Dependable Computer Systems (Advances in Intelligent and Soft Computing 97), pp.301–311.
6-th Int. Conf. on Dependability of Complex Systems, Brunow Palace, Poland, June 27 – July 1, 2011.

DOI 10.1007/978-3-642-21393-9_23.

Incremental Composition of Software Components

W.M. Zuberek

Department of Computer Science, Memorial University,
St.John's, NL, Canada A1B 3X5

and

Department of Applied Informatics, University of Life Sciences,
02-787 Warszawa, Poland

email: wlodek@mun.ca

Abstract In component-based systems, two interacting components are compatible if all sequences of services requested by one components can be provided by the other component. In the case of several components interacting with a single provider, as is typically the case in client–server computing, the requests from different components can be interleaved and therefore verifying component compatibility must check all possible interleavings of requests from all interacting components. Incremental composition of interacting components eliminates this need for exhaustive combinatorial checking of the interleavings by imposing some restrictions on the interleavings. The paper introduces simple conditions which must be satified by the interacting components for their composition to be incremental and illustrates the concepts using simple examples of interactions.

1 Introduction

Component-base software engineering is one of promising approaches to the development of large-scale software systems [2]. The success of this approach relies, however, on the automated and easily verifiable composition of components and their services [14]. While manual and ad hoc strategies toward component integration have met with some success in the past, such techniques do not lend themselves well to automation. A more formal approach toward the assessment of component compatibility and interoperability is needed. Such a formal approach would permit an automated assessment and would also help promote the reuse of existing software components. It would also significantly enhance the assessment of component substitutability when an existing component is replaced by an improved one, and the replacement is not supposed to affect the functionality of the remaining parts of the system [4].

Components can be considered as the basic functional units and the fundamental data types in architectural design [27]. Components represent high-

level software models; they must be generic enough to work in a variety of contexts and in cooperation with other components, but they also must be specific enough to provide easy reuse.

Primary reasons for component production and deployment are [14]: separability of components from their contexts, independent component development, testing and later reuse, upgrade and replacement in running systems. Component compositionality is often taken for granted. Compositionality, however, is influenced by a number of factors. Component technologies are not entirely independent of particular hardware and operating platforms, programming languages or the specific middleware technology in which they are based. Ideally, the development, quality control, and deployment of software components should be automated similarly to other engineering domains, which deal with the construction of large systems composed of well-understood elements with predictable properties and under acceptable budget and timing constraints [24]. For software engineering, such a situation does not seem to belong to the forseeable future yet.

In this work, two interacting components are considered compatible if any sequence of services requested by one component can be provided by the other. This concept of compatibility can be easily extended to a set of interacting components, then, however, the requests from different components can be interleaved, so any verification of the behavior of the composed system must check all interleavings which can be created by the interacting components. These interleavings can be controlled by specific frameworks for component compositions which can vary from simple syntactic expansions of the GenVoca model [3], to models resembling higher-order programming [5] and dynamic interconnections of distributed processes [22]. Despite significant research efforts, a comprehensive model of software composition is still to come [17].

Incremental composition of interacting components introduces a restriction on the form of interleaving of requests coming from several components, and eliminates the need for exhausting combinatorial checking of the behavior of the composed system. In incremental composition, if a requesting and providing components are compatible, the requesting component can be added to other interacting components without any adverse effect on the behavior of the system. On the other hand, the performance of a system composed in such a way may not be fully used.

Several formal models of component behavior have been proposed in the literature. They include finite automata [9] [10] [28], predicates [29], process calculi [8] [26] and especially labeled Petri nets [1] [12] [16] [19]. Some approaches are built on the concept of subtyping derived from object–oriented programming. They use the interface type to define a subtyping relation between components [7] [21]. Various forms of those types exist, starting with the classical interface type [11] and adding behavioral descriptions such as automata [9]. Related research shows that the resulting approach may be too restrictive for practical applications [29].

Petri nets [23] [25] are formal models of systems which exhibit concurrent activities with constraints on frequency or orderings of these activities. In labeled Petri nets, labels, which represent services, are associated with el-

ements of nets in order to identify interacting components. Well-developed mathematical theory of Petri nets provides a convenient formal foundation for analysis of systems modeled by Petri nets.

This chapter is a continuation of previous work on component compatibility and substitutability [12] [13] [30]. Using the same linguistic specification of component behavior as before [30], the paper introduces icremental component composition and shows that for such a composition, properties of the composed systems can be verified without the exhaustive checking of all possible interleavings of component languages. Simple criteria for incremental composition of components are also given and illustrated by a few examples.

