Component Compatibility and its Verification

Donald C. Craig and Wlodek M. Zuberek
Computer Science Department
Memorial University of Newfoundland
Canada A1B 3X5
Email: {donald,wlodek}@cs.mun.ca

Abstract

An approach to verification of component compatibility is proposed in which each component's behaviour (at its interfaces) is represented by a labeled Petri net in such a way that the sequences of services (provided or requested) correspond to sequences of labels assigned to occurring transitions. The behaviour of a component can thus be defined as the language of its modeling net. Two interacting components are compatible if and only if all possible sequences of services requested by one of these two components can be satisfied by the other component; in other words, two components are compatible if the language of the requesting component is a subset of the language of the component providing the services. Verification of this simple relation depends upon the class of languages defining the behaviours of the components. If the languages are regular, the verification of compatibility is straightforward. For non-regular languages, a more elaborate approach is needed in which a net model composed of the interacting components is checked for the absence of deadlocks. Some applications of the proposed approach are also discussed.

1. Introduction

As the demands placed on modern day software systems continue to increase, the construction of large-scale software architectures becomes more complex [19, 6]. While the continued development of programming languages, paradigms and patterns of increasing level of abstraction have helped combat some of this complexity [2, 3, 14], there is a growing trend towards focusing efforts on the development of new architectural strategies to assist in the building of software systems [7, 1]. Proposals to help mitigate the complexity inherent in the design, development and maintenance of software architectures include applying principles of Component-Based Software Development (CBSD) and the integration of Commercial Off The Shelf (COTS) software [5]. Using these approaches, pre-fabricated subsystems with well-defined functionality are combined and deployed so as to create a fully functioning software system that satisfies the requirements.

As the discipline of software engineering continues to adopt a component-based approach towards the construction of complex software architectures, the need to assess the compatibility and interoperability of the individual software components is becoming increasingly important during the integration phase of the software production process. While manual and ad hoc strategies towards component integration have met with some success in the past, these techniques do not lend themselves well to automation. Clearly, a more formal approach towards the assessment of component compatibility and interoperability is desired. This may permit the automated assessment of interoperability using mechanical techniques and may help promote the reuse of existing software components.

Unfortunately, assessing the interoperability of pre-existing software components software during the integration phase is a challenging problem. This paper presents a formal approach to determine if two or more components are compatible with each other. This technique is independent of any particular software language, application framework or component technology. The method of verification proposed in this paper abstracts away low-level attributes in favour of assessing interoperability by identifying compatible sequences of operations between component interfaces.

While a number of definitions of *component* exist in the literature [8], many of them share common attributes. In particular, the concepts of *interfaces*, *services*, *deployment* and *composition* permeate most definitions. While such definitions are plentiful, attempts to formally represent the specification of components occur less frequently in the literature. This paper provides a means to formally specify the behaviour of components in the sense of the sequence of services that they either provide or require. Using this formal specification, a means to verify their subsequent compatibility is then proposed.

Second International Conference on Digital Telecommunications (ICDT'07)
0-7695-2910-0/07 $20.00 © 2007 IEEE

The organization of this paper is as follows: In Section 2, the concept of interface languages and how it relates to component compatibility is introduced. Formal means to verify the compatibility of software components are then described in Section 3. Examples are presented in Section 4 followed by conclusions and future work in Section 5.

2. Component Compatibility

Central to the notion of a component is the concept of an *interface*. Through the interface, the component interacts with its environment and with other components. At a fundamental level, an interface is a collection of services that a component either provides to other components or a collection of services that it requires in order to fulfill its duties. With this in mind, it is helpful to consider two classes of component interfaces, namely, *provider* interfaces and *requester* interfaces. Provider interfaces are typically reactive in that they respond to the service demands of requester interfaces. Requester interfaces are active elements of a software system that initiate connections to the provider interfaces that are needed to satisfy the requester's objectives.

