Dependable Computer Systems (Advances in Intelligent and Soft Computing 97), pp.319–330.
7-th Int. Conf. on Dependability of Complex Systems, Brunow Palace, Poland; 25-29 June 2012.

DOI 10.1007/978-3-642-30662-4_21.

Service Renaming in Component Composition

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, the behavior of components is usually described at component interfaces and the components are characterized as requester (active) and provider (reactive) components. Two interacting components are considered compatible if all possible sequences of services requested by one component can be provided by the other component. This concept of component compatibility can be extended to sets of interacting components, however, in the case of several requester components interacting with one or more provider components, as is typically the case of cleint–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. Service renaming is proposed as a method of systematic eliminating of such unexpected effects and streamlining component compositions.

1 Introduction

In component-based systems, components represent high-level software abstraction which must be generic enough to work in a variety of contexts and in cooperation with other components, but which also must be specific enough to provide easy reuse [9].

Primary reasons for using components are [8]: separability of components from their contexts; independent component development, testing and later reuse; upgrade and replacement in running systems. Component composability is often taken for granted, while it actually is influenced by a number of factors such as operating platforms, programming languages or the specific middleware technology in which the components 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 [14]. For this to happen, automated component-based software engineering must resolve a number of issues, including efficient verification of component compatibility.

The behavior of components is usually described at component interfaces [16] and the components are characterized as requester (active) and provider (reactive) components. Although several approaches to checking component composability have been proposed [1] [4] [10] [12], further research is needed to make these ideas practical [8]. Usually two interacting components are considered compatible if all sequences of services requested by one component can be provides by the other component. In the case of several components interacting with a single provider, as is typically the case of internet applications (e.g., client-server systems), the requests from different components can be interleaved and then verifying component compatibility must check all possible interleavings of requests from all interacting components for possible conflicts. Such interleaving of service requests can lead to unexpected behavior of the composed system; e.g., a deadlock can occur. Service renaming is proposed as a systematic method of eliminating request conflicts and streamlining component composition.

The idea of service renaming for the elimination of conflicting requests can be illustrated by the following simple example. Let the languages of two requester components be specified by regular expressions "`(abc)*`" and "`(bac)*`" ("`a`", "`b`" and "`c`" are services requested by the components), and let the language of the corresponding provider component be "`((ab + ba)c)*`". It can be easily checked that both requester components are compatible with the provider [20], however, one of possible interleavings of the requests is "`(ab(bc + ac))*`", which - if allowed - results in a deadlock. Service renaming which eliminates this deadlock replaces, for example, "`a`" by "`A`" in the sequence "`bac`" and "`b`" by "`B`" in the sequence "`abc`", and then any interleaving of "`(aBc)*`" with "`(bAc)`" is compatible with the provider "`((aB + bA)c)*`".

The paper is continuation of previous work on component compatibility and substitutability [7] [19] [20] [21]. Using the same formal specification of component behavior in the form of component languages, the paper proposes an approach to identify component conflicts in component composition and systematic renaming of services as a conflict removal method.

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

Petri nets [13] [15] 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 recalls the concept of component languages as a characterization of component's behavior. Component languages are used in Section 3 to define

component compatibility. Service renaming is described in Section 4 while Section 5 concludes the chapter.

2 Modeling component behavior

The behavior of a component, at its interface, can be represented by a cyclic labeled Petri net [6] [20]:

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

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 [11]. Therefore, they are significantly more general than languages defined by finite automata [5], 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.

3.1 Bounded case

In the case of bounded nets, checking compatibility of a single requester with a single provider components 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 $\mathcal{G}(\mathcal{M}_p)$, which is described in detail in [20]. For the case of several requester components $\mathcal{M}_i, i = 1, ..., k$, interacting with a single provider component $\mathcal{M}_p$, first the compatibility of each requester with the provider is checked in [20]. Then the interleaving of requests are checked for progress, and is a deadlock is discovered, the set of interacting

components cannot be compatible. For simplicity, the family of requester components is represented by a vector $\mathbf{N}_r$ with individual components $\mathbf{N}_r[1]$, $\mathbf{N}_r[2]$, ... $\mathbf{N}_r[k]$. Similarly, the markings for $\mathbf{N}_r$ are denoted be a vector $\mathbf{m}_r$ with individual marking functions $\mathbf{m}_r[1]$, $\mathbf{m}_r[2]$, ... $\mathbf{m}_r[k]$.

