New Results in Dependability and Complex Sytems (Advances in Intelligent and Soft Computing 224);
8-th Int. Conf. on Dependability of Complex Systems; Brunow Palace, Poland; Sept. 8-13, 2013, pp.543-552.

DOI 10.1007/978-3-319-00945-2_50.

Shuffle–based verification of component compatibility

Włodek M. Zuberek

Department of Computer Science, Memorial University,
St.John's, NL, Canada A1B 3X5
email: wlodek@mun.ca

Abstract. Similarly as in earlier work on component compatibility, the behavior of components is specified by component interface languages. The shuffle operation is introduced to represent possible interleavings of service requests that originate at several concurrent components. The paper shows that the verification of component compatibility is possible without the exhaustive analysis of the state space of interacting components. Exhaustive analysis of state spaces was the basis of earlier approaches to compatibility verification.
Keywords: software components, component-based systems, component composition, component compatibility, compatibility verification, shuffle operation, labelled Petri nets.

1 Introduction

In component-based systems [8], two interacting components, one requesting services and the other providing them, are considered compatible if all possible sequences of services requested by one component can be provided by the other one. This concept of component compatibility can be extended to sets of interacting components, however, in the case of several requester components, as is typically the case of client–server applications, the requests from different components can be interleaved and then verifying component compatibility must take into account all possible interleavings of requests. Such interleaving of requests can lead to unexpected behavior of the composed system, e.g. a deadlock can occur [17], [18].

The behavior of components is usually described at component interfaces [14] and the components are characterized as requester (active) and provider (reactive) components. Although several approaches to checking component composability have been proposed [1], [2], [3], [9], [11], [15], further research is needed to make these ideas practical [7].

The paper is an extension of previous work on component compatibility and substitutability [6], [16], [17], [18]. Using the same formal specification of component behavior in the form of interface languages defined by labeled Petri nets, the paper extends the linguistic approach to the verification of component compatibility. A shuffle operation is proposed to represent the interleavings of requests originating at concurrent components. This shuffle of requests is matched with the (interface) language of the service provider. If the provider languages matches the interleavings of the requester components, the components are considered compatible, otherwise some correcting procedure is required (in the form of redesign of some components or additional constraints which preventing some interleavings to happen).

Since interface languages are usually infinite, their compact finite specification is needed for effective verification, comparisons and other operations. Labeled Petri nets [16], [17] are used as such specification.

Petri nets [12], [13] 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 elements 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.

Section 2 introduces the shuffle operation applied to sequences of requests as well as collections of such sequences. Interface languages as the description of component's behavior are recalled in Section 3, while Section 4 provides the linguistic version of component compatibility. Section 5 illustrates the shuffle-based verification of components compatibility and Section 6 concludes the paper.

2 Shuffle and swap

The shuffle operation, used in mathematical linguistics [10] to merge strings, is used here to represent the interleaving of requests from several components. So, if x_i and x_j are sequences of requests from components "i" and "j", then $\texttt{shuffle}(x_i, x_j)$ denotes the set of sequences of merged requests in which all elements of x_i and x_j occur in their original order, but the elements of x_i can be arbitrarily interleaved with the elements of x_j. For example:

$$\texttt{shuffle}(ab, cd) = \{abcd, acbd, acdb, cabd, cadb, cadb, cdab\}.$$

The set of "shuffled" strings can also be created by successive applications of a swap operation to the concatenated string xy, where each swap operation "swaps" (changes the positions of) two adjacent symbols provided one of these two symbols is an element of x and the other is an element of y, so a simple swap (`sswap`):

$$\texttt{sswap}(abcd) = \{acbd\}$$

while the consecutive simple swap operation can be applied in three different ways, so:

$$\texttt{sswap}(acbd) = \{cabd, abcd, acdb\}$$

Let a (general) swap operation be the reflexive, transitive closure of the simple swap operation. Then:

$$\texttt{shuffle}(x, y) = \texttt{swap}(xy).$$

The shuffle operation can be naturally extended to sets of sequences:

$$\texttt{shuffle}(A, B) = \{\texttt{shuffle}(x, y) \mid x \in A \wedge y \in B\}.$$

Moreover, it can be observed that:

$$\texttt{shuffle}(x, \texttt{shuffle}(y, z)) = \texttt{shuffle}(\texttt{shuffle}(x, y), z)$$

so the operation can be generalized as:

$$\texttt{shuffle}(x, y, z, \cdots)$$

as well as:

$$\texttt{shuffle}(A, B, C, \cdots).$$

3 Component behavior

The behavior of a component, at its interface, can be represented by a cyclic labeled Petri net [5], [6], [17]:

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

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 can easily capture 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 [10]. Therefore, they are significantly more general than languages defined by finite automata [4], but their compatibility verification is also more difficult than in the case of regular languages.

