Third Workshop on Modelling of Objects, Components, and Agents. Aarhus, Denmark, 11-13 October 2004, pp. 117–130

Modelling and Verification of Compatibility of Component Composition

Donald C. Craig and Wlodek M. Zuberek
Department of Computer Science
Memorial University of Newfoundland
St. John's, Canada A1B 3X5

Abstract

Two components are compatible if any sequence of operations requested by one of these components can be provided by the other component. If the set of all requested sequences is denoted by $\mathcal{L}_R$ and the set of all provided sequences of operations by $\mathcal{L}_P$, then the two components are compatible if $\mathcal{L}_R \subseteq \mathcal{L}_P$. This paper uses Petri nets to model the interface behaviours of interacting components (*i.e.* the languages $\mathcal{L}_R$ and $\mathcal{L}_P$) and formally defines the composition of components. Compatibility of components is verified by checking if the composed models contain deadlocks. Simple examples illustrate the proposed approach.

Keywords : component compatibility, component interfaces, software architecture, component-based software, Petri nets, deadlock detection

1 Introduction

The difficulties involved in the development of large-scale software architectures are well documented and, over the years, numerous strategies have been developed to help mitigate these difficulties [1]. Object-oriented programming [2] and numerous architectural description languages [3] have been introduced in order to make the development of software systems more tractable. During recent years, component based software engineering (CBSE) has been emerging as viable means of software construction whereby pre-manufactured software substructures with well-defined interfaces are designed and implemented and subsequently incorporated into larger software systems [4]. While this approach has met with some degree of success, there remains the problem of determining compatibility between components.

Classical techniques of determining compatibility have typically focused on compile-time metrics such as consistency between the numbers and types of method arguments and on appropriate use of a method return type. While such static checks are clearly important, they are insufficient in establishing the dynamic or behavioural compatibility between two software components. For example, it is possible for a server component to provide methods that exactly match the method requirements of a client component. However, if the service component imposes a rigid ordering upon the sequence of these method calls that are not adhered to by the client, it is still possible for the two components to exhibit conflicting behaviours. Such conflicts result in component incompatibility.

This paper provides the foundation for a formal model of component interaction by representing component interfaces using Petri nets [5, 6]. Interface compatibility is established by determining those interfaces that, when connected, are free of deadlock. The "requires" and "provides" relationships will be discussed in the context of the intersection of formal languages generated by the corresponding Petri nets in a component's deployment environment. By treating component behaviour as a language, compatibility between components is tested and verified.

A model of component interfaces that employs Petri nets and the notion of interface languages are introduced in Section 2. Section 3 describes the composition of component interfaces and provides a formal framework for establishing compatibility between two components using the Petri net model and deadlock detection. In Section 4, some examples that demonstrate the model are provided. Finally, Section 5 provides some applications of the model and discusses future work.

2 Petri Net Component Models

Several attempts have been made to define a component: many of these attempts have been summarized in [7]. Informally, a component can be thought of as a cohesive logical unit of abstraction with a well-defined interface that provides services to its environment. In order to behave correctly, the component would also likely require the services of other components in its environment. Some attempts have been made to formally define a component and its behaviour. Such definitions have even made extensive use of Petri nets [8].

For the purposes of this model, the low-level, internal behaviour of the component will be disregarded as it is not important in the formalism discussed below. The focus of attention is on the behaviour at the scope of the components' interfaces and not the internal dynamics of the components themselves. While it is certainly true that there may be an inseparable relationship between a component's internal behaviour and the dynamics manifested at the component's interface, this model will concentrate only upon the interface itself. The relevant behavioural properties that are necessary to ensure compatibility between components manifest themselves at the components' interface, thereby rendering communications that are strictly internal to the component irrelevant for the purposes of this discussion. Therefore, this proposal is not so much a model of a component as it is a model of a component's interface.

2.1 Interface Models

A component's interface is defined in terms of a labelled Petri net:

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

In this definition, P_i and T_i are disjoint sets of places and transitions, respectively, $A_i \subseteq P_i \times T_i \cup T_i \times P_i$ is a set of directed arcs, S_i is an alphabet representing a set of services which are associated with labelled transitions, $\ell_i : T_i \rightarrow S_i \cup \{\varepsilon\}$ is a labelling function (ε is the empty label), finally, $m_i : P_i \rightarrow \{\ 0, 1, \dots\ \}$ is the initial marking function. The Petri net model representing an individual interface must be deadlock-free. Although it is not necessary for the presented approach, it is assumed that the interfaces are represented by cyclic nets. A simple example of an appropriately marked and labelled interface is presented in Figure 1.

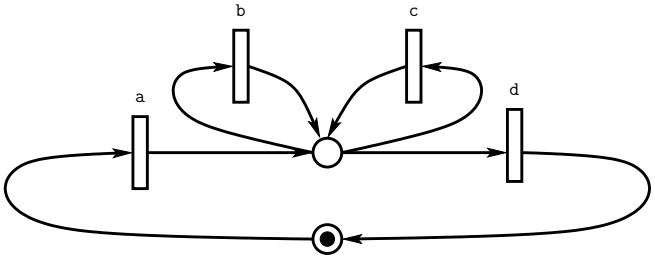

Figure 1: A component interface with services a,b,c and d

Note that a simple loop, using an unlabelled transition, can be introduced in the interface net at the initially marked place. This cycle can represent ancillary processing done by the component that is not directly related to the interface interaction. This processing may also involve the component's eventual termination, if desired.

