3-rd Int. Conf. on Dependability of Computer Systems, Szklarska Poreba, Poland, 26-29 June 2008, pp.293-300.
Copyright © 2008 IEEE (DOI 10.1109/DepCoS-RELCOMEX.2008.37).

Performance analysis of component–based systems

W.M. Zuberek and I. Bluemke

Department of Computer Science
Memorial University
St.John's, Canada A1B 3X5

Institute of Informatics
Warsaw University of Technology
00-661 Warsaw, Poland

Abstract

Dependability assessment of component-based systems must include verification of temporal and performance requirements as they can be of primary importance for many real-time and embedded systems. This paper uses labeled timed Petri nets as models of the behavior of components at their interfaces. These component models are systematically composed into an integrated model of the system which is used for verification of temporal characteristics and performance analysis.

1 Introduction

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 [9], [14]. 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. Clearly, a more formal approach toward the compatibility and interoperability assessment is needed. Such a formal approach would permit and assessment based on automaed 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 [3], [12], 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 [13]. 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.

This paper extends previous work on using Petri net models for the representation of components' behavior at their interfaces. It is known that languages defined by (labeled) Petri nets include regular languages, some context–free and even context–sensitive languages [10]. Therefore, they are significantly more general than languages defined by finite automata [4], however the verification of component compatibility is significantly more difficult than in the case of regular languages [7], [5].

Petri net models proposed earlier for the verification of component compatibility [6], [7], are extended by temporal characteristics of provided services. This makes it possible to study the performance aspects of modeled systems and to verify their time-critical behavior.

Similarly as before, interacting components are composed in such a way that their interoperability can be assessed systematically. Reachability analysis, structural methods and even simulation techniques can be used for performance analysis of the composed model. The proposed 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.

Section 2 recalls the besic concepts of timed Petri nets. Section 3 is a brief introduction to component modeling using Petri nets. Composition of component models is formally defined in Section 4 while Section 5 contains a simple example illustrating the proposed approach. Concluding remarks are provided in Section 6.

2 Timed Petri Nets

Petri nets are known as a simple and convenient formalism for modeling systems that exhibit parallel and concurrent activities [10], [11]. In Petri nets, these activities are represented by the so called *tokens* which can move within a (static) graph–like structure of the net. More formally, a marked place/transition Petri net $\mathcal{M}$ is defined as $\mathcal{M} = (\mathcal{N}, m_0)$, where the structure $\mathcal{N}$ is a bipartite directed graph, $\mathcal{N} = (P, T, A)$, with a set of places P, a set of transitions T, a set of directed arcs A connecting places with transitions and transitions with places, $A \subseteq T \times P \cup P \times T$, and an initial marking function m_0 which assigns nonnegative numbers of tokens to places of the net, $m_0 : P \to \{0, 1, ...\}$.

A place is shared if it is connected to more than one transition. A shared place p is free–choice if the sets of places connected by directed arcs to all transitions sharing p are identical. Each free–choice place determines a free–choice class of transitions sharing it. A shared place which is not free–choice is a conflict place and transitions sharing conflict places are called conflicting transitions.

In order to study performance aspects of Petri net models, the duration of activities must also be taken into account and included into model specifications. In timed nets [15], occurrence times are associated with transitions, and transition occurrences are real–time events, i.e., tokens are removed from input places at the beginning of the occurrence period, and they are deposited to the output places at the end of this period (sometimes this is also called a three–phase firing mechanism as opposed to one–phase instantaneous occurrences of transitions in stochastic nets [1], [2] and time nets [8]). All occurrences of enabled transitions are initiated in the same instants of time in which the transitions become enabled (although some enabled transitions cannot initiate their occurrences). If, during the occurrence period of a transition, the transition becomes enabled again, a new, independent occurrence can be initiated, which will overlap with the other occurrence(s). There is no limit on the number of simultaneous occurrences of the same transition (sometimes this is called infinite occurrence semantics). Similarly, if a transition is enabled "several times" (i.e., it remains enabled after initiating an occurrence), it may start several independent occurrences in the same time instant.

More formally, a timed Petri net is a triple, $\mathcal{T} = (\mathcal{M}, c, f)$, where $\mathcal{M}$ is a marked net, c is a choice function which assigns choice probabilities to free-choice classes of transitions and relative occurrence frequencies to conflicting transitions, and f is a timing function which assigns an (average) occurrence time to each transition of the net, $f : T \to \mathbf{R}^+$, where $\mathbf{R}^+$ is the set of nonnegative real numbers.

