ACTA UNIVERSITATIS AGRICULTURAE ET SILVICULTURAE MENDELIANAE BRUNENSIS

Volume LXI 280 Number 7, 2013
http://dx.doi.org/10.11118/actaun201361072515

APPLYING OF COMPONENT SYSTEM DEVELOPMENT IN OBJECT METHODOLOGY

Milan Mišovič, Oldřich Faldík

Received: May 3, 2013

Abstract

MIŠOVIČ MILAN, FALDÍK OLDŘICH: *Applying of component system development in object methodology.* Acta Universitatis Agriculturae et Silviculturae Mendelianae Brunensis, 2013, LXI, No. 7, pp. 2515–2522

In the last three decades, the concept and implementation of component-based architectures have been promoted in software systems creation. Increasingly complex demands are placed on the software component systems, in particular relating to the dynamic properties. The emergence of such requirements has been gradually enforced by the practice of development and implementation of these systems, especially for information systems software.

Just the information systems (robust IS) of different types require that target software meets their requirements. Among other things, we mean primarily the adaptive processes of different domains, high distributives due to the possibilities of the Internet 2.0, acceptance of high integrity of life domains (process, data and communications integrity), scalability, and flexible adaptation to process changes, a good context for external devices and transparent structure of the sub-process modules and architectural units.

Of course, the target software of required qualities and the type robust cannot be a monolith. As commonly known, development of design toward information systems software has clearly come to the need for the software composition of completely autonomous, but cooperating architectural units that communicate with each other using messages of prescribed formats.

Although for such units there were often used the so called subsystems and modules, see (Jac, Boo, Rumbo, 1998) and (Arlo, Neus, 2007), their abstraction being gradually enacted as the term component. In other words, the subsystems and modules are specific types of components.

In (Král, Žeml, 2000) and (Král, Žeml, 2003) there are considered two types of target software of information systems. The first type – there are SWC (Software Components), composed of permanently available components, which are thought as services – Confederate software. The second type – SWA (Software Alliance), called semi Confederate, formed during the run-time of the software system and referred to as software alliance.

In both of these mentioned publications there is delivered deep philosophy of relevant issues relating to SWC / SWA as creating copies of components (cloning), the establishment and destruction of components at software run-time (dynamic reconfiguration), cooperation of autonomous components, programmable management of components interface in depending on internal components functionality and customer requirements (functionality, security, versioning).

Nevertheless, even today we can meet numerous cases of SWC / SWA existence, with a highly developed architecture that is accepting vast majority of these requests. On the other hand, in the development practice of component-based systems with a dynamic architecture (i.e. architecture with dynamic reconfiguration), and finally with a mobile architecture (i.e. architecture with dynamic component mobility) confirms the inadequacy of the design methods contained in UML 2.0. It proves especially the dissertation thesis (Rych, Weis, 2008).

Software Engineering currently has two different approaches to systems SWC / SWA. The first approach is known as component-oriented software development CBD (Component based Development). According to (Szyper, 2002) that is a collection of CBD methodologies that are heavily focused on the setting up and software components re-usability within the architecture. Although CBD does

