Models and Methodology of System Dependability, ed. W. Zamojski et.al., pp.175–184;
5-th Int. Conf. on Dependability of Computer Systems, Brunow, Poland, June 29 - July 1, 2010.

DOI 10.1007/978-83-7493-5260-5.

Checking Compatibility and Substitutability of Software Components

W.M. Zuberek

Department of Computer Science and *Department of Applied Informatics*
Memorial University *University of Life Sciences*
St.John's, Canada A1B 3X5 *02-787 Warsaw, Poland*

Abstract

In component-based systems, two components are compatible if all possible sequences of services requested by one component can be provided by the other component. It has been recently shown that for verification of compatibility, the behavior of interacting components, at their interfaces, can be modeled by labeled Petri nets with labels representing the requested and provided services. Such component models are then composed and the composition operation is designed in such a way that component incompatibilities are manifested as deadlocks in the composed model. Compatibility verification is thus performed through deadlock analysis of the composed models. Component compatibility is also used for the verification of component substitutability; if the new component is compatible with all components that interact with the old component, it can safely replace the old one.

Keywords: software components, component-based systems, component compatibility, compatibility verification, component substitutability, Petri nets.

1. INTRODUCTION

In complex software architectures, the need to assess the compatibility and interoperability of the individual software components is becoming increasingly important during the integration phase of the software production process. 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 compatibility and interoperability assessment is needed. Such a formal approach would permit an assessment based on automated techniques 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 whole system.

Petri nets [15] [14] are formal models of systems which exhibit concurrency or synchronizations of activities. Examples of such systems include multiprocessor systems, distributed databases, manufacturing and transportation systems, and many others. One important aspect of such systems is the existence (or absence) of deadlocks, i.e., a possibility of reaching a state in which no activity can be continued. Absence of deadlocks is critical in systems which are expected to operate in a continuous way, such as life-support systems, supervisory control systems (e.g., in a nuclear plant), transportation control systems, and so on. A systematic and efficient method of deadlock detection is of primary importance for such systems [2].

In component–based systems, two interacting components are 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. Recently, an approach to verification of component compatibility has been proposed in which the behavior of individual components (at component interfaces) was modeled by labeled Petri nets [9]. Moreover, the composition of interacting components was designed in such a way that all component incompatibilities were manifested by deadlocks in the composed model. Consequently, the verification of component compatibility is performed by deadlock analysis of the composed net model.

Several techniques exist to ensure substitutability between components [17]. All these 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 [3]. Various forms of those types exist, starting with the classical interface type [7] and adding behavioral descriptions such as automata [4]. Related research shows that the resulting approach may be too restrictive [17].

This chapter extends the approach proposed earlier for component compatibility to study component substitutability. Two types of substitutability are defined. Strong substitutability guarantees that a replacement of a compatible component is also compatible with its environment. Contextual substitutability requires the compatibility verification of the new component in the original setting.

Section 2 introduces Petri net models of component behavior. These models are used in Section 3 to discuss component compatibility and in Section 4 to characterize component substitutability. Section 5 concludes the chapter.

2. PETRI NET MODELS OF COMPONENT BEHAVIOR

The behavior of a component, at its interface, can be represented by a cyclic labeled Petri net [8] [9]:

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

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).

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 must 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 $Out(p) = \{tT \mid (p,t) \in A\}$; 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 $\ell(t_{i_1} t_{i_2} ... t_{i_k}) = \ell(t_{i_1})\ell(t_{i_2})...\ell(t_{i_k})$.

Interface languages defined by Petri nets include regular languages, some context–free and even context–sensitive languages [14]. 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 $\mathcal{M}_i$ is compatible with a provider component $\mathcal{M}_j$ if and only if all sequences of services requested by $\mathcal{M}_i$ can be provided by $\mathcal{M}_j$, i.e., if and only if:

$$\mathcal{L}(\mathcal{M}_i) \subseteq \mathcal{L}(\mathcal{M}_j).$$

If the languages of interacting requester and provider components are regular, checking the compatibility is relatively straightforward because the compatibility relation can be expressed as:

$$\mathcal{L}(\mathcal{M}_i) \cap \overline{\mathcal{L}(\mathcal{M}_j)} \ = \ \emptyset,$$

where $\emptyset$ denotes the empty set, and $\overline{\mathbf{A}}$ is the complement of the set $\mathbf{A}$. Since the class of regular languages is closed under the operations of complementation and set intersection, compatibility can be verified by performing the corresponding operations on finite automata representing the requester and provider languages [12]. Since the interface automata can be quite large, the product operation should be performed in a way that eliminates all inessential pairs of states (i.e., pairs of states which cannot be reached from the initial state). This can easily be done as a straightforward modification of the "standard" product operation.