The occurrence times of transitions can be either deterministic or stochastic (i.e., described by some probability distribution function); in the first case, the corresponding timed nets are referred to as D–timed nets, in the second, for the (negative) exponential distribution of firing times, the nets are called M–timed nets (Markovian nets). In both cases, the concepts of state and state transitions have been formally defined and used in the derivation of different performance characteristics of the model [15].

In timed nets, the occurrence times of some transitions may be equal to zero, which means that the occurrences are instantaneous; all transitions with zero occurrence time are called immediate (while the others are called timed). Since the immediate transitions have no tangible effects on the (timed) behavior of the model, it is convenient to 'split' the set of transitions into two parts, the set of immediate and the set of timed transitions, and to first perform all occurrences of the (enabled) immediate transitions, and then (still in the same time instant), when no more immediate transitions are enabled, to start the occurrences of (enabled) timed transitions. It should be noted that such a convention effectively introduces the priority of immediate transitions over the timed ones, so the conflicts of immediate and timed transitions are not allowed in timed nets. Detailed characterization of the behavior or timed nets with immediate and timed transitions is given in [15].

3 Component model

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

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

where $(P_i, T_i, A_i, m_i, c_i, f_i)$ is a timed Petri net, 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).

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, all providers must be ε–conflict–free, *i.e.*:

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

(the last condition could be used in a more relaxed form which 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*. Interface languages are used to define component compatibility and to verify the compatibility of interacting components [6], [7].

Although some performance characteristics can also be derived from interfaces languages of interacting components, a more general approach is used, based on performance analysis of the composed model. In this way all effects of potential concurrent executions of requester and provider components can easily be taken into account.

4 Component composition

Verification of temporal behavior of interacting components is performed by analyzing the composed model of interacting components. The composition operation follows the CORD (compatible or deadlocked) composition proposed in [5], [6], [7] for the verification of component compatibility.

4.1 Composition of a single requester wiit a single provider

A requester interface $\mathcal{M}_i = (P_i, T_i, A_i, m_i, c_i, f_i, S_i, \ell_i)$ composed with a provider interface $\mathcal{M}_j = (P_j, T_j, A_j, m_j, c_j, f_j, S_j, \ell_j)$, where $P_i \cap P_j = T_i \cap T_j = \emptyset$, creates a net $\mathcal{M}_{ij}$.

The composition is based on service transitions, i.e., those transitions in the p-interface and r-interface that have non-empty labels. Let:

$$\hat{T}_i = \{ \; t \in T_i : \ell_i(t) \neq \varepsilon \; \}, \qquad \hat{T}_j = \{ \; t \in T_j : \ell_j(t) \neq \varepsilon \; \}.$$

For a single service (denoted by "a"), the composition is outlined in Fig.1, which shows a fragment of requester and provider interfaces before and after composition [5] [6].

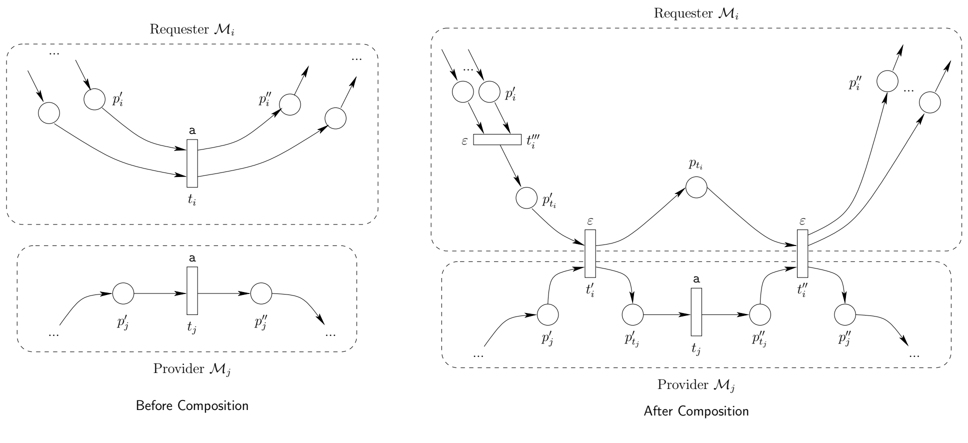

Fig.1. Composition of an r-interface with a p-interface.

