Int. Conf. on Dependability of Computer Systems, Szklarska Poreba, Poland, 25-27 May 2006, pp.11-18.
Copyright © 2006 IEEE (DOI 10.1109/DEPCOS-RELCOMEX.2006.13).

Compatibility of Software Components – Modeling and Verification

D.C. Craig and W.M. Zuberek

Department of Computer Science
Memorial University, St. John's, Canada A1B 3X5
{donald,wlodek}@cs.mun.ca

Abstract: Component-based software engineering (CBSE) has been emerging as a promising approach to the development of large-scale software architectures in which software components with well-defined interfaces can be quickly assembled into complex software systems. However, assembled components must be compatible in the sense that any sequence of operations requested by one of the interacting components must be provided by the other component(s). Component incompatibility may result in subtle software failures which are difficult to detect and correct. A formal model of component interaction is proposed by representing component behaviors by labeled Petri nets. These net models are designed in such a way that component incompatibility is manifested by deadlocks in the net model of interacting components. Reachability-based as well as structural methods of deadlock detection are discussed. A simple example illustrating the proposed approach is provided.

Keywords: software architecture, software components, component compatibility, Petri nets, deadlock detection.

1 Introduction

Numerous strategies have been proposed to address the difficulties and challenges involved in the development of large-scale software systems [3]. Object-oriented programming [8] and ADLs (Architecture Descriptions Languages) [14] have been introduced to mitigate the complexities of development and integration of dependable software systems.

The concepts of systematic software decomposition into modules and reuse of modules have been aggressively promoted as a means to handle the complexity inherent in the design and implementation of software systems [15, 16, 19]. Unfortunately, while this approach was initially very appealing, the design and implementation of modules that are simultaneously generic and useful, can be challenging. Also, truly generic software entities can be very difficult for software designers to efficiently deploy, thereby limiting the advantages gained by module reuse.

Design patterns [4] have been proposed as an attempt to make software development more template-oriented. Design patterns originate from recognizing the frequent occurrence of similar design structures across several successful software systems. These design structures can then be generalized and documented, thereby creating a library of patterns. These patterns, once shared with the development community, can then be adapted and reused for similar problems in other domains. For example, the *Composite* pattern can be easily adapted to represent the hierarchical composition of graphic elements in a visualization product or can be used to represent the hierarchical composition of hardware components in a CAD package. While design patterns can, in theory, transcend all levels of software representation, they are most commonly employed in the context of object-oriented and component-based software development [8].

Scenario-based software analysis has also met with some success in the comprehension and maintenance of software systems [11]. In this strategy, the various activities that the system is required to support are identified. These so-called "system uses" are developed from the perspective of both the different end-users and the developers of the systems. By analyzing software from these two perspectives, multiple views of the system can be derived and studied. Scenarios can be used to determine whether an existing software system successfully satisfies its qualitative requirements in domain specific areas. High degrees of coupling and low degrees of cohesion, both of which can negatively impact the design of a system, can also be found by identifying locations in the software where scenario interaction and interdependence are at their greatest.

Somewhat related to this is the method of Aspect Oriented Programming (AOP) [7]. Under this paradigm, functionality that is employed by several software subsystems is identified as *cross-cutting concerns*. For example, functionality that involves writing diagnostic or debugging information to a file or database would be regarded as a cross-cutting concern since it has the potential to be used by a large number of subsystems. Other cross-cutting concerns can involve aspects related to authentication and database transactions. AOP involves the identification of locations in the code base where cross-cutting concerns or aspects arise (these locations are called *join points*) and the injection of appropriate code segments that implement the aspects into those join points. This injection of code, called *weaving*, is most effectively done automatically by software tools.