Prior to addressing the issue of component compatibility, it is necessary to develop a formal model for a component interface and to introduce the concept of interface languages. This model will form the basis of a technique which can be employed to determine if two or more components are compatible.

2.1. Component Interfaces and Interface Languages

While component interfaces may be represented using several abstractions and languages [8], this paper will demonstrate the viability of using Petri nets as a foundation for the description of component interfaces. To formally represent an interface and the services that it contains, a labeled Petri net [18] can be used:

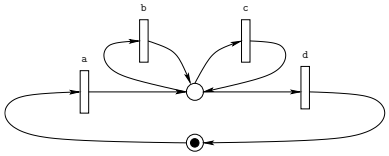
$$\mathcal{M}_i = (P_i, T_i, A_i, L_i, \ell_i, m_i, F_i)$$

where (P_i, T_i, A_i, m_i) is a deadlock-free, marked Petri net, L_i is an alphabet representing a set of services which are associated with transitions by a labeling function $\ell_i : T_i \rightarrow L_i \cup \{\varepsilon\}$, where ε is the empty label, $\varepsilon \notin L_i$, and F_i is a set of *final* markings. Note that from an implementation perspective, Petri nets can be described using an XML-based language [21].

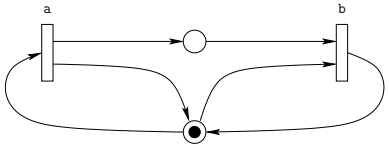
In a Petri net, the set of places (P_i) and transitions (T_i) are connected together by directed arcs (A_i) to form a bipartite graph. Initially, one or more of the places are marked with a token as denoted by its initial marking function m_i. A transition is *enabled* and can *fire* if all of the places that lead into it (*i.e.*, the *input* places) have a token. As a transition fires, a token is removed from each of the transition's input places and new tokens are placed in each of the places leading from the transition (*i.e.*, the *output* places), which typically results in a new marking of the net. A deadlock occurs when a marking reachable from the initial marking has no transitions enabled.

An *interface language* is any sequence of service identifiers (or labels) that can be executed by a requester or provider component. The interface language is characterized by the sequences of labeled transitions of a Petri net that are fired as part of the operation of the component. More formally, the language of $\mathcal{M}_i = (P_i, T_i, A_i, L_i, \ell_i, m_i, F_i)$, denoted by $\mathcal{L}(\mathcal{M}_i)$, is the set of all strings over L_i obtained by labeling firing sequences which begin with m_i and end at one of the final markings.

This interface language essentially defines the behaviour of the component at its interface. As a result, one can think of the high-level behaviour (and, consequently, the compatibility) of a component in terms of the languages that are manifested at the component's interfaces. The language can be regular or non-regular. For example, consider the simple Petri nets of Figure 1. By convention, places and transitions



(a) Interface with regular language



(b) Interface with non-regular language

Figure 1. Sample Petri net interfaces

in a Petri net are represented, respectively, by circles and rectangles connected together by directed arcs. Tokens are denoted by a small black circle within a place. In both examples, the set of final markings is defined to be the set containing the initial marking. Figure 1(a) shows an interface with a language defined by the regular expression $(\mathtt{a}(\mathtt{b}|\mathtt{c})^*\mathtt{d})^*$. Figure 1(b) shows an interface with a non-regular language, in which each 'a' symbol is followed by a matching 'b' symbol. This interface language could represent the pairing of service invocations in which each service invocation acquiring a resource must be matched with a corresponding service invocation that releases the resource. This interface language may also represent nested database transactions, for example.

We can subsequently define component compatibility in the context of these interface languages. More specifically, a requester interface $\mathcal{M}_i$ and a provider interface $\mathcal{M}_j$ are *compatible* iff:

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

In other words, a requester can demand only a sequence of services that the provider interface is capable of performing. If the requester makes demands that are beyond the provider's capabilities, then the two components are deemed incompatible, by this definition.