For compatibility verification of components with non-regular behavior, the direct verification of the compatibility relation cannot be used because the class of non-regular languages is not closed under complementation. In such cases the compatibility of interacting components can be verified by composing the component models into one model and checking the properties of this model. The composition, however, can be performed in several ways, resulting in models with different properties.

The COSY–style composition [13] uses the fusion of transitions labeled by the same services (with some additional elements to distinguish repeated requests of the same service). The consequence of such an approach is that the composition corresponds to the intersection of languages of the provider and requester interfaces:

$$\mathcal{L}(\mathcal{M}_i) \cap \mathcal{L}(\mathcal{M}_j).$$

Compatibility is thus verified by checking the equality:

$$\mathcal{L}(\mathcal{M}_i) \cap \mathcal{L}(\mathcal{M}_j) \;=\; \mathcal{L}(\mathcal{M}_i)$$

which is as difficult as the verification of the original compatibility relation.

The idea behind the CORD (compatible or deadlocked) composition [8] is to make the language of composed interfaces equal to the language of the requester, or to create a deadlock if the requested sequence of services cannot be provided by the other component. The verification of component compatibility is thus equivalent to deadlock detection in the composed model.

Known methods of deadlock detection in Petri net models include reachability analysis which systematically explores all possible states that can be reached from the initial state(s), looking for deadlock states, and structural analysis which analyzes the structure of Petri net models to predict deadlock existence (or absence). Reachability analysis is quite straightforward, but can be used only for models with finite and reasonably small state spaces [14]. Structural analysis uses *siphons* for deadlock detection [6] [11] [16].

Siphon–based analysis does not depend upon the number of reachable markings and can be used for deadlock detection in nets with infinite state spaces, but it can easily become quite inefficient if the number of siphons is large (for some net models, the number of siphons grows exponentially with the net size). It appears, however, that instead of analyzing all siphons, only a small set of minimal siphons provides the same information about the absence (or existence) of deadlocks. Moreover, the large number of siphons can be significantly reduced by simple reductions of net models which do not affect the existence (or absence) of deadlocks, making the siphon-based deadlock detection quite attractive from a practical point of view [10].

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.

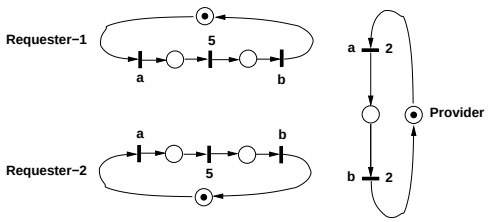

Fig.1. Two requesters and a single provider.

The composed model is shown in Fig.2. Since the composed net is free from deadlocks, the components are compatible. Indeed, it can be observed that the provider as well as both requester languages in this particular case can be described by a regular expression `(ab)*`. The provider simply performs the requested sequence of services on the "first come first served" basis or randomly choses the requester if the two requests for "`a`" occur simultaneously.

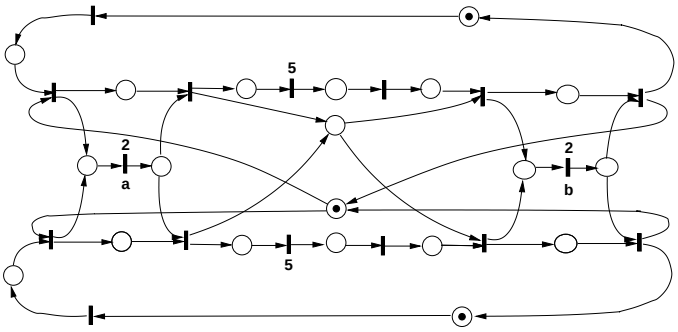

Fig.2. Composition of two requesters and a single provider.