In any software system, there will naturally be many components and each component can have several interfaces. In order to represent communication between components, the interfaces are divided into *provider* interfaces (*p-interfaces*) and *requester* interfaces (*r-interfaces*) [9].[1]

In the context of a provider interface, a labelled transition can be thought of as a *service* provided by that component. Labelled transitions on the provider essentially denote an entry point into the component. It should be noted that it is possible to have unlabelled transitions on an interface (denoted by ε in the labelling function ℓ_i above). Such transitions may be needed to implement behavioural logic of the interface and do not actually constitute a service.

It is assumed that each service in each p-interface has exactly one labelled representation, so as to prevent ambiguity in the interaction of the component interfaces:

$$\forall t_i, t_j \in T : \ell(t_i) = \ell(t_j) \Rightarrow t_i = t_j.$$

The label assigned to a transition represents a service or some unit of behaviour. For example, the label could conceivably represent a conventional function or method call. The return type and parameters are all encapsulated or abstracted by the label and are of no concern to the model as a whole. It will be assumed that if the p-interface requires parameters from the r-interface, then the appropriate number and types parameters will be delivered by the r-interface. Similarly, it will be assumed that the p-interface will generate an appropriate return value to the r-interface, if required.

Another assumption is that if an r-interface requests any arbitrary service a of a provider component that supports that particular service via its p-interface, then the provider component will be able to satisfy that service (*i.e.* the component servicing the request will not fail due to lack of resources or software faults, for example).

2.2 Interface Languages

Some proposals have restricted interface behaviour by basing the protocol on regular languages, or modest variations thereof [10]. However, by employing Petri nets, this model allows for significant flexibility in the protocol language between components, indeed

[1]Note that this model does not prevent a component from having a provider interact with a requester interface belonging to the same component. This would be an example of a recursive or *feedback* component.

even permitting interface languages recognizable by Turing machines. For example, the protocol languages could conceivably be context-free, which, in the context of modelling the behaviour of a relatively simple data structure, such as a stack, could be quite useful.

Possible sequences of services provided by a p-interface are determined by firing sequences in the Petri net model of an interface, $\mathcal{M}_i = (P_i, T_i, A_i, S_i, \ell_i, m_i)$. That is to say, $\sigma = t_{i_1} t_{i_2} \dots t_{i_k}$ is an initial firing sequence in $\mathcal{M}_i$ iff there exists a sequence of markings $m_{i_0}, m_{i_1}, \dots, m_{i_k}$ such that t_{i_ℓ} is enabled (or fireable) by $m_{i_{\ell-1}}$, m_{i_ℓ} is obtained by firing t_{i_ℓ} in $m_{i_{\ell-1}}$ for $i = 1, 2, \dots, k$, and $m_{i_0} = m_i$ is the initial marking function of $\mathcal{M}_i$. The set of all initial firing sequences of $\mathcal{M}_i$ is denoted by $\mathcal{F}(\mathcal{M}_i)$. Formally, the (initial) firing sequence σ is defined as:

$$\sigma = t_{i_1} t_{i_2} \dots\ t_{i_k} \Leftrightarrow m_{i_0} = m_i\ \wedge\ (\forall\ 0 < j \leq k : t_{i_j} \in E(m_{i_{j-1}})\ \wedge\ m_{i_{j-1}} \overset{t_{i_j}}{\rightarrow} m_{i_j}),$$

where $E(m)$ is the set of transitions enabled by m.

Finally, the language of $\mathcal{M}_i$, denoted by $\mathcal{L}(\mathcal{M}_i)$, is the set of all strings over S_i obtained by labelling complete initial firing sequences:

$$\mathcal{L}(\mathcal{M}_i) = \{\ \ell(\sigma) \mid \sigma \in F(\mathcal{M}_i)\ \wedge\ \ell(\sigma) \text{ is a complete sequence of operations }\}$$

where $\ell(t_{i_1} \dots t_{i_k}) = \ell(t_{i_1}) \dots \ell(t_{i_k})$. A *complete sequence of operations* represents a firing of all relevant transitions involved in a provider/requester interaction. As an example, the language associated with the behaviour of the interface presented in Figure 1 is $(\mathtt{a}(\mathtt{b}|\mathtt{c})^*\mathtt{d})^*$. In this case, a complete sequence in a provider/requester interaction would be any repetition of a sequence which started with `a` and ended with `d` and had any number of intervening `b` or `c` operations.

3 Component Composition and Compatibility

Component composition and compatibility assessment using Petri net models is well established in the literature [11, 12]. Related to this area is the composition and interoperability of web services [13] and verification of workflow composition [14]. While the method presented herein shares concepts with those presented in the literature, especially with respect to the deadlock-free nature of the composition of compatible nets, this paper proposes another method of composition and compatibility assessment that is fundamentally different from those proposed by earlier efforts. In particular, the composition strategy is based on sharing the labels rather than elements of net models, so the interface is composed of services rather than messages or message channels. Interface compatibility is determined by studying the languages generated by the labelled transitions of the provider and requester Petri net interface.

Compatibility of two components is dictated primarily by the behaviour at their respective interfaces. For two components to interact, the provided and requested services must be compatible with one another. This means that not only must all the services required by the requester be made available by the provider, but that the sequence of services that the requester demands must be compatible with the sequence that the provider imposes upon the services being invoked.

This leads to the following definition of *compatible interfaces*: The interface models of requester $\mathcal{M}_i$ and provider $\mathcal{M}_j$ are compatible iff $\mathcal{L}(\mathcal{M}_i) \subseteq \mathcal{L}(\mathcal{M}_j)$. Observe that this definition implies that the alphabet S_j must be a superset of S_i, $S_i \subseteq S_j$.

