Considerations and Ideas in Component Programming - Towards to Formal Specification

William Steingartner, Valerie Novitzká, Martina Benčková, Peter Pražňák
Faculty of Electrical Engineering and Informatics
Technical University in Košice
Letná 9, 04200 Košice, Slovakia
{william.steingartner, valerie.novitzka, martina.benckova}@tuke.sk, praznak@minv.sk

Abstract. *Programming in present can be characterized as a composition of some prepared components. New market was opened for providing components for various applications. However, every component library uses different standards, methods and requirements for using its products. Another problem is different terminology and a lack of some common standards. This situation can be compared with the period before setting open systems standards. We assume that analyzing of common principles of component programming can help to find unified formal description and it enables to construct formal models serving for solving and preventing troubles in composing appropriate components into working program systems. In our paper we try to present our analysis and preliminary ideas for achieving these aims.*

Keywords. component programming, non-classical logics, category theory

1 Introduction

In the last decades, new method of software development - component oriented programming - has become popular. It can be shortly characterized as a construction of program system from existing software entities by composition. At the first sight, component oriented programming can be viewed as a continuation of object oriented programming, but there are many important differences. Component oriented programming enables reusing of prepared components in quite different program systems. Reusability requires some independence of components and some rules how to compose and deploy them to achieve a working program system.

In the past, program systems were developed from the beginning, only with help of some tools and libraries. The libraries of procedures can be considered as the first examples of the simple form of component programming. This approach has some advantages, e.g. simple modifications, employing knowledge and skills of clients, etc. But software developed in this manner had high cost and it cannot be optimally used out of given institution. Involving some new functions in such systems, e.g. web access, local or global interoperability can be very difficult for programming teams. Present world is characteristic with rapidly changing requirements and this classical programming approach is not able to keep up with it.

Now, there are several companies providing component libraries that can be used in different systems as needed. Such purchased software can be adapted to user requirements only slightly, it often enforces great changes in the work mode in user institution. But all administration, upgrading evolution and interoperability is out of users. Among such known companies providing components of wide spectrum belong OMG (Object Management Group), CORBA (Common Object Request Broker Architecture), Microsoft COM (Component Object Model), CLR (Common Language Runtime) and many others. These companies compete on market; they have no common rules and no common standards for provided components. Development of various components, their accessibility and reusability leads to the situation when component-oriented programming becomes a necessity, the main programming method in software engineering.

There are many publications and web pages about component programming, composition of components, and methods for solving arisen problems. Most of them deal with implementation details and issues, only small number of them tries to provide some principal, exact formal approach. Also in the terminology, as in every rapidly evolved discipline, is some confusion.

The aim of our research is to find and formulate an appropriate formal framework for specifying and modeling component program systems with provable properties. Therefore our effort is to formulate unified terminology, to analyze particular basic notions and their properties, and to define formal description of basic principles for composition of components. This paper contains short analysis and our ideas considerations about formal description and modeling component-oriented program systems.

In this paper, in section 2 we analyze differences be-

tween objects and components. Section 3 contains basic notions used in component programming, starting with interfaces, composition, interactions, contracts and dependencies. In section 4 we present our preliminary ideas, proposed formal tools and methods how to formulate formal framework serving for provable description and modeling of program systems composed from components.

2 Component versus object

The notion of component is one of the notions with many different definitions. First, we consider differences between objects and components, we present definition of a component and we mention some properties that can be considered within the work with components.

Components are often compared with objects and/or classes in object oriented programming. Both objects and components provide their services trough some access points of some types. Between objects and components are some interactions that can be described by some patterns and frames. The basic difference between components and objects we list in the following text.

The main difference between component oriented programming and object oriented programming is that component oriented programming places importance on interfaces and composition while object oriented programming on classes and objects [17, 19, 34].

An object can be viewed as

- a unit of instantiation having unique identity sufficient to its identification;
- it has externally observable state;
- it encapsulates its behavior, only interface is visible, not implementation details.

A component

- is an independent deployable entity by composition;
- it has to have all important features, it must be deployed whole, not partially;
- composition is successful when some agreements and environment requirements are satisfied;
- has hidden implementation details from user;
- has involved interactions with environment by well-defined typed ports;
- can be a subject of separate compilation;
- has no externally observable state; its initial state is established after its deployment.