The composition of an r-interface $\mathcal{M}_i$ with a p-interface $\mathcal{M}_j$, with the same sets of services $S = S_i = S_j$, is a net $\mathcal{M}_{ij} = (P_{ij}, T_{ij}, A_{ij}, m_{ij}, c_{ij}, f_{ij}, S, \ell_{ij})$ where:

$$P_{ij} = P_i \cup P_j \cup \{\ p_{t_i}, p'_{t_i} : t_i \in \hat{T}_i\ \} \cup \{\ p'_{t_j}, p''_{t_j} : t_j \in \hat{T}_j\ \};$$

$$T_{ij} = T_i \cup T_j - \hat{T}_i \cup \{\ t'_i, t''_i, t'''_i : t_i \in \hat{T}_i\ \};$$

$$\begin{aligned} A_{ij} = A_i \cup A_j - P_i \times \hat{T}_i - \hat{T}_i \times P_i - P_j \times \hat{T}_j - \hat{T}_j \times P_j \cup \\ \{\ (p'_i, t'''_i), (t'''_i, p'_{t_i}), (p'_{t_i}, t'_i), (t'_i, p_{t_i}), (p_{t_i}, t''_i), (t''_i, p''_i) : \\ t_i \in \hat{T}_i\ \wedge\ p'_i \in \mathrm{Inp}(t_i)\ \wedge\ p''_i \in \mathrm{Out}(t_i)\ \} \cup \\ \{\ (p'_j, t'_i), (t'_i, p'_{t_j}), (p'_{t_j}, t_j), (t_j, p''_{t_j}), (p''_{t_j}, t''_i), (t''_i, p''_j) : \\ t_i \in \hat{T}_i\ \wedge\ t_j \in \hat{T}_j\ \wedge\ \ell_j(t_j) = \ell_i(t_i) \wedge \\ p'_j \in \mathrm{Inp}(t_j)\ \wedge p''_j \in \mathrm{Out}(t_j)\ \}; \end{aligned}$$

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

$$\forall t \in T_{ij} : c_{ij}(t) = \begin{cases} c_i(t), & \text{if } t \in T_i, \\ c_j(t), & \text{if } t \in T_j, \\ 1.0, & \text{otherwise;} \end{cases}$$

$$\forall t \in T_{ij} : f_{ij}(t) = \begin{cases} f_i(t), & \text{if } t \in T_i, \\ f_j(t), & \text{if } t \in T_j, \\ 0, & \text{otherwise;} \end{cases}$$

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

where $\mathrm{Inp}(t)$ is the set of t's input places and $\mathrm{Out}(t)$ is the set of its output places.

For each service, the composition merges the corresponding service transitions of the requester and the provider, and introduces two new places in the provider and two new places and three new transitions in the requester. The first new transition/place pair of the requester (t'''_i and p'_{t_i}) allows each requester to initiate and control its interaction with the provider without any effect from the provider. The other place (p_{t_i}) is enveloped by two transitions (t'_i and t''_i) that straddle the boundary between each requester and the provider. These elements help coordinate each requester's access to the provider's service transition (the requester can request the same service several times). 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 service.

All new transitions are assigned empty labels. The marking function of the composition is based upon the markings of the interacting components – all new places introduced by the composition are unmarked. Also, the composition introduces only conflict–free places, so the choice probabilities are all equal to 1.0, and the occurrence times are equal to zero.

4.2 Composition of multiple requesters with multiple providers

In multirequester composition, several requester interfaces interact concurrently with the same provider interface. For example, multiple web clients connecting to a web server would constitute a multirequester composition.

For a single service (denoted by "a"), the multirequester composition is outlined in Fig.2 which shows two requester interfaces and a provider interface before and after composition. It can be observed that the composition is a systematic extension of the single requester case.