Note that this definition of compatibility does not consider low-level details such as the numbers and type of parameters required by a service or its return value, if any. While these attributes are certainly important, they can typically be checked through static analysis of the programming language used to implement the system. The definition of compatibility above attempts to address the *dynamic* compatibility of the interfaces as it relates to the sequence of services that are requested and provided between two or more components.

3. Verification of Component Compatibility

When the languages of two or more interfaces are known, verifying if they are compatible essentially amounts to determining whether or not the interface language of the requester is a subset of the interface language of the provider. To determine this, several approaches can be used, depending upon whether the interface languages of the components are regular or non-regular.

3.1. Regular Interface Languages

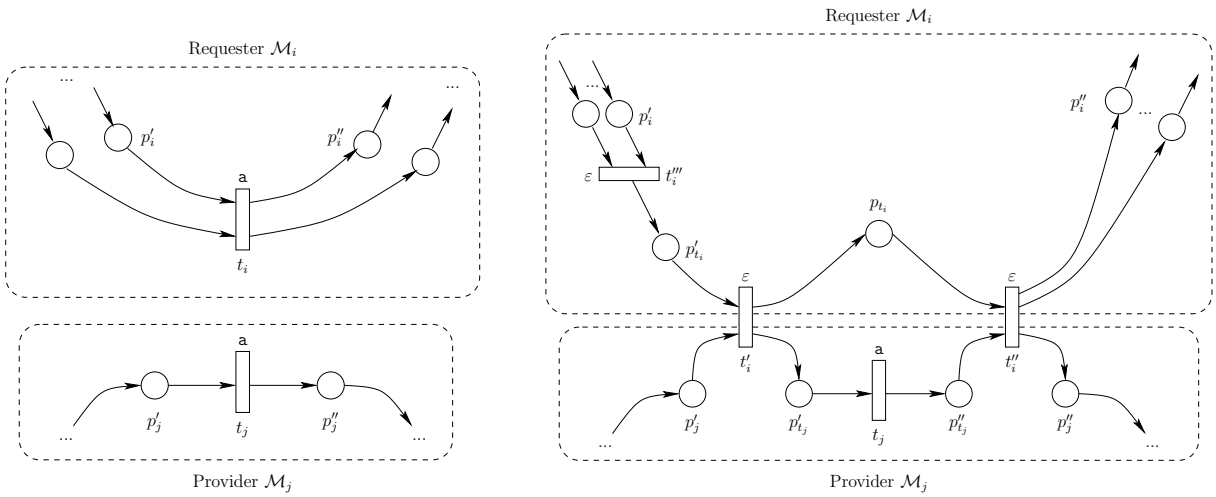
If only two components are being tested for compatibility and if the Petri nets have finite behaviours (*i.e.*, the number of tokens that can be in any one place is limited by some value k), then the interface language is regular and the net modeling the interface of the component can be transformed into a deterministic finite automaton. In many cases, the finite automaton defining the language of an interface may be derived directly from the net representing the interface. More generally, each reachable marking of the Petri net is represented by a state in the corresponding automaton. The labels of the arcs between the states are derived from the labels of the transitions that fire between successive markings of the Petri net. The initial state and final (accepting) states of the automaton are represented by the initial marking and final marking(s) of the original Petri net, respectively.

When the nets that represent the requester and provider interfaces have been converted to an equivalent automaton, compatibility can be verified by simple operations on the interface languages. Because regular languages are closed under complementation and intersection and since $\mathbf{L}_1 \subseteq \mathbf{L}_2 \Leftrightarrow \mathbf{L}_1 \cap \overline{\mathbf{L}_2} = \emptyset$, where $\overline{\mathbf{L}}$ is the complement of $\mathbf{L}$, all that needs to be done to determine if the requester language is a subset of the provider language is to build an appropriate automaton that recognizes $\mathcal{L}(\mathcal{M}_i) \cap \overline{\mathcal{L}(\mathcal{M}_j)}$ and check if the language is empty. This can be achieved by complementing the accepting and non-accepting states of the provider's corresponding finite automaton and using the *product construction* technique [15] to build an automaton that recognizes the intersection of the requester's automaton and the complement of the provider's automaton. If the set of reachable accepting states in the resulting automaton is empty, then the two interfaces are compatible. Otherwise, they are incompatible.