The following logical function $CheckProgressB$ is used when all requester and provider languages are defined by bounded marked Petri nets $(\mathcal{N}_i, m_i)$, $i = 1, ..., k$, and $(\mathcal{N}_p, m_p)$, respectively. The function performs exhaustive analysis of possible interleavings of requests, checking the progress of the composed model; if there is no progress (which means, a deadlock has been created), FALSE is returned. In the pseudocode below, New is a sequence (a queue) 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 CheckProgressB(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, n) ∉ Analyzed then
            Analyzed := Analyzed ∪ {(m, n)};
            noprogress := true;
            for i := 1 to k do
                Symbols1 := Enabled(N_r[i], SkipEps(N_r[i], m[i]));
                Symbols2 := Enabled(N_p, SkipEps(N_p, n));
                if Symbols1 ∩ Symbols2 ≠ {} then
                    noprogress := false;
                    m' := m;
                    for each x in Symbols1 ∩ Symbols2 do
                        m'[i] := next(N_r[i], m[i], x)
                        append(New, (m', next(N_p, n, x))
                    od
                fi
            od;
            if noprogress return FALSE fi
        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.

Example. Fig.1 shows a simple configuration of two (cyclic) requester components and a single provider of three services named `a`, `b` and `c`.

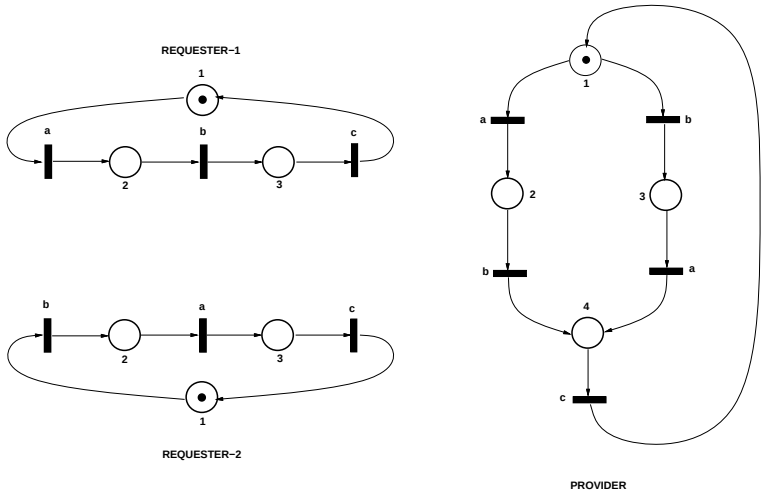

Fig.1. Two requesters and a single provider.

In this case, the languages of the requesters are described by regular expressions "`(abc)*`" and "`(bac)*`" and the language of the provider by "`((ab+ba)c)*`". It can be easily checked that both requesters are compatible with the provider; the languages "`(abc)*`" and "`(bac)*`" are subsets of the language "`((ab+ba)c)*`".

As indicated in the introduction, the combined requests from both requester components are not compatible with the provider shown in Fig.1. For example, if the first request from requester-1 (i.e., "`a`") is followed by the first request from requester-2 (i.e., "`b`"), the composed system becomes deadlocked because further requests are "`b`" (from requester-1) and "`a`" (from requester-2) while the only provided service at this stage is "`c`".

The steps performed by the function *CheckProgressB* for the nets shown in Fig.1 are illustrated in a table, in which the first column, "*conf*", identifies the configuration of the model, while the last column, "ℓ", indicates

the next configuration, reached in effect of the requested/provided service shown in column "x"; columns $\mathbf{m}$ and n show the markings of the requester and provider nets, respectively; i indicates the requester component (used for interleaving), as in function *CheckProgressB*, similarly to the remaining columns of the table:

$conf$	$\mathbf{m}$	n	i	$Symbols1$	$Symbols2$	x	$next(\mathbf{N}_r[i], \mathbf{m}[i], x)$	$next(\mathcal{N}_p, n, x)$	ℓ
0	[(1,0,0),(1,0,0)]	(1,0,0,0)	1	$\{a\}$	$\{a, b\}$	a	(0,1,0)	(0,1,0,0)	1
			2	$\{b\}$	$\{a, b\}$	b	(0,1,0)	(0,0,1,0)	2
1	[(0,1,0),(1,0,0)]	(0,1,0,0)	1	$\{b\}$	$\{b\}$	b	(0,0,1)	(0,0,0,1)	3
			2	$\{b\}$	$\{b\}$	b	(0,1,0)	(0,0,0,1)	4
2	[(1,0,0),(0,1,0)]	(0,0,1,0)	1	$\{a\}$	$\{a\}$	a	(0,1,0)	(0,0,0,1)	4
			2	$\{a\}$	$\{a\}$	a	(0,0,1)	(0,0,0,1)	5
3	[(0,0,1),(1,0,0)]	(0,0,0,1)	1	$\{c\}$	$\{c\}$	c	(1,0,0)	(1,0,0,0)	0
			2	$\{b\}$	$\{c\}$				–
4	[(0,1,0),(0,1,0)]	(0,0,0,1)	1	$\{b\}$	$\{c\}$				–
			2	$\{a\}$	$\{c\}$				–
5	[(1,0,0),(0,0,1)]	(0,0,0,1)	1	$\{a\}$	$\{c\}$				–
			2	$\{c\}$	$\{c\}$	c	(1,0,0)	(1,0,0,0)	0

No component can progress in configuration 4, so this is a deadlock configuration. Consequently the components shown in Fig.1 cannot be compatible. It can be observed that this deadlock configuration can be reached from configuration 0 by requesting service "`a`" (by requester-1) and then in configuration 1, requesting service "`b`" (by requester-2). Configuration 4 can also be reached from configuration 0 by first requesting service "`b`" (by requester-2) and then, in configuration 2, service "`a`" (by requester-2).

It should also be noted that in configurations 3 and 5, only one of the requester components does not progress, so these configurations are not deadlocks.

3.2 Unbounded case

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 requesters as well as for the provider nets by combining the markings together. More specifically, for each analyzed pair of markings $(\mathbf{m}, n)$, an additional check is performed if the set *Analyzed* contains a pair of markings, which is componentwise smaller than $(\mathbf{m}, n)$ and from which $(\mathbf{m}, n)$ is reachable; if the set *Analyzed* contains such a pair, analysis of $(\mathbf{m}, n)$ is discontinued. This additional check is performed by a logical function *Reachable*$((\mathbf{m}, n), Analyzed)$, in which the first argument is a vector of marking functions (which - in this particular case - can be considered as a single marking function obtained by concatenation of all consecutive elements of

$\mathbf{m}$). As in the bounded case, $\mathbf{N}_r$ is a vector of requester nets $[\mathcal{N}_1, \mathcal{N}_2, ..., \mathcal{N}_k]$, also denoted $\mathbf{N}_r[i], i = 1, ..., k$, and $\mathbf{m}_r$ is a vector of marking functions for nets $\mathbf{N}_r[i], i = 1, ..., k$:

```
proc CheckProgressU(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, n) ∉ Analyzed then
            Analyzed := Analyzed ∪ {(m, n)};
            noprogress := true;
            if not Reachable((m, n), Analyzed) then
                for i := 1 to k do
                    Symbols1 := Enabled(N_r[i], SkipEps(N_r[i], m[i]));
                    Symbols2 := Enabled(N_p, SkipEps(N_p, n));
                    if Symbols1 ∩ Symbols2 ≠ {} then
                        noprogress := false;
                        m' := m;
                        for each x in Symbols1S†⇳⌊↾⇕∫∈ do
                            m'[i] := next(N_r[i], m[i], x)
                            append(New, (next(m', next(N_p, n, x))
                        do
                    fi
                od;
                if noprogress return FALSE fi
            fi
        fi
    od;
    return TRUE
end;
```

As in the bounded case, the function *CheckProgressU* returns FALSE if there is a sequence of service requests which cannot be satisfied by the provider component.

Example. Fig.2 shows another model composed of two requester components with languages "`(abc)*`" and "`(ab*c)*`" and an unbounded provider which accepts any sequence of requests of services "`a`", "`b`" and "`c`" such that any prefix of this sequence (including the whole sequence) contains not less requests for service "`a`" than for service "`c`"; the language of this provider is nonregular.

Both requester components are compatible with the provider as their languages are subsets of the provider's language.