3.1 Composition of Interfaces

Informally, the composition can be modelled by "melding" the r-interface, $\mathcal{M}_i$, and p-interface, $\mathcal{M}_j$, together into a single Petri net $\mathcal{M}_{ij} = (P_{ij}, T_{ij}, A_{ij}, S_i, \ell_{ij}, m_{ij})$, assuming $P_i \cap P_j = T_i \cap T_j = \emptyset$.[2] The composition is denoted by $\mathcal{M}_i \rhd \mathcal{M}_j$ with an r-interface as the left-hand argument and a p-interface as the right-hand argument.

The definition of $\mathcal{M}_{ij}$ is based on those transitions in the p-interface and r-interface that have non-empty labels. Let:

$$\begin{aligned} \hat{T}_i &= \{\, t \in T_i : \ell_i(t) \neq \varepsilon \,\}, \\ \hat{T}_j &= \{\, t \in T_j : \ell_j(t) \neq \varepsilon \,\}. \end{aligned}$$

In the composed net, in addition to the existing places in the two interfaces, three more places are added for each requested/provided service. One place is added to the r-interface domain of the resulting net and two places are added to the p-interface domain of the combined net. The purpose of these three places is to act as synchronization points between the requester and provider:

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

The pairs of places added to the p-interface limit the number of additional places introduced during composition when multiple r-interfaces are allowed to interact with a single p-interface.

With respect to the transitions, all those transitions in the r-interface that have non-empty labels are replaced by a pair of transitions which envelop the additional place introduced in the r-interface domain above:

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

The new places are connected to the transitions as shown in Figure 2.

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

The labelling function of the composed net is defined by combining the labelling functions of the two interfaces:

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

Finally, the composite net has the following marking function which is based upon the markings of the interface nets of the underlying pair of interacting components:

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

Note that the composition technique described in this section precludes the possibility of value failures since all parameter types and return values associated with each

[2] It should be observed that the condition $i \neq j$ is not needed because an r-interface cannot be a p-interface.

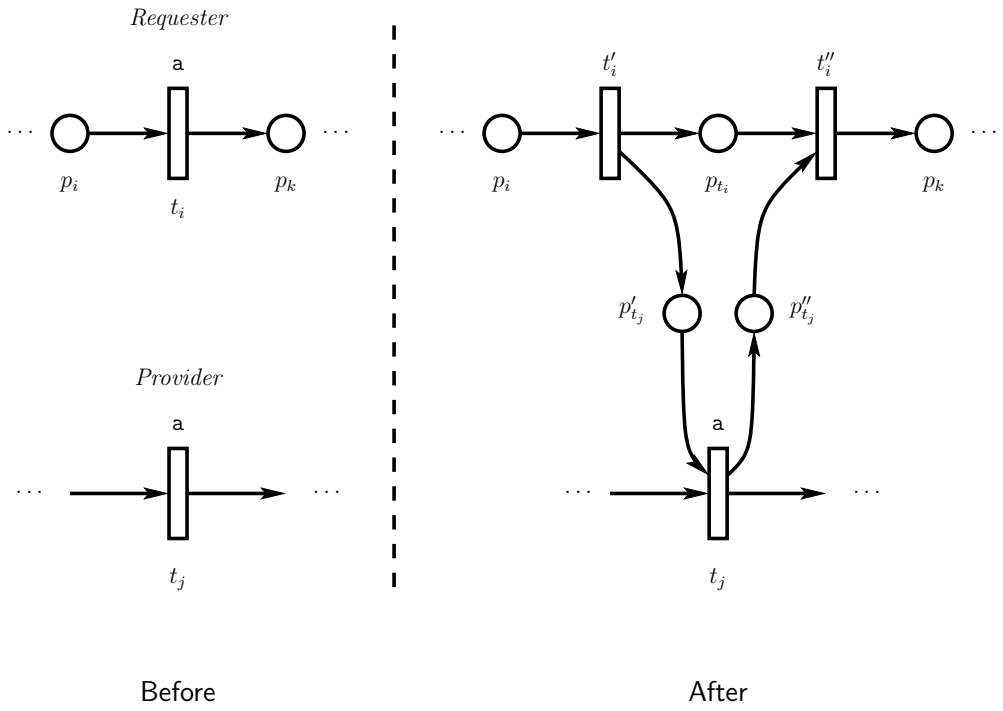

Figure 2: Composing component interfaces

service are abstracted by the transition label — the model assumes that traditional static checking has been performed and that no errors of this nature are possible. Omission failures are successfully identified because if a service is required by the requester, then the corresponding service must be offered by the provider, otherwise the two nets can be immediately deemed to be incompatible. The current model cannot, unfortunately, handle timing failures, but this may be addressed in the future by employing timed Petri nets.

3.2 Compatibility Verification

Interface compatibility means that each sequence of service requests (from an r-interface) is matched by a sequence of identical services in the corresponding p-interface.