4. COMPONENT SUBSTITUTABILITY

Component substitutability is usually defined as the possibility of replacing a component $\mathcal{C}_{old}$ of a system by another component $\mathcal{C}_{new}$ without disrupting the operation of the system [1].

Substitutability of a **provider component** depends upon the relationship between the language of the original component, $\mathcal{L}^{(r)}_{old}$ and the language of the new component, $\mathcal{L}^{(r)}_{new}$. If

$$\mathcal{L}^{(p)}_{old} \subseteq \mathcal{L}^{(p)}_{new}$$

the new component is substitutable for the old one, and can replace it without any adverse effect on the whole system. Sometimes such a substitutability is called strict substitutability [1].

On the ther hand, if

$$\mathcal{L}^{(p)}_{new} \subset \mathcal{L}^{(p)}_{old}$$

the compatibility of the new component must be verified with the set of all interacting requester components. Sometimes this is called contextual substitutability [1].

For **requester components** the relations are different. If

$$\mathcal{L}^{(r)}_{new} \subseteq \mathcal{L}^{(r)}_{old}$$

the new component is substitutable for the old one, and it can replace it without any adverse effect on the whole system.

If, however,

$$\mathcal{L}^{(r)}_{old} \subset \mathcal{L}^{(r)}_{new}$$

the compatibility of the new component must be verified with the set of all interacting requester components.

Example. Fig.3 shows a model of a component which still requires that each operation `b` is preceded by an operation `a`, but which also allows the operations `a` of several requesters to be performed before any of the corresponding `b` operations (which is not allowed in model shown in Fig.1). Consequently, the language of the provider in Fig.3 is a superset of the provider's language in Fig.1, so the provider in Fig.3 is substitutable for the one in Fig.1.

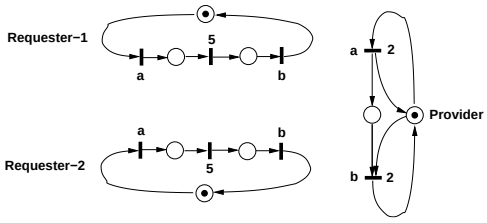

Fig.3. Two requesters with a modified provider.

The composed system is shown in Fig.4; it can be checked that the net in Fig.4 is free from deadlocks.

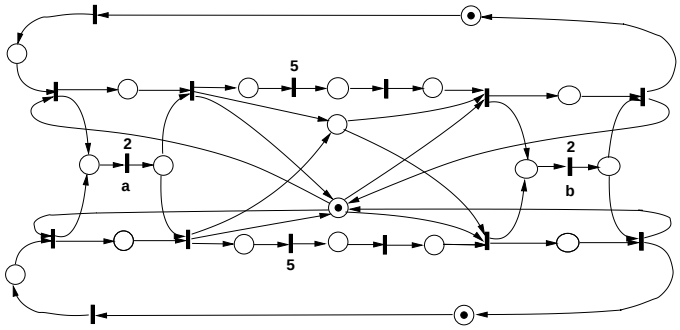

Fig.4. Composition of two requesters and a modified provider.

Fig.5 shows another (simple) configuration of two requesters and a single provider in which requester-2 is modified in a way which removes the requirement that any request of service `b` is always preceded by a request of service `a` - the language of the modified requester is `(a|b)*`.

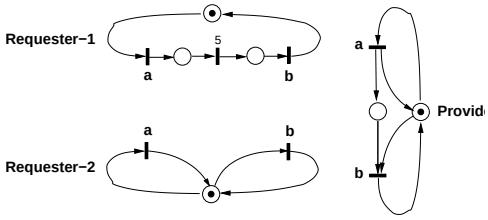

Fig.5. Modified requester-2 as well as provider.

In this case, the composed net is not free from deadlocks, as shown in Fig.6.