3.2. Non-Regular Interface Languages

When verifying the compatibility of more than two components or when the languages of the requester and/or provider are not regular, a more elaborate way of assessing compatibility must be employed. To determine if the components are compatible, their respective interfaces can be composed as shown in Figure 2. Figure 2(a) shows two identically labeled transitions of a requester and provider interface before composition. Figure 2(b) shows the resulting Petri net fragment after the composition. For each service being composed, the service transition associated with the requester is removed and three new transitions with ε (empty) labels are added. Note that the labeled transition associated with the service in the provider remains the same, but its connectivity is modified after the composition. Four additional places are introduced. The three places surrounding the labeled service transition in the provider (p_{t_i}, p'_{t_j}, and p''_{t_j}) allow for the coordination between a requester and the provider when multiple requesters are interacting with the provider. The new transition (t'''_i) and adjacent place (p'_{t_i}) introduced in the requester preserve any free choice structures in the requester. The purpose of this transition/place pair is to ensure that the sequence of services that can be solicited by the requester is not restricted by the composition with the provider.

Given this composition, it can be shown that the verification of the compatibility relation (*i.e.*, that $\mathcal{L}(\mathcal{M}_i) \subseteq \mathcal{L}(\mathcal{M}_j)$), is equivalent to checking the absence of deadlocks in the composed net [12]. There are a couple of strategies that can be used to determine if the composed net contains a deadlock. The appropriate strategy to use depends upon whether the composed net is bounded or unbounded. If the composition results in a bounded net, then reachability analysis of the composed net can be performed to determine if there are any deadlocks in the composed net. This can be done by executing a breadth-first traversal of all the markings that are reachable from the initial marking. New mark-



(a) Interfaces before composition

(b) Interfaces after composition

Figure 2. Sample Petri net interfaces

ings reachable from a given marking can be determined by firing each enabled transition and keeping track of the resulting markings. The same technique can then be iteratively applied to each of the new markings. This process can be continued until all the reachable markings have been found. If a marking is found in which no transition can fire, then a deadlock has been found and the components cannot be compatible.

If the set of reachable markings is excessively large due to the state explosion problem [20], for example, or if the composed net is unbounded, then structural techniques may be employed to determine if the composed net contains a deadlock. It is known that after a net deadlocks, all the remaining places constitute an unmarked (or empty) siphon [17]. A *siphon* is simply a set of places in which the transitions that lead immediately to the places (*i.e.*, the set of *input* transitions) are a subset of the transitions that lead directly out of the places (*i.e.*, the set of *output* transitions). Two classes of siphons that are of particular importance for deadlock analysis include *minimal siphons*, which are siphons that do not contain another siphon as a proper subset, and *basis siphons*, which are siphons that cannot be represented as a union of other siphons [4].