4 Component compatibility

Interface languages of interacting components are used to define the compatibility of components. For a pair of interacting components, a requester component "r" and a provider component "p" are compatible if and only if all sequences of services requested by "r" can be provided by "p", i.e., if and only if:

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

In the case of several requester components, indicated by subscripts "$i \in I$" where I is an index set, interacting with a single provider component "p", the component compatibility requires that all sequences of (interleaved) requests be satisfied by the provider, so in a straightforward case:

$$\texttt{shuffle}(\mathcal{L}_i | i \in I) \subseteq \mathcal{L}_p.$$

Often however, some requests cannot be satisfied when they are requests and are delayed because some other operations performed by the provider component. In such cases the services can be provided in a sequence which is different from the sequence of requests. Therefore, it is convenient to decompose the sequence of requests x in two parts y and z, $x = yz$, the initial part y which is served in the order of requests, and the remaining part z where the services are provided in an order different than requested. And then the component compatibility condition is:

$$\forall x \in \mathtt{shuffle}(\mathcal{L}_i | i \in I)) : x \in \mathcal{L}_p \;\vee\; x = yz \wedge \{y\} \circ \mathtt{swap}(z) \cap \mathcal{L}_p \neq \emptyset$$

where $\circ$ denotes set concatenation, and $\emptyset$ is the empty set.

5 Example

A simple system of two requesters and a single provider is shown in Fig.1 [18]. The interface language of the provider is described by a regular expression:

$$\mathcal{L}_p = ((\mathtt{ab} + \mathtt{ba})\mathtt{c})^*$$

and the language of the (interleaved) requests from two requesters is:

$$\mathcal{L}_r = (\mathtt{shuffle}(\mathtt{abc}, \mathtt{bac}))^*.$$

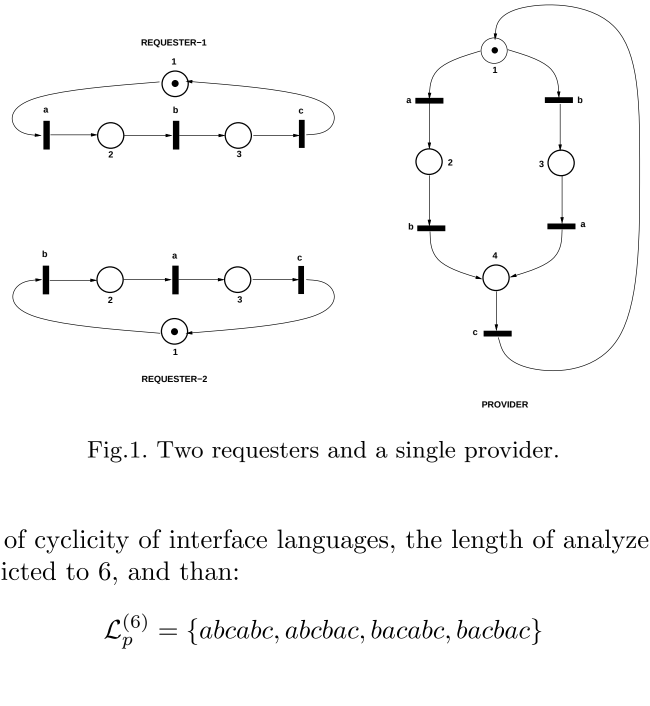

Fig.1. Two requesters and a single provider.

Because of cyclicity of interface languages, the length of analyzed sequences can be restricted to 6, and than:

$$\mathcal{L}_p^{(6)} = \{abcabc, abcbac, bacabc, bacbac\}$$

and there are 10 different strings in $\mathcal{L}_r^{(6)}$):
$\mathcal{L}_r^{(6)} = \{ababcc, abacbc, abbacc, abbcac, abcbac,$
$babacc, babcac, baabcc, baacbc, bacabc\}$.

