COMPATIBILITY OF SOFTWARE COMPONENTS - MODELLING AND VERIFICATION

DONALD C. CRAIG

Compatibility of Software Components — Modelling and Verification

by

© Donald C. Craig

A thesis submitted to the
School of Graduate Studies
in partial fulfilment of the
requirements for the degree of
Doctor of Philosophy

Department of Computer Science
Memorial University of Newfoundland

May 2007

St. John's Newfoundland

Your file Votre référence
ISBN: 978-0-494-31312-1
Our file Notre référence
ISBN: 978-0-494-31312-1

Abstract

The area of Component Based Software Engineering (CBSE) is rapidly emerging as a means of mitigating the complexity faced by software architects during the design and maintenance of large software systems. Unfortunately, given the substantial number of components that may be deployed in a given software architecture, successfully establishing compatible interaction amongst components can be a difficult problem to solve. The purpose of this work is to show that compatibility between components may be determined by developing a formal model to describe component interfaces and their behaviour. In addition to promoting reuse and substitutability in the design and maintenance of software systems, this approach may also have a significant effect on the reliability and trustworthiness of software systems.

At a fundamental level, a component can be regarded as a cohesive logical unit of abstraction with well-defined interfaces that provide services to its environment or request such services. This work sets the foundation for a formal model of component composition by using Petri nets to represent the behaviour of component interfaces. Compatibility is established by verifying that interfaces can satisfy all requested sequences of operations. The *requires* and *provides* relationships are discussed in the context of formal languages generated by the corresponding labelled Petri net models. The compatibility of the interfaces is determined by examining various structural and reachability properties of the net obtained by the composition of the interfaces.

As commercial components become increasingly available and the web services industry becomes more vibrant, formal compatibility assessment is an important step toward the construction of large, distributed software systems.

Acknowledgements

I would like to thank Dr. Wlodek Zuberek for his supervision during my program. His guidance, patience and support were crucial for the completion of this thesis. The financial support from the *Natural Sciences and Engineering Research Council of Canada* (NSERC) throughout my research is also gratefully acknowledged. I am also grateful to Drs. Xiaobu Yuan and Miklós Bartha for agreeing to serve on my supervisory committee and for their continuous encouragement and advice.

Thanks also to Dr. Paul Gillard for encouraging me to take on the challenge of the Ph.D. program here at Memorial and for helping me secure funding during my initial years of study. I would like to express my gratitude to the School of Graduate Studies for their generous financial support throughout my program.

I would also like to thank the members of my oral comprehensive examination committee, including Drs. Tony Middleton, Theo Norvell and Brad de Young. They offered constructive suggestions and useful advice which helped me focus my research. The examiners of this thesis, Drs. Adrian Fiech, Ryszard Janicki and Krishnamurthy Vidyasankar provided many helpful and constructive suggestions. I am grateful to them for their thorough analysis of my work and their detailed recommendations. I believe that their contributions have helped strengthen this work considerably.

I am also indebted to the current department head Dr. Wolfgang Banzhaf, former interim Computer Science Department head Mrs. Jane Foltz, Miss. Elaine Boone and the rest of the administrative staff and faculty of the Department of Computer Science for their continuing support and assistance over the past several years.

Finally, I would like to thank my fellow graduate students for their companionship and moral support over the years — I wish them all the best of luck in their programs and future endeavours.

Contents

List of Tables

List of Figures

To my parents

Chapter 1

Introduction

1.1 Motivation

The challenges and difficulties associated in the development of large-scale software projects are well documented [96, 111] as are the analyses of project failures [36]. Over the years, numerous strategies have been developed to help mitigate these difficulties. Object-oriented programming [12] and numerous architectural description languages [71] have been introduced in order to make the development of software systems more tractable. During recent years, component based software engineering (CBSE) [16] has been emerging as viable means of software construction whereby pre-manufactured software structures with well-defined interfaces are designed and implemented, and subsequently incorporated into larger software systems [45]. While this approach has met with some degree of success, there remains the problem of determining compatibility between components.

In his provocative paper, *No Silver Bullet — Essence and Accident in Software*

Engineering [14], Frederick Brooks Jr. identified four essential difficulties that impede the construction of large software systems: complexity, conformity, changeability and invisibility. Of these four so-called "essences," the problem of *complexity* is often regarded as the most difficult to address and subsumes the other three.