Section 2 recalls the concept of component languages as a characterization of component's behavior. Component languages are used in Section 3 to define component compatibility. Incremental composition is described in Section 4 while Section 5 concludes the chapter.

2 Petri net models od component behavior

The behavior of a component, at its interface, can be represented by a cyclic labeled Petri net [12] [30]:

$$\mathcal{M}_i \ = \ (P_i, T_i, A_i, S_i, m_i, \ell_i, F_i),$$

where P_i and T_i are disjoint sets of places and transitions, respectively, A_i is the set of directed arcs, $A_i \subseteq P_i \times T_i \cup T_i \times P_i$, S_i is an alphabet representing the set of services that are associated with transitions by the labeling function $\ell_i : T_i \to S_i \cup \{\varepsilon\}$ (ε is the "empty" service; it labels transitions which do not represent services), m_i is the initial marking function $m_i : P_i \to \{0, 1, ...\}$, and F_i is the set of final markings (which are used to capture the cyclic nature of sequences of firings).

Sometimes it is convenient to separate net structure $\mathcal{N} = (P, T, A)$ from the initial marking function m.

In order to represent component interactions, the interfaces are divided into *provider* interfaces (or p-interfaces) and *requester* interfaces (or r-interfaces). In the context of a provider interface, a labeled transition can be thought of as a service provided by that component; in the context of a requester interface, a labeled transition is a request for a corresponding service. For example, the label can represent a conventional procedure or method invocation. It is assumed that if the p-interface requires parameters from the r-interface, then the appropriate number and types of parameters are delivered by the r-interface. Similarly, it is assumed that the p-interface provides an appropriate return value, if such a value is required. The equality of symbols representing component services (provided and requested) implies that all such requirements are satisfied.

For unambiguous interactions of requester and provider interfaces, it is required that in each p-interface there is exactly one labeled transition for each provided service:

$$\forall t_i, t_j \in T : \ell(t_i) = \ell(t_j) \neq \varepsilon \Rightarrow t_i = t_j.$$

Moreover, to express the reactive nature of provider components, all provider models are required to be ε–conflict–free, *i.e.*:

$$\forall t \in T \ \forall p \in Inp(t) : Out(p) \neq \{t\} \Rightarrow \ell(t) \neq \varepsilon$$

where $Out(p) = \{t \in T \mid (p,t) \in A\}$; the condition for ε–conflict–freeness could be used in a more relaxed form but this is not discussed here for simplicity of presentation.

Component behavior is determined by the set of all possible sequences of services (required or provided by a component) at a particular interface. Such a set of sequences is called the *interface language*.

Let $\mathcal{F}(\mathcal{M})$ denote the set of firing sequences in $\mathcal{M}$ such that the marking created by each firing sequence belongs to the set of final markings F of $\mathcal{M}$. The interface language $\mathcal{L}(\mathcal{M})$, of a component represented by a labeled Petri net $\mathcal{M}$, is the set of all labeled firing sequences of $\mathcal{M}$:

$$\mathcal{L}(\mathcal{M}) \ = \ \{\ell(\sigma) \mid \sigma \in \mathcal{F}(\mathcal{M})\},$$

where $\ell(t_{i_1} t_{i_2} ... t_{i_k}) = \ell(t_{i_1})\ell(t_{i_2})...\ell(t_{i_k})$.

By using the concept of final markings, interface languages reflect the cyclic behavior of (requester as well as provider) components.

Interface languages defined by Petri nets include regular languages, some context–free and even context–sensitive languages [18]. Therefore, they are significantly more general than languages defined by finite automata [10], but their compatibility verification is also more difficult than in the case of regular languages.

3 Component compatibility

Interface languages of interacting components can be used to define the compatibility of components; a requester component $\mathcal{M}_r$ is compatible with a provider component $\mathcal{M}_p$ if and only if all sequences of services requested by $\mathcal{M}_r$ can be provided by $\mathcal{M}_p$, i.e., if and only if:

$$\mathcal{L}(\mathcal{M}_r) \subseteq \mathcal{L}(\mathcal{M}_p).$$

Checking the inclusion relation between the requester and provider languages defined by Petri nets $\mathcal{M}_r$ and $\mathcal{M}_p$ can be performed by systematic checking if the services requested by one of the interacting nets can be provided by the other net at each stage of the interaction. In the case of bounded

nets, the checking procedure performs a breadth–first traversal of the reachability graph $\mathcal{G}(\mathcal{M}_r)$ verifying that for each transition in $\mathcal{G}(\mathcal{M}_r)$ there is a corresponding transition in $\mathcal{G}(\mathcal{M}_p)$.

