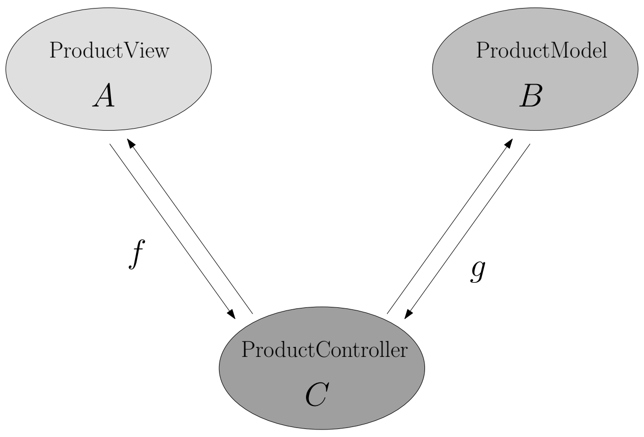

Fig. 1 Categories in MVC model

4.2. Contract

A useful way to view interface specifications is as contracts acting between a client of an interface and a provider of an implementation of the interface. The contract states what the client needs to do to use the interface. It also states what the provider has to implement to meet the services promised by the interface. On the level of an individual operation of an interface, there is a particularly popular contractual specification method. The two sides of the contract can be captured by specifying pre-and postconditions for the operation. The client has to establish the precondition before calling the operation, and the provider can rely on the precondition being met whenever the operation is called. The provider has to establish the postcondition before returning to the client and the client can rely on the postcondition being met whenever the call to the operation returns.

In Fig. 2 is a component A and a component B, an interface I, a contract C_1 and a contract C_2. Component A sends data to component B and Component B sends data to component A. Both components communicate with interface I. Component A receives only integer data type and component B receives only data which its value is more than zero. Because of these conditions we use contract. Contract C_1 receive all data from A but sends only number more than zero to interface I. Contract C_2 receives all data a sends integer data to interface I.

We define interface $I = \langle T, M, C\rangle$ where T is name of type, M is name of method and C is name of contract.

```
$getNumber = \langle T, M, C\rangle$
$T = int$
$M = number : Component\ A \longrightarrow int$
$C =$ if $isNumber : int$ then
    $getNumber : Component\ B$
 else $exception$
$getNumber = \langle T, M, C\rangle$
$T = int$
$M = number : Component\ B \longrightarrow int$
$C =$ if $isMoreThanZero : int$ then
    $getNumber : Component\ A$
 else $exception$
```

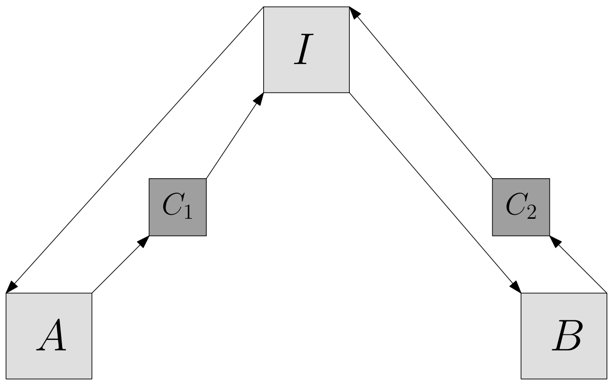

Fig. 2 Contracts in component model

4.3. Contracts and Dependencies

To be a system composed from components health, some additional information for successful composition is needed. This information is known as

- contracts and
- context dependencies.

Contracts create a common basis for successful composition and interactions between components in a program system. The basic conditions stated by contracts involve:

- correlations between ports constructing data and the ports extracting these data;
- requirements to the ports of other components that should be satisfied for making given component working;

Contracts can be considered as specifications of non-functional requirements. Their role is to state obligations for achieving desirable behavior of components in program systems [31, 21]. The first formal specification of contracts follows from algebraic specifications of abstract data types, i.e. signatures and axioms formulated in some logical system extended with some constraints specific for given component. Among simple examples of contracts we can list component interoperability, pre- and postconditions of Hoare's logic, invariants, etc. [15, 18, 22].

A contract is a pair

$$(A, G)$$

where A is a specification of assumptions and G is a specification of guarantees.

- *Assumptions* contain requirements on environment of a component and
- *guarantees* formulate what provide a components if assumptions are satisfied.

Both specifications of assumptions and guarantees can be formulated in some specification theory. If we consider for a component its assumption specification A and its guarantee specification G as abstract data type, we can say that $A <: G$, i.e. assumption specification is "subtype" of guarantee specification [9]. Composition of two components leads to working system if all assumptions and guarantees of both components are satisfied. Contracts for interaction between two composed components are illustrated in Fig. 3.