Agile methods [2] are also becoming more relevant in both research and industry. Agile methods deemphasize the *predictive* nature of the traditional waterfall software life-cycle in favour of a more *adaptive* style of software development which can more readily contend with rapidly changing requirements. This style promotes more frequent releases of code, greater collaboration with the intended consumers of the software and greater communication between the software developers themselves. As a result, agile methods appear to be most effective in relatively small, collocated teams of about a dozen developers. One of the more successful variants of the agile methodology is *extreme programming* [1, 18], which emphasizes the notions of pair-programming, test-driven development, unit testing and continuous integration of software, amongst many other aspects.

Currently, concepts related to *software architecture* [9] are becoming more widespread in addressing the complexity associated with software development. Software architecture attempts to tie together many of the more recent trends in software developments, including object–oriented design patterns and scenario–based software analysis. Software architecture uses *components* as the fundamental building blocks of software systems.

Components can be considered as the primary functional units and the fundamental data types in architectural designs. The connections between components serve to determine the flow of control and to provide a context or environment for the components. Components are a means of representing a high–level software models; they must be generic enough to work in a variety of contexts and in cooperation with other components, but also specific enough to provide easy reuse.

This paper provides a foundation for a formal model of component interaction by representing the behaviour of components by their interface languages, *i.e.*, sets of all possible sequences of services (required or provided). These languages are modelled by labelled Petri nets. Component compatibility is established by determining those components which, when connected, are free of deadlock.

Section 2 introduces Petri net models of components. Composition of components and verification of component compatibility is discussed in Section 3, while Section 4 illustrates the proposed approach by a simple example of component composition. Section 5 concludes the paper.

2 Component models

Informally, a component is 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. Several attempts have been made to formally define a component and its behaviour, some of which made extensive use of Petri nets [21].

For the purpose of component compatibility, internal details of components can be disregarded, and the component's behaviour can be represented at its interface(s). It is believed that all essential effects of component's internal behaviour can be satisfactorily represented at the component's interface, significantly simplifying the model.

2.1 Petri net models

A component's behaviour at its interface is represented by a labelled Petri net:

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

where P_i and T_i are disjoint sets of places and transitions, $A_i \subseteq P_i \times T_i \cup T_i \times P_i$ is a set of directed arcs, S_i is an alphabet representing a set of services which are associated with transitions by the labelling function $\ell_i : T_i \to S_i \cup \{\varepsilon\}$ (ε is the "empty" service; it labels transitions which do not represent services provided or requested by an interface), and m_i is the initial marking function $m_i : P \to \{0, 1, ...\}$.

In order to represent component interactions, the interfaces are divided into *provider* interfaces (or p–interfaces) and *requester* interfaces (or r–interfaces). In the context of a provider interface, a labelled transition can be thought of as a service provided by that component; in the context of a requester interface, a labelled transition is a request for a corresponding service. In both cases, Petri net models of a component's behaviour are designed in such a way that the sequences of possible services correspond to the firing sequences of (labelled) transitions. For example, the model shown in Fig.1(a) can be a model of a provider interface, and that shown in Fig.1(b), a requester interface.

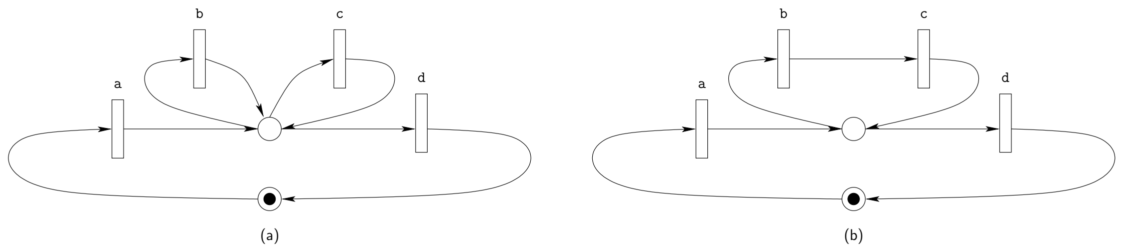

Figure 1: Example provider and requester interfaces

It is required that in each p–interface there is exactly one labelled transition for each provided service:

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

The label assigned to a transition represents a service or some unit of behaviour. For example, the label can represent a conventional procedure or method invocation. It is assumed that if the p–interface requires parameters from the r–interface, then the appropriate number and types of parameters by be delivered by the r–interface. Similarly, it is assumed that the p–interface provides an appropriate return value, if required. The equality of symbols representing component (requested and provided) services implies that all such requirements are satisfied.