Many definitions of components are in literature. For instance, among newer formulations in [4, 5, 29] belongs the following characterization: a component is a software implementation that can be deployed independently and it is a subject to composition by the third parties. This definition does not express all important properties of components. Sometimes, a component is compared with a black box, because of hiding implementation details, it can be reused and it can have some rules to be deployed in a program system in such a manner that the whole system provides desired results.

Components can be considered as some classes known from object oriented paradigm and their deploying and instantiations lead to some kind of objects. However, components have more properties than objects, even a component can involved several objects [17, 35].

Often, to guarantee functionality of a component in given program system, some own user code, additional information and external files have to be used. Components can be generic, i.e. substituting parameters (of proper types) by appropriate arguments enable their using for different purposes [2, 13, 31].

There are other comparisons of components with existing entities in software engineering. For instance, a component is similar to module as it is known from programming language Modula [37] or to package in Ada [7]. Both support separate compilation that enables type checking across modules. Packages can be also generic and instantiated by substituting (type) parameters with arguments.

The world of components is still in evolution. We try to characterize some so called stable common features as some starting points for our formal approach.

3 Basic Notions of Component Oriented Programming

In this section we state the terminology and we characterize basic notions needed for our approach. We start with interfaces, visible parts of components, then we analyze composition and interactions between components and we characterize necessary information for successful composition, i.e. contracts and dependencies.

3.1 Interfaces

A component is often characterized as a black box with only visible part called *interface*. All information, i.e. data structures, data types, procedures and ports without their implementation details constitute an interface [38]. Only the content of an interface defines what can be used in program system after deploying a component. Hidden part contains implementation details that are non-accessible for users. An interface contains

- *typed data structures* usable and movable between components;
- *operations* over typed data structures;
- *typed ports* that are access points for moving data. Every port is input/output for receiving/sending data to corresponding port(s) of other component(s) and serves for data of some specific type.

An interface is often specified as an abstract data type. Typed ports are very important part of any interface because cooperation between components can be performed only through corresponding ports. Ports serve as end points of interactions, they enable transfer of data of some type in required direction [24]. Components are the subjects of composition to build working program system.

3.2 Composition of component system

Composition of components requires first to construct some *composition model*. It can be considered as a basic architecture expressing the roles of components, their interconnecting, relations between them and the rules of composing components together. Relations between components are called *interactions* [9].

It is not precisely established how a model for component systems have to be built [32]. However, it is usable to extract the most important features of a program system, to specify necessary known interconnections between them and to encapsulate them into appropriate structure with exactly defined properties. Such model creates a basis for independence and cooperation of components on some abstract level. It can be characterized as a holistic view of a program system, defining invariants, i.e. properties characterizing all the actual program systems constructed on the base of this model. A model can be based on principal considerations on functionality and it should state some decision policy ensuring correct interoperability of components. In such complex and large systems as component-based program systems are, it is obvious to construct models in hierarchical mode with several layers.

Certainly, in any model we abstract from a computer architecture or implementation details. For our purposes we consider a model as a collection of

- component frameworks and
- the rules of interoperability between these frameworks.

Component frameworks contain specification of interfaces and typed ports together with decision policy on the component level. The rules of interoperability state the conditions for successful cooperation between components in the form of defining interactions. In the following subsection we try to classify known and frequently occurred interactions in the frame of component model.

3.3 Interactions

It is interesting that different publications provide different classifications of interactions. In the following literature [36, 32, 1, 20] authors form various patterns of interactions. On this base we try to classify interactions by three criteria:

- interactions based on pre- and post- conditions;
- interactions that are dealing with concurrency and indeterminism;
- interactions based on what makes choice and what follows-up.

The first group encapsulates interactions specified on the base of Hoare's pre- and post- conditions. If c_1, c_2 and c_3 are preconditions and/or postconditions, p is an input or output port, we can distinguish the following interaction patterns:

- *send pattern sequence*: before executing function $send$ it is satisfied pre-condition c_1, after it post-condition c_2 is satisfied. This pattern is illustrated in Figure 1;
- *pre- blocking send*: in this pattern a message is sent through port p. After pre-condition c_1 until message is not consumed, the sender component blocks all activities as it is in Figure 2;
- *post- blocking send*: it is similar pattern as before but sender component blocks all activities after satisfying post-condition c_2 until c_3. This pattern is in Figure 3;
- *receive message*: in this pattern receiver blocks all activities until whole sent message is received;
- *lossy receive*: if the input port of receiver is not able to accept sent message, a message is lost.