not show high theoretical approach, nevertheless, it is classified under the general evolution of SDP (Software Development Process), see (Sommer, 2010) as one of its two dominant directions.
From a structural point of view, a software system consists of self-contained, interoperable architectural units – components based on well-defined interfaces. Classical procedural object-oriented methodologies significantly do not use the component meta-models, based on which the target component systems are formed, then. Component meta-models describe the syntax, semantics of components. They are a system of rules for components, connectors and configuration. Component meta-models for dynamic and mobile architectures also describe the concept of rules for configuration changes (rules for reconfiguration). As well-known meta-models are now considered: *Wright* for static architecture, *SOFA* and *Darvin* for dynamic architecture and *SOFA 2.0* for mobile architecture, see (Rych, Weis, 2008).
The CBD approach verbally defines the basic terms as component (primitive / composite), interface, component system, configuration, reconfiguration, logical (structural) view, process view (behavioral), static component architecture, dynamic architecture, mobile architecture (fully dynamic architecture), see (IEEE Report, 2000) and (Crnk, Chaud, 2006).
The CBD approach also presents several ADL languages (Architecture Description Languages) which are able to describe software architecture. The known languages are integration ACME and UML (Unified Modeling Language), see (Garl, Mon, Wil, 2000) and (UNIFEM, 2005).
The second approach to SWC / SWA systems is formed on SOA, but this article does not deal with it consistently.
SOA is a philosophy of architecture. SOA is not a methodology for the comprehensive development of the target software. Nevertheless, SOA successfully filled the role of software design philosophy and on the other hand, also gave an important concept linking software components and their architectural units – business services. SOA understands any software as a Component System of a business service and solved life components in it. The physical implementation of components is given by a Web services platform. A certain lack of SOA is its weak link to the business processes that are a universally recognized platform for business activities and the source for the creation of enterprise services.
This paper deals with a specific activity in the CBD, i.e. the integration of the concept of component-based system into an advanced procedural, object-oriented methodology (Arlo, Neust, 2007), (Kan, Müller, 2005), (Krutch, 2003) for problem domains with double-layer process logic. There is indicated an integration method, based on a certain meta-model (Applying of the Component system Development in object Methodology) and leading to the component system formation. The mentioned meta-model is divided into partial workflows that are located in different stages of a classic object process-based methodology. Into account there are taken the consistency of the input and output artifacts in working practices of the meta-model and mentioned object methodology. This paper focuses on static component systems that are starting to explore dynamic and mobile component systems.
In addition, in the contribution the component system is understood as a specific system, for its system properties and basic terms notation being used a set and graph and system algebra.

component, component system, interface, static, dynamic and mobile architecture, component meta-model, object methodology

METHODS AND RESULTS

1 Generally about Component systems

Significant attention to the component system after its installation and its run-time has resulted in the recognition of three types of component software systems: the static, dynamic and mobile one.

The Static Component System is characterized by restrictions: after installing, the system excludes the possibility of creating any new interconnection between components, whereas existing connections are immutable and irrevocable.

Unlike the static architecture, dynamic architecture allows the creation and cancelling components and their connections in the run-time of the system.In addition, for a dynamic architecture there are certain rules for the evolutionary development of the component system during its run-time, made even during its system development. Under these rules components and connections between components are being formed and interfered during the system run-time.

Nevertheless, the mobile architecture is the dynamic architecture, but in addition it is characterized by the mobility of components. Components can change their context in the system logical structure at the run-time of their implementation. This is again in accordance with rules established at the system development time and in accordance with the requirements of components functionality changes.

Development and implementation requirements of software component systems have focused

considerably just on dynamic quality of these systems and components mobility in the last decade. It is a focus on the possibility how to secure a wide behavior modification of components and their interconnections in the component system during its run-time. Therefore, specialists of informatics effort to find appropriate solutions for the following problems mentioned in the list, both in practical and in theoretical – formal areas. Naturally, as for this issue there can exist not only more different approaches, but even more achievements, passing through both areas, practical and theoretical, see (Rych, Weis, 2008).

Problems of dynamic component-based systems:

1. *The concept of dynamics of the component-based software system, its system attributes and their specificities.*
2. *Find the method of dynamic creation, behavior modification and the possibility of removing components from the system.*
3. *Suggest a notational means for the purpose of formulating dynamic requirements for the software component system.*
4. *Suggest an apparatus for modeling of the software component system dynamics that accepts notation of dynamic requirements.*
5. *Find the method of dynamic contexts modification of individual components (the component mobility), i.e. the establishment and removing of interconnections of components at the run-time of their implementation.*

Thus, as we have said, we are going to investigate the issue of component-based systems with static component systems. We introduce our multiset formulation of relevant terms related to component-based systems. This will not only refine the used terms but we will try to use the formalization for new pieces of knowledge. Many of the definitions of static component systems are given in (Crn, Mag, Lar, 2002) describing the static component model WRIGHT. The basic elements of the component system are components themselves, component connections (links), interfaces of all system components and auxiliary connectors. As we move out of the plane of the implementation, thus auxiliary connectors we will not be further considered.

The basis of the component system is a component. A very useful definition of component implementation is shown in UML 2.0.