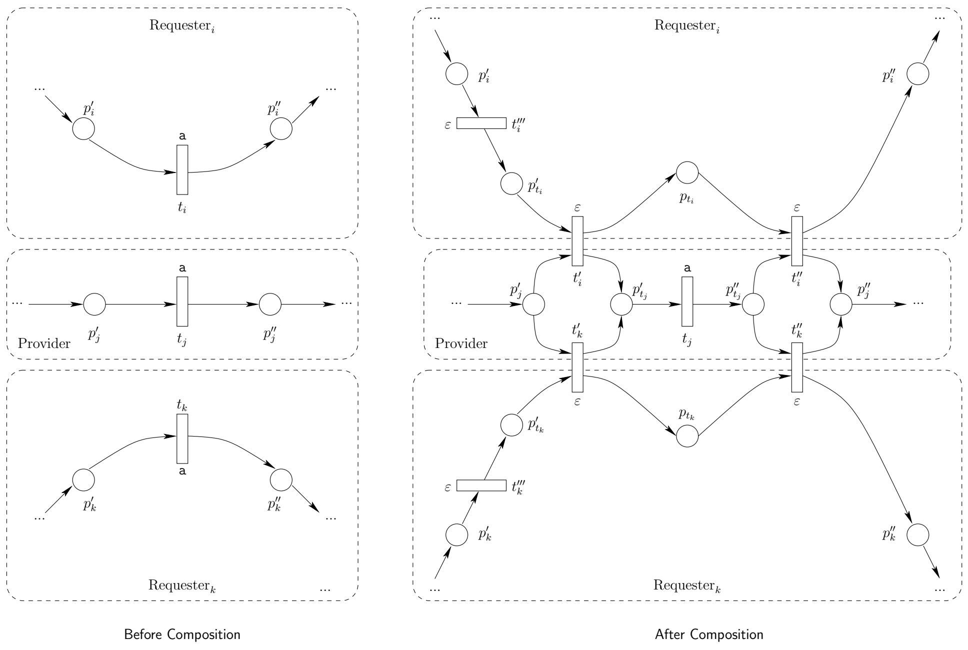

Fig.2. Multirequester composition.

The composition introduces conflict places in sequentialization of (potentially concurrent) accesses of several requesters to the same provider's service. All such conflicts are resolved in a random way by using equal probabilities for all conflicting requests.

Let the family of requesters be denoted $\mathcal{M}_I = \{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_k\}$. For simplicity it is assumed that all components use one common set of services, so $\mathcal{M}_i = (P_i, T_i, A_i, m_i, c_i, f_i, S, \ell_i)$, $i = 1, ..., k$. Similarly, let the family of providers be denoted $\mathcal{M}_J = \{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_\ell\}$, $\mathcal{M}_j = (P_j, T_j, A_j, m_j, c_j, f_j, S, \ell_j)$, $j = 1, ..., \ell$. As in the case of single requester, the composition is based on service transitions. Let:

$$\hat{T}_i = \{\ t \in T_i : \ell_i(t) \neq \varepsilon\ \}, i \in I, \quad \hat{T}_I = \bigcup_{i \in I} \hat{T}_i;$$
$$\hat{T}_j = \{\ t \in T_j : \ell_j(t) \neq \varepsilon\ \}, j \in J, \quad \hat{T}_J = \bigcup_{j \in J} \hat{T}_j.$$

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

$$T_I = \bigcup_{i \in I} T_i, \quad P_I = \bigcup_{i \in I} P_i, \quad A_I = \bigcup_{i \in I} A_i;$$
$$T_J = \bigcup_{j \in J} T_j, \quad P_J = \bigcup_{j \in J} P_j, \quad A_J = \bigcup_{j \in J} A_j.$$

The composition of a family of r-interfaces, $\mathcal{M}_I = \{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_k\}$, with a family of p-interfaces $\mathcal{M}_J = \{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_\ell\}$, is a net $\mathcal{M}_{IJ} = (P_{IJ}, T_{IJ}, A_{IJ}, L, m_{IJ}, c_{IJ}, f_{IJ}, S, \ell_{IJ})$

where:

$$P_{IJ} = P_I \cup P_J \cup \{\ p_{t_i}, p'_{t_i} : t_i \in \hat{T}_i \ \wedge \ i \in I \ \} \cup \{\ p'_{t_j}, p''_{t_j} : t_j \in \hat{T}_j \ \wedge \ j \in J\};$$

$$T_{IJ} = T_I \cup T_J - \hat{T}_I \cup \{\ t'_i, t''_i, t'''_i : t_i \in \hat{T}_i \ \wedge \ i \in I \ \};$$