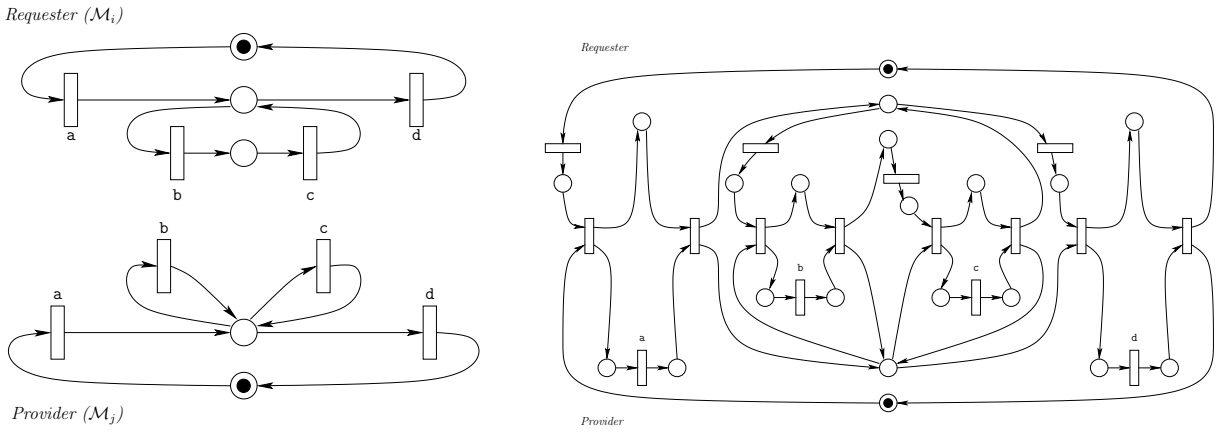
If no minimal siphon can be emptied of its tokens, then the composed net is known to be deadlock-free [9] and the composition is verified to be compatible. If, however, at least one minimal siphon can be emptied, then further analysis is required to determine if a deadlock exists in the net. It must be determined if there is a sequence of minimal or basis siphons that, when emptied, produce a deadlock. If no such sequence exists, then, again, the net is confirmed to be deadlock-free. The deadlock detection algorithm uses both minimal and basis siphons in conjunction with linear programming [10] in order to determine if some sequence of minimal/basis siphons can be emptied in order to produce a deadlock. For efficiency reasons, the set of siphons to be emptied is initially limited to the set of minimal siphons, which are usually much smaller in number when compared to the basis siphons. If no deadlock is found, the basis siphons are then used to find a deadlock by employing the same technique. Because all siphons in a net can be represented as a union of one or more basis siphons, if no combination of basis siphons can be emptied to produce a deadlock, then the net is shown to be deadlock-free and the composition is verified to be compatible.

As the composed net gets increasingly larger, extracting the basis and minimal siphons can become more complex [11]. As a result, for efficiency reasons, the composed net can typically be reduced by finding and eliminating redundant paths. It can be shown that removing these redundant paths does not affect the behaviour of the remaining net in terms of the presence or absence of deadlocks.

Note that the composition strategy described above and the subsequent checking for deadlocks (either by reachability analysis or by structural techniques) can also be performed on interfaces that are regular in their languages. Examples of the verification techniques are presented in the next section.

4. Examples

To demonstrate the compatibility verification approach described above, consider the case of a database client and server. When performing operations on a database, a



(a) Database interfaces before composition

(b) Database interfaces after composition

Figure 3. Composition of database interfaces

database server (or provider) expects clients (or requesters) to start a transaction, perform the necessary read and/or write operations and terminate the transaction. This model of operation can be represented by the interfaces presented in Figure 3(a) where `a` and `d` represent the opening and closing of the transaction, respectively, and `b` and `c` represent the reading and writing operations, respectively. In this particular case, the requester always uses the write operation immediately after the read, while the provider permits reads and/or writes in any order. Clearly, both of these interfaces are regular: the requester interface, $\mathcal{M}_i$, implements the interface language defined by the regular expression $(\mathtt{a}(\mathtt{bc})^*\mathtt{d})^*$ and the provider $\mathcal{M}_j$ implements the regular language as defined by the regular expression $(\mathtt{a}(\mathtt{b}|\mathtt{c})^*\mathtt{d})^*$. A simple inspection of the two languages reveals that the provider interface language subsumes the requester interface language: $\mathcal{L}(\mathcal{M}_i) \subseteq \mathcal{L}(\mathcal{M}_j)$, and the two components are compatible. Using a more formal technique to determine compatibility, an automaton that recognizes the intersection of $\mathcal{L}(\mathcal{M}_i)$ with $\overline{\mathcal{L}(\mathcal{M}_j)}$ can be constructed using the product construction technique. It can be shown that this results in a five-state finite automaton (excluding the "error" state and non-reachable states) in which there are no accepting states reachable from the initial state, thereby showing that the composition is compatible.