A component in the Unified Modeling Language *"represents a* modular *part of a system that encapsulates its content and the manifestation of which is replaceable within its* environment. *A component defines its* behavior *in terms of provided and required* interfaces*".*

As such, a component serves as a type, the conformance of which is defined by these provided and required interfaces. One component may be replaced by a second one only when both are type conformant.

By other words: A component may be replaced by another if and only if their provided and required interfaces are identical. This idea is the underpinning for the plug-and-play *capability of* component-based *systems and promotes* software reuse.

The following definition is somewhat different from that introduced implementation definition because it is oriented for the sake of a theoretical approach to these concepts and is very useful. It is influenced by effort rather to go away from the object implementing.

Definition 1

Component **C** is called an ordered threesome **F**, $\mathbf{I}_{in}$, $\mathbf{I}_{out}$, where **F** is a procedural component basis (process behavior), $\mathbf{I}_{in}$ is a finite set of all required interfaces and $\mathbf{I}_{out}$ is a finite set of interfaces being offered. Component **C** is written in the form **C = (F,** $\mathbf{I}_{in}$, $\mathbf{I}_{out}$**)**.

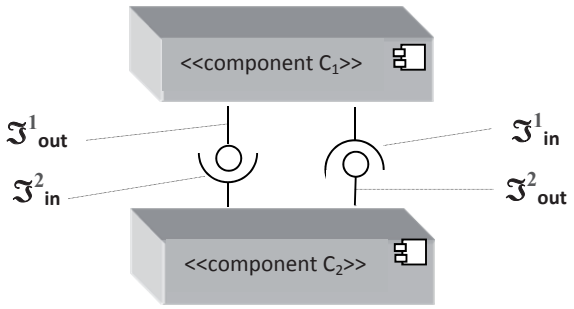
In accordance with the above mentioned idea, later we formally define the basic concepts, such as component system, interconnection, conformance (weak / strong).

Definition 2

Interconnection of two components $\mathbf{C}_1$ a $\mathbf{C}_2$ in given order, is an element of binding relation $\mathfrak{R}$ that is written in graphical form $\mathfrak{I}^1_{out}$ —○– $\mathfrak{I}^2_{in}$ or in form $\mathfrak{I}^2_{out}$ —○– $\mathfrak{I}^1_{in}$.

We can use more abstract forms ($\mathfrak{I}^1_{out}$ $\mathfrak{R}$ $\mathfrak{I}^2_{in}$), ($\mathfrak{I}^2_{out}$ $\mathfrak{R}$ $\mathfrak{I}^1_{in}$), or ($\mathfrak{I}^1_{out}$, $\mathfrak{I}^2_{in}$), ($\mathfrak{I}^2_{out}$, $\mathfrak{I}^1_{in}$).

The binding properties of relation $\mathfrak{R}$ depend on its implementation definition. Sometimes there are especially valued its properties of reflexivity, symmetry and transitivity, which can be very beneficial in the implementation practice.



1: *Illustration of* C_1 *and* C_2 *interconnection*

Definition 3

Two components **C1**, **C2** are weakly type conformal, according to binding relation $\mathfrak{R}$ if they have at least one interface of the same type $\mathfrak{R}$ and consistent process behavior.

It requires equality $\mathfrak{I}^1_{in} = \mathfrak{I}^2_{in}$ or equality $\mathfrak{I}^1_{out} = \mathfrak{I}^2_{out}$ and equality of their process behavior $\mathbf{F}_1 \approx \mathbf{F}_2$, where $\mathfrak{I}^1_{in} \in \mathbf{I}^1_{in}$, $\mathfrak{I}^2_{in} \in \mathbf{I}^2_{in}$, $\mathfrak{I}^1_{out} \in \mathbf{I}^1_{out}$, $\mathfrak{I}^2_{out} \in \mathbf{I}^2_{out}$. By other words, components $\mathbf{C}_1$, $\mathbf{C}_2$ are weakly type conformal, if they have the same at least one interface and consistent process behavior.

The weak type conformity does not suffice for components replacement without life disruption of the component system.

Definition 4

Two components **C1**, **C2** are of strong type conformance if there exists the equality between all their offered interfaces and all their required interfaces and even consistent procedural behavior.