The steps used by the function *CheckProgressU* for the components shown in Fig.2 are illustrated in the following table:

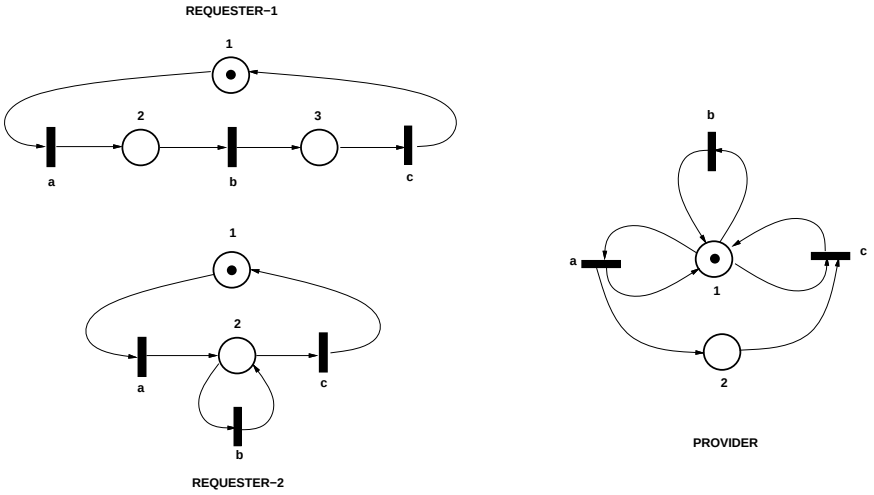

Fig.2. Two requesters with an unbounded provider.

It can be easily checked that both requester components shown in Fig.2 are compatible with the provider. The steps used by *CheckProgressU* for the components shown in Fig.2 are illustrated by the following table:

$conf$	$\mathbf{m}$	n	i	$Symbols1$	$Symbols2$	x	$next(\mathbf{N}_r[i], \mathbf{m}[i], x)$	$next(\mathcal{N}_p, n, x)$	ℓ
0	[(1,0,0),(1,0)]	(1,0)	1	$\{a\}$	$\{a, b\}$	a	(0,1,0)	(1,1)	1
			2	$\{a\}$	$\{a, b\}$	a	(0,1)	(1,1)	2
1	[(0,1,0),(1,0)]	(1,1)	1	$\{b\}$	$\{a, b, c\}$	b	(0,0,1)	(1,1)	3
			2	$\{a\}$	$\{a, b, c\}$	a	(0,1)	(1,2)	4
2	[(1,0,0),(0,1)]	(1,1)	1	$\{a\}$	$\{a, b, c\}$	b	(0,1,0)	(1,2)	4
			2	$\{b, c\}$	$\{a, b, c\}$	b	(0,1)	(1,1)	2
						c	(1,0)	(1,0)	0
3	[(0,0,1),(1,0)]	(1,1)	1	$\{c\}$	$\{a, b, c\}$	c	(1,0,0)	(1,0)	3
			2	$\{a\}$	$\{a, b, c\}$	a	(0,1)	(1,2)	4
4	[(0,0,1),(0,1)]	(1,2)	1	$\{c\}$	$\{a, b, c\}$	c	(1,0,0)	(1,1)	2
			2	$\{b, c\}$	$\{a, b, c\}$	b	(0,1)	(1,2)	4
						c	(1,0)	(1,1)	3

Since there is no deadlock configuration, the components shown in Fig.2 are compatible.

4 Service renaming

The progress of interactions for the model shown in Fig.1 can be illustrated by a "request graph" shown in Fig.3, in which the nodes are configurations of the model (from the column "*conf*" of the table following Fig.1 and the edges are labeled by the services requested/provided by the interacting components in

the form "`x/i`" where "`x`" is the service and "`i`" is the index of the requester component. It should be observed that Fig.3 is a graphical representation of the table following Fig.1 with node 4 representing the deadlock.

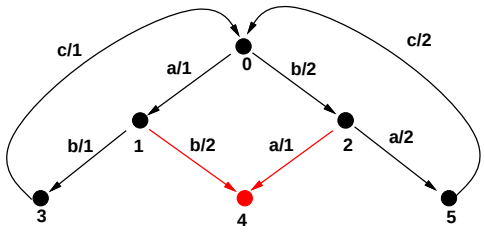

Fig.3. Request graph for Fig.1.

There are two paths leading (from the initial node 0) to the deadlock node; one is "`(a/1,b/2)`" and the second is "`b/2,a/1)`", as discussed in the example in Section 3.1. Moreover, the cycles including nodes 0–1–3–0 and 0–2–5–0 represent the compatibility of single requester components with the provider.