3.1 Bounded models

The following logical function *CheckBounded* can be used for compatibility checking if the requester and provider languages are defined by bounded marked Petri nets $(\mathcal{N}_r, m_r)$ and $(\mathcal{N}_p, m_p)$, respectively. The function performs exhaustive analysis of the marking spaces of its two argument marked nets checking, at each step, if all service that can be requested by the first argument net are available in the second net. In the pseudocode below, New is a sequence (a queue) of pairs of markings to be checked, *head* and *tail* are operations on sequences that return the first element and remaining part of the sequence, respectively, $append(s, a)$ appends an element a to a sequence s, *Analyzed* is the set of markings that have been analyzed, $Enabled(\mathcal{N}, m)$ returns the set of labels of transitions enabled in the net $\mathcal{N}$ by the marking m (including ε if the enabled transitions include transitions without labels), and $next(\mathcal{N}, m, a)$ returns the marking obtained in the net $\mathcal{N}$ from the marking m by firing the transition labeled by x):

```
proc CheckBounded(N_r, m_r, N_p, m_p);
begin
    New := (m_r, m_p);
    Analyzed := {};
    while New ≠ {} do
        (m, n) := head(New);
        New := tail(New);
        if m ∉ Analyzed then
            Analyzed := Analyzed ∪ {m};
            Symbols1 := Enabled(N_r, SkipEps(N_r, m));
            Symbols2 := Enabled(N_p, SkipEps(N_p, n));
            if Symbols1 ∩ Symbols2 = {} then return FALSE fi;
            for each x in Symbols1 do
                if x ∈ Symbols2 then
                    append(New, (next(N_r, m, x), next(N_p, n, x))
                fi
            od
        fi
    od;
    return TRUE
end;
```

The function *SkipEps(m)* advances the marking function m through all transitions labeled by ε:

proc $SkipEps(\mathcal{N}, m)$;
begin
while $\varepsilon \in Enabled(\mathcal{N}, m)$ **do** $m := next(\mathcal{N}, m, \varepsilon)$ **od**;
return m
end;

where the ε parameter of the function *next* refers to any transition enabled by m that is labeled by ε.

The function *CheckBounded* returns TRUE if the language of $(\mathcal{N}_r, m_r)$ is a subset of the language defined by $(\mathcal{N}_p, m_p)$; otherwise FALSE is returned.

Example. Fig.1 shows a simple configuration of two (cyclic) requester components and a single provider of two services named `a` and `b`. In both requester components, the requested services are separated by some "local" operations.

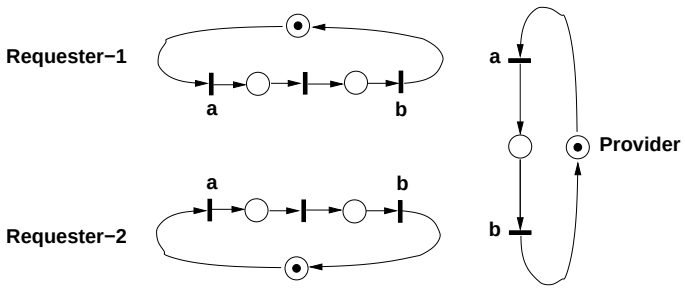

Fig.1. Two requesters and a single provider.

In this case, the languages of all components are the same, and are sequences of service `a` followed by service `b`. They can be described by a regular exapression `(ab)*`.

For the Requester-1 and Provider nets shown in Fig.1, the steps performed by the function *CheckBounded* can be illustrated in the following table:

m	n	$Symbols1$	$Symbols2$	x	$next(\mathcal{N}_r, m, x)$	$next(\mathcal{N}_p, n, x)$
(1,0,0)	(1,0)	$\{a\}$	$\{a\}$	a	(0,1,0)	(0,1)
(0,1,0)	(0,1)	$\{b\}$	$\{b\}$	b	(1,0,0)	(1,0)

Since in each case, the (only) symbol of $Symbols1$ is also an element of $Symbols2$, the returned result of checking is TRUE.