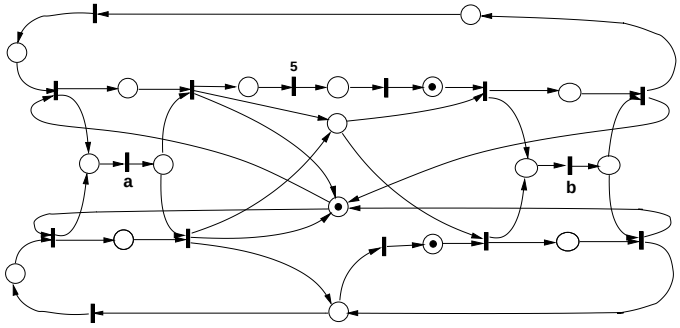

Fig.6. Composed net for modified requester and provider;
the deadlock is indicated by the marking function.

5. SUBSTITUTABILITY CHECKING

It is known that the languages defined by Petri nets include regular language, some context-free languages and even some context-sensitive languages [14]. Therefore checking the inclusion relation between two languages defined be Petri nets may not be a straightforward task.

The following logical function *CheckBounded* can be used if the compared languages are nonregular, but are defined by bounded Petri nets $\mathcal{N}_1$ and $\mathcal{N}_2$ (m_1 and m_2 are the initial marking functions of the two nets, 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 a set of markings that have been analyzed, $Enabled(\mathcal{N}, m)$ returns the set of labels of transitions enabled in the net $\mathcal{N}$ with the marking m, $next(\mathcal{N}, m, a)$ returns the marking obtained in the net $\mathcal{N}$ from the marking x by firing the transition labeled by x):

proc $CheckBounded(\mathcal{N}_1, m_1, \mathcal{N}_2, m_2)$;
begin
  $New := (m_1, m_2)$;
  $Analyzed := \{\}$;
  while $New \neq \{\}$ **do**
    $(m, n) := head(New)$;
    $New := tail(New)$;
    if $m \notin Analyzed$ **then**
      $Analyzed := Analyzed \cup \{m\}$;
      $Symbols1 := Enabled(\mathcal{N}_1, m)$;
      $Symbols2 := Enabled(\mathcal{N}_2, n)$;
      for each x **in** $Symbols1$ **do**
        if $x \in Symbols2$ **then**
          $append(New, (next(\mathcal{N}_1, m, x), next(\mathcal{N}_2, n, x))$
        else
          return FALSE
        fi
      od
    fi
  od;
  return TRUE
end;

The function returns TRUE if the language of $\mathcal{N}_1$ is a subset of the language defined by $\mathcal{N}_2$; otherwise FALSE is returned.

For the provider nets shown in Fig.1 and Fig.3, the steps performed by the function *Check* can be illustrated in the following table:

m	n	$Symbols1$	$Symbols2$	x	$next(\mathcal{N}_1, m, x)$	$next(\mathcal{N}_2, n, x)$
(1,0)	(1,0)	$\{a\}$	$\{a\}$	a	(0,1)	(1,1)
(0.1)	(1,1)	$\{b\}$	$\{a, b\}$	b	(1,0)	(1,0)

If the net $\mathcal{N}_1$ is unbounded, the procedure is a bit more complex the detection of unboundedness needs to be built into it. For each analyzed marking m, an additional check is performed if the set *Analyzed* contains a marking, which is componentwise smaller than m and from which m is reachable; if the set *Analyzed* contains such a marking, analysis of m is discontinued because it is one of an infinite sequence of reachable markings:

```
proc CheckUnbounded(N1, m1, N2, m2);
begin
    New := (m1, m2);
    Analyzed := {};
    while New ≠ {} do
        (m, n) := head(New);
        New := tail(New);
        if m ∉ Analyzed then
            cont := true;
            for each r in Analyzed do
                if reachable(r, m) ∧ m ≥ r then cont := false fi
            od;
            if cont then
                Analyzed := Analyzed ∪ {m};
                Symbols1 := Enabled(N1, m);
                Symbols2 := Enabled(N2, n);
                for each x in Symbols1 do
                    if x ∈ Symbols2 then
                        append(New, (next(N1, m, x), next(N2, n, x))
                    else
                        return FALSE
                    fi
                od
            fi
        fi
    od;
    return TRUE
end;
```

6. CONCLUDING REMARKS

Component compatibility and substitutability are closely related; a component $\mathcal{C}_{new}$ is substitutable for a component $\mathcal{C}_{old}$ only if it is compatible with all components that interact with $\mathcal{C}_{old}$. Sometimes a strict substitutability relation can be established between $\mathcal{C}_{new}$ and $\mathcal{C}_{old}$ and then $\mathcal{C}_{new}$ remains compatible with any environment in which $\mathcal{C}_{old}$ is compatible. If strict substitutability cannot be established, the compatibility of $\mathcal{C}_{new}$ must be verified in any specific configuration of components interacting with $\mathcal{C}_{old.}$

Strict substitutability is based on the relation between languages of the two components. Sometimes this relation can be derived from the extensions or modifications performed during the development of $\mathcal{C}_{new}$. If this is not the case, component meta-representations can be used for component comparisons.