Corollary 1
The language of the composition of two interfaces with the same alphabet S, an r-interface $\mathcal{M}_i$ and a p-interface $\mathcal{M}_j$, $\mathcal{M}_i \rhd \mathcal{M}_j$, is the intersection of $\mathcal{L}(\mathcal{M}_i)$ and $\mathcal{L}(\mathcal{M}_j)$,

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

The corollary is a straightforward consequence of the definition of interface composition. In particular, note that the composition technique leaves the structure of both interfaces essentially intact. The primary difference is that token firing is interleaved over the sequence of services of each interface. Ultimately, within the isolated context of each interface, the flow relation remains undisturbed as a result of the composition. Therefore any string generated by the resulting composition can also be generated by each interface. Corollary 1 is supported by the following corollary:

Corollary 2
Let $\mathcal{M}_{ij} = \mathcal{M}_i \rhd \mathcal{M}_j$ and $S_i = S_j$. Each firing sequence σ_i in $\mathcal{M}_i$ such that $\ell(\sigma_i) \in \mathcal{L}(\mathcal{M}_{ij})$ corresponds to a firing sequence $\hat{\sigma}_i$ in $\mathcal{M}_{ij}$ which is obtained from $\sigma_i = t_{i_1} t_{i_2} \ldots t_{i_k}$ by replacing each occurrence of each transition t_i such that $\ell(t_i) \neq \varepsilon$ by a triple of transitions t'_i, t_i and t''_i where $\ell(t'_i) = \ell(t''_i) = \varepsilon$, so $\ell(\sigma_i) = \ell(\hat{\sigma}_i)$.

This corollary is another consequence of the definition of interface composition, as shown in Figure 2.

Theorem 1
Two deadlock-free interfaces with the same alphabet S, an r-interface $\mathcal{M}_i$ and a p-interface $\mathcal{M}_j$ are incompatible iff the composition $\mathcal{M}_{ij} = \mathcal{M}_i \rhd \mathcal{M}_j$ contains a deadlock.

Proof:
An r-interface $\mathcal{M}_i$ is incompatible with a p-interface $\mathcal{M}_j$ if $\mathcal{L}(\mathcal{M}_i) \supset \mathcal{L}(\mathcal{M}_j)$.

1. $\mathcal{L}(\mathcal{M}_i) \supset \mathcal{L}(\mathcal{M}_j) \Rightarrow \mathcal{M}_{ij}$ contains a deadlock.

   If $\mathcal{L}(\mathcal{M}_i) \supset \mathcal{L}(\mathcal{M}_j)$, there exists an initial firing sequence $\hat{\sigma}$ in $\mathcal{M}_i$ such that $\ell(\hat{\sigma}) \notin \mathcal{L}(\mathcal{M}_{ij})$. The sequence $\hat{\sigma}$ cannot be an initial firing sequence in $\mathcal{M}_{ij}$, which means that it must contain a transition t_k which is not enabled in $\mathcal{M}_{ij}$, so $\hat{\sigma}$ must lead to a deadlock in $\mathcal{M}_{ij}$.

2. $\mathcal{M}_{ij}$ contains a deadlock $\Rightarrow \mathcal{L}(\mathcal{M}_i) \supset \mathcal{L}(\mathcal{M}_j)$.

   By contradiction. The claim is not true, so $\mathcal{M}_{ij} = \mathcal{M}_i \rhd \mathcal{M}_j$ contains a deadlock and $\mathcal{L}(\mathcal{M}_i) \subseteq \mathcal{L}(\mathcal{M}_j)$. If $\mathcal{M}_{ij}$ contains a deadlock, then there exists a finite firing sequence $\sigma = t_{i_1} t_{i_2} \ldots t_{i_k}$ such that $E(m_k) = \emptyset$ where $m_i \stackrel{\sigma}{\mapsto} m_k$. However, in $\mathcal{M}_i$, σ can be continued ($\mathcal{M}_i$ does not contain a deadlock), so the deadlock can be due only to composition with $\mathcal{M}_j$, *i.e.*, $\ell(\sigma) \notin \mathcal{L}(\mathcal{M}_j)$. This however contradicts the assumption $\mathcal{L}(\mathcal{M}_i) \subseteq \mathcal{L}(\mathcal{M}_j)$.

In summary, the issue of component interface compatibility can be reduced to a problem of detecting deadlocks in a net that results from the composition of two interfaces. It is believed that this model can be extended to handle several requesters interacting with a single provider, possibly through the introduction of coloured Petri nets. Needless to say, as the number of interacting components increases, the problem of deadlock detection in the resulting net becomes increasingly difficult. This challenge may be mitigated by adopting an incremental approach towards interface composition.

4 Examples

This section provides some examples which demonstrate the composition of provider and requester component interfaces using the construction technique presented in the previous sections. To demonstrate the concepts, the example of a database client interacting with a database server will be used. The examples will also highlight the importance of being able to represent interface protocols whose languages are not regular.

