7-th Int. Conference on Application of Concurrency to System Design, Bratislava, Slovakia, July 10–13, 2007

Multicomponent Compatibility and its Verification

(Extended abstract)

Donald C. Craig and Wlodek M. Zuberek
Department of Computer Science
Memorial University
St.John's, Canada A1B 3X5
{donald, wlodek}@cs.mun.ca

Software architecture has been introduced with promise of better re-use of software, greater flexibility, scalability and higher quality of software services [1] [10]. Software architecture uses components as the basic building blocks of software systems.

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 [12]. To be composable with other (third-party) components, a component needs to be sufficiently self-contained. Also, it needs a clear specification of what it requires and what it provides. In other words, a component needs to encapsulate its implementation and interact with its environment by means of well-defined interfaces.

The interface of a component defines the component's access points [12]; these points allow clients of a component to access the services provided by the component. Each access point may provide a different service, catering to different client needs. In order to represent component interactions, the interfaces are divided into *provider* interfaces and *requester* interface, or simply providers and requesters.

Two interacting components are compatible if all services that are requested by one component are provided by the other component. Such "static" compatibility can usually be checked quite easily, but it does not prevent more subtle errors which are due to some limitations on the ordering of services. Therefore a "dynamic" (or behavioral) compatibility is used, and two components are compatible in the behavioral sense if all possible sequences of services requested by one of the interacting components can be provided by the other component. The collections of sequences of services can be regarded as interface languages (over the alphabet of requested and provided services), and then a requester component $\mathcal{C}_i$ is compatible with a provider component $\mathcal{C}_j$ if

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

If the interface languages are regular (in the sense of Chomsky [7]), the behavior of components can be represented by finite automata [2], [6], and the compatibility relation can be verified easily by operations on such automata [7]. In many cases, however, the interface languages are non-regular; for example, if a component implements a stack, the language of PUSH and POP stack operations is context-free. Models which are more expressive than finite automata are needed for representing non-regular interface languages. In [4], [5], labeled Petri nets (with labels assigned to transitions) have been proposed as behavioral models of components at their interfaces:

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

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 indicate the end of sequences of firings). It is required that in each net representing a provider interface there is exactly one labeled transition for each provided service so as to eliminate ambiguity of requested services.

Let $\mathcal{F}(\mathcal{M})$ denote the set of such firing sequences in $\mathcal{M}$ which create one of the final markings in the set F. The interface language of a component represented by a labeled Petri net $\mathcal{M}$, $\mathcal{L}(\mathcal{M})$, is the set of all labeled firing sequences in $\mathcal{F}(\mathcal{M})$:

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

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 all regular languages as well as some context-free languages and even context-sensitive languages [9].

For the case of non-regular languages, 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 [8] 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 of a requester modeled by $\mathcal{M}_i$ with a provider modeled by $\mathcal{M}_j$ corresponds to the intersection of their languages:

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

The verification of the compatibility is thus checking if the intersection is equal to the requester language:

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

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

The idea behind the CORD (compatible or deadlocked) composition [4] is to make the language of composed interfaces equal to the language of the requester, or to create a deadlock when the requested sequence of services cannot be provided by the other component. The verification of the component compatibility is thus equivalent to deadlock detection in the composed model which, at least for some models, can be done quite efficiently.

The case of a single requester interacting with a single provider is discussed in [5]. For multiple providers, the consistency of each provider (with the same requester(s)) is a sufficient condition for the consistency of the whole multiprovider configuration. For multiple requesters, the composition must take into account the concurrent interactions of different requesters with the provider. For a single service (named "a"), the composition of two requesters with a single provider is outlined in Fig.1 and Fig.2.