Another way to demonstrate compatibility is to compose the two interfaces as described in Section 3.2. The resulting composed net is shown in Figure 3(b). Reachability analysis of this composed net shows that there are fifteen reachable markings, none of which is dead, thereby demonstrating that the net is deadlock-free and the components are compatible. This can also be verified by performing structural analysis of the net, which shows that there are 64 minimal siphons, none of which can be emptied of their tokens, again implying that the composed net is deadlock-free and the components are indeed compatible.

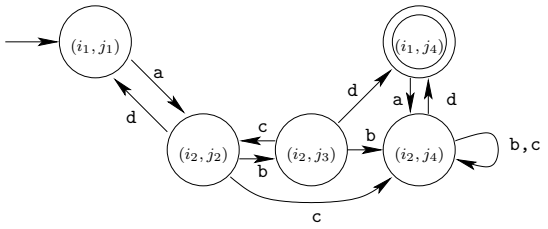
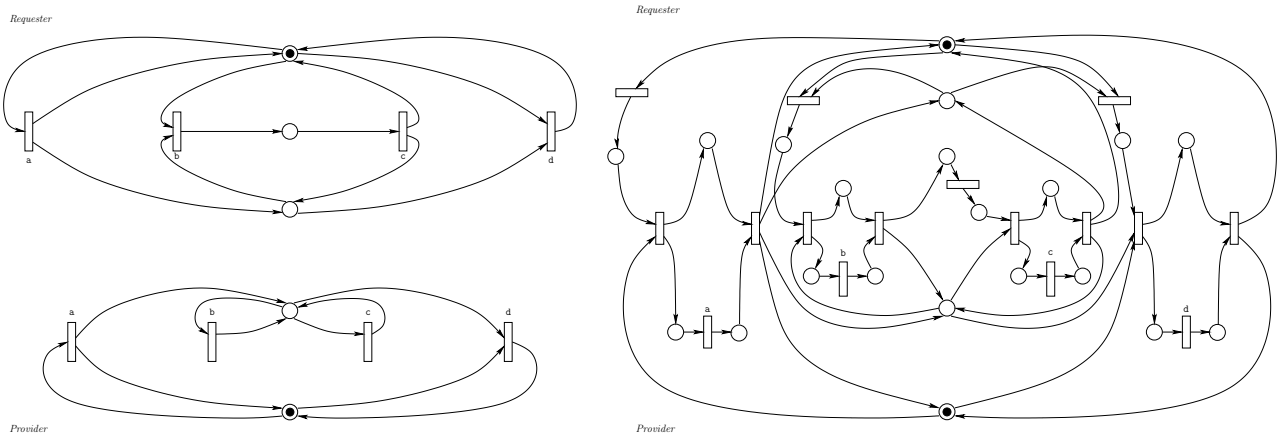


Figure 4. Automaton recognizing $\mathcal{L}(\mathcal{M}_i) \cap \overline{\mathcal{L}(\mathcal{M}_j)}$

If the requester and the provider are transposed and recomposed so that the interface language of the requester, $\mathcal{L}(\mathcal{M}_i)$, becomes the language denoted by the regular expression $(\mathtt{a}(\mathtt{b}|\mathtt{c})^*\mathtt{d})^*$ and the interface language of the provider, $\mathcal{L}(\mathcal{M}_j)$, becomes the language denoted by the expression $(\mathtt{a}(\mathtt{bc})^*\mathtt{d})^*$, then an incompatibility results — the language of the requester is now a proper superset of the language of the provider and the requirement for compatibility, $\mathcal{L}(\mathcal{M}_i) \subseteq \mathcal{L}(\mathcal{M}_j)$, is no longer satisfied.