This indicates a very strong demand for their sets of offered and required interfaces, i.e. $\mathbf{I^1}_{out} = \mathbf{I^2}_{out}$ and $\mathbf{I^1}_{in} = \mathbf{I^2}_{in}$. The strong type conformance guarantees components replacement without disruption of the component system.

Definition 5

The arranged three $\mathbf{C}_{com}$, **L**, $\mathbf{I}_S$ is called a component system $\mathbf{C}_S$, where

$\mathbf{C}_{com}$ is the set of all components of the system, **L** is the set of all interconnections between system components and $\mathbf{I}_S$ is the set of all component interfaces of the component system. Any component system is written in the form $\mathbf{C}_S = (\mathbf{C}_{com}, \mathbf{L}, \mathbf{I}_S)$.

The set $\mathbf{I}_S$ will be defined as an ordered pair $\mathbf{I}_S = (\boldsymbol{U}^m_{i=1}\boldsymbol{I}^i_{out}, \boldsymbol{U}^n_{i=1}\boldsymbol{I}^i_{out})$

Very often, the set **L**, given by Cartesian notation $\mathbf{L} \subseteq (\boldsymbol{U}^m_{i=1}\boldsymbol{I}^i_{out} \times \boldsymbol{U}^n_{i=1}\boldsymbol{I}^i_{out})$, is called the configuration of the component system. Any configuration change is labeled as reconfiguration. Among other things, in the component system there are just as many interconnections as there is the number of dimensionality of **L**. Then $|\boldsymbol{L}| \leq \boldsymbol{m} \times \boldsymbol{n}$, where $\boldsymbol{m} = |\boldsymbol{U}^m_{i=1}\boldsymbol{I}^i_{out}|$, $\boldsymbol{n} = |\boldsymbol{U}^n_{i=1}\boldsymbol{I}^i_{out}|$. Naturally, the dimension of **L** depends on the implementationally defined binding relation $\mathfrak{R}$. This means that not all the interfaces must be bound by this relation. On the other hand, more types of binding relations $\mathfrak{R}$ can be introduced in the component system.

1.1 Implementation view of the Components interface

Interfaces between components can be of different nature. The most commonly used is the nature of the data and activity. Frequently, we can meet the application of heterogeneous composition and activity of the data elements. Investigation if the interfaces $\mathfrak{I}^1_{in}$ and $\mathfrak{I}^2_{out}$ are interconnected is possible only on the basis of serially comparison of their corresponding elements. Naturally, for this comparison the implementational character of elements (data types with the same scope, activities defined in the components) is used. Rough equality of all corresponding interface elements of $\mathfrak{I}^1_{in}$ and $\mathfrak{I}^2_{out}$ or $\mathfrak{I}^1_{out}$ and $\mathfrak{I}^2_{in}$ two components C_1 and C_2, ensures the strong type of their conformance.

Be $\mathfrak{I}^1_{in} = (\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n)$, $\mathfrak{I}^2_{out} = (\mathbf{e'}_1, \mathbf{e'}_2, \ldots, \mathbf{e'}_n)$, where e_i and e'_i are notation of their elements. The review of connectivity of interfaces can be converted to the test of Boolean logical expression $\mathbf{Q} = (\mathbf{e}_1 \approx \mathbf{e'}_1) \wedge (\mathbf{e}_2 \approx \mathbf{e'}_2) \wedge \ldots \wedge (\mathbf{e}_n \approx \mathbf{e'}_n)$, $\mathbf{m} \geq \mathbf{n}$. A similar notation can also be used for the interfaces $\mathfrak{I}^1_{out}$ and $\mathfrak{I}^2_{in}$.

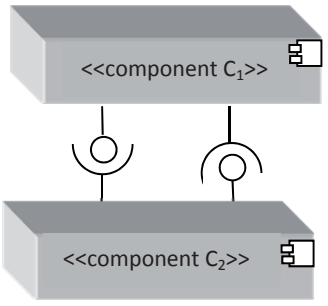
1.2 System tasks over Component systems

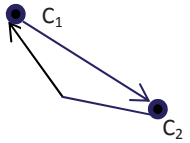
In the context of static systems we meet with many system tasks. We will notice only two of them, the task on the structure and the task on the integration of component system development into the framework of the advanced object-oriented methodology.