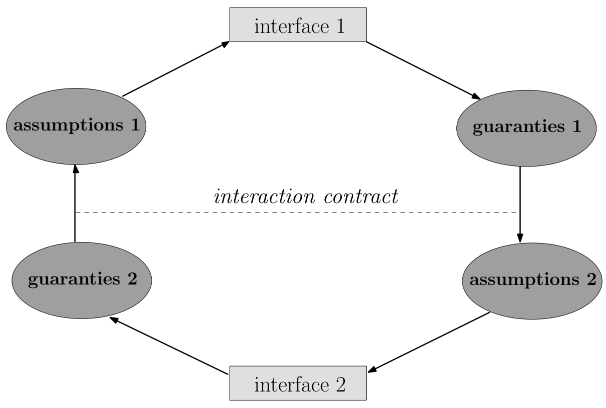

Fig. 3 Interaction contract

The mostly used specification methods for specifying interaction contracts are:

- *trace-based specifications* [4] where assumption and guarantees are specified as sets of runs and
- *modal contracts* [25] based on modal *may*- and *must*-transitions.

Contracts are not enough for successful and working composition of components. Another information is needed: *context dependencies*. Dependency can be characterized as follows: if a component C_1 uses a component C_2, we say that C_1 depends on C_2. Dependencies formulate conditions for reusing and upgrading components. An environment consisting of a set of components together with their context dependencies is called *repository*. Context dependencies involve:

- composition context dependencies - requirements on environment for successful composition;
- deployment context dependencies - possible platforms (hardware and software) where component can work.

Due to different application domains there are many ways how to build program systems from components. Most of the components can be considered as data structures with explicit or implicit data types, subprograms, collections of subprograms, etc. One component can require some modification in another one. This relation between new and existing components is one of the forms of dependencies. A simple example of dependencies between components is e.g. in [33]. Let C_1 and C_2 be components with corresponding explicitly typed data structures. Authors classify known and frequently occurred context dependencies as:

- *data dependencies*: a value in C_1 can influence data of C_2, or data of C_1 can be used for computing in C_2;
- *type dependencies*: type definitions and their changes in C_1 can influence data types in C_2;
- *subprogram dependencies*: executing some procedure or function of component C_1 by calling with data from C_2 causes this kind of dependencies;
- *source file dependencies*: if some source file serves as a common source for both components C_1 and C_2 in program system;
- *source location relationships*,
- *time and space dependencies*, and many others.

Deployment context dependencies express hardware and software platform for successful deploying of components. Here belong hardware architectures, operating systems and their versions, representations of built-in data, etc. Deployment context dependencies become important in implementation process and we will think they are hardly formalizable.

In newer literature, e.g. in [1] dependencies are classified into two groups:

- positive dependencies,
- negative dependencies called *conflicts*.

Components relationships are described by *dependency graph*, i.e. oriented graph where nodes are components and oriented edges express dependencies or conflicts. A system is said to be healthy when all components have their dependencies satisfied and all conflicts unsatisfied.

From the previous analysis of dependencies follows that dependencies can be described by first order formulae in some appropriate logical system.

After this short analyzing and classifying of contracts and dependencies we can extend the definition of components with the following part: A software component is a unit of composition with contractually specified access points and explicit context dependencies. Contracts and composition context dependencies are stable parts in component-based programming and they seem to be the first adepts for rigorous formalization.

5. THE NEXT GOALS

The aim of our research is to formulate formal framework for specifying and modeling component program systems. This idea we presented in [29]. Because of complexity of this problem, it would be reasonable to construct this framework hierarchically with three layers illustrated in Fig. 4.

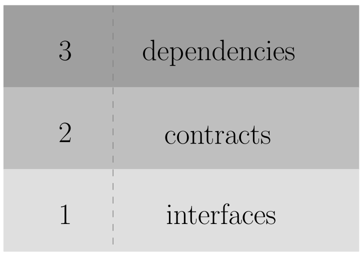

Fig. 4 Proposed layers for formal framework

The layer 1 concerns with interfaces and interactions between components. An interface can be specified using algebraic specification consisting from signature and axioms. A signature contains

- typed ports;
- specifications of operations.

Axioms can be written as obvious in equational logic or in predicate linear logic. First layer component system then can be modeled as a category of interfaces, where objects are representations of interfaces (Σ-algebras) and morphisms are interactions between components.

