A Formal Model for Representing Component Interfaces and Their Interaction

Donald C. Craig

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

***Abstract*— The field of Component Based Software Engineering (CBSE) is emerging as a means of mitigating the complexity faced by software architects during the design and maintenance of software systems. Unfortunately, successfully determining compatible interaction amongst components can be a difficult problem to solve. Establishing whether two components are compatible with one another may be facilitated by developing a formal model to describe component interfaces and, in particular, how they interact with each other. This model can be extended to represent the interaction between an arbitrary number of components and may also be used to model the components hierarchically in the context of a larger software architecture.**

This paper establishes a formal model of component interaction by representing component interfaces as Petri nets. Interface compatibility can be established by determining those interfaces that, when connected, are free of deadlock. Such work could lead to the formation of autonomous software systems in which each component is endowed with a meta-knowledge of the services it requires and provides. By giving software the ability to self-connect to other components in its environment, we allow for the possibility of self-assembling software systems; thereby potentially increasing the degree of automation in the construction of software systems.

I. 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. 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. By treating component behaviour as a language, compatibility between components can be tested and verified.

A model of component interfaces that employs Petri nets and the notion of interface languages are introduced in Section II. Section III 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 IV, some examples that demonstrate the model are provided. Finally, Section V discusses some of the pragmatic issues that arise as a result of the proposed model.

II. Petri Net Component Models

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. For the purposes of the model presented in this paper, the low-level, internal behaviour of the component will be disregarded as it is not important in the formalism discussed below. 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. Therefore, this proposal is not so much a model of a component as it is a model of a component's interface.

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

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

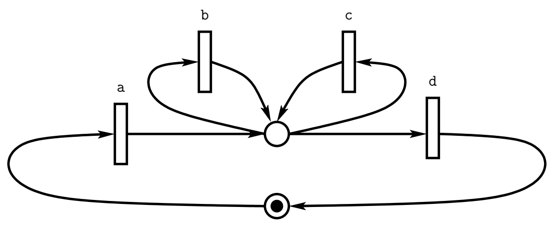

Fig. 1. A component interface with services a,b,c and d

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.

In a given software system, there will typically be several components. In order to represent communication between components, the interfaces are partitioned into *provider* interfaces (*p-interfaces*) that supply services and *requester* interfaces (*r-interfaces*) that require services [7].

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:

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

As mentioned, 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. 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).

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 $\mathcal{F}(\mathcal{M}_i)$.

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: $\mathcal{L}(\mathcal{M}_i) = \{ \ell(\sigma) \mid \sigma \in F(\mathcal{M}_i) \wedge \ell(\sigma)$ *is a complete sequence of operations* $\}$ where $\ell(t_{i_1} \dots t_{i_k}) = \ell(t_{i_1}) \dots \ell(t_{i_k})$ and *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.

III. Component Composition and Compatibility

Some prior work has already been attempted in the areas of component composition and compatibility assessment using Petri net models [8], [9]. Related to this area is the composition and interoperability of web services [10] and verification of workflow composition [11]. 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 interfaces.

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 observation regarding 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, $S_i \subseteq S_j$.

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 $P_i \cap P_j = T_i \cap T_j = \emptyset$. The composition is denoted by $\mathcal{M}_i \triangleright \mathcal{M}_j$ 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. First, we define $\hat{T}_i$ and $\hat{T}_j$ to be the transitions in the requester and provider, respectively, that have labels assigned to them:

$$\hat{T}_i = \{\ t \in T_i : \ell_i(t) \neq \varepsilon\ \},$$

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

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; A_{ij} is defined as follows:

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

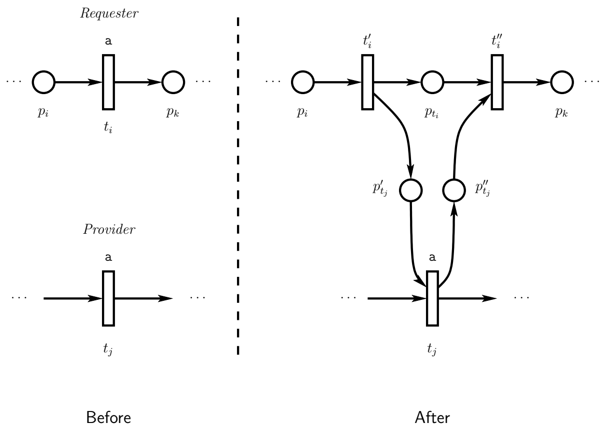

Fig. 2. Composing component interfaces

The labelling function of the composed net is defined by combining the labelling functions of the p- and r-interfaces. The marking of the composite net is based upon the markings of the interface nets of the underlying pair of interacting components:

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

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

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

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

This observation 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. It can also be observed that 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 $\mathcal{M}_{ij} = \mathcal{M}_i \triangleright \mathcal{M}_j$ contains a deadlock. In essence, the issue of component interface compatibility can be reduced to a problem

