DepCoS-RELCOMEX 2014 - Proceedings of the 9-th International Conference (Advances in Intelligent and Soft Computing 286), Brunow Palace, Poland, June 30 – July 4, 2014, pp.521-529.

DOI 10.1007/978-3-319-07013-1_50.

Model fusion for compatibility verification of software components

W.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, defined by labeled Petri nets. In the case of composition of concurrent components, the requests from different components can be interleaved, and - as shown earlier - such interleaving can result in deadlocks in the composed system even if each pair of interacting components is deadlock–free. Therefore the elements of a component–based system are considered compatible only if the composition is deadlock–free. This paper formally defines model fusion, which is a composition of net models of individual components that represents the interleaving of interface languages of interacting components. It also shows that the verification of component compatibility can avoid the exhaustive analysis of the composed state space.
Keywords: software components, component-based systems, component composition, component compatibility, compatibility verification, model fusion, labelled Petri nets.

1 Introduction

In component-based software development the functionality of a software system is decomposed among loosely coupled independent software components. Such a reuse-oriented approach to defining and implementing software systems has recently been extensively studied as it is believed to be a starting platform for service orientation [4], [10].

In component-based systems [10], two interacting components, one requesting services and the other providing them, are considered compatible if all possible sequences of services needed by the requesting 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 concurrent requester components, as is typically the case for 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, i.e. a deadlock can occur indicating component incompatibility [21], [22].

The behavior of components is usually described at component interfaces [17] and the components are characterized as requester (active) and provider (reactive) components. Although several approaches to checking component composability have been proposed [1], [2], [4], [11], [13], [18], further research is needed to make these ideas practical [9].

The paper is an extension of previous work on component compatibility and substitutability [8], [20], [21], [22]. Using the same formal specification of component behavior in the form of interface languages, the paper addresses the verification of component compatibility. Since interface languages are usually infinite, their compact finite specification is needed for effective processing. Labeled Petri nets [20], [21] are used as such specification.

Petri nets [14], [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 control component interactions. Well-developed mathematical theory of Petri nets provides a convenient formal foundation for analysis of systems modeled by Petri nets.

Model fusion is proposed to represent the interactions of components. The fusion operation is performed by merging Petri net transitions that request and provide the same service. Interleavings of requests from concurrent components result in different execution paths in the combined model. The components are compatible, if the combined model is deadlock–free. Deadlock–freeness can be verified by structural methods, avoiding the exhaustive state space analysis because model fusion preserves structural (some) properties of the fused elements. If the interacting components are not compatible, some correcting steps are required (in the form of redesign of some components or additional constraints which prevent some interleavings to occur).

Section 2 recalls the concept of interface languages as the description of component's behavior and Section 3 presents a linguistic version of component compatibility. Section 4 illustrates the proposed concepts by a simple example and shows that compatibility verification for some classes of systems can be performed using structural analysis of the fused model. Section 5 concludes the paper.

2 Component Behavior

The behavior of a component, at its interface, can be represented by a cyclic labeled Petri net [7], [8], [21]:

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

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 [12]. 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 are used to define the compatibility of components. 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, "r_i", $i \in I$, interacting with a single provider component "p", the component compatibility requires that all sequences of (interleaved) requests be satisfied by the provider, so:

$$\mathcal{L}_I \subseteq \mathcal{L}_p$$

where $\mathcal{L}_I$ is the language of interleavings of requester languages $\mathcal{L}_i, i \in I$. It should be observed that $\mathcal{L}_I$ does not necessarily contain all possible interleavings of requests because some requests cannot be satisfied immediately upon request and are delayed until some other operations and/or their sequences are performed by the provider component. All such restrictions are represented by the fused component models.

4 Model Fusion

Model fusion is used to combine separate models of interacting components into one model which represents the possible behaviors of the interacting components. The general idea is sketched in Fig.1 where a single transition representing service "a" in the provider component is combined with requests of this service in two requester components - the provider transition is conceptually "fused" (or merged) with corresponding transitions of the requester components.

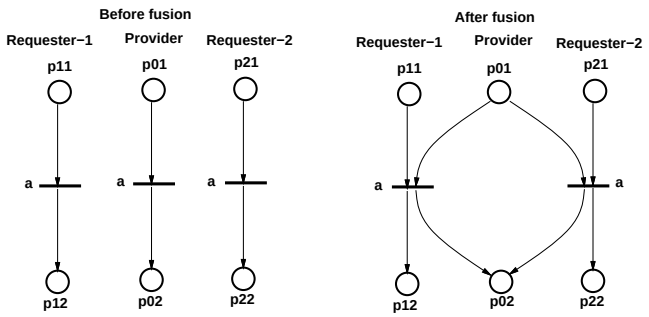

Fig.1. Model fusion for service "a".