Petri net models of component interfaces must be deadlock–free. Typically, models of component interfaces are cyclic nets, as shown in Fig.1.

2.2 Component compatibility

Component's behaviour can be represented by the set of all possible sequences of services (required or provided by a component). Such a set of sequences is called the *interface language* [6], and, for an interface modelled by a net $\mathcal{M}$, is denoted by $\mathcal{L}(\mathcal{M})$. Interface languages of interacting components can be used to define the compatibility of components; a requester component $\mathcal{M}_i$ is compatible with a provider component $\mathcal{M}_j$ if and only if all sequences of services requested by $\mathcal{M}_i$ can be provided by $\mathcal{M}_j$, *i.e.*, if and only if $\mathcal{L}(\mathcal{M}_i) \subseteq \mathcal{L}(\mathcal{M}_j)$.

Component models shown in Fig.1(a) (p–interface) and Fig.1(b) (r–interface) are compatible; the requester language can be described by a regular expression `(a(bc)*d)*` and the provider language by `(a(b+c)*d)*`.

3 Component composition

Component composition and compatibility assessment of Petri net models has been studied in the past [17, 12]. Related to this area is the composition and interoperation of web services [13] and verification of workflow composition [20].

Since the interface languages of interacting components are usually represented by net models (for example, developed from component specifications), component compatibility should be verified using such models. A composition of net models is used for this purpose.

3.1 Compositions of interfaces

Informally, the composition of component interfaces is performed by "merging" an r–interface $\mathcal{M}_i$ with a corresponding p–interface $\mathcal{M}_j$ into a single net $\mathcal{M}_{ij}$, assuming $P_i \cap P_j = \emptyset$ and $T_i \cap T_j = \emptyset$. The composition is formally defined in [6], and is outlined in Fig.2.