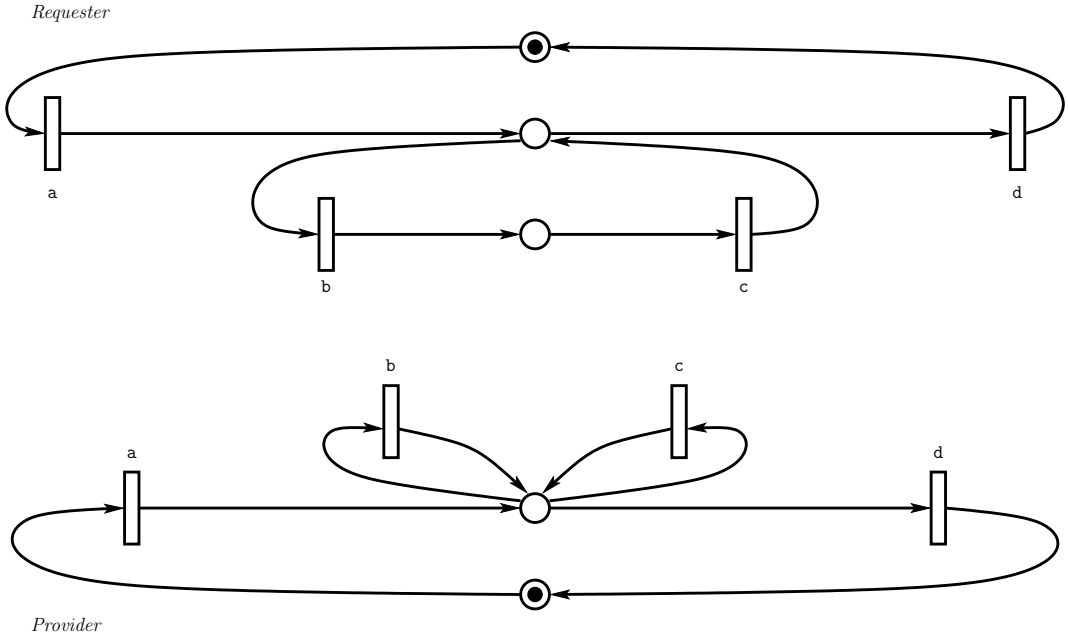

Figure 3: Database requester and provider interfaces

4.1 Database Transactions

As a simple example of the composition of a requester and provider interface modelled as Petri nets, consider Figure 3 which represents a simple database client (requester) and a database server (provider).

The first operation or requested service is denoted by a which could represent a service that opens the database and prepares it for queries, for example. The interface then requests a sequence of operations in which each operation b is followed by a corresponding operation c (these could represent *read* and *write* operations to the database, respectively). Finally, the requester invokes service d which could represent the closing of the database. The behaviour of the requester interface can therefore be represented by the regular language $(\mathtt{a}(\mathtt{bc})^*\mathtt{d})^*$. The provider interface, which represents the database server, imposes the restriction that the a service must be invoked first followed by any sequence of b and/or c services, followed finally by the d service. The behaviour of the database server is therefore denoted by the language $(\mathtt{a}(\mathtt{b}|\mathtt{c})^*\mathtt{d})^*$.

The composition of interfaces shown in Figure 3 results in the net shown in Figure 4. This composition is achieved by using the construction technique presented in Section 3.1.

It should be noted that the composition given in Figure 4 enforces the restriction that the database must be opened by the client (requester) before any operations take place upon the database server (provider). Similarly, the requester must close the database in order to satisfy the constraints of the provider. In other words, the net resulting from the composition will be deadlocked if the requester did not perform the a operation first and the d operation last, thereby implying that under such circumstances, the two components would be incompatible.

As another example of a deadlock situation, consider the case where the p-interface and r-interface from the previous example are swapped, so the language of the requester is $(\mathtt{a}(\mathtt{b}|\mathtt{c})^*\mathtt{d})^*$ and the provider's language is $(\mathtt{a}(\mathtt{bc})^*\mathtt{d})^*$, and the two nets are recomposed.

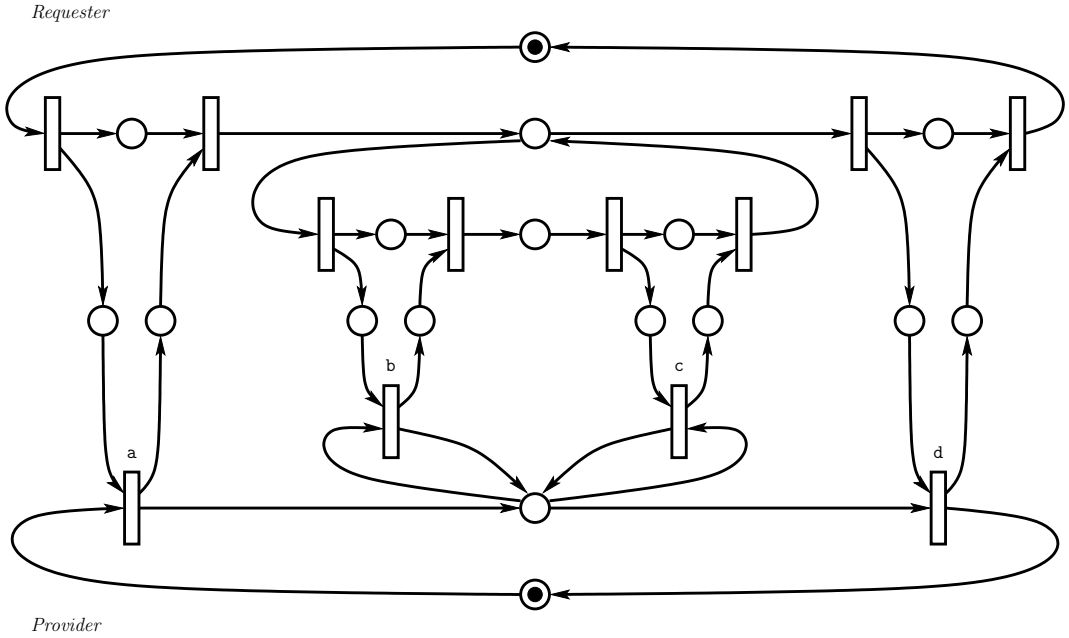

Figure 4: Composition of database requester and provider interfaces

The resulting net would exhibit deadlock as demonstrated by the composition shown in Figure 5. This results in a deadlock situation when the requester invokes service `c` immediately after invoking `a` but the provider requires that service `b` be invoked before service `c` can be requested. This deadlock demonstrates incompatibility between the two interfaces. In this case, the language of the requester is a superset of the language of the provider.