Verification of the component consistency checks the sequences of requests in $\mathcal{L}_r^{(6)}$ (in the following table, symbol subscripts indicate the original component requesting the service) looking for a matching sequence in $\mathcal{L}_p^{(6)}$:

$x \in \mathcal{L}_r^6$	matching in $\mathcal{L}_p^{(6)}$
$a_1b_2a_2b_1c_1c_2$	$\{ab\} \circ \texttt{swap}(a_2b_1c_1c_2) \cap \mathcal{L}_p^{(6)} = \emptyset$
$a_1b_2a_2b_1c_2c_1$	$\{ab\} \circ \texttt{swap}(a_2b_1c_2c_1) \cap \mathcal{L}_p^{(6)} = \emptyset$
$a_1b_1b_2a_2c_1c_2$	$cbac \in \texttt{swap}(b_2a_2c_1c_2)$ and $ab \circ cbac \in \mathcal{L}_p^{(6)}$
$a_1b_1b_2a_2c_2c_1$	$cbac \in \texttt{swap}(b_2a_2c_2c_1)$ and $ab \circ cbac \in \mathcal{L}_p^{(6)}$
$a_1b_2b_1a_2c_1c_2$	$\{ab\} \circ \texttt{swap}(a_2b_1c_1c_2) \cap \mathcal{L}_p^{(6)} = \emptyset$
$a_1b_2b_1a_2c_2c_1$	$\{ab\} \circ \texttt{swap}(a_2b_1c_2c_1) \cap \mathcal{L}_p^{(6)} = \emptyset$
$a_1b_1b_2c_1a_2c_2$	$cbac \in \texttt{swap}(b_2c_1a_2c_2)$ and $ab \circ cbac \in \mathcal{L}_p^{(6)}$
$a_1b_2b_1c_1a_2c_2$	$\{ab\} \circ \texttt{swap}(a_2b_1c_1c_2) \cap \mathcal{L}_p^{(6)} = \emptyset$
$a_1b_1c_1b_2a_2c_2$	$abcbac \in \mathcal{L}_p^{(6)}$
$b_2a_1b_1a_2c_1c_2$	$\{ba\} \circ \texttt{swap}(b_1a_2c_1c_2) \cap \mathcal{L}_p^{(6)} = \emptyset$
$b_2a_1b_1a_2c_2c_1$	$\{ba\} \circ \texttt{swap}(b_1a_2c_2c_1) \cap \mathcal{L}_p^{(6)} = \emptyset$
$b_2a_2a_1b_1c_1c_2$	$cabc \in \texttt{swap}(a_1b_1c_1c_2)$ and $ba \circ cabc \in \mathcal{L}_p^{(6)}$
$b_2a_2a_1b_1c_2c_1$	$cabc \in \texttt{swap}(a_1b_1c_2c_1)$ and $ba \circ cabc \in \mathcal{L}_p^{(6)}$
$b_2a_1b_1c_1a_1c_2$	$\{ba\} \circ \texttt{swap}(b_1c_1a_1c_2) \cap \mathcal{L}_p^{(6)} = \emptyset$
$b_2a_1b_1c_2a_1c_1$	$\{ba\} \circ \texttt{swap}(b_1c_2a_1c_1) \cap \mathcal{L}_p^{(6)} = \emptyset$
$b_2a_2a_1c_2b_1c_1$	$cabc \in \texttt{swap}(a_1c_2b_1c_1)$ and $ba \circ cabc \in \mathcal{L}_p^{(6)}$
$b_2a_1a_2c_2b_1c_1$	$\{ba\} \circ \texttt{swap}(a_2c_2b_1c_1) \cap \mathcal{L}_p^{(6)} = \emptyset$
$b_2a_2c_2a_1b_1c_1$	$bacabc \in \mathcal{L}_p^{(6)}$

All rows which do not have a matching sequence indicate component incompatibilities.

For example, for the first sequence of requests, $x = a_1b_2a_2b_1c_1c_2$, x is decomposed into ab (which can be matched by the provider) and the remaining sequence $a_2b_1c_1c_2$, for which the `swap` operation creates the set:

$$\{a_2b_1c_1c_2, a_2b_1c_2c_1, a_2c_2b_1c_1, b_1a_2c_1c_2, b_1a_2c_2c_1, b_1c_1a_2c_2\}$$

or, removing the subscripts:

$$\{abcc, acbc, bacc, bcac\}$$

and then the intersection:

$$\{ab\} \circ \{abcc, acba, baccbcac\} \cap \mathcal{L}_p^{(6)}$$

is empty indicating the incompatibility.

The second sequence is basically identical, and so on.