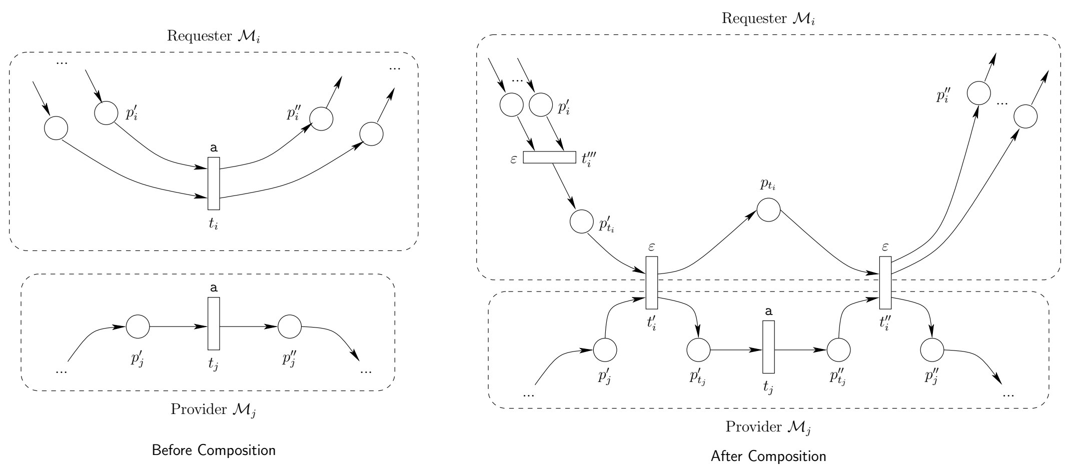

Figure 2: Composition of an r–interface and p–interface

The composition of a requester and a provider interfaces introduce two new places (p_{t_i}, p'_{t_i} in Fig.2) and three transitions (t'_i, t''_i, t'''_i in Fig.2) for each service request in the r–interface. The transition-place pair t'''_i and p'_{t_i} allows the requester to initiate the interaction with the provider and to direct the ensuing sequence of operations. The other place (p_{t_i}) and transitions (t'_i and t''_i) serve to coordinate the requester's interaction with the provider at the service point. The place p_{t_i} constitutes an *implicit* place. Generally, during net analyses, this place can be removed without adversely affecting the underlying behaviour of the net.

Composition of several requester interfaces with a single provider interface is outlined in Fig.3. In the case of concurrent requests, one service request is selected randomly but priorities can easily be implemented by additional elements of the model.

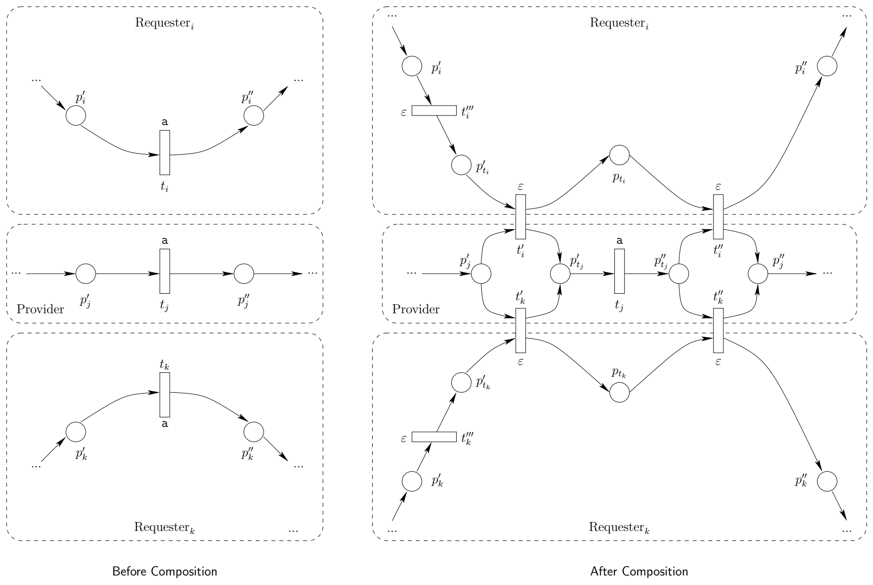

Figure 3: Composition of multiple r–interfaces with a p–interface

3.2 Compatibility verification

It can be shown that components are compatible if and only if the composition of component models is free of deadlocks [6]. Component compatibility verification can thus be performed by checking deadlock freeness in the composed net models.

There are two basic approaches to deadlock detection in Petri nets, one uses reachability analysis and the other is based on structural properties of nets. The reachability–based approach performs exhaustive analysis of the space of reachable markings, looking for markings with the empty set of enabled transitions. The approach is quite straightforward, but it is effective only for models with relatively small spaces of reachable markings. Structural methods determine the existence or absence of deadlocks on the basis of subnets of the original net and the initial marking function. In particular, it is known that an unmarked siphon (*i.e.*, such a subset of places for which the set of their input transitions is a subset of the set of their output transitions) is a necessary condition for a deadlock [5]. Therefore, once the set of (proper) net siphons is determined, and all these siphons are marked by the initial marking function, what remains to be checked is if there exists a firing sequence of transitions for which a siphon becomes unmarked. Linear programming can be used for this purpose [6]. More specifically, for each proper siphon, linear programming can be used to find a firing vector which reduces the token count in the siphon to zero. If such a vector exists, it needs to be verified for feasibility, *i.e.*, if, for the given initial marking function m_0, there is a firing sequence which corresponds to the firing vector determined by linear programming. If there such a sequence exists, a deadlock can occur in the composed model, so the components are incompatible.