The task of the structure (Structure of Component system) is extremely important for us to be able to investigate the internal behavior of components in the component system. The task of the structure converts the component system $\mathbf{C}_S = (\mathbf{C}_{com}, \mathbf{L}, \mathbf{I}_S)$ in a graph form. The obtained structural graph $\mathbf{G} = (\mathbf{U}, \mathbf{H}, \mathbf{f})$, $\mathbf{f}: \mathbf{H} \rightarrow \mathbf{U}^2$, which is given as the incidence of nodes and edges, is defined according to the following transformation prescription:

1. Each component $\mathbf{C} \in \mathbf{C}_S$ becomes a node (isolated / non-isolated) of a graph **G**.
2. Be component **C1** already a node of the graph **G**. Connection $(\mathfrak{I}^1_{out}, \mathfrak{I}^2_{in})$ of component **C1** to another component **C2** gives rise to oriented edge **(C1, C2)**.
3. Reverse connection $(\mathfrak{I}^2_{out}, \mathfrak{I}^1_{in})$, component **C2** to **C1** gives rise to an oriented edge **(C2, C1)**. Multiplicity of interconnections between components leads to the multigraph **G**.

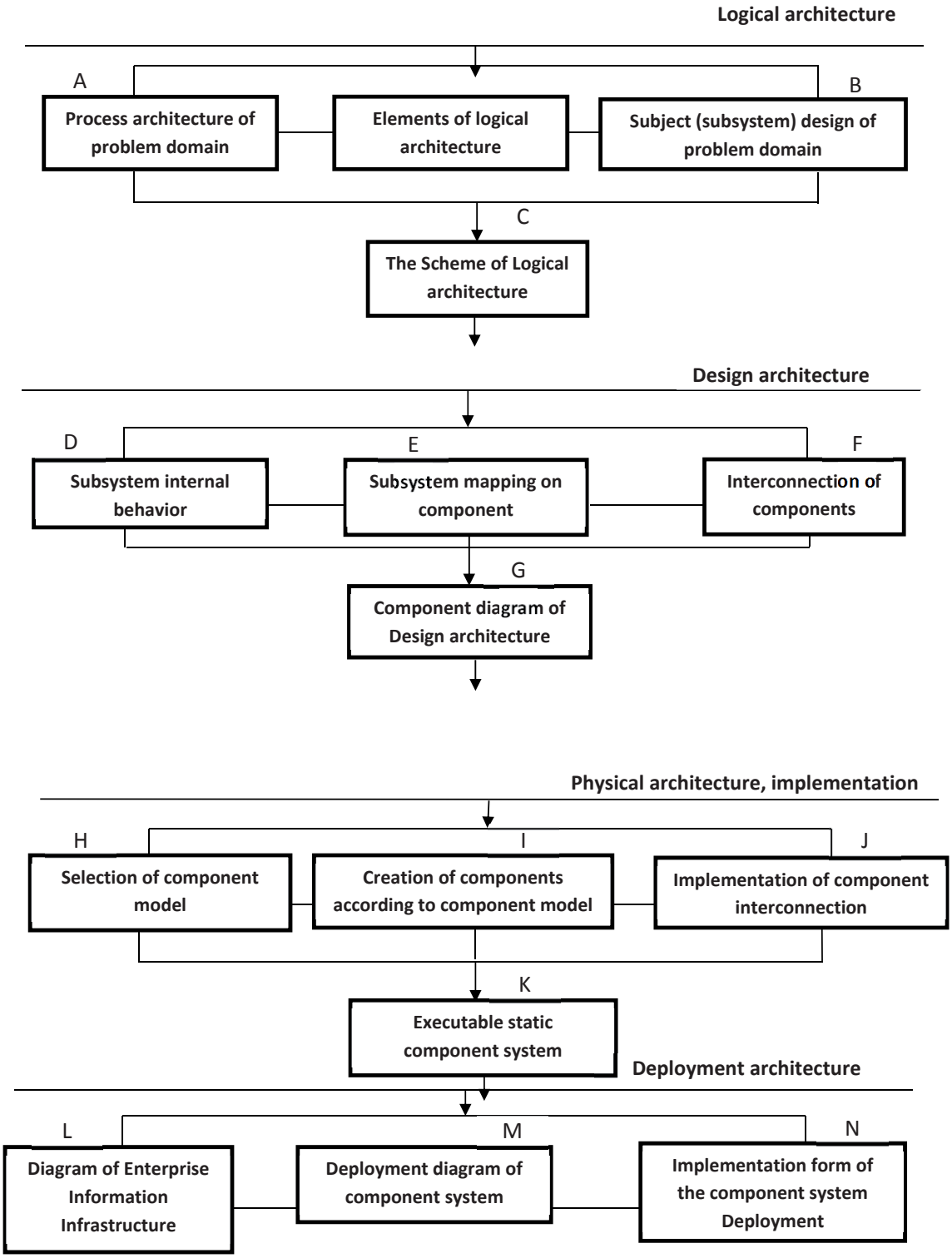
It is understandable that the importance of G increases with an increasing number of components and interconnections between them. Creating a graph G can be conferred to a very capable software application. The result is not only a graph **G** but also a set of graph paths that represent a chain