4.2 Database with Nested Transactions

The example discussed in the previous subsection models a "flat-transaction" system in which *open* and *close* pairs do not nest. Client interaction with a database component that supports nested transactions can also be represented by the model. This highlights the importance of a model being able to represent the context-free nature of the interaction between the client and server as each "opening" of a nested transaction must be matched against a corresponding "closing" of the transaction, which is a behaviour that cannot be modelled with a regular language.

A requester and provider interface that employ nested database transactions can be represented by the Petri nets given in Figure 6.[3] The provider interface keeps track of the number of opened transactions by accumulating a corresponding number of tokens in its top-most place. Similarly, the number of tokens in the bottom-most place of the requester indicate how many transactions have been opened.

These two interfaces can then be composed by applying the composition strategy described in Section 3.1. The resulting net, shown in Figure 7 does not exhibit any deadlock, therefore implying that the the interfaces are indeed compatible.

[3]Note that in the figure, the requester Petri net prohibits the opening of a new transaction in between the `b` service and `c` service. This can be easily rectified by introducing a new arc from the `b` transition to the top-most place in the requester

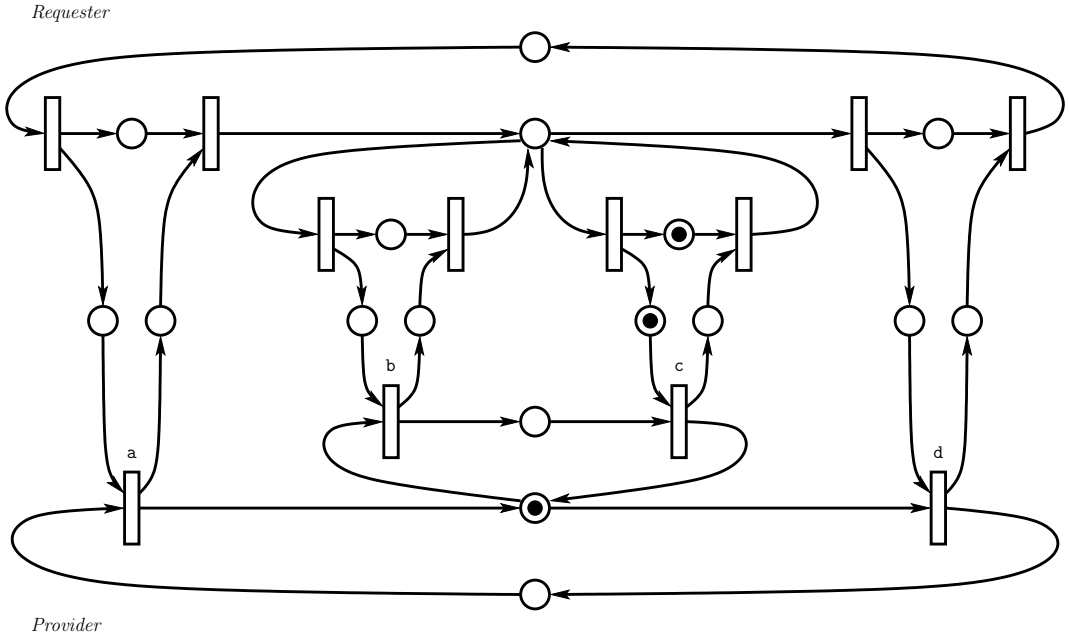

Figure 5: Composition of database requester and provider interfaces resulting in deadlock

Interfaces whose behaviours are much more complicated can also be represented by the model. Indeed, with the introduction of inhibitor arcs in the Petri net of an interface, any interface protocol whose language is recognizable by a Turing machine can be modelled by this approach.

5 Applications and Future Work

Currently, work is underway to design and implement software tools to model the interfaces of generic components. These utilities could conceivably be used to model the dynamic interactions of several components in a theoretical software system. The viability of the interface net model in the context of a software system could be tested using such tools.

Many conventional applications are not overly time dependent with respect the interactions amongst components and modest latencies between component interaction (whether due to hardware or network limitations) are usually acceptable. However, in the case of embedded real-time systems, timing issues are of paramount importance. The model proposed above is insufficient in determining temporal compatibility of two components. Fortunately, through the use of timed Petri nets [15], such timing aspects can be easily added to the model.

Hierarchical composition of components may be represented by the proposed model by constructing a hierarchy of Petri net interfaces [16]. Instead of acting as peers, interfaces can serve as mediators between different levels of a software hierarchy leading to both vertical and horizontal communication within a deployed component-based architecture during compatibility checking. The model presented in this paper is also amenable to establishing the feasibility of replacing an existing component with a new component in a hierarchy. This substitutability aspect can serve to make the upgrading of