More formally, if the provider component is modeled by

$$\mathcal{M}_p \;=\; (P_p, T_p, A_p, S, m_p, \ell_p, F_p),$$

and the requester components are modeled by

$$\mathcal{M}_i \;=\; (P_i, T_i, A_i, S, m_i, \ell_i, F_i), \quad i \in I,$$

where S is a common set of services and all other sets (of places, transitions, etc.) are disjoint, then the combined (or "fused") model is

$$\mathcal{M}_c \;=\; (P_c, T_c, A_c, S, m_c, \ell_c, F_c),$$

where:

$$P_c = P_p \cup P_I, \quad P_I = \bigcup_{i\in I} P_i;$$

$$T_c = T_p - T_0 \cup T_I. \;\; T_0 = \{t \in T_p \mid \ell_p(t) \in S\}, \quad T_I = \bigcup_{i\in I} T_i;$$

$$A_c = A_p - A_0 \cup A_I \cup A_{pr}, \quad A_0 = P_c \times T_0 \cup T_0 \times P_c, \quad A_I = \bigcup_{i\in I} A_i,$$

$$A_{pr} = \{(p_x, t_{ik}) \mid p_x \in P_p \wedge (p_x, t) \in A_0 \wedge \ell_i(t_{ik}) = \ell_p(t)\} \cup$$

$$\{(t_{ik}, p_y) \mid p_y \in P_p \wedge (t, p_y) \in A_0 \wedge \ell_i(t_{ik}) = \ell_p(t)\};$$

$$\forall p \in P_c : m(p) = \begin{cases} m_p(p), & \text{if } p \in P_p, \\ m_i(p), & \text{if } p \in P_i, i \in I; \end{cases}$$

$$\forall t \in T_c : \ell(t) = \begin{cases} \ell_p(t), & \text{if } t \in T_p. \\ \ell_i(t), & \text{if } t \in T_i, i \in I; \end{cases}$$

$$F_c = F_p \cup F_I, \quad F_I = \bigcup_{i\in I} F_i;$$

and $Inp(t)$ and $Out(t)$ are, respectively, the input and the output sets places of transition t. The composed model is obtained by deleting all labeled transitions in the provider model (the set T_0) with all arcs connected to these transitions (the set A_0), and adding arcs similar to the deleted ones but connecting places of the provider model with labeled transitions of all requester models (the set A_{pr}).

The composed net can be analyzed by typical methods used for analysis of Petri net models. In particular, for some classes of models, structural analysis can be used for verification of deadlock freeness [19], [16] avoiding the exhaustive state–based model analysis.

It can be observed that the fusion operation preserves the place invariants [15] of the model, as well as their siphons. More specifically, if a collection of component models is covered by a set of place invariants, than the same set of invariants covers the fused model. Moreover, if all place invariants are marked, than no deadlock can occur in the fused model, so the components are compatible. The following example illustrates this property in greater detail.

5 Example

A simple system of two requesters and a single provider is shown in Fig.2. The interface language of Requester-1 is simply `(ab)*`, the language of Provider describes the behavior of (unbounded) stack with services "`a`" and "`b`" corresponding to operations "push" and "pull', and the language of Requester-2 is that of bounded stack of capacity 2. Both stacks (i.e., Provider and Requester-2) in Fig.2 are empty. Obviously, the languages of Requester-1 and Requester-2 are (proper) subsets of the language of Provider.

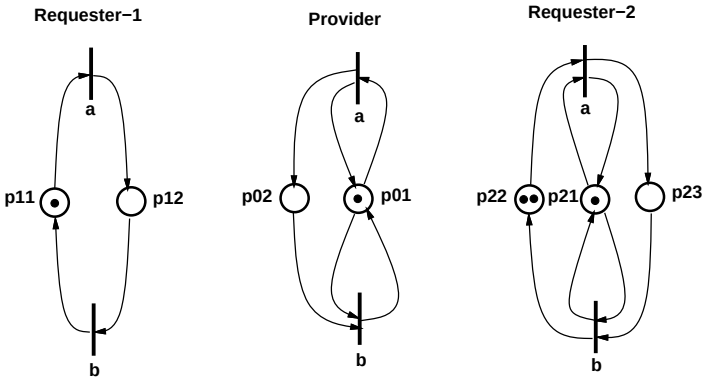

Fig.2. Net models of two requesters and a single provider.

Fig.3 shows the combined (or fused) model of the system from Fig.2. Structural analysis can be used to check its deadlock freeness (i.e., compatibility of components).

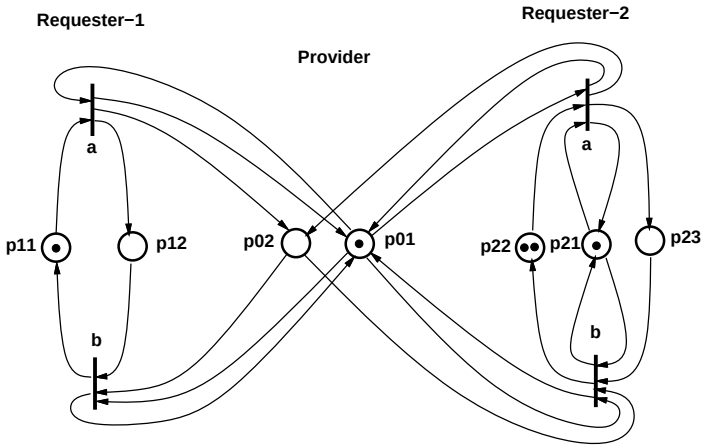

Fig.3. A combined model of two requesters and a single provider.