Attempts to address the complexity inherent in the software design process have met with mixed success. For example, visual programming languages and environments [95, 113] attempt to allow software designers to model software the same way hardware designers create circuits. However, because the design and creation of software is very much a mental exercise, completely accurate physical representation is not possible, thereby limiting the scope of problems that can be solved by visual programming techniques.

Artificial intelligence and expert-systems [66] have also been presented as possible answers to the problem of complexity. However, until we, as software designers, are able to justify all the reasons *why* a particular software design is more suitable than another *and* until we are able to enumerate those reasons as a series of logical rules suitable for consumption by a machine, AI and expert systems approaches will only be applicable in the design of specific domain systems. Expert human designers acknowledge that there is a certain level of art in the construction of well-designed software systems; extracting and distilling these qualitative features and representing them concisely and accurately has proven to be elusive.

While the advent of high-level languages has produced significant productivity gains in the area of software implementation and deployment, high-level languages have not contributed significantly to a reduction in the inherent complexity of software analysis and design. As designs of modern day software systems have grown increasingly complex, popular implementation languages such as Java, C and C++ are still relatively low-level when compared to the high-level abstractions necessary to solve nontrivial problems.

Consequently, a formal, high-level approach towards the composition of software subsystems would lessen the development burden on software architects and thus be amenable to the construction of larger, more complex software systems. In particular, a formal model to describe the composition of software entities and to determine their resulting compatibility is crucial in the construction of large software systems. This work is an attempt to establish a foundation for such an approach.

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 or more software components. For example, it is possible for a server component to provide methods that exactly match the static 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.

1.2 Research Objectives

The primary goal of this research is to provide a formal model of component interaction by representing the behaviour of components at their interfaces using Petri nets [78, 86]. Interface compatibility is established by determining those interfaces which can satisfy all possible sequences of requested operations. The "requires" and "provides" relationships are discussed in the context of formal languages generated by the corresponding Petri nets in a component's deployment environment. By analyzing the structural and dynamic properties of the Petri net representing the composition of the components' interfaces, compatibility between components is tested and verified.

Of particular importance in the development of software systems is whether two separate software modules, one of which relies on the services of the other, can successfully interact with one another to fulfill their requirements. One of the primary objectives of this work is to provide a formal definition of compatibility in terms of the languages manifested by the interfaces. With this objective satisfied, a formal means to actually assess or verify that two or more components satisfy this property can then be found by composing the nets and showing the resulting net is free of deadlocks. Multiple strategies can be used for deadlock detection, depending upon the structural and behavioural properties of the composed net; this work describes the advantages and disadvantages of each method. To help mitigate some of the complexity associated with the analysis of the composed net, various net reduction algorithms are proposed to limit the number of the elements in the net. More complicated interactions between multiple providers and multiple requesters are also discussed both formally and with examples.

This work is not focused on decomposition strategy of software design nor is it focused on the actual construction of the atomic elements themselves. Rather, the goal of this work is to facilitate the determination that two or more software entities can successfully be composed to achieve a useful goal in the context of a larger software system. Ultimately, it is hoped that this work may provide the necessary infrastructure to build autonomous self-assembling software systems that may evolve independently.

1.3 Practical Implications

Initially, the implications of this research should increase the reliable construction of software systems. From an industrial perspective, software integration and reuse are two primary challenges facing the pragmatic construction of large-scale software systems. During the early stages of development, components should be reused as a cost-saving measure so as to reduce re-implementation of commonly used functionality. Compatibility assessment can help determine whether a pre-existing software entity can be reused in a particular environment. Later, when independently developed software modules are integrated to form larger software architectures, it is important to assess the compatibility of these modules so that the developers can be assured that the software structures are able to communicate effectively with each other. The strategy presented in this work may be able to provide quantitative metrics which can be used to assess compatibility of software components.

In the area of software maintenance, substitutability allows upgrades of software systems already deployed in a production environment. Whether new components are acting as traditional clients or servers in a multi-tier architecture or the components are in a peer network of components, it is vital that they are able to operate correctly in their deployed environment. Again, compatibility assessment is crucial in this area.

Eventually, this research may help facilitate the further advancement of self-organizing applications [34] in which applications evolve to adapt to changing environments or requirements. This could lead to the construction of software systems that can dynamically reconfigure themselves to adapt to changing conditions as dictated by their context. Closely related to this is the notion of *software evolution* [22] in which sub-elements of software projects are replaced over time to satisfy new demands placed upon a software system. The feasibility of such systems depends largely upon the dynamic integration of separate software entities, during which compatibility evaluation must be performed.