For the renamed services, the steps performed by the function *CheckProgressB* are illustrated in the following table:

$conf$	$\mathbf{m}$	n	i	$Symbols1$	$Symbols2$	x	$next(\mathbf{N}_r[i],\mathbf{m}[i],x)$	$next(\mathcal{N}_p,n,x)$	ℓ
0	[(1,0,0),(1,0,0)]	(1,0,0,0)	1	$\{a\}$	$\{a,b\}$	a	(0,1,0)	(0,1,0,0)	1
			2	$\{b\}$	$\{a,b\}$	b	(0,1,0)	(0,0,1,0)	2
1	[(0,1,0),(1,0,0)]	(0,1,0,0)	1	$\{B\}$	$\{B\}$	B	(0,0,1)	(0,0,0,1)	3
			2	$\{b\}$	$\{B\}$				–
2	[(1,0,0),(0,1,0)]	(0,0,1,0)	1	$\{a\}$	$\{A\}$				–
			2	$\{A\}$	$\{A\}$	A	(0,0,1)	(0,0,0,1)	4
3	[(0,0,1),(1,0,0)]	(0,0,0,1)	1	$\{c\}$	$\{c\}$	c	(1,0,0)	(1,0,0,0)	0
			2	$\{b\}$	$\{c\}$				–
4	[(1,0,0),(0,0,1)]	(0,0,0,1)	1	$\{a\}$	$\{c\}$				–
			2	$\{c\}$	$\{c\}$	c	(1,0,0)	(1,0,0,0)	0

The deadlock node from Fig.3 has been eliminated, so the components – after service renaming – are compatible and can be composed into a deadlock–free system.

It can be observed that there are several other service renamings which result in the same behavior of composed system, for example "`(Abc)*`", "`(Bac)*`" and "`(Ab+Ba)c)*`" as well as "`(ABc)*`", "`(bac)*`" and "`(AB+ba)c)*`", so some other criteria can be taken into account in using service renaming for eliminating conflicting requests.

5 Concluding remarks

In component–based systems, when several requester components are interacting with one or more provider components, the requests from different components can be interleaved and then the properties of the composed system can differ significantly from the properties of components. As shown in Section 3.1, the compatibilities of pairs of interacting components are not sufficient – in general case – for the compatibility of composed system. Consequently, the compatibility of each composition must be verified independently of the compatibility of interacting pairs of components. Straightforward algorithms for such verifications are outlined in Section 3.

In the case of incompatibilities (represented by deadlocks in the composed system), service renaming has been proposed as a systematic approach to eliminating conflicting requests.

Practical service renaming can be performed by connectors [2] or component adaptors [3] [17].

The discussion (in Section 3) was restricted to systems of several requester components interacting with a single provider component because it can be shown that systems with several provider components can be decomposed into several systems, each with one provider component, and analyzed one after another.

It can be observed that the proposed service renaming can be used for restricting request interleaving that may be required for incremental composition [20].

Acknowledgements 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. Bracciali A, Brogi A, Canal C (2005) A formal approach to component adaptations. The Journal of Systems and Software, vol.74, no.1, pp.45-54
4. Broy M (2006) A theory of system interaction: components, interfaces, and services. In: "Interactive Computations: The New Paradigm", Springer-Verlag, pp.41-96
5. 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
6. 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
7. 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

8. 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
9. 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
10. 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
11. Hopcroft J E, Motwani R, Ullman J D (2001) Introduction to automata theory, languages, and computations (2 ed.). Addison–Wesley
12. 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
13. Murata T (1989) Petri nets: properties, analysis, and applications. Proceedings of the IEEE, vol.77, no.4, pp.541-580
14. Nierstrasz O, Meijler T (1995) Research directions on software composition. ACM Computing Surveys, vol.27, no.2, pp.262-264
15. Reisig W (1985) Petri nets – an introduction (EATCS Monographs on Theoretical Computer Science 4). Springer-Verlag
16. Szyperski C (2002) Component software: beyond object-oriented programming (2 ed.). Addison–Wesley Professional
17. 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
18. 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
19. 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
20. 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
21. Zuberek W M, Bluemke I, Craig D C (2010) Modeling and performance analysis of component-based systems. Int. Journal of Critical Computer-Based Systems, vol.1, no.1-3, pp.191-207