The second kind of interactions concerns concurrency and cases of internal and external indeterminisms. The difference between internal and external indeterminism can be roughly specified as follows: internal indeterminism does not depend on some decision or situation in environment; in external indeterminism chosen activity depend on some external decision arising from environment. In this kind of interactions we distinguish the following patterns:

- *concurrent send (furcation)*: message is sent into two different receiver ports concurrently;
- *concurrent receive (rendezvous)*: two messages are receiving from two different ports concurrently,
- *sending choice (free choice)*: one message from two ready messages is sent according to choice of sender;

- *receiving choice (dependent choice)*: one of two sent messages is received depending on environment; it is external indeterminism;
- *internal choice*: it is combination of the two possibilities above, the service does not make decision, neither the environment. This case is internal indeterminism.

The third kind of interactions contains interactions based on what makes choice and what follows. In this kind we distinguish:

- *sending choice receiving follow-up*: the sender makes a choice and get information to the environment and then the environment send a message depending on the initial choice;
- *receiving choice sending follow-up*: the environment affects a choice of sender;
- *sending choice sending follow-up*;
- *receiving choice receiving follow-up*;
- *internal choice sending follow-up*.

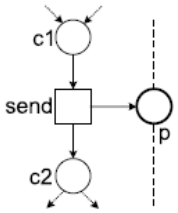

Figure 1: Send pattern (sequence)

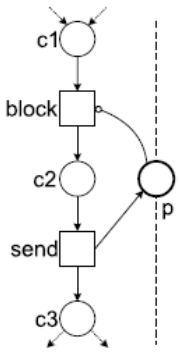

Figure 2: Pre- blocking send

The interactions above to be executed we have to use typed ports. However, there can arise such situations in real world when ports are faulty. These not desired events can be caused by some data or transmission error, mismatching of protocols or incompatible data models. These situation have to be carefully treated because they cause shutting down interactions between components[21].

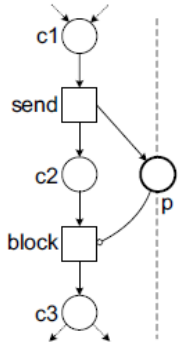

Figure 3: Post- blocking send

3.4 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 [33, 23]. 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 post conditions of Hoare's logic, invariants, etc. [12, 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.

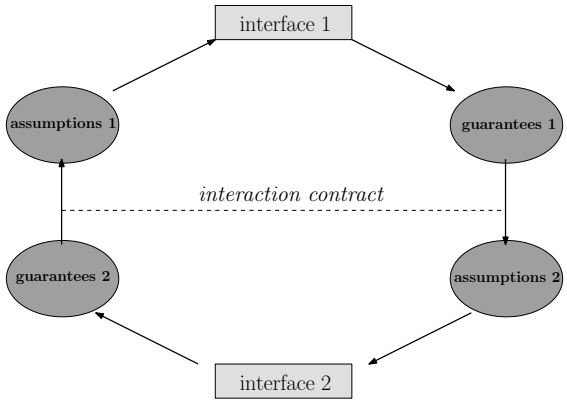

Figure 4: Interaction contract

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 types, we can say that $A <: G$, i.e. assumption specification is "subtype" of guarantee specification [10]. 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 is illustrated in Figure 3.4.
The mostly used specification methods for specifying interaction contracts are:

- *trace-based specifications* [8] where assumption and guarantees are specified as sets of runs and
- *modal contracts* [30] based on modal *may*- and *must*- transitions.

Contracts are not enough for successful and working composition of components. Another information is nedded: *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 component. 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 [36]. 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 of C_1 can influence data of C_2, or data of C_1 can be used for computing 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 [3] 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.

4 Towards Formalization of Component Composition

In this section we present our preliminary ideas how formalize principles and properties of program systems constructed by composition of components. First we give reason about formal tools we would like to use and then we sketch our approach.

After our good experience with several logical systems we assume that for description of some basic principles and properties can serve some logic with great expressive power. Using of logical system has another advantage; descriptions can be proved by corresponding deduction calculus. We choose *linear logic* [11] formulated by J.-Y. Girard. Linear logic (LL) can express dynamics of systems and handling with such resources as time and space [26]. Among useful properties of LL become the properties of linear connectives:

- multiplicative conjunction $A \otimes B$ expresses that actions A and B can be performed simultaneously;
- additive conjunction $A\&B$ expresses dependent choice, external indeterminism;
- multiplicative disjunction $A \parr B$ expresses that if A is not perform then B is perform and vice versa;
- additive disjunction $A \oplus B$ expresses free choice, internal indeterminism;
- linear implication $A \multimap B$ has special properties: a resource A is consumed after performing linear implication, and action B follows after A;
- linear negation expresses consuming of a resource A or a reaction of action A.