2: *Illustration of subsequent construction of G*

of interconnected components. In addition, the application can provide a lot of useful information (cycles in interconnection, redundant interface, etc.). Thus we have formed a general idea about the life of the component in the component system. Based on these strings we can prepare functional sequence diagrams for programmers and design a management component for all the component system.



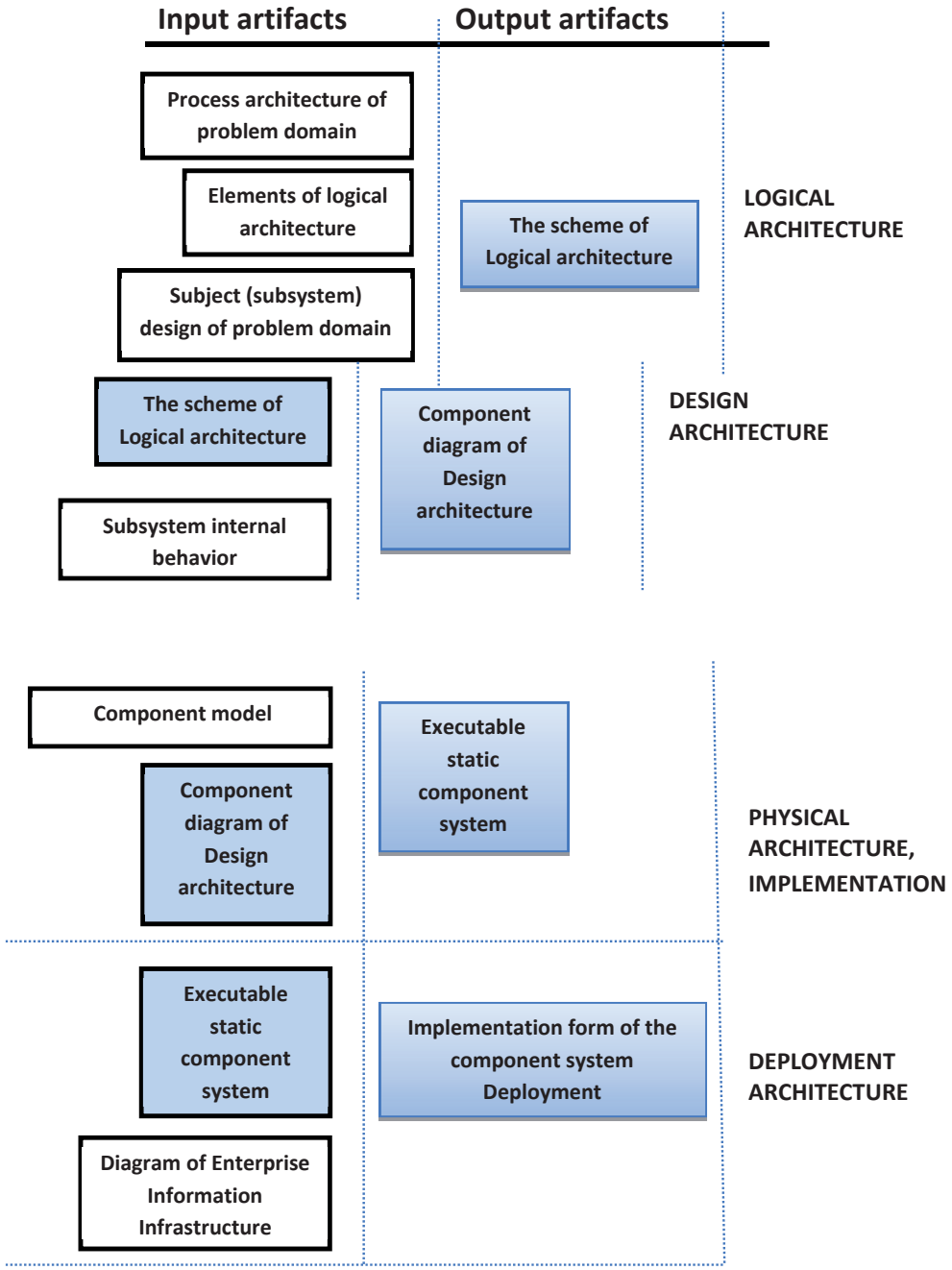
3: *Four workflow parts of meta-model*

2 Development of Component systems

The basis of our considered integration towards component-based system development into object methodology is not only a specific meta-model design but its implementation in practice. Its integration into the advanced object methodology lies in the proper nesting of its working practices in the parent workflow methodology used. This integration is then based on the consistency of the input and output meta-model artifacts and the input and output artifacts of the object methodology.

The proposed meta-model, see Figure 3, consists of four parts – workflows: Logical Architecture, Design Architecture (conceptual architecture), Physical Architecture (implementation architecture), and Deployment Architecture. It is true that meta-model applying for the selected problem domain usually causes implementation differences but the basic features of the meta-model are maintained. Furthermore, there is maintained using of various parts of the meta-model in specified stages of the advanced object-oriented methodology.

The aim of the Logical architecture is to make use of process and data views and look at the domain subjects to transfer the relevant characteristics of the problem domain into ***a Scheme of logical***



4: *Input and output artifacts of meta-model workflow*

architecture. This scheme is complemented by a number of general elements (computer icons data base, control, etc.), so the logic of the target software becomes even more marked. The above mentioned scheme is the primary view of the target software architecture. It is very useful for the programmers, although it makes only a logical knowledge.