1.4 Outline of Thesis

Software development methodologies are briefly presented in Chapter 2 which also provides an overview of the concept of software architectures in general. Chapter 3 describes the concept of component-based software engineering, a software development strategy which is gaining wider acceptance in the construction of sophisticated software systems. This chapter also reviews some current component-based systems used in practice. The important features of this methodology, which form the basis of the remainder of the thesis, are emphasized. Various informal and formal definitions of components are presented in Chapter 4. Various properties related to Petri nets in general are also proposed as well as algorithms for net reductions and deadlock detection. Also in this chapter, a formal model of component interfaces that employs Petri nets is introduced. Moreover, the notion of interface languages is proposed and elaborated upon. Chapter 5 describes the different strategies that can be used to compose component interfaces and presents formal frameworks for establishing compatibility between two components. The proposed framework is based on the Petri net models of component interfaces and deadlock detection in the composed model. In Chapter 6, some examples that demonstrate the proposed approach are provided. Finally, Chapter 7 concludes the thesis and discusses future work.

Chapter 2

Software Development Methodologies

Prior to discussing software architectures in general, and component based systems in particular, this chapter provides context to the complexities of software development. The following section is devoted to a high-level review of the techniques used to address the complexities associated with the development of large-scale software systems. Numerous paradigms and methodologies have evolved to address inherent difficulties associated with the engineering of large software systems [18]. Recent strategies have attempted to raise the level of abstractions at which the software designer and, subsequently, implementer operate.

2.1 Background of Software Development Trends

By briefly studying the historical evolution of software representation, we can attempt to extrapolate future trends in software development. Representations of modern software originated with machine code, the lowest-level representation to which any other software representation is usually translated prior to execution. This representation of hardware operations and their corresponding arguments as a series of bits, while offering ultimate flexibility, is very error-prone as a development language. To counteract this deficiency, symbolic languages were created to represent hardware instructions and their arguments. While certainly less error-prone than raw machine-code, the symbolic languages, also referred to as *assembler* [53], offered limited advantages in terms of raising the level of abstraction — each symbol is essentially mapped directly onto a hardware instruction, there is typically no concept of data types and control flow is quite arbitrary.

With the advent of compilers, higher-level representations of software were possible. Programming languages such as FORTRAN use program statements that more closely model the corresponding mathematical domain. Each programming statement could conceivably be mapped to several low-level machine instructions, but the programmer is kept isolated from these details, thereby making the programming task easier. These programming languages also introduced a set of fundamental set of data types which could be easily manipulated by the programmer. The use of arbitrary flow of control was also discouraged in favour of more restricted looping constructs and function calls. This led to programs with a greater degree of structure and modularity and hence served to increase program comprehension.

Eventually, the importance of data encapsulation began to arise, in which the fundamental representation of data structures was concealed behind a well-defined interface. Access to a data structure's composite data elements could only be made indirectly via the interface. Consequently, the designer of the data type could change the internal representation without adversely affecting the users of the data type as the interface would remain the same. These so-called abstract data types (ADTs) [99] gave rise to object-based programming.

In an effort to promote the reuse of code, the concepts of classes and inheritance were introduced in languages such as SIMULA-67 and SMALLTALK. By allowing a derived class to inherit and extend the behaviour of a base class, developers are encouraged to extend existing classes rather than developing their own. Other features, such as run-time binding of function calls (polymorphism), further relieve the programmer from creating tedious and error-prone dispatch tables. The combination of data encapsulation, inheritance and polymorphism gave rise to object-oriented programming [12].

The trend described above suggests two simultaneous developments in the evolution of software representation. The first is an attempt to raise the level of abstraction by placing more responsibilities on the translation tools. Features such as type checking, exception handling and dynamic dispatch become the responsibility of the compile-time and run-time environments. These advances allow the developer to concentrate more fully on aspects directly related to solving a given problem. The second trend, and perhaps counter-intuitively, is that new software representations tend to be more restrictive or constrained than their predecessors. Arbitrary control flow is sacrificed in favour of more restricted looping or recursive constructs. Direct access to encapsulated data elements that implement a more complex data structure is prohibited in favour of using a more abstract interface instead. By imposing well-defined restrictions upon the data representations and control flow, the software representation as a whole becomes more tractable, less arbitrary and, therefore, more comprehensible for the developer.