In general, the composition of a family of requesters $\mathcal{M}_I = \{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_k\}$, $I = \{1, \ldots, k\}$, with a single provider $\mathcal{M}_j$ results in the net $\mathcal{M}_{Ij}$. Its definition is based on service transitions in the requester and provider interfaces:

$$\hat{T}_j = \{\, t \in T_j : \ell_j(t) \neq \varepsilon \,\},$$

$$\hat{T}_i = \{\, t \in T_i : \ell(t) \neq \varepsilon \,\}, i \in I; \quad \hat{T}_I = \bigcup_{i \in I} \hat{T}_i.$$

Also, let the set of all the requesters' transitions (both labeled and unlabeled), all places, all arcs and all final markings be denoted, respectively, as:

$$T_I = \bigcup_{i \in I} T_i, \;\; P_I = \bigcup_{i \in I} P_i, \;\; A_I = \bigcup_{i \in I} A_i, \;\; F_I = \bigcup_{i \in I} F_i.$$

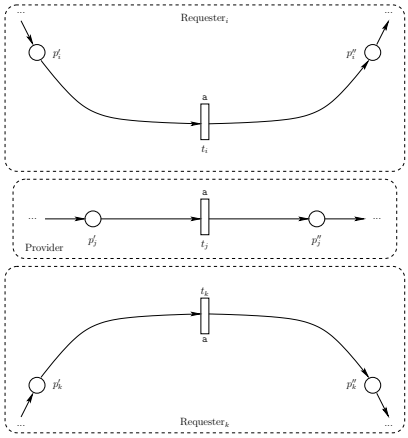

Fig.1. Multirequester interaction (before composition).

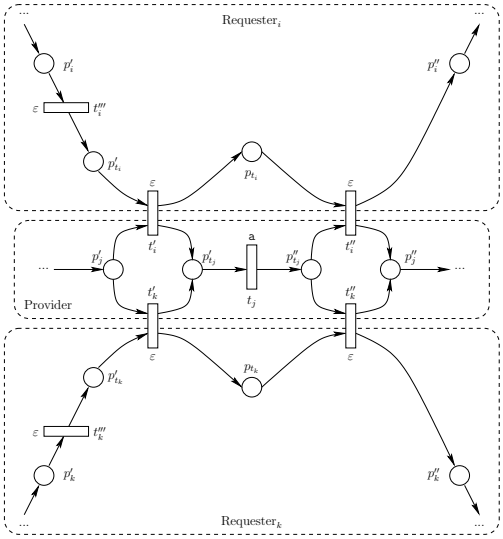

Fig.2. Multirequester interaction (after composition).

The composed model $\mathcal{M}_{Ij} = (P_{Ij}, T_{Ij}, A_{Ij}, S_{Ij}, m_{Ij}, \ell_{Ij}, F_{Ij})$ is defined as follows:

$$\begin{aligned}
P_{Ij} \;=\;& P_I \cup P_j \cup \{\, p_{t_i}, p'_{t_i} : t_i \in \hat{T}_i \;\wedge\; i \in I \,\} \cup \\
& \{\, p'_{t_j}, p''_{t_j} : t_j \in \hat{T}_j \,\};
\end{aligned}$$

$$\begin{aligned}
T_{Ij} \;=\;& T_I \cup T_j - \hat{T}_I \cup \\
& \{\, t'_i, t''_i, t'''_i : t_i \in \hat{T}_i \;\wedge\; i \in I \,\};
\end{aligned}$$

$$\begin{aligned}
A_{Ij} \;=\;& A_I \cup A_j - \\
& P_I \times \hat{T}_I - \hat{T}_I \times P_I - P_j \times \hat{T}_j - \hat{T}_j \times P_j \cup \\
& \{\, (p'_i, t'''_i), (t'''_i, p'_{t_i}), (p'_{t_i}, t'_i), (t'_i, p_{t_i}), \\
& \quad (p_{t_i}, t''_i), (t''_i, p''_i) : t_i \in \hat{T}_i \;\wedge\; i \in I \;\wedge \\
& \quad p'_i \in \text{Inp}(t_i) \;\wedge\; p''_i \in \text{Out}(t_i) \,\} \cup \\
& \{\, (p'_j, t'_i), (t'_i, p'_{t_j}), (p'_{t_j}, t_j), (t_j, p''_{t_j}), \\
& \quad (p''_{t_j}, t''_i), (t''_i, p''_j) : t_j \in \hat{T}_j \wedge \\
& \quad t_i \in \hat{T}_i \;\wedge\; i \in I \;\wedge\; \ell_j(t_j) = \ell_i(t_i) \wedge \\
& \quad p'_j \in \text{Inp}(t_j) \;\wedge\; p''_j \in \text{Out}(t_j) \,\};
\end{aligned}$$