4 Example

This examples considers the composition of a database provider interface with an interface requesting the services of the database. The interfaces are the same as shown in Fig.1. In this particular context, the a and d operations could represent services that open and close the database, respectively; the b and c operations could represent services that read from and write to database, respectively. The composition of the two interfaces is shown in Fig.4.

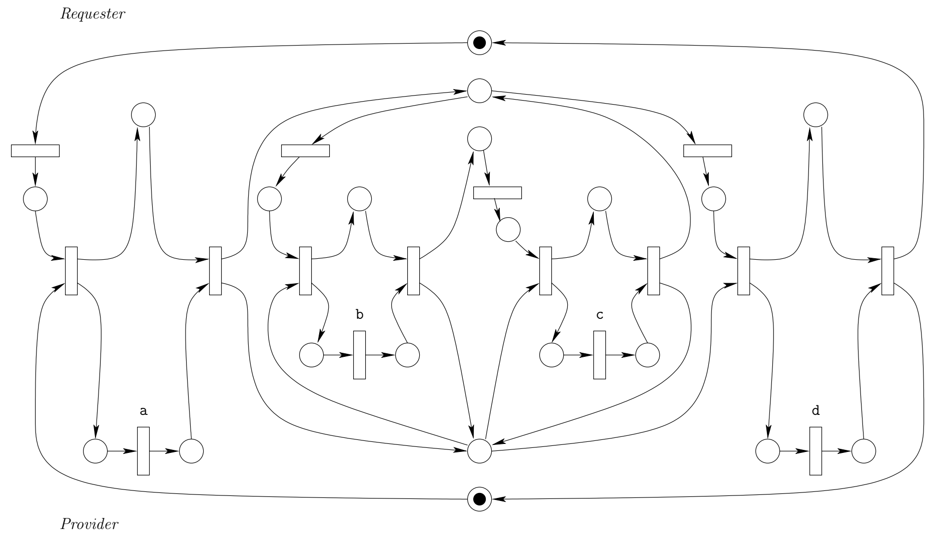

Figure 4: Compatible composition

This composed net contains fifteen reachable markings, none of which results in deadlock, thereby verifying the compatibility of the two component interfaces. This is expected intuitively, because the requester performs each write operation only after doing a corresponding read operation, while the database server imposes no order on the sequences of read and write operations; the language of the r–interface is a subset of the p–interface's language.

Switching the models of requester and provider interfaces, *i.e.*, using model shown in Fig.1(a) as an r–interface, and that from Fig.1(b) as a p–interface, results in a composed model in which deadlocks can occur; Fig.5 shows the composed net and one of its (three) deadlocks.

The firing sequence which leads to the deadlock shown in Fig.5 is (t_1, t_5, a, t_6, t_3). In this case, the requester demands a sequence of operations that cannot be satisfied by the provider, so the two components are not compatible. The deadlocks can be identified by using either reachability analysis (there are 17 reachable markings) or by using structural techniques and linear programming, as described in Section 3. In this second case, one of the siphons that become unmarked by the firing sequence (t_1, t_5, a, t_6, t_3) is $\{p_1, p_5, p_3, p_2, p_6, p_7, p_{15}, p_9, p_{10}, p_4\}$. The other deadlocks are reached by firing sequences $(t_1, t_5, a, t_6, t_2, t_7, b, t_8, t_4)$ and $(t_1, t_5, a, t_6, t_2, t_7, b, t_8, t_2)$.