$$\begin{aligned} A_{IJ} = A_I \cup A_J - P_I \times \hat{T}_I - \hat{T}_I \times P_I - P_J \times \hat{T}_J - \hat{T}_J \times P_J \cup \\ \{\ (p'_i, t'''_i), (t'''_i, p'_{t_i}), (p'_{t_i}, t'_i), (t'_i, p_{t_i}), (p_{t_i}, t''_i), (t''_i, p''_i) : \\ t_i \in \hat{T}_i \ \wedge \ i \in I \ \wedge \ p'_i \in \mathrm{Inp}(t_i) \ \wedge \ p''_i \in \mathrm{Out}(t_i)\ \} \cup \\ \{\ (p'_j, t'_i), (t'_i, p'_{t_j}), (p'_{t_j}, t_j), (t_j, p''_{t_j}), (p''_{t_j}, t''_i), (t''_i, p''_j) : \\ t_j \in \hat{T}_j \ \wedge \ j \in J \ \wedge \ t_i \in \hat{T}_i \ \wedge \ i \in I \ \wedge \ \ell_j(t_j) = \ell_i(t_i) \ \wedge \\ p'_j \in \mathrm{Inp}(t_j) \ \wedge \ p''_j \in \mathrm{Out}(t_j)\ \}; \end{aligned}$$

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

$$\forall t \in T_{IJ} : c_{Ij}(t) = \begin{cases} c_i(t), & \text{if } t \in T_i \ \wedge \ i \in I, \\ c_j(t), & \text{if } t \in T_j \ \wedge \ j \in J, \\ 1.0, & \text{otherwise;} \end{cases}$$

$$\forall t \in T_{IJ} : f_{Ij}(t) = \begin{cases} f_i(t), & \text{if } t \in T_i \ \wedge \ i \in I, \\ f_j(t), & \text{if } t \in T_j \ \wedge \ j \in J, \\ 0, & \text{otherwise;} \end{cases}$$

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

It can be shown that if a family of r-interfaces $\mathcal{M}_I = \{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_k\}$ is compatible with a p-interface, $\mathcal{M}_j$, then each r-interface $\mathcal{M}_i, i = 1, 2, \ldots, k$, is also compatible with $\mathcal{M}_j$.

The presented multirequester composition model can represent pragmatic features of traditional software architectures. For example, the notion of resource exhaustion can be represented by initially marking a provider with a finite number of tokens in the place connected to its first operation. As requesters connect with the provider, the provider's tokens are transferred from this place to implement the interaction. When this place becomes unmarked, the provider is operating at full capacity and cannot serve more requests concurrently. Any future requesters connecting with the provider would have to wait until an earlier requester completes interacting with the provider.

Another observation is that the nature of the composition makes it impossible for a requester to perform its operations in any order that is different from the one imposed by the provider. Although the service transitions are ultimately shared by all requesters, the orders in which each requester can access the services is consistent with the order imposed by the provider.

5 Example

Fig.3(a) 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 a "local" operations with the execution time equal to 5 time units (shown as an additional description of the corresponding transitions). It is assumed (quite arbitrarily) that the execution time of both provided services is equal to 2 time units (shown as descriptions attached to the transitions within the provider component). The composed model is shown in Fig.3(b).

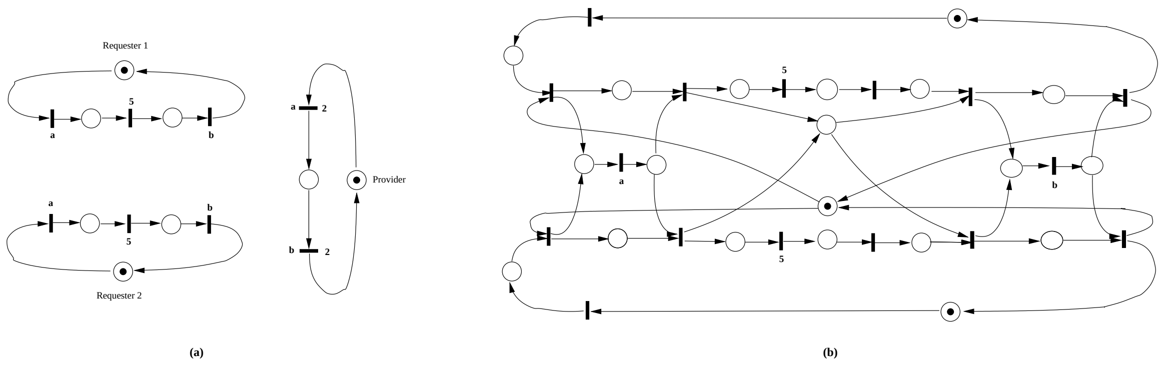

Fig.3. Composition of two requesters with a single provider.