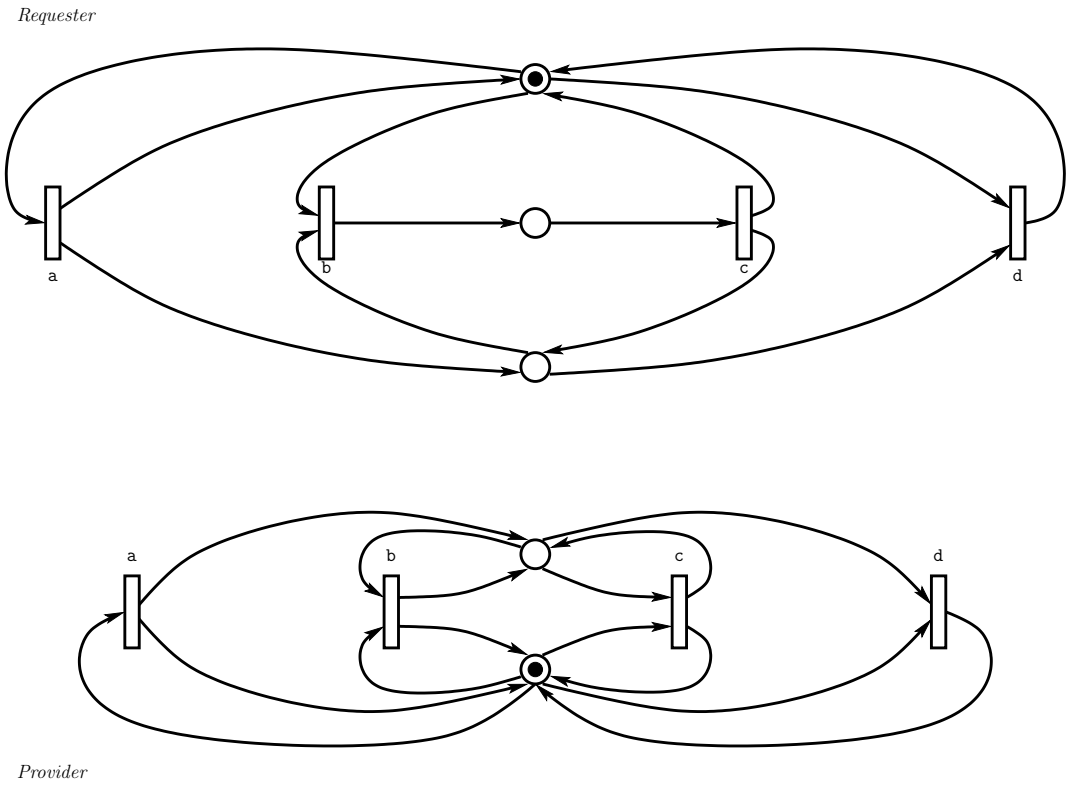

Figure 6: Database requester and provider interfaces using nested transactions

existing systems easier and may also help promote reuse in a software architecture [17].

One issue not fully addressed by this paper is how can one construct the Petri net for an interface when given the corresponding code that implements the interface? Static analysis of the code can, at the very least enumerate the services provided or requested by a component's interface. Static analysis may also reveal, to a limited degree, the sequence of service invocations. However, to accurately determine the complete set of sequences in which the services occur, a dynamic approach must be taken during which all branches of execution must be exercised before a complete Petri net can be deduced.

Another important pragmatic concern that has yet to be resolved is *when* should the compatibility check occur. If the compatibility check can be deferred as late as when the component is deployed into a running environment, then this could allow for the possibility of a software architecture that can dynamically reconfigure itself, potentially giving rise to autonomous, self-assembling software systems that exhibit behaviours that are consistent with a formal requirements specification. Naturally, issues related to state transfer from an old component to a new component would have to be addressed before this possibility can become a reality.

6 Concluding Remarks

Determining the degree to which components are compatible with one another is a multi-faceted problem that, in the general case, requires a comprehensive understanding of both the static and dynamic nature of the components involved. However, by abstracting away the internal, low-level behaviour of components and concentrating solely upon