3.2 Unbounbded models

For the unbounded case, compatibility checking must include checking the unboundedness condition (a marked net $(\mathcal{N}, m_0)$ is unbounded if there exist

markings m' and m'' reachable from m_0 such that m'' is reachable from m' and m'' is componentwise greater or equal to m'). This condition is checked for the requester as well as for the provider nets by combining these two markings together. More specifically, for each analyzed pair of markings (m, n), an additional check is performed if the set *Analyzed* contains a pair of markings, which is componentwise smaller than (m, n) and from which (m, n) is reachable; if the set *Analyzed* contains such a pair, analysis of (m, n) is discontinued. This additional check is performed by a logical function $Reachable((m, n), Analyzed)$:

proc $CheckUnbounded(\mathcal{N}_r, m_r, \mathcal{N}_p, m_p)$;
begin
  $New := (m_r, m_p)$;
  $Analyzed := \{\}$;
  while $New \neq \{\}$ **do**
    $(m, n) := head(New)$;
    $New := tail(New)$;
    if $(m, n) \notin Analyzed$ **then**
      $Analyzed := Analyzed \cup \{(m, n)\}$;
      $Symbols1 := Enabled(\mathcal{N}_r, SkipEps(\mathcal{N}_r, m)$;
      $Symbols2 := Enabled(\mathcal{N}_p, SkipEps(\mathcal{N}_p, n)$;
      if $Symbols1 \cap Symbols2 = \{\}$ **then return** FALSE **fi**;
      if not $Reachable((m, n), Analyzed)$ **then**
        for each x **in** $Symbols1$ **do**
          if $x \in Symbols2$ **then**
            $append(New, (next(\mathcal{N}_r, m, x), next(\mathcal{N}_p, n, x))$
          fi
        od
      fi
    fi
  od;
  return TRUE
end;

Example. Fig.2 shows a modified model of a provider which still requires that each operation `b` is preceded by an operation `a`, but which also allows several operations `a` to be performed before any of the corresponding `b` operations is requested (which is not allowed in model shown in Fig.1). This provider net is unbounded.

Checking the compatibility of Requester-1 and Provider in Fig.2, performed by *CheckUnboundedOne*, can be illustrated by the following table:

m	n	$Symbols1$	$Symbols2$	x	$next(\mathcal{N}_r, m, x)$	$next(\mathcal{N}_p, n, x)$
(1,0,0)	(1,0)	$\{a\}$	$\{a\}$	a	(0,1,0)	(1,1)
(0,1,0)	(1,1)	$\{b\}$	$\{a, b\}$	b	(1,0,0)	(1,0)

Again, since for each case the (single) element of *Symbols1* is also an element of *Symbols2*, the result of checking is TRUE.

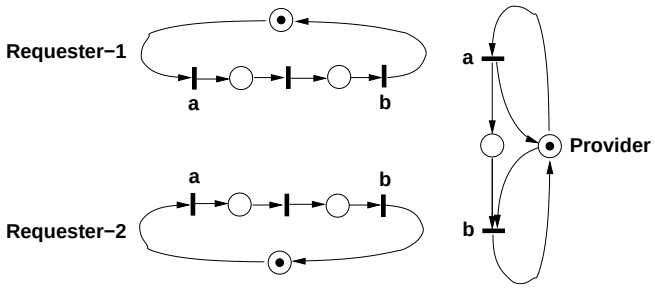

Fig.2. Two requesters with a modified provider.

4 Incremental composition

Incremental composition of interacting components takes advantage of the cyclic nature of component behavior, and allows the interleaving at the level of cycles. So, taking this behavioral cyclicity into account, the condition of compatibility of interacting components can be rewritten as:

$$\mathcal{L}_r^* \subseteq \mathcal{L}_p^*$$

where $\mathcal{L}_r$ is the single-cycle language of the requester component and $\mathcal{L}_p$ is the single-cycle language of the provider component. These single-cycle languages can be just sequences of services (requested or provided) but normally they are more sophisticated and can even be infinite.