Although open questions remain (i.e., how to get interface models of components), it is believed that the linguistic characterization of interacting components constitutes a simple, intuitive and interesting approach to analysis of component-based systems.

Acknowledgement

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

References

[1] BELGUIDOUM M., DAGNAT F., *Formalization of component substitutability*; Electronic Notes in Theoretical Computer Science, vol.215, pp.75-92, 2008.

[2] BORDBAR B., OKANO K., *Testing deadlock-freeness in real-time systems: a formal approach*; Formal Approaches to Software Testing (Lecture Notes in Computer Science 3395) pp.95-109, 2004.

[3] BRADA P., VALENTA L., *Practical verification of component substitutability using subtype relation*; Proc. Int. Conf. on Software Engineering and Advanced Applications (SEAA'06), pp.38-45, 2006.

[4] CERNA I., VAREKOVA P., ZIMMEROVA B., *Component substitutability via equivalencies of component-interaction automata*; Proc. Int. Workshop on Formal Aspects of Component Software (FACS'06), pp.115-130, 2006.

[5] CHAKI S., CLARKE S.M., GROCE A., JHA S., VEITH H., *Modular verification of software components in C*; IEEE Trans. on Software Engineering, vol.30, no.6, pp.388-402, 2004.

[6] CHU F., XIE X., *Deadlock analysis of Petri nets using siphons and mathematical programming*; IEEE Trans. on Robotics and Automation, vol.13, no.6, pp.793-804, 1997.

[7] COSTA SECO J. , CAIRES L., *A basic model of typed components*; Proc. 14-th European Conf. on Object-Oriented Programming, London, UK, pp.108-128, 2000.

[8] CRAIG D.C., *Compatibility of software components – modeling and verification*; Ph.D. Thesis, Department of Computer Science, Memorial University, St.John's, Canada A1B 3X5, 2006.

[9] CRAIG D.C., ZUBEREK W.M., *Compatibility of software components – modeling and verification*; Proc. Int. Conf. on Dependability of Computer Systems, Szklarska Poreba, Poland, pp.11-18, 2006.

[10] CRAIG D.C., ZUBEREK W.M., *Petri nets in modeling component behavior and verifying component compatibility*; Proc. Int. Workshop on Petri Nets and Software Engineering, Siedlce, Poland, pp.160-174, 2007.

[11] EZPELETA J., COLOMBO J.M., MARTINEZ J., *A Petri net based deadlock prevention policy for flexible manufacturing systems*; IEEE Trans. on Robotics and Automation, vol.11, no.2, pp.173-184, 1995.

[12] HOPCROFT J.E., MOTWANI R., ULLMAN J.D., *Introduction to automata theory, languages, and computations* (2 ed.); Addison-Wesley 2001.

[13] JANICKI R., LAUER P.E., *Specification and analysis of concurrent systems – the COSY approach*; Springer-Verlag 1992.

[14] MURATA T., *Petri nets: properties, analysis, and applications*; Proceedings of the IEEE, vol.77, no.4, pp.541-580, 1989.

[15] REISIG W., *Petri nets – an introduction* (EATCS Monographs on Theoretical Computer Science 4); Springer-Verlag 1985.

[16] SILVA M., TERUEL E., COUVREUR J., *Linear algebra in and linear programming techniques for the analysis of place/transition net systems*; Lectures on Petri nets – basic models (Lecture Notes in Computer Science 1491), pp.309-373, Springer-Verlag 1998.

[17] ZAREMSKI A.M., WANG J.M., *Specification matching of software components*; ACM Trans. on Software Engineering and Methodology, vol.6, no.4, pp.333-369, 1997.