The role of second layer is to consider also contracts. There are two possibilities:

- to extend specifications of interfaces by assumptions and guarantees or
- to formulate them by formulae in predicate linear logic.

Both solutions enable to protect ”subtype” relation between assumptions and guarantees. This layer can restrict possible interactions between components to satisfy contracts.

On the third layer dependencies will be introduced. Dependencies we would like to formulate as predicates of other formulae in predicate linear logic and model in an appropriate category.

Our preliminary idea how particular layers are interconnected is to use some functors or natural transformations with suitable properties.

6. CONCLUSION

The size and complexity of software systems is growing as well as the demands of customers, component-based programming is the appropriate way to deal with this problem. When creating a system of components it is important in their proper interaction, and hence its description plays an essential role. After deeper analysis we can say that category theory provides an efficient way to describe connecting components and thus improve the quality of software. In our next research we want to focus on classification of categories that will be used in construction of formal model of component system.

ACKNOWLEDGEMENT

This work has been supported by Grant No. 002TUKE-4/2017: Innovative didactic methods of education process at university and their importance in increasing education mastership of teachers and development of students competences.

REFERENCES

[1] ABATE, P. – BOENDER, J. – Di COSMO, R. – ZACCHIROLI, S.: Strong dependencies between software components, Tech. rep.0002, 7th Framework Programme fp7-ict-2007, Universitè Paris Diderot, 2009.

[2] ALDRICH, J. – CHAMBERS, C. – NOTKIN, D.: Architectural Reasoning in ArchJava, *In: Proceedings of the 16th European Conference on Object-Oriented Programming* (London, UK, UK, 2002), ECOOP ’02, Springer-Verlag, pp. 334–367.

[3] BARBOSA, L.: *Components as coalgebras*, PhD thesis, Departamento de Informática, Escola de Engenharia, Universidade do Minho, 2001.

[4] BENVENISTR, A. – CAILLAUD, B. – FERRARI, A. – MANGERUCA, L. – PASSERONE, R. – SOFRONIS, C.: Formal methods for components and objects, Springer-Verlag, Berlin, Heidelberg, 2008, Multiple Viewpoint Contract-Based Specification and Design, pp. 200–225.

[5] BIDINGER, P. – LECLERCQ, M. – QUÉMA, V. – SCHMITT, A. – STEFANI, J.-B.: Dream types: A domain specific type system for component-based message-oriented middleware, *SIGSOFT Software Engineering Notes*, Vol. 31, No. 2 (Sept 2005).

[6] BOSCH, J. – SZYPERSKI, C. – WECK, W.: *Component-Oriented Programming*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 70–78.

[7] BRUNETON, E. – COUPAYE, T. – LECLERCQ, M. – QUÉMA, V. – STEFANI, J.-B.: *An Open Component Model and Its Support in Java*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 7–22.

[8] CHEN, X. – HE, J. – LIU, Z. – ZHAN, N.: A model of component-based programming, *In: Proceedings of the 2007 International Conference on Fundamentals of Software Engineering* (Berlin, Heidelberg, 2007), FSEN’07, Springer-Verlag, pp. 191–206.

[9] ENSELME, D. – FLORIN, G. – LEGOND-AUBRY, F.: Design by contracts: Analysis of hidden dependencies in component based applications, *Journal of Object Technology*, Vol. 3, No. 4 (2004), pp. 23–45.

[10] FABRESSE, L. – BOURAQADI, N. – DONY, C.: Component-oriented programming: From requirements to language support, *In Marcus Denker and Gabriela Arevalo editors, Proceedings of the 4th Smalltalks’2010 Conference* (2010).

[11] FABRESSE, L. – DONY, C. – HUCHARD, M.: Foundations of a simple and unified component-oriented language, *Computer Languages, Systems and Structures*, Vol. 34, No. 2-3 (July 2008), pp. 130–149.

[12] GALINEC, D. – STEINGARTNER, W. A look at observe, orient, decide and act feedback loop, pattern-based strategy and network enabled capability for organizations adapting to change, *In: Acta Electrotechnica et Informatica*, Vol. 13, No. 2 (2013).

[13] GOGUEN, J.: Categorical foundations for general systems theory, *In: Advances in Cybernetics and System Research* (1973), Transcripta Books, pp. 121–130.

[14] GRANSTRÖM, J.: A new paradigm for component-based development, *Journal of Software*, Vol. 7, No. 5 (May 2012).

[15] HAN, J.: An approach to software component specification, In *Proceedings of 1999 International Workshop on Component Based Software Engineering* (Los Angeles, USA, 1999).