It is known that a deadlock in a Petri net corresponds to an unmarked siphon [6] (a siphon is a subset of places for which the set of output transitions is a superset of the set of input transitions). Consequently, a deadlock freeness can be verified by checking that the siphons cannot become unmarked (linear programming can be used to for such checking [19]). Moreover, since all siphons are composed of a rather small number of basis siphons [3], verification can be restricted to basis siphons only.

The net shown in Fig.3 has three such siphons with the following subsets of places:

siphon	*places*
1	p_{11}, p_{12}
2	p_{11}, p_{01}, p_{21}
3	$p_{22}.p_{23}$

Since all these siphons are actually marked place invariants, they cannot become unmarked (place invariants preserve the markings). Consequently, the deadlock cannot occur in the net shown in Fig.3, so the interacting components are compatible.

It can be observed that the original models of components are covered by place invariants (in the example shown in Fig.2, the model of Requester-1 is a single place invariant, the model of Provider is covered by two place invariants, and the model of Requester-2 is also covered by two place invariants). The combined model is covered by the same place invariants because the fusion process preserves the place invariants. Efficient methods of compatibility verification can use such properties for simplifying the verification process.

Morel general structural approach to deadlock analysis is discussed in [19].

6 Concluding Remarks

The paper shows that the verification of component compatibility based on the exhaustive analysis of the "state space", as discussed in [20] and [21], can be replaced by structural analysis of the model obtained by "fusion" of models of interacting components.

It is expected that the proposed verification of component compatibility can be quite efficient although this aspect needs to be studied in greater detail.

It should be noticed that the discussion of component compatibility 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.

Similarly, the requester and provider components can use other components in a sort of hierarchical structure. Since model fusion combines all component models into a single net model, there are no restrictions on such structures.

Also, an important aspect of component compatibility is its incremental verification. The approach described in this paper is not incremental (with the exception of some cases, for example, when all models of all components are

covered by place invariants), 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.: Checking component composability. Proc. 5-th Int. Symp. on Software Composition (LNCS 4089), pp.18-33 (2006)
2. Baier, C., Klein, J., Klueppenholz, S.: Modeling and verification of components and connectors. In: "Formal Methods for Eternal Networked Software Systems" (LNCS 6659), pp.114-147 (2001)
3. Boer, E.T., Murata, T.: Generating basis siphons and traps of Petri nets using the sign incidence matrix, IEEE Trans. on Circuits and Systems, I – Fundamental Theory and Applications, vol.41, no.4, pp.266-271 (1994)
4. Broy. M.: A theory of system interaction: components, interfaces, and services. In: "Interactive Computations: The New Paradigm", Springer-Verlag, pp.41-96 (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. 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)
8. 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)
9. Crnkovic, I., Schmidt, H. W., Stafford, J., Wallnau, K.: Automated component-based software engineering. The Journal of Systems and Software, vol.74, no.1, pp.1-3 (2005)
10. Garlan, D.: 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 (2003)

11. Henrio, L., Kammueller, F., Khan, M. U.: A framework for reasoning on component composition. Proc. 8-th Int. Symp. on Formal Methods for Components and Objects (LNCS 6286), pp.41-69 (2009)
12. Hopcroft, J, E,, Motwani, R., Ullman, J. D.: Introduction to automata theory, languages, and computations (2 ed.). Addison–Wesley (2001)
13. Leicher, A., Busse, S., Suess, J. G.: Analysis of compositional conflicts in component-based systems. Proc. 4-th Int. Workshop on Software Composition; Edinburgh, UK (LNCS 3628), pp.67-82 (2003)
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 (1995)
16. Reisig, W.: Understanding Petri nets – modeling techniques, analysis methods, case studies, Springer-Verlag (2013)
17. Szyperski, C.: Component software: beyond object-oriented programming (2 ed.). Addison–Wesley Professional (2002)
18. 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)
19. Zuberek, W. M.: Siphon-based verification of component compatibility, Proc. 4-th Int. Conference on Dependability of Computer Systems (DepCoS-09); Brunow Palace, Poland, pp.123-132 (2009)
20. Zuberek, W, M.: Checking compatibility and substitutability of software components. In: "Models and Methodology of System Dependability", Oficyna Wydawnicza Politechniki Wroclawskiej, ch.14, pp.175-186 (2010)
21. Zuberek, W. M.: Incremental composition of software components. In: "Dependable Computer Systems" (Advances in Intelligent and Soft Computing 97), Springer-Verlag, pp.301-311 (2011)
22. Zuberek, W. M.: Service renaming in component composition. In: "Complex Systems and Dependability" (Advances in Intelligent and Soft Computing 170); Springer-Verlag, pp.319-330 (2012)