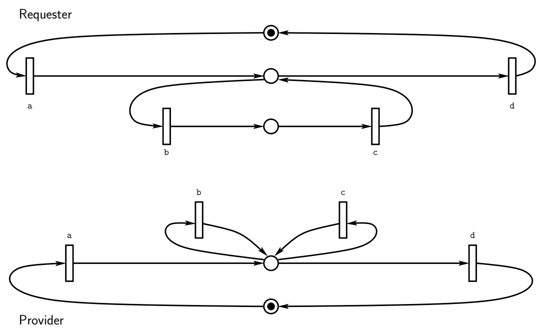

Fig. 3. Database requester and provider interfaces

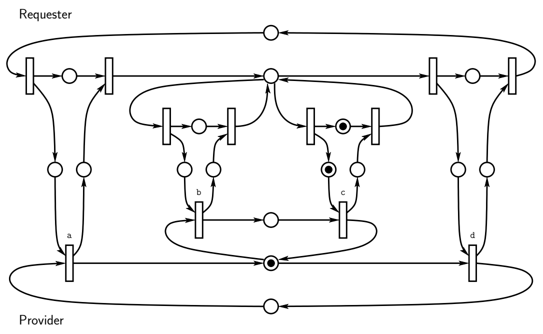

Fig. 5. Deadlock in incompatible interfaces

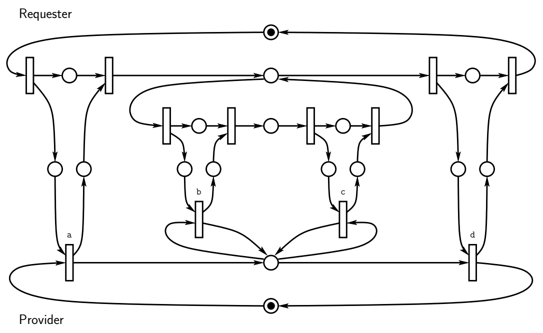

Fig. 4. Interface composition

of detecting deadlocks in a net that results from the composition of two interfaces.

IV. Example

As a simple example 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 is shown in Figure 4.

As an example of a deadlock situation, consider the case where the p-interface and r-interface from the previous example are swapped and then recomposed. 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.

The composition strategy can also be used to model nested database transactions in which the open and close pairs can be nested. The protocol in this case would be context free and not regular. 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. This feature allows for the modelling of a wide variety of protocol interactions between components in a software system.

V. Pragmatic Considerations

An 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.

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, for example, 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 [12], such timing aspects can be easily added to the model.

Hierarchical composition of components may be represented in this model by constructing a hierarchy of Petri net interfaces [13]. 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 deployment environment. This hierarchical strategy more accurately reflects the design and development of large software systems. 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.

VI. 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 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. 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 author wishes to thank Dr. Wlodek Zuberek for providing guidance on the composition strategy and for providing numerous suggestions on an earlier version of this paper.

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. Taylor, "A framework for classifying and comparing architecture description languages," in *Software Engineering — ESEC/FSE '97*, ser. Lecture Notes in Computer Science. Springer-Verlag, 1997, vol. 1301, pp. 60–76.

[4] G. Heineman and W. 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*, vol. 77, no. 4, pp. 541–580, Apr 1989.

[7] S. Moschoyiannis and M. Shields, "Component-based design: Towards guided composition," in *Proceedings of Third International Conference on Application of Concurrency to System Design*, J. Lilius, F. Balarin, and R. J. Machado, Eds. IEEE Computer Society, Jun 2003, pp. 112–131.

[8] C. Sibertin-Blanc, "A compositional partial order semantics for petri net components," in *Application and Theory of Petri Nets 1993*, ser. Lecture Notes in Computer Science, M. Marsan, Ed. Springer-Verlag, 1993, vol. 691, pp. 377–396.

[9] E. Kindler, "A compositional partial order semantics for petri net components," in *Application and Theory of Petri Nets 1997*, ser. Lecture Notes in Computer Science, P. Azéma and G. Balbo, Eds. Springer-Verlag, 1997, vol. 1248, pp. 235–252.

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

[11] W. van der Aalst, "Workfbw verifi cation: Finding control-fbw errors using petri-net-based techniques," in *Business Process Management: Models, Techniques, and Empirical Studies*, ser. Lecture Notes in Computer Science, W. van der Aalst, J. Desel, and A. Oberweis, Eds. Springer-Verlag, 2000, vol. 1806, pp. 161–183.

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

[13] R. Fehling, "A concept of hierarchical petri nets with building blocks," in *Advances in Petri Nets 1993*, ser. Lecture Notes in Computer Science, G. Rozenberg, Ed. Springer-Verlag, 1993, vol. 674, pp. 148–169.