This can be verified by the construction of an automaton that recognizes $\mathcal{L}(\mathcal{M}_i) \cap \overline{\mathcal{L}(\mathcal{M}_j)}$, as shown in Figure 4. (Note that the "error" state and non-reachable states are not shown.) The automaton has an accepting state reachable from the initial state which implies that $\mathcal{L}(\mathcal{M}_i) \cap \overline{\mathcal{L}(\mathcal{M}_j)} \neq \emptyset$ and $\mathcal{L}(\mathcal{M}_i) \not\subseteq \mathcal{L}(\mathcal{M}_j)$ (for example, the automaton accepts $\mathtt{abd} \in \mathcal{L}(\mathcal{M}_i)$ but $\mathtt{abd} \notin \mathcal{L}(\mathcal{M}_j)$). Because the requester can demand a sequence of service operations that are beyond the capabilities of the provider, the two components are not compatible.

As another verification of incompatibility, the two interfaces can be composed using the composition technique de-



(a) Non-regular interfaces before composition

(b) Non-regular interfaces after composition

Figure 5. Composition of non-regular interfaces

scribed above. As expected, this results in a Petri net that contains a deadlock. Reachability analysis shows that there are eighteen possible reachable markings, three of which end in deadlock. One of the deadlocks occurs when the requester attempts to perform a write operation immediately after opening the transaction. This sequence is forbidden by the provider. The presence of a deadlock can also be verified through structural analysis.

As an example of components with non-regular languages, consider the two interfaces shown in Figure 5. These two interfaces are similar to those presented in Figure 3, except that they allow for an arbitrary nesting of the open/close transaction pairs (*i.e.*, the service operations `a` and `d` may nest but will always occur in pairs).

Because these languages are not regular, finite automaton and product construction techniques cannot be used. Furthermore, because the composition of interface nets results in an unbounded net, reachability analysis cannot be used either. As a result, structural analysis of the composed net must be performed to determine if these two components are compatible. There are a total of sixteen minimal siphons in the composed net, shown in Figure 5(b), and it can be demonstrated, using linear programming, that the minimum number of tokens in each of these siphons is always greater than zero, therefore implying the deadlock-freeness of the net and, consequently, verifying the compatibility of the two components.

5. Conclusions and Future Work

Verifying the compatibility of two or more components is an important area that must be addressed if the subsequent integration of software components is to be successful. This work described a formal mechanism to test for the interoperability of software components by considering the sequences of services that are requested and provided at a component's interface. By treating the interfaces as Petri nets and the sequences of services as formal languages, the compatibility of the components involved can be assessed. When the languages are regular, automata theory can be used to determine interoperability; non-regular languages require more sophisticated techniques involving a formal composition of the Petri nets that represent the interfaces and subsequent analysis of the resulting net for deadlock.

For this technique to be useful, vendors of software components must be willing to distribute appropriate information regarding the implementation of their software from which a Petri net can be obtained for verification of compatibility. Some vendors may be reluctant to disclose anything other than the minimal application programming interface of their software component libraries. They may fear that disclosing too many internal details would reveal too much of the underlying behaviour of their components, thereby possibly compromising the commercial viability of their proprietary component library. However, if the software is based upon well established protocols with well known pre- and post-conditions, this may not be a considerable obstacle as the interface nets would be identical across several different vendor implementations. Of course, this is not a problem for free and open source software.

Other specifications for component interfaces, including NETCONF [13] may also be possible. However, it is not immediately clear if such a specification would lead to the same properties as exhibited by the Petri net approach demonstrated by this paper, particularly with respect to the equivalence of deadlock-freeness and compatibility. Further investigation would be necessary.