Performance analysis of the composed model shown in Fig.3(b) results in the utilization of the service transitions (labeled by `a` and `b` in Fig.3(b)) equal to 0.2222, which can easily be deduced (in this particular case) from the analysis of each of the requester modules as their requested services are provided in strictly sequential manner.

If this performance of the provider is unsatisfactory, the provider component can be redesigned allowing more independence of the provided services. Fig.4(a) shows a model of a component which still requires that each operation `b` is preceded by an operation `a`, but which also allow 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.3). Consequently, when one of the requesters performs its local operations separating the `a` and `b` service requests, the second requester can have its services provided.

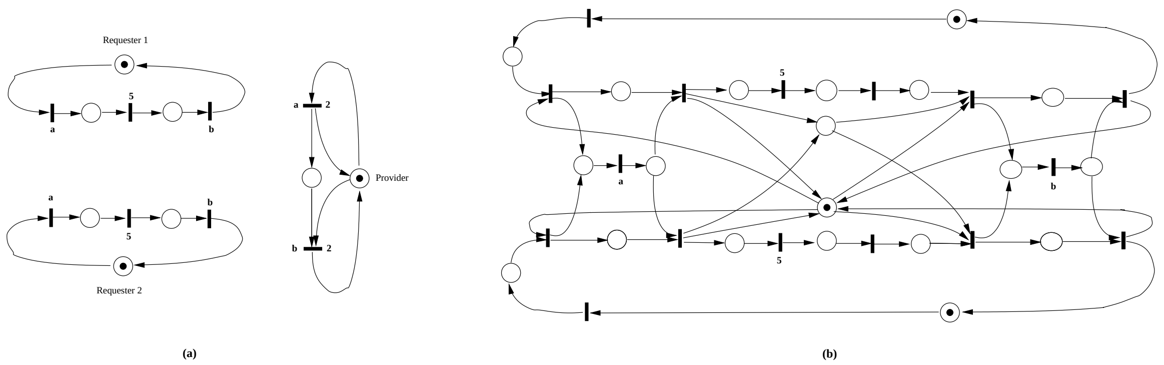

Fig.4. Composition of two requesters with a modified provider.

The utilization of the service transitions in Fig.4(b) is 0.3636, so it is significantly improved with respect to the solution shown in Fig.3.

6 Concluding Remarks

The paper proposes an extension of earlier work on component compatibility. The extension introduces temporal characteristics of services provided by interacting components and this allows to study the performance properties of component-based systems.

The derived models of realistic systems can be quite complicated and their performance analysis can become a non-trivial task. Fortunately, there are several techniques that can be used for analysis of Petri net models. For small models, reachability analysis can be used for all kinds of performance and verification aspects. For some classes of models, structural analysis can very efficient avoiding the difficulties (and in particular, the "state explosion") of some models. For large models, the simulation techniques can offer the robustness that is not provided by the other techniques.

The proposed approach is based on the integrated model of interacting components. An interesting research question would be to to develop an incremental model development as it would allow a software architect to "reuse" the verification steps for subsystems that are not affected by modifications.

Another interesting related problem is how to obtain Petri net models of components; could such models be generated (automatically) from component specifications or, perhaps, from the implementation code?

Acknowledgement

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

References

[1] M. Ajmone Marsan, G. Conte, and G. Balbo, "A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems"; *ACM Trans. on Computer Systems*, vol.2, no.2, pp.93-122, 1984.

[2] F. Bause and P.S. Kritzinger, *Stochastic Petri nets – an introduction to the theory* (Academic Studies in Computer Science); Vieweg Publ. 1996.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] P.M. Merlin and D.J. Farber, "Recoverability of communication protocols – implications of a theoretical study"; *IEEE Trans. on Communications*, vol.24, no.9, pp.1036–1049, 1976.

[9] B. Meyer, "The grand challenge of trusted components"; Proc. 25-th Int. Conf. on Software Engineering, Portland, OR, pp.660-667, 2003.

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

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

[12] 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.

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

[14] C. Szyperski, "Component technology – what, where, and how?"; Proc. 25-th Int. Conf. on Software Engineering, Portland, OR, pp.684-693, 2003.

[15] W.M. Zuberek, "Timed Petri nets – definitions, properties and applications"; *Microelectronics and Reliability* (Special Issue on Petri Nets and Related Graph Models), vol.31, no.4, pp.627-644, 1991.