[16] KNIGHTEN, R. L.: Notes on category theory, 2007.

[17] KOSTECKI, R. P.: An introduction to topos theory, Tech. rep., Institute of Theoretical Physics, University of Warsaw, 2011.

[18] KOZACZYNSKI, W.: Composite nature of component, *In: International Workshop on Component-Based Software Engineering* (1999), pp. 73–77.

[19] LIU, Y. – CUNNINGHAM, H. C.: Boxscript: A component-oriented language for teaching, *In: Proceedings of the 43rd Annual Southeast Regional Conference - Volume 1* (New York, NY, USA, 2005), ACM-SE 43, ACM, pp. 349–354.

[20] LIU, Y. – CUNNINGHAM, H. C.: Java in the box: implementing the boxscript component language, *In: Proceedings of the 45th Annual Southeast Regional*

Conference, 2007, Winston-Salem, North Carolina, USA, March 23-24, 2007 (2007), pp. 47–52.

[21] MESSABIHI, M. – ANDRÉ, P. – ATTIOGBÉ, C.: Multilevel contracts for trusted components, *In: The 7th International Conference on Software Engineering Advances* (2012), ICSEA, pp. 71–85.

[22] MEYER, B.: Applying design by contract, *Computer*, Vol. 25, No. 10 (2002), pp. 40–51.

[23] NOVITZKÁ, V. – SLODIČÁK, V.: *Categorical structures and their applications in informatics*, Equilibria, 2010, (in Slovak).

[24] PETRE, L.: Components vs objects, TUCS Technical Reports No. 370, Turku Centre for Computer Science, 2000.

[25] RACLET, J.-B. – BADOUEL, E. – BENVENISTE, A. – CAILLAUD, B. – LEGAY, A. – PASSERONE, R.: A modal interface theory for component-based design, *Fundam. Inf.*, Vol. 108, No. 1-2 (2011), pp. 119–149.

[26] ROUVOY, R. – MERLE, P.: Leveraging component-oriented programming with attribute-oriented programming, In: *11th International ECOOP Workshop on Component-Oriented Programming* (2003), WCOP'06, pp. 10–18.

[27] SECO, J. – SILVA, R. – PIRIQUITO, M.: ComponentJ: A component-based programming language with dynamic reconfiguration, *Computer Science and Information Systems*, Vol. 5, No. 2 (2008), pp. 63–86.

[28] SPACEK, P. – DONY, C. – TIBERMACINE, C.: A component-based meta-level architecture and prototypical implementation of a reflective component-based programming and modeling language, *In: Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering* (New York, NY, USA, 2014), CBSE'14, ACM, pp. 13–22.

[29] STEINGARTNER, W. – NOVITZKÁ, V. – BENČKOVÁ, M. – PRAZŇÁK, P.: Considerations and ideas in component programming - towards to formal specification, *In: Central European Conference on Information and Intelligent Systems* (2014), CECIIS, pp. 332–339.

[30] SZYPERSKI, C.: *Component Software beyond Object- Oriented Programming*, ACM press, New York, USA, 2005.

[31] URTING, D. – BAELEN, S. V. – HOLVOET, T. – BERBERS, Y.: Embedded software development: Components and contracts, 2001.

[32] WANG, A. J. A. – QIAN, K.: *Component-Oriented Programming*, Wiley-Interscience, 2005.

[33] WILDE, N.: Program dependencies, Tech. report SEI-CM-26, Carnegie Mellon, 1990.

Received February 19, 2017, accepted May 30, 2017

BIOGRAPHIES

William Steingartner works as Assistant Professor of Informatics at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia. He defended his Ph.D. thesis "The Rôle of Toposes in Informatics" in 2008. His main fields of research are semantics of programming languages, category theory, compilers, data structures and recursion theory. He also works with software engineering.

Davorka Radaković works as teaching assistant at Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad. She received her Diploma degree in Mathematics at Faculty of Sciences, University of Novi Sad, Serbia, in 2001 and become Magister of Computer Sciences from the same University in 2010 with thesis "A modular extensible platform for dynamic geometry". Her scientific research is focusing a development of a platform for dynamic geometry.

Valerie Novitzká works as a Full Professor of Informatics at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia. Her fields of research include semantics of programming languages, non-classical logical systems and their applications in computing science. She also works with type theory and behavioral modeling of large program systems based on categories.

Mohamed Ali M. Eldojali is a Ph.D. student at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia. His main field of research covers Coalgebraic Models for Component Based Program Systems.