

# Compatibility and performance analyses for component-based systems

W.M. Zuberek\*

## Summary

As software engineering continues to adopt a component-based approach toward the construction of increasingly complex software architectures, the need to assess the compatibility and interoperability of the individual software components is becoming critical during the integration phase of the software production process [5], [9]. This assessment includes performance analysis of integrated systems and verification of temporal requirements which can be of primary importance for many real-time and embedded systems. 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.

Components represent high-level software models; they must be generic enough to work in a variety of contexts and in cooperation with other components, but they also must be specific enough to provide easy reuse. Although there are many (informal) component definitions [2], [7], very few attempts have been made to formalize these concepts. One aspect which many component definitions have in common is the notion of an interface that defines the component's access points [8]. These access points allow other components to use the services provided by a component. Normally, a component can have multiple interfaces corresponding to its different access points.

The behavior of a component, at its interface, can be represented by a (cyclic) labeled timed Petri net [4], [10]:

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

where  $P_i$  and  $T_i$  are disjoint sets of places and transitions, respectively,  $A_i$  is the set of directed arcs,  $A_i \subseteq P_i \times T_i \cup T_i \times P_i$ ,  $m_i$  is the initial marking function  $m_i : P_i \rightarrow \{0, 1, \dots\}$ ,  $c_i$  is a choice function which assigns probabilities to free-choice classes of transitions and relative occurrence frequencies to conflicting transitions,  $f$  is a timing function which assigns an (average) occurrence time to each transition of the net,  $f : T \rightarrow \mathbf{R}^+$ , where  $\mathbf{R}^+$  is the set of nonnegative real numbers,  $S_i$  is an alphabet representing the set of services that are associated with transitions by the labeling function  $\ell_i : T_i \rightarrow S_i \cup \{\varepsilon\}$  ( $\varepsilon$  is the "empty"

---

\*Department of Computer Science, Memorial University, St. John's, NL, Canada A1B-3X5, and Department of Applied Informatics, University of Life Sciences, 02-787 Warsaw, Poland, [wlodek@mun.ca](mailto:wlodek@mun.ca)

service; it labels transitions which do not represent services), and  $F_i$  is a set of final markings (which are used to indicate the end of firing sequences).

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 reflect their reactive nature, all provider models must be  $\varepsilon$ -conflict-free, *i.e.*:

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

where  $\text{Inp}(t)$  and  $\text{Out}(p)$  are the sets of input places of  $t$  and output transitions of  $p$ , respectively (this condition could be used in a more relaxed form which is not elaborated 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  $\mathcal{F}(\mathcal{N})$  denote the set of firing sequences in  $\mathcal{N}$  such that the marking created by each firing sequence belongs to the set of final markings  $F$  of  $\mathcal{N}$ . The interface language of a component represented by a labeled Petri net  $\mathcal{N}$ ,  $\mathcal{L}(\mathcal{N})$ , is the set of all labeled firing sequences of  $\mathcal{N}$ :

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

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

Interface languages defined by Petri nets include regular languages, some context-free and even context-sensitive languages [6]. Therefore, they are significantly more general than languages defined by finite automata [3], but their compatibility verification is also more difficult than in the case of regular languages.

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

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

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.

Component composition proposed in [4] is defined in such a way that the incompatibility of components results in deadlocks in the composed model. Verification of component compatibility is thus performed by checking the existence of deadlocks in the composed model. For small and bounded nets this can be done by using reachability analysis; for unbounded nets and nets with large marking spaces an efficient approach was proposed which is based on reduced sets of minimal and basis siphons because a small set of minimal siphons provides the same information about the absence (or existence) of deadlocks as the original set of siphons. Moreover, the 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 [4].

Temporal characteristics associated with interface models can be used to study the performance aspects of modeled systems and to verify their time-critical behavior. As for compatibility checking, interacting components are composed into a single integrated model and then reachability analysis, structural methods and even simulation techniques can be used for performance analysis of the model. Such an approach is sufficiently flexible to allow multiple “client-like” and “server-like” components to be combined in a variety of ways to achieve different specification goals.

Component compatibility is closely related to component substitutability, which 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,  $\mathcal{L}_{old}^{(p)}$  and the language of the new component,  $\mathcal{L}_{new}^{(p)}$ . If

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

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

On the other hand, if

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

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

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

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

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

If, however,

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

the compatibility of the new component must be verified as in the case of contextual substitutability of a provider component.

Prediction of performance properties of a system after substitution of one or more components usually requires the composition of all components and performance analysis of composed systems. Only in some cases structural properties allow to predict the system's performance without the integration of its components.

## References

- [1] M. Belguidoum, F. Dagnat, "Formalization of component substitutability"; *Electronic Notes in Theoretical Computer Science*, vol.215, pp.75-92, 2008.
- [2] A.W. Brown, "An overview of components and component-based development"; in "Advances in Computers", vol.54 – Trends in Software Engineering, pp.1-34, Academic Press 2001.
- [3] S. Chaki, E.M. Clarke, A. Groce, S. Jha, H. Veith, "Modular verification of software components in C"; *IEEE Trans. on Software Engineering*, vol.30, no.6, pp.388-402, 2004.
- [4] D.C. Craig, W.M. Zuberek, "Verification of component behavioral compatibility"; Proc. Second Int. Conf. on Dependability of Computer Systems, Szklarska Poreba, Poland, pp.294-301, 2007.
- [5] B. Meyer, "The grand challenge of trusted components"; Proc. 25-th Int. Conf. on Software Engineering, Portland, OR, pp.660-667, 2003.
- [6] T. Murata, "Petri nets: properties, analysis and applications"; *Proceedings of IEEE*, vol.77, no.4, pp.541-580, 1989.
- [7] C. Szyperski, "Component software and the way ahead"; in "Foundations of component-based systems", G.T. Leavens, M. Sitaraman (eds.), pp.1-20, Cambridge University Press 2000.
- [8] C. Szyperski (with D. Gruntz, S. Murer), "Component software: beyond object-oriented programming" (2 ed.); Addison-Wesley 2002.
- [9] C. Szyperski, "Component technology – what, where, and how?"; Proc. 25-th Int. Conf. on Software Engineering, Portland, OR, pp.684-693, 2003.
- [10] W.M. Zuberek, I. Bluemke, "Performance analysis of component-based systems"; Proc. Third Int. Conf. on Dependability of Computer Systems, Szklarska Poreba, Poland, pp.293-300, 2008.