While this approach can be used to ensure that the se-

quences of operations between two components are compatible, further work is needed to ensure that subtler aspects related to component interoperability are also satisfied before optimal reuse and automated software assembly can be realized. Compatibility issues related to the lower-level semantics of the operations as well as the underlying communication protocols will also have to be addressed at some point. Also note that this work does not adequately deal with issues related to the discovery and deployment of software components in a system's runtime environment. This work may also be applied to other areas outside of the construction of traditional large-scale software systems. For example, the composition model presented here may also be adapted to handle web services interoperability [16] or service-oriented architectures. In the area of embedded systems, precise timing information between the invocation and completion of services is vital to the reliability of these systems. While this model does not support temporal attributes, it can be adapted for this purpose by using timed Petri nets [22].

The approach presented in this paper is an initial, but important step in the advancement of the design and construction of large-scale software systems. Assessing compatibility of larger, more complex systems may also be possible if composed nets are reduced by removing redundant net elements. Quantitatively investigating the level of complex interaction between components that can be successfully analyzed by the approach presented in this paper is worthy of future consideration. Establishing a well defined and formal method for verifying the interoperability of two or more components can serve to enhance reuse of software components in a given software architecture. This may facilitate the development of complex software systems in the future.

Acknowledgment

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

References

[1] S. Albin. *The Art of Software Architecture: Design Methods and Techniques*. Wiley Publishing Inc., 2003.

[2] K. Beck. *Extreme Programming explained : Embrace Change*. Addison-Wesley, 2000.

[3] B. Boehm and R. Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley Pearson Education, 2004.

[4] E. Boer and T. Murata. Generating basis siphons and traps of Petri nets using the sign incidence matrix. *IEEE Transactions on Circuits and Systems — I: Fundamental Theory and Applications*, 41(4):266–271, Apr. 1994.

[5] A. W. Brown. *Large-Scale Component-Based Development*. Prentice Hall, 2000.

[6] D. Bugden. *Software Design (2^{nd} edition)*. Pearson Addison-Wesley, 2003.

[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[8] D. G. C. Szyperski and S. Murer. *Component Software: Beyond Object-Oriented Programming (2^{nd} edition)*. Addison-Wesley, 2002.

[9] F. Chu and X. Xie. Deadlock analysis of Petri nets using siphons and mathematical programming. *IEEE Transactions on Robotics and Automation*, 13(6):793–804, Dec. 1997.

[10] J. Colom and M. Silva. Improving the linearly based characterization of p/t nets. In *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 113–145. Springer-Verlag, 1990.

[11] R. Cordone, L. Ferrarini, and L. Piroddi. Enumeration algorithms for minimal siphons in Petri nets based on place constraints. *IEEE Transactions on Systems, Man, and Cybernetics — Part A: Systems and Humans*, 35(6):844–854, Nov. 2005.

[12] D. Craig. *Compatibility of Software Components — Modeling and Verification*. PhD thesis, Memorial University of Newfoundland, 2006.

[13] R. Enns. NETCONF Configuration Protocol. Internet Draft, `http://www.ietf.org/rfc/rfc4741.txt`, December 2006.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[15] R. M. J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation (2^{nd} edition)*. Addison-Wesley, 2001.

[16] 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.

[17] S. Melzer and J. Esparza. Checking system properties via integer programming. In *Programming Languages and Systems — ESOP'96*, volume 1058 of *Lecture Notes in Computer Science*, pages 250–264. Springer-Verlag, 1996.

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

[19] I. Sommerville. *Software Engineering (6^{th} edition)*. Addison-Wesley, 2001.

[20] A. Valmari. The state explosion problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.

[21] M. Weber and E. Kindler. The petri net markup language. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Petri Net Technology for Communication-Based Systems*, volume 2472 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 2003.

[22] W. Zuberek. Timed petri nets – definitions, properties and applications. *Microelectronics and Reliability (Special Issue on Petri Nets and Related Graph Models)*, 31(4)::627–644, 1991.