The above compatibility condition can be simplified to:

$$\mathcal{L}_r \subseteq \mathcal{L}_p.$$

For the case of k requester components interacting with a single provider, for incremental composition the combined language of requesters becomes:

$$(\mathcal{L}_{r1} \cup \mathcal{L}_{r2} \cup \cdots \cup \mathcal{L}_{rk})^*$$

and then the simplified compatibility condition is:

$$\mathcal{L}_{r1} \cup \mathcal{L}_{r2} \cup \cdots \cup \mathcal{L}_{rk} \subseteq \mathcal{L}_p$$

which is equivalent to

$$\mathcal{L}_{r1} \subseteq \mathcal{L}_p \wedge \mathcal{L}_{r2} \subseteq \mathcal{L}_p \wedge \cdots \wedge \mathcal{L}_{rk} \subseteq \mathcal{L}_p$$

so, instead of checking the compatibility of interleaved requests, it is sufficient to check if each requester component is compatible with the provider. Consequently, the incremental composition eliminates the need for exhaustive combinatorial checking of the behavior of the composed system.

It can be observed that a straightforward criterion for incremental composition is that the set *First(L_r)* of leading symbols of the single-cycle languge $\mathcal{L}_r$ is disjoint with the set *Follow(L_p)* of non-leading symbols of the single-cycle language $\mathcal{L}_p$:

$$First(\mathcal{L}_r) \cap Follow(\mathcal{L}_p) \ = \ \{\}$$

where (S is the set of services required and provided by the components):

$$First(\mathcal{L}) \ = \ \{a \in S \mid \exists x \in S^* : ax \in \mathcal{L}\},$$

$$Follow(\mathcal{L}) \ = \ \{a \in S \mid \exists x \in S^+, y \in S^* : xay \in \mathcal{L}\}.$$

Example. The languages of Requester-1, Requester-2 and Provider shown in Fig.1 are `(ab)*`, their single-cycle languages are `(ab)`, so $First(\mathcal{L}_r) = \{\mathtt{a}\}$, $Follow(\mathcal{L}_p) = \{\mathtt{b}\}$, and $First(\mathcal{L}_r) \cap Follow(\mathcal{L}_p) = \{\}$, so the composition is incremental.

For Fig.2, the languages of Requester-1 and Requester-2 are also `(ab)*`, but the language of Provider is nonregular. In this case, $First(\mathcal{L}_r) = \{\mathtt{a}\}$, $Follow(\mathcal{L}_p) = \{\mathtt{a}, \mathtt{b}\}$, and $First(\mathcal{L}_r) \cap Follow(\mathcal{L}_p) \neq \{\}$, so the composition of these models is not incremental and requires a more detailed verification.

5 Concluding remarks

Incremental composition eliminates the exhaustive verification of the behavior of the composed system by restricting the behavior of interacting components. A simple criterion can be used to check if the interacting components satisfy the requirement of incremental composition.

A different approach, called component adaptation is proposed in [6]. Its main idea is to identify mismatches of interacting components and to generate (on the basis of formal specification of components) component adaptors which eliminate the identified mismatches. The formal foundation for such an approach is provided in [28].

A similar approach is proposed in [20].

Different languages, tools, and environments that support some degree of component–oriented development, support some kinds of components and component composition, but no common model exists. Therefore it is difficult to compare different approaches in a unifor way and is difficult to reason about inteoperability between languages and platforms. More research in this area is expected.

On the other hand, an interesting (and challenging) task that needs to be addressed is the derivation of Petri net behavioral models of components. The derivation should be automated using either component formal specifications or component implementations.

Acknowledgements The Natural Sciences and Engineering Research Council of Canada partially supported this research through grant RGPIN-8222. Helpful remarks of three anonymous reviewers are gratefully acknowledged.