To eliminate incompatibilities existing in this example, service renaming was proposed in [18], for example, the provider language can be (formally) changed from:

$$((\mathtt{ab}+\mathtt{ba})\mathtt{c})*$$

to:

$$((\mathtt{aB}+\mathtt{bA})\mathtt{c})*$$

by introducing services `A` and `B` as renamed services `a` and `b`, respectively, and changing the languages of the requesters n a similar way, to `(aBc)*` and `(bAc)*`, respectively. After such renaming the interleavings of the two requesters are:

$$\begin{aligned}\mathtt{shuffle}(aBc, bAc) = \{&abABcc, abAcBc, abBAcc, abBcAn, aBbAcc, aBbcAc,\\ &aBcbAc, baBacc, baBcAc, baABcc, baAcBc, bAaBcc, bAacBc, bAcaBc\}\end{aligned}$$

and the verification of the component compatibility follows the same steps as in the previous case. For example, the first sequence $abABcc$ can be decomposed into the leading a and the remaining $bABcc$ with the following set of swapped sequences:

$$\begin{aligned}\mathtt{swap}(b_2A_2B_1c_1c_2) = \{&b_2B_1A_2c_1c_2, b_2B_1c_1A_2c_2,\\ &B_1b_2A_2c_1c_2, B_1b_2c_1A_2c_2, B_1c_1b_2A_2c_2\}\end{aligned}$$

Since, in this case::

$$\mathcal{L}_p^{(6)} = \{aBcaBc, aBcbAc, bAcaBc, bAcbAc\}$$

the compatibility is verified as:

$$a \circ \text{swap}(b_2A_2B_1c_1c_2) \cap \mathcal{L}_p^{(6)} \;=\; \{a_1B_1c_1b_2A_2c_2\} \;\neq\; \emptyset,$$

so the incompatibility has been removed.

It should be observed that the initial decomposition of the analyzed sequence is not necessary; its purpose is to simplify the verification process by reducing the length of the analyzed sequence.

The remaining sequences are verified similarly, as shown in the following table.

$x \in \mathcal{L}_r^6$	matching in $\mathcal{L}_p^{(6)}$
$a_1b_2A_2B_1c_1c_2$	$BcbAc \in \mathtt{swap}(b_2A_2B_1c_1c_2)$ and $a \circ BcbAc \in \mathcal{L}_p^{(6)}$
$a_1b_2A_2B_1c_2c_1$	$BcbAc \in \mathtt{swap}(b_2A_2B_1c_2c_1)$ and $a \circ BcbAc \in \mathcal{L}_p^{(6)}$
$a_1b_2A_2c_2B_1c_1$	$BcbAc \in \mathtt{swap}(b_2A_2c_2B_1c_1)$ and $a \circ BcbAc \in \mathcal{L}_p^{(6)}$
$a_1b_2B_1A_2c_1c_2$	$BcbAc \in \mathtt{swap}(b_2B_1A_2c_1c_2)$ and $a \circ BcbAc \in \mathcal{L}_p^{(6)}$
$a_1b_2B_1A_2c_2c_1$	$BcbAc \in \mathtt{swap}(b_2B_1A_2c_2c_1)$ and $a \circ BcbAc \in \mathcal{L}_p^{(6)}$
$a_1b_2B_1c_1A_2c_2$	$BcbAc \in \mathtt{swap}(b_2B_1c_1A_2c_2)$ and $a \circ BcbAc \in \mathcal{L}_p^{(6)}$
$a_1B_1b_2A_2c_1c_2$	$cbAc \in \mathtt{swap}(b_2A_2c_1c_2)$ and $aB \circ cbAc \in \mathcal{L}_p^{(6)}$
$a_1B_1b_2A_2c_2c_1$	$cbAc \in \mathtt{swap}(b_2A_2c_2c_1)$ and $aB \circ cbAc \in \mathcal{L}_p^{(6)}$
$a_1B_1b_2c_1A_2c_2$	$cbAc \in \mathtt{swap}(b_2c_1A_2c_2)$ and $aB \circ cbAc \in \mathcal{L}_p^{(6)}$
$a_1B_1c_1b_2A_2c_2$	$aBcbAc \in \mathcal{L}_p^{(6)}$
$b_2a_1A_2B_1c_1c_2$	$AcaBc \in \mathtt{swap}(a_1A_2B_1c_1c_2)$ and $b \circ AcaBc \in \mathcal{L}_p^{(6)}$
$b_2a_1A_2B_1c_2c_1$	$AcaBc \in \mathtt{swap}(a_1A_2B_1c_2c_1)$ and $b \circ AcaBc \in \mathcal{L}_p^{(6)}$
$b_2a_1A_2c_2B_1c_1$	$AcaBc \in \mathtt{swap}(a_1A_2c_2B_1c_1)$ and $b \circ AcaBc \in \mathcal{L}_p^{(6)}$
$b_2a_1B_1A_2c_1c_2$	$AcaBc \in \mathtt{swap}(a_1B_1A_2c_1c_2)$ and $b \circ AcaBc \in \mathcal{L}_p^{(6)}$
$b_2a_1B_1A_2c_2c_1$	$AcaBc \in \mathtt{swap}(a_1B_1A_2c_2c_1)$ and $b \circ AcaBc \in \mathcal{L}_p^{(6)}$
$b_2a_1B_1c_1A_2c_2$	$AcaBc \in \mathtt{swap}(a_1B_1c_1A_2c_2)$ and $b \circ AcaBc \in \mathcal{L}_p^{(6)}$
$b_2A_2a_1B_1c_1c_2$	$caBc \in \mathtt{swap}(a_1B_1c_1c_2)$ and $bA \circ AcaBc \in \mathcal{L}_p^{(6)}$
$b_2A_2a_1B_1c_2c_1$	$caBc \in \mathtt{swap}(a_1B_1c_2c_1)$ and $bA \circ AcaBc \in \mathcal{L}_p^{(6)}$
$b_2A_2a_1c_2B_1c_1$	$caBc \in \mathtt{swap}(a_1c_2B_1c_1)$ and $bA \circ AcaBc \in \mathcal{L}_p^{(6)}$
$b_2A_2c_2a_1B_1c_1$	$bAcaBc \in \mathcal{L}_p^{(6)}$