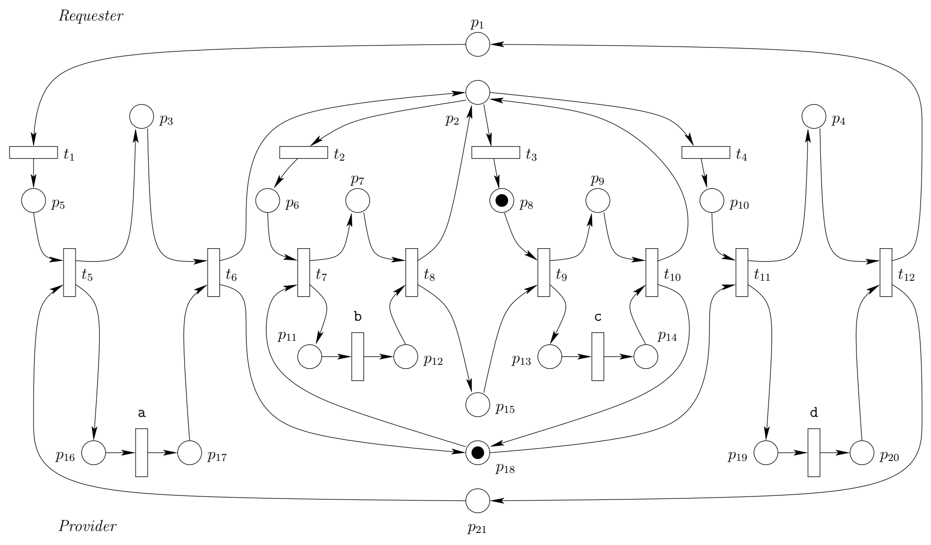

Figure 5: Incompatible composition resulting in deadlock

5 Concluding Remarks

Component compatibility is a multifaceted problem which requires a comprehensive understanding of the static as well as dynamic nature of components involved. However, abstracting the internal, low–level behaviour of components, and concentrating solely upon the behaviour exhibited at component interfaces, makes it possible to (formally) verify the compatibility of components by checking if the composed model is free of deadlocks.

For small models, reachability analysis of the composed model seems to be the simplest approach to deadlock detection. For larger models as well as for unbounded nets, structural analysis should be used, however, more efficient methods for structural analysis are needed as the currently known ones are not really efficient.

The proposed approach can be extended in many ways, for example, temporal characteristics of components can be included into net models and used for performance analysis [22]. Similar components could be "folded" by introducing token attributes, as in colored Petri nets [10].

The strategy described in this paper represents an initial, but important step in the continuing evolution of the design and construction of dependable software systems. Establishing a well defined and formal method for determining the extent to which two or more components are able to successfully interact can serve to significantly enhance reuse of software components in a given software architecture. Ultimately, this may contribute to the reliable evolution of a deployed component-based software system.

Acknowledgements

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

References

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

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

[3] D. Bugden. *Software Design*. Pearson Addison-Wesley, second edition, 2003.

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

[5] 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, 1997.

[6] D.Craig. *Compatibility of Software Components — Modelling and Verification*. Ph.D. Thesis. Memorial University of Newfoundland, 2005.

[7] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, Inc., 2000.

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

[9] D. Garlan and D. E. Perry. Introduction to the special issue on software architecture. *IEEE Transactions of Software Engineering*, 21(4):269–274, Apr 1995.

[10] K. Jensen. Coloured Petri nets. In *Advanced Course on Petri Nets 1986*, volume 254 of *Lecture Notes in Computer Science* pages 248–299. Springer-Verlag, 1987.

[11] R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario-based analysis of software architecture. *IEEE Software*, 13(6):47–55, Nov 1996.

[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] N. Medvidovic and R. 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.

[15] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec 1972.

[16] D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, 11(3):259–266, Mar 1985.

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

[18] D. H. Steinberg and D. W. Palmer. *Extreme Software Engineering: A Hands-on Approach*. Pearson/Prentice Hall, 2004.

[19] P. Tonella. Concept analysis for module restructuring. *IEEE Transactions of Software Engineering*, 27(4):351–363, Apr 2000.

[20] W. 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 Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 161–183. Springer-Verlag, 2000.

[21] W. van der Aalst, K. van Hee, and R. 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.

[22] W.M 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.