References

1. Aalst van der W M P, Hee van K M, Torn van der R A (2002) Component-based software architecture: a framework based on inheritance of behavior. Science of Computer Programming, vol.42, no.2-3, pp.129-171
2. Attiogbé C, André P, Ardourel G (2006) Checking component composability. Proc. 5-th Int. Symp. on Software Composition (Lecture Notes in Computer Science 4089), pp.18-33
3. Batiry D, Singhal V, Thosmas J, Dasari S, Geract B, Sirkin M (1994) The Gen Voca model of software system generators. IEEE Software, vol.11, n.5, pp.89-94
4. Belguidoum M, Dagnat F (2008) Formalization of component substitutability. Electronic Notes in Theoretical Computer Science, vol.215, pp.75-92
5. Bracha G, Cook W (1990) Mixin-based inheritance. Proc. Joint ACM Conf. on Object-Oriented Pogramming, Systems, Languages and Applications and the European Conf. on Object-Oriented Programming, pp.303-311
6. Bracciali A, Brogi A, Canal C (2005) A formal approach to component adaptations. The Journal of Systems and Software, vol.74, n.1, pp.45-54
7. Brada P, Valenta L (2006) Practical verification of component substitutability using subtype relation. Proc. Int. Conf. on Software Engineering and Advanced Applications (SEAA'06), pp.38-45
8. Canal C, Pimentel E, Troya J M (2001) Compatibility and inheritance in software architectures. Science of Computer Programming, vol.41, no.2, pp.105-138
9. Cerna I, Varekove P, Zimmerova B (2006) Component substitutability via equivalencies of component-interaction automata. Proc. Int. Workshop on Formal Aspects of Component Software (FACS'06), pp.115-130
10. Chaki S, Clarke S M, Groce A, Jha S, Veith H (2004) Modular verification of software components in C. IEEE Trans. on Software Engineering, vol.30, no.6, pp.388-402
11. Costa Seco J, Caires L (2000) A basic model of typed components. Proc. 14-th European Conf. on Object-Oriented Programming, London, UK, pp.108-128
12. Craig D C, Zuberek W M (2006) Compatibility of software components – modeling and verification. Proc. Int. Conf. on Dependability of Computer Systems, Szklarska Poreba, Poland, pp.11-18
13. Craig D C, Zuberek W M (2007) Petri nets in modeling component behavior and verifying component compatibility. Proc. Int. Workshop on Petri Nets and Software Engineering, Siedlce, Poland, pp.160-174
14. Crnkovic I, Schmidt H W, Stafford J, Wallnau K (2005) Automated component-based software engineering. The Journal of Systems and Software, vol.74, n.1, pp.1-3
15. Garlan D (2003) Formal modeling and analysis of software architecture: components, connectors, and events. Proc. Third Int. School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures (SFM 2003), Bertinoro, Italy (Lecture Notes in Computer Science 2804), pp.1-24
16. Hameirlain N (2007) Flexible behavioral comatibility and substitutability for component protocols: a formal specification. Proc. 5-th Int. Conf. on Software Engineering and Formal Methods, London, England, pp.391-400
17. Henrio L, Kammueller F, Khan M U (2009) A framework for reasoning on component composition. Proc. 8-th Int. Symp. on Formal Methods for Components and Objects, Eindhoven, The Netherlands (Lecture Notes in Computer Science 6286), pp.41-69

18. Hopcroft J E, Motwani R, Ullman J D (2001) Introduction to automata theory, languages, and computations (2 ed.). Addison-Wesley
19. Karlsson D, Eles P, Peng Z (2002) Formal verification on a component-based reuse methodology. Proc. 15-th Int. Symp. on System Synthesis, Kyoto, Japan, pp.156-161
20. Leicher A, Busse S, Suess J G (2005) Analysis of compositional conflicts in component-based systems. Proc. 4-th Int. Workshop on Software Composition; Edinburgh, UK (Lecture Notes in Computer Science 3628), pp.67-82
21. Liskov B, Wing J (1994) A behavioral notion of subtyping. ACM Trans. on Programming Languages and Systems, vol.19, no.6, pp.1811-1841
22. Magee J, Dulay N, Kramer J (1995) Specifying distributed software architectures. Proc. 5-th European Software Engineering Conference, Sitges, Spain (Lecture Notes in Computer Science 989), pp.137-153
23. Murata T (1989) Petri nets: properties, analysis, and applications. Proceedings of the IEEE, vol.77, no.4, pp.541-580
24. Nierstrasz O, Meijler T (1995) Research directions on software composition. ACM Computing Surveys, vol.27, no.2, pp.262-264
25. Reisig W (1985) Petri nets – an introduction (EATCS Monographs on Theoretical Computer Science 4). Springer-Verlag
26. Suedholt M (2005) A model of components with non-regular protocols. Proc. 4-th Int. Workshop on Software Composition; Edinburgh, UK (Lecture Notes in Computer Science 3628), pp.99-113
27. Szyperski C (2002) Component software: beyond object-oriented programming (2 ed.). Addison–Wesley Professional
28. Yellin D M, Strom R E (1997) Protocol specifications and component adaptors. ACM Trans. on Programming Languages and Systems, vol.19, no.2, pp.292-333
29. Zaremski A M, Wang J M (1997) Specification matching of software components. ACM Trans. on Software Engineering and Methodology, vol.6, no.4, pp.333-369
30. Zuberek W M (2010) Checking compatibility and substitutability of software components. Models and Methodology of System Dependability, ch.14, pp.175-186, Oficyna Wydawnicza Politechniki Wroclawskiej, Wroclaw