Design Architecture **workflow** participates in workflows made by the object methodology: Requirements, Analysis of analytical and design classes and then uses the already defined contents, i.e. the internal behavior of all subsystems (step D). Based on the range of content, step E consists of subsystems mapping to components and step F finds component interconnected scheme by ***the Scheme of logical architecture***. The result is a system diagram ***Component diagram of design architecture***.

The ***Physical architecture*** of the workflow, that is the component model selection (step H) for the implementation (programming, step I) components of the system. It performs the interconnection between components, completes programming and creates static components and executable target software system – ***Executable static component system***.

Deployment workflow task provides implementation of the component system based on ***Deployment diagram*** and designs implementation component on the enterprise Information Infrastructure as an ***Implementation form of the component system Deployment***.

The following Figure 4 illustrates the views of the meta-model workflow on the basis of the inputs and outputs artifacts.

RESULTS AND DISCUSSION

Currently there are several approaches to the partial problem of component systems. The first result of this paper is using a systematic approach to component systems and on this basis explaining the meaning of their system properties. Then there is performed a formalization of relevant concepts related to static component systems, such as: *component*, *interface*, *connection of components*, *component systems*, *configuration*, *reconfiguration*, *the orchestration of components* on the set theory, graph platform and using a system algebra. Of course, it is questionable if this approach is comprehensive enough and can be used for notation of the new knowledge of such systems, especially dynamic component systems and systems with components mobility.

The use of the set platform and system view provides precise definitions of known concepts and procedures how to implement the two most important tasks over component systems (structural task and development of component-based system integration into advanced object methodologies).