Every formula in (LL) can be considered as action or a resource and can be represented as type. Extending linear propositional logic with predicate symbols, linear term and quantifiers we get first order predicate linear logic (PLL) with high expressive power. PLL covers classical first order logics. Sequent calculus of linear logic enables proving of constructed formulae and to provide a strong tool for verification purposes.

Formal description in linear logic has to be modeled in some appropriate structure. We prefer as a basis symmetric monoidal categories as very useful mathematical structures providing wide spectrum of interesting properties. In [14] is constructed categorical model for object oriented program system with some indication how to extend it for complex program systems. Indeed, PLL can be extended by modal operators as we made in [27, 28].

Category theory provide two possibilities for modeling component base program systems [25, 15]. If we are interesting in component composition then it is suitable to model component interfaces are 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 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 (for example [16]), it is suitable to use coalgebras [6], where polynomial endofuctor is constructed to model behavior of a system step by step. This endofunctor is constructed over category of states.

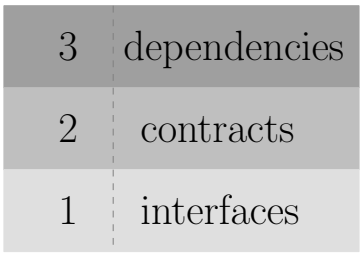

Figure 5: Proposed layers for formal framework

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

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 PLL. 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 PLL.

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 PLL 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.

5 Conclusion

Component based programming is an actual and rapidly evolving paradigm of programming. There are many useful research results in the frame of implementation, composition of components and eliminating arisen problems. We analyzed basic principles of this new method and formulated basic notions. Our analysis and discussion serves for our aim - to prepare a suitable formal framework for verifiable formal description and modeling component based program systems. Our future research will be concerned to working out in detail particular ideas presented in this paper.

6 Acknowledgments

This work has been supported by KEGA grant project No. 050TUKE-4/2012: "Application of Virtual Reality Technologies in Teaching Formal Methods", and by the Slovak Research and Development Agency under the contract No. APVV-0008-10: "Modelling, simulation and implementation of GPGPU-enabled architectures of high-throughput network security tools."

References

[1] AALST W.M.P van, MOOIJ A.J., STAHL C., WOLF K.: *Service Interaction: Patterns, Formalization, and Analysis*, In: Proc. of 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinoro, Italy, 1-6 June 2009, LNCS Vol.5569, Springer, 2009, pp.42-88.

[2] AALST W.M.P van der, HEE K.M. van, R.A. TOORN R.A van der: *Component-Based Software Architectures: A Framework Based on Inheritance of Behavior*, Science of Computer Programming, Vol. 42, No. 2-3, Elsevier, 2002, pp.129-171.

[3] ABATE P., BOENDER J., DI COSMO R., ZACCHIROLI S.: *Strong Dependencies between Software Components*, Tech.Rep.0002, 7th Framework Programme FP7-ICT-2007, University Paris Diderot, 2009

[4] AOYAMA M.: *New Age of Software Development: How Component- Based Software Engineering Changes the Way of Software Development*, International Workshop on CBSE, ICSE'98, 1998.

[5] BACHMANN F., BASS L., BUHMAN C., COMELLA-DORDA S, LONG F., ROBERT J., SEACORD R., WALLNAU K.: *Technical Concepts of Component-Based Software Engineering*, Volume II, Carnegie Mellon, Techn.Report CMU/SEI-2000-TR-008, ESC-TR-2000-007, May 2000.

[6] BARBOSA, L.: *Components as coalgebras*, PhD. Thesis, University Minho, 2001.

[7] BARNES J.: *Programming in Ada 2005*, Addison-Wesley, 2005.

[8] BENVENISTE A. et al: Multiple viewpoint contract-based specification and design, LNCS 5382, Springer-Verlag, 2007, pp.200-225.

[9] COUNCILL B., HEINEMAN G.T.: *Definition of a Software Component and Its Elements*, Ch.1, Component-based software engineering, Addison-Wesley Longman, 2001, pp.5-19.

[10] 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.