$$\forall t \in T_{Ij} : \ell_{Ij}(t) \;=\; \begin{cases} \ell_i(t), & \text{if } t \in T_i \;\wedge\; i \in I, \\ \ell_j(t), & \text{if } t \in T_j, \\ \varepsilon, & \text{otherwise;} \end{cases}$$

$$\forall p \in P_{Ij} : m_{Ij}(p) \;=\; \begin{cases} m_i(p), & \text{if } p \in P_i \;\wedge\; i \in I, \\ m_j(p), & \text{if } p \in P_j, \\ 0, & \text{otherwise;} \end{cases}$$

$$F_{Ij} \;=\; \{m : P_{Ij} \to \{0, 1, \ldots\} \mid m \downarrow P_I \in F_I \;\wedge\; m \downarrow P_j \in F_j \;\wedge\; \forall p \in P_{Ij} - P_I - P_j : m(p) = 0\};$$

where the binary operator $\downarrow$ restricts its lefthand argument (which is a function) to the domain indicated as its righthand argument; Inp(t) is the set of input places of t and Out(t) is the set of t's output places.

For each service, the composition merges the labeled service transitions of all requesters with the corresponding service of the provider, and introduces two new places in the provider and two new places and three new transitions in each requester. The first new transition/place pair of the requesters (t_i''' and p'_{t_i}) allow each requester to initiate and control its interaction with the provider without any effect from the provider. The other place (p_{t_i}) enveloped by two transitions (t'_i and t''_i) is on the boundary between each requester and the provider. These elements help coordinate each requester's access to the provider's service transition. The two new provider's places (p'_{t_j} and p''_{t_j}) with the requester places p_{t_i}, p_{t_k} perform serialization of accesses to the provider's shared service transition.

If the composed model is bounded and its marking space is reasonably small, reachability analysis can be used for checking the absence of deadlocks. If, however, the net is unbounded or the marking space is unreasonably large, structural methods (using siphons and linear programming techniques [3], [11]) can be used. The drawback of siphon–based methods is, however, that the composed model usually contains a large number of siphons even when only minimal and basis siphons are used. This large number of siphons can be significantly reduced by performing simple, deadlock–preserving transformations of nets, and in particular, the reduction of parallel and alternate paths [4]. These transformations reduce "equivalent siphons", i.e., such siphons which, for all reachable markings, are either marked or unmarked in the same way.

Experience shows that (for systems of 5 to 10 components [4]) the performance of the proposed approach is quite satisfactory, although many further improvements are possible. For example, the reductions of parallel and alternate paths do not guarantee that all "equivalent siphons" are eliminated, so further reduction may be possible. Also, it can be shown [4], that the ordering of analyzed siphons can affect the performance the deadlock detection process; predicting the most efficient ordering of siphons can be an interesting research topic.

An interesting related problem is how to obtain Petri net models of components; would it be practical to generate such models from component specifications or, perhaps, from the implementation code?

Acknowledgments

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada through Grant RGPIN-8222.

References

[1] S.T. Albin, "The art of software architecture: design methods and techniques"; Wiley 2003.

[2] 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.

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

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

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

[6] T.A. Henzinger, "Automata for specifying component interfaces"; Proc. 8-th Int. Conference on Implementation and Application of Automata (Lecture Notes in Computer Science 2759), pp.1-2, Springer-Verlag 2003.

[7] J.E. Hopcroft, R. Motwani, J.D. Ullman, "Introduction to automata theory, languages, and computations" (2 ed.); Addison Wesley 2001.

[8] R. Janicki, P.E. Lauer, "Specification and analysis of concurrent systems – the COSY approach"; Springer-Verlag 1992.

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

[10] M. Shaw, D. Garlan, "Software architecture: perspectives on an emerging discipline"; Prentice Hall 1996.

[11] M. Silva, E. Teruel, J. Couvreur, "Linear algebra in and linear programming techniques for the analysis of place/transition net systems"; in "Lecture on Petri nets - basic models" (Lecture Notes in Computer Science 1491), pp.309-373, Springer-Verlag 1998.

[12] C. Szyperski (with D. Gruntz, S. Murer), "Component software: beyond object–oriented programming" (2 ed.); Addison-Wesley 2002.