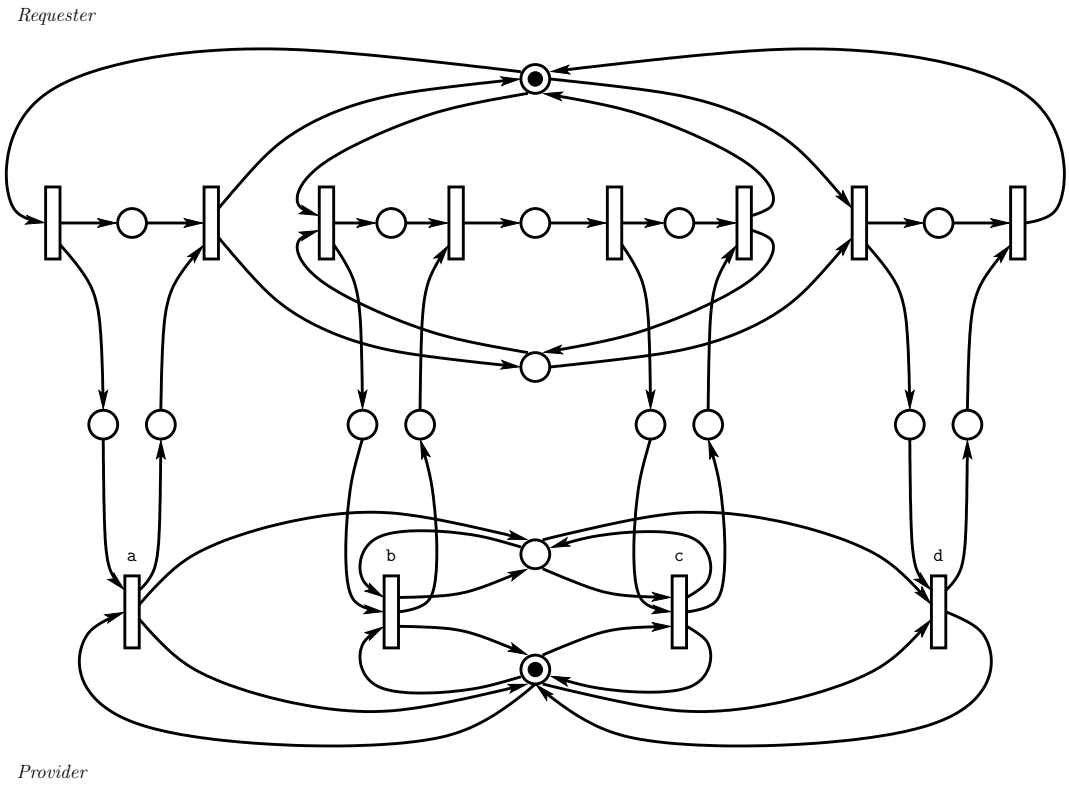

Figure 7: Composition of database requester and provider interfaces using nested transactions

the static and dynamic nature exhibited at their respective interfaces, one can establish whether or not the two components will be able to communicate effectively. This paper presents a formal strategy for composing two components by integrating the Petri nets that represent their interfaces into a single net. If the resulting net does not exhibit deadlock, then the two components are compatible and can function effectively together.

There are several approaches to deadlock detection in Petri nets. The most general one is based on exhaustive exploration of the state space; deadlock states are states which have no *next states*, so they are easily identifiable, but the generation of the entire state space may exhibit the so-called state-explosion phenomenon, *i.e.* the number of states can be an exponential function of some model parameters (*e.g.* the number of interfaces). More efficient methods identify deadlocks on the basis of structural properties and, in particular, siphons and traps [18, 19]. Models of interfaces presented in this paper are composed of multiple cyclic subnets, it is thus expected that structural methods can be used for deadlock detection in this context.

Establishing a well defined and formal method for determining the extent to which two components are able to successfully interact will serve to significantly enhance reuse of software components in a given software architecture and can contribute to the reliable evolution of a deployed component-based software system.

Acknowledgement

The authors are thankful for a number of interesting remarks and comments provided by four anonymous reviewers.

Partial support from the *Natural Sciences and Engineering Research Council of Canada*, through grant RGPIN 8222, is gratefully acknowledged.

References

[1] I. Sommerville. *Software Engineering, 6th edition*. Addison-Wesley, 2001.

[2] G. Booch. *Object-Oriented Analysis and Design with Applications, 2nd edition*. Benjamin/Cummings Publishing Company, Inc., 1994.

[3] N. Medvidovic and R.N. Taylor. A framework for classifying and comparing architecture description languages. In *Software Engineering — ESEC/FSE '97*, volume 1301 of *Lecture Notes in Computer Science*, pages 60–76. Springer-Verlag, 1997.

[4] G.T. Heineman and W.T. Council. *Component Based Software Engineering: Putting the Pieces together*. Addison-Wesley, 2001.

[5] W. Reisig. *Petri-Nets: An Introduction*. Springer-Verlag, 1985.

[6] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.

[7] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming, 2nd edition*. Addison-Wesley, 2002.

[8] W.M.P van der Aalst, K.M. van Hee, and R.A. van der Toorn. Component-based software architectures: A framework based on inheritance of behaviour. *Science of Computer Programming*, 42(2–3):129–171, Feb/Mar 2002.

[9] S. Moschoyiannis and M.W. Shields. Component-based design: Towards guided composition. In Johan Lilius, Felice Balarin, and Ricardo J. Machado, editors, *Proceedings of Third International Conference on Application of Concurrency to System Design*, pages 112–131. IEEE Computer Society, Jun 2003.

[10] R.H. Reussner. Enhanced component interfaces to support dynamic adaption and extension. In Ralph H. Sprague, editor, *Proceedings of the 34th Hawaii International Conference on System Sciences*, pages 1–9. IEEE Computer Society, 2001.

[11] C. Sibertin-Blanc. A compositional partial order semantics for petri net components. In M.A. Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 377–396. Springer-Verlag, 1993.

[12] E. Kindler. A compositional partial order semantics for petri net components. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 235–252. Springer-Verlag, 1997.

[13] A. Martens. Usability of web services. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering Workshops*, pages 182–190. IEEE Computer Society, 2003.

[14] W.M.P van der Aalst. Workflow verification: Finding control-flow errors using petri-net-based techniques. In W. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empircal Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 161–183. Springer-Verlag, 2000.

[15] W.M. Zuberek. Petri nets and timed petri nets in modeling and analysis of concurrent systems, November 2003. Faculty Research Forum — 25^{th} Anniversary of the Department of Computer Science at Memorial University, St. John's, Newfoundland and Labrador, Canada A1B 3X5.

[16] R. Fehling. A concept of hierarchical petri nets with building blocks. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1993*, volume 674 of *Lecture Notes in Computer Science*, pages 148–169. Springer-Verlag, 1993.

[17] D. Garland, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17—26, Nov 1995.

[18] Z. Li and M. Zhou. Elementary siphons of petri nets and their application to deadlock prevention in flexible manufacturing systems. *IEEE Transactions on Systems, Man, and Cybernetics — Part A: Systems and Humans*, 34(1):38–51, Jan 2004.

[19] J. Ezpeleta, J.M. Couvreur, and M. Silva. A new technique for finding a generating family of siphons, traps and st-components: Application to colored petri nets. In G. Rozenberg, editor, *Advances in Petri Nets 1993*, volume 674 of *Lecture Notes in Computer Science*, pages 126–147. Springer-Verlag, 1993.