In this case, all sequences of requests are matched by the provider component, so the components are compatible.

6 Concluding remarks

The paper shows that the verification of component compatibility based on the exhaustive analysis of the “state space”, as discussed in [16] and [17], can be replaced by a simple analysis of languages that describe the sets of request sequences that can be generated by interacting compoments. In fact, once the interface languages are known, the behavioral models of components are not needed at all.

It is believed that the proposed verification of component compatibility can be quite efficient since many symmetries and partial orders can be taken into account. All these properties are not addressed in this paper.

It should be noticed that the discussion was restricted to a single provider component. In the case of several providers, each provider can be considered independently of other, so a single provider case is not really a restriction.

Also, an important aspect of component compatibility is its incremental verification. The approach described in this paper is not incremental but may provide a foundation for an incremental approach.

The paper did not address the question of deriving behavioral models of components (which is common to all component-based studies). Such models, at least theoretically, could be derived from formal component specifications, or perhaps could be obtained through analyzing component implementations. Since the component compatibility verification proposed in this paper does not require the use of the underlying component models (they are used only to define the interface languages), these interface languages could also be determined experimentally, by executing the components and collecting the information about the sequences of service requests.

Acknowledgement

The Natural Sciences and Engineering Research Council of Canada partially supported this research through grant RGPIN-8222.

References

1. Attiogbe C, Andre P, Ardourel G (2006) Checking component composability. Proc. 5-th Int. Symp. on Software Composition (LNCS 4089), pp.18-33
2. Baier C, Klein J, Klueppenholz S (2011) Modeling and verification of components and connectors. In: "Formal Methods for Eternal Networked Software Systems" (LNCS 6659), pp.114-147
3. Broy M (2006) A theory of system interaction: components, interfaces, and services. In: "Interactive Computations: The New Paradigm", Springer-Verlag, pp.41-96
4. 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
5. 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
6. 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
7. Crnkovic I, Schmidt H W, Stafford J, Wallnau K (2005) Automated component-based software engineering. The Journal of Systems and Software, vol.74, no.1, pp.1-3
8. 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) (LNCS 2804), pp.1-24
9. 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 (LNCS 6286), pp.41-69
10. Hopcroft J E, Motwani R, Ullman J D (2001) Introduction to automata theory, languages, and computations (2 ed.). Addison–Wesley
11. 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 (LNCS 3628), pp.67-82

12. Murata T (1989) Petri nets: properties, analysis, and applications. Proceedings of the IEEE, vol.77, no.4, pp.541-580
13. Reisig W (1985) Petri nets – an introduction (EATCS Monographs on Theoretical Computer Science 4). Springer-Verlag
14. Szyperski C (2002) Component software: beyond object-oriented programming (2 ed.). Addison–Wesley Professional
15. 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
16. Zuberek W M (2010) Checking compatibility and substitutability of software components. In: Models and Methodology of System Dependability, Oficyna Wydawnicza Politechniki Wroclawskiej, ch.14, pp.175-186
17. Zuberek W M (2011) Incremental composition of software components. In: Dependable Computer Systems (Advances in Intelligent and Soft Computing 97), Springer-Verlag, pp.301-311
18. Zuberek W M (2012) Service renaming in component. In: Complex Systems and Dependability (Advances in Intelligent and Soft Computing 170); ed. W. Zamojski, J. Kacprzyk, J. Mazurkiewicz, J. Sugier, T. Walkowiak, Springer-Verlag, pp.319-330