SUMMARY AND RECOMMENDATION

This paper deals with static component systems and presents a system view. On its basis the familiar terms are defined, up till now defined verbally. The set platform and presented meta-model have provided the possibility to resolve the structural task and development of component-based system integration within advanced object methodologies.

Based on the proposed meta-model, a modified methodology with integrated meta-model could be developed and we can recommend using this methodology especially in teaching of the subject Software Engineering, and for creating software in small software companies.

REFERENCES

ARLOW, J., NEUSTAD, I., 2007: *UML 2 a unifikovaný proces vývoje aplikací*. Brno: Computer Press. 566 p. ISBN 978-80-251-1503-9. [cit. 17. 02. 2013].

BELL, M., 2008: *Service-oriented Modeling (SOA) Service Analysis, Design, and Architecture*. USA: John Wiley & Sons, 452. ISBN 978.0-470-14111-3.

CRNKOVIC, I., CHAUDRON, M., LARSSON, S., 2006: Component-based development process and component lifecycle. In: *International Conference on Software Engineering Advances, ICSEA'06*. Tahiti, French Polynesia: Tahiti University, 12 p. IEEE.

CRNKOVIC, I., LARSON, M. P. H., 2002: *Building Reliable Component-based Software Systems*. Norwood: Artech House, Computing library, 589 p. ISBN 1580535585, 9781580535588. [cit. 17. 04. 2013].

ČERNÁ, I.,VAŘEKOVÁ, P., ZIMMEROVÁ, B., 2006: *Component interaction – automata modeling Language. Technical Report FIMU-RS-2006-08*. Brno: Masaryk University Brno.

GARLAN, D., MONROE, R.T., WILE, D., 2000: *Architectural description of component-based systems*. Foundations of Component – Based Systems chapter 3, pages 47–68. New York: Cambridge University Press, 22 p.

JACOBSON, I., BOOCH, G., RUMBAUGH, J., 1998: *The Unified Software Development Process*. Amsterdam: Addison Wesley Longman, Inc. 463 p. ISBN 0-201-57169-2.

KANISOVÁ, H., MULLER, M., 2007: *UML srozumitelně*. Brno: Computer Press. 176 p. ISBN 80-251-1083-4.

KRÁL, J., ŽEMLIČKA, M., 2000: Autonomous components. In: *SOFSEM 2000. Theory And Practice of Informatics, volume 1963 of Lecture Notes in Computer Science*: 375–383. [cit. 19.04.2013].

KRÁL, J., ŽEMLIČKA, M., 2003: Software confederations and alliances. CEUR Workshop. In: *CAiSE Short Paper Proceedings*, 74: 229–232. [cit. 19. 03. 2013].

KRUCHTEN, P., 2003: *The Rational Unified Process, An introduction*. USA: Pearson Education. 310 p. ISBN 0-321-19770-4.

Recommended practice for architectural description of software intensive systems, 2000: Technical Report IEEE P1471–2000, The Architecture Working Group of the Software Engineering Committee, Standards Department, IEEE, Piscataway, New Jersey, USA. [cit. 18. 04. 2013].

RYCHLÝ, M., WEISS, P., 2008: Modeling of Service-oriented Architecture: From Business process to Service realization. *Third International conference on Evaluation of Novel Approaches to Software Engineering Proceedings*. Poland: Institute for Systems and Technologies of information, Control and Communication. ISBN 978-989-8111-28-9. [cit. 17. 02. 2013].

SOMMERVILLE, I., 2010: *Software Engineering,* 9th ed. London: Publisher PEARSON. 773 p. ISBN 10: 0-13-705346-0.

SZYPERSKI, C., 2002: *Component Software: Beyond Object-Oriented Programming*. 2nd edition. New York: Addison Wesley Professional, 589 p. ISBN 0-321-75302-X.

Unified modeling language specification, version 1.4.2. Document format l-05-04-01, 2005: The Object Management Group. [cit. 19. 03. 2013].

Address

prof. RNDr. Milan Mišovič, CSc., Bc. Oldřich Faldík, Department of Informatics, Mendel University in Brno, Zemědělská 1, 613 00 Brno, Czech Republic, e-mail: misovic@mendelu.cz, xfaldik@node.mendelu.cz