[11] GIRARD, J.Y.: *Linear logic*, Theoretical Computer Science, Vol. 50, No.1, 1987, pp. 1-102.

[12] HAN J.: *An Approach to Software Component Specification*, Melbourne, 2004.

[13] HATCLIFF J., DENG W., DWYER M.B., JUNG G., RANGANATH V.: Cadena: *An Integrated Development, Analysis, and Verification Environment for Component Based Systems*, Software Engineering, 2003, pp.160-172.

[14] JENČÍK M., Mihályi D.: *Category for component-based program system*, Electrical Engineering and Informatics, Košice, 2012, pp. 575-579.

[15] Knighten R.L.:, *Notes on Category Theory*, MIT, 2011.

[16] Harasthy, T., Turan, J., Ovsenik, L.: *Road line detection based on Optical Correlator*, In: 36th International Convention on Information & Communication Technology Electronics & Microelectronics (MIPRO), 2013, Opatija, 20-24 May 2013, pp. 298-300, IEEE 2013

[17] *Komponentenprogrammierung und Middleware-Component programming with C# and .NET*, `http://www.dcl.hpi.uni-potsdam.de/LV/Components04/VL7/04_NET-components.pdf`, April 2014.

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

[19] KWON O., YOON S., SHIN G.: *Component-Based Development Environment: An Integrated Model of Object-Oriented Techniques and Other Technologies*, In: International Workshop on Component- Based Software Engineering, 1999, pp. 2252-2256.

[20] MAIN M., SAVITCH W.: *Data Structures and Other Objects Using C++*, Addison-Wesley Longman, 2010.

[21] MARIANI L.: *A Fault Taxonomy for Component-based Software*, Electronic Notes in Theoretical Computer Science, Vol. 82, No. 6, 2003, pp. 55-65.

[22] Meyer B.: *Applying design by contract*, Computer, Vol. 25, No.10, 2002, pp.40-51

[23] Messabihi M., André P., Attiogbé C.: *Multilevel Contracts for Trusted Components*, In: The 7th International Conference on Software Engineering Advances, ICSEA, 2012, pp.71-85.

[24] NING J.Q.: *A Component Model Proposal*, International Workshop on Component- Based Software Engineering, USA, 17-18 May 1999, pp.13-17.

[25] NOVITZKÁ V., SLODIČÁK V.: *Categorical structures and their applications in informatics*, Equilibria, 2011 (in Slovak).

[26] NOVITZKÁ V., MIHÁLYI D.: *Resource-oriented programming based on linear logic*, In: Acta Polytechnica Hungarica, Vol.4, No.2, 2007, pp.157-166.

[27] MIHÁLYI D., NOVITZKÁ V.: *Towards the Knowledge in Coalgebraic model of IDS*, Computing and Informatics. Vol.33,No.1,2014, pp. 61-78.

[28] MIHÁLYI D., NOVITZKÁ V.: I*ntrusion Detection System Episteme*, Central European Journal of Computer Science. Vol.2,No.3,2012, pp. 214-220.

[29] POLBERGER D.: *Component technology in an embedded system*, Lund University, USA, 2009.

[30] RACLET J.B. et al: *A modal interface theory for component-based design*, Fundam. Inform, Vol.108, No.1-2, 2011, pp.119-149.

[31] RAYMOND K.: *Reference Model of Open Distributed Processing (RM-ODP): Introduction*, University of Queensland, Brisbane, Australia, 1996

[32] SZYPERSKI C., GRUNTZ D., MURER S.: *Component Software beyond Object- Oriented Programming*, ACM Press, New YORK, USA, 2002.

[33] URTING D., BAELEN S. van, HOLVOET T.: Yolande BERBERS, *Embedded software development: components and contracts*, Belgium, 2001.

[34] WANG A., Kai QIAN K.: *Component-oriented programming*, Wiley-Interscience, 2005.

[35] WEGNER P.: *Concepts and Paradigms of object-oriented programming*, *ACM SIGPLAN OOPS*, Vol. 1 No. 1, 1990 pp. 7-87.

[36] WILDE N., *Program Dependencies*, Carnegie Mellon, Tech.Report SEI-CM-26, 1990.

[37] WIRTH N.: Programming in Modula-2, 4th edition, Springer-Verlg, 1989.

[38] YACOUB S., AMMAR H., Ali MILI A.: *A Model for Classifying Component Interfaces*, ICSE'99 West Virginia, USA, May 1999.