The concepts of systematic module decomposition and reuse mentioned earlier have been aggressively promoted as a means to counteract the complexity inherent in the design and implementation of software systems [82, 83, 105]. Unfortunately, while this approach was initially very appealing, the effort required to design and implement a module that is simultaneously generic and useful, can be overwhelming. Also, truly generic software entities can be very difficult for software designers to efficiently deploy, thereby limiting the advantages gained by module reuse.

The importance of examining successful software systems and documenting their common design decisions also cannot be ignored in the evolution of the software development process. Such documentation has led to the creation of design patterns [21] which 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 [38].

Scenario-based software analysis has also met with some success in the comprehension and maintenance of software systems [54]. 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 [54].

Somewhat related to this is Aspect Oriented Programming (AOP) [26, 31]. Under this paradigm, functionalities that are employed by several software subsystems are 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 [10] are also becoming more relevant in both research and industry. Agile methods include *adaptive software development* [46], which strongly emphasizes the iterative nature of the development process while maintaining focus on the required feature sets. *Extreme programming* [7, 98] is another example of an agile method that has been successful in emphasizing the benefits of pair-programming, test-driven development, unit testing and continuous integration of software, amongst many other aspects. 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.

Currently, concepts related to *software architecture* are becoming more widespread as a means of addressing the complexity associated with software development [5, 40]. Software architecture attempts to tie together many of the more recent trends in software development, including object-oriented design patterns and scenario-based software analysis, in an attempt to make the production of large-scale software easier and more effective. Software architectures are discussed in more detail in next section.

The paradigms and methodologies described above by no means constitute an exhaustive list of all the practices in the software design and development field today; however they do provide an overview of current techniques which are being employed to facilitate design and implementation of software systems.

2.2 Software Architecture

Software architecture [1] represents an attempt to limit the complexity of software development by studying a software system at a very high-level of abstraction. Details regarding low-level abstractions such as APIs, protocols, algorithms and data structures, for example, are elided in favour of a more general view of the design. Software representation, consequently, is described using architecture description languages (ADLs), that offer a wider, more abstract view of software systems and may even be used to represent evolving software architectures [76].

When thinking of software architecture, it is sometimes useful to apply analogies with other domains where the concept of an architecture is better understood. For example, in the context of computer hardware, the architecture can be thought of as an interconnected collection of smaller functional entities (or *building blocks*). However, unlike physical architectural domains, a functioning software project does not have a physical manifestation. Therefore, many "real world" analogies relating to software architecture have been deemed to be inadequate and may actually misrepresent the numerous nuances associated with a software system [4].

2.2.1 Definitions

Over the years, many definitions of software architecture have been proposed. Indeed, the Software Engineering Institute (SEI), based in Carnegie Melon, currently lists in excess of one hundred definitions of "software architecture" at their website:

`http://www.sei.cmu.edu/architecture/definitions.html`

These definitions are almost always informal and quite broad. They attempt to offer guidelines, as opposed to rigid formal definitions, in an effort to establish a foundation for software architecture as a viable area of research and study.

In developing a definition for software architecture, Shaw and Garlan [91] identified several issues associated with the structure of a software system. These issues include: *component organization*, *global control structures*, *protocols*, *assignment of functionality to design elements*, *composition of design elements*, *physical distribution*, *scaling and performance*, *dimensions of evolution* and *selection among design alternatives*. They summarize these issues with the following description of software architecture [91]:

> "Abstractly, software architecture involves the description of elements from which systems are built, interactions amongst those elements, patterns that guide their composition, and constraints on these patterns. In general, a particular system is defined in terms of a collection of components and interactions amongst those components."

This definition, while certainly comprehensive, is probably overly ambitious. For example, the phrase "patterns that guide their composition, and constraints on these patterns" is probably better left to the domain of *architectural styles*. Architectural styles arise by applying the concepts of design patterns and idioms to software architecture. For example, architectural styles such as client/server, pipe-and-filter and blackboard architectures are commonplace in the software community, but a definition of software architecture itself should not limit itself by enforcing a particular pattern. A more succinct definition of software architecture was offered in a discussion group at the SEI during 1994 [40]:

> "The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time."

The above definition reprises of the concept of a component and their corresponding interactions. It also introduces the importance of the design and subsequent maintenance of software systems. Once a software system has been deployed, changes to its required operation are almost inevitable. For example, problems in the implementation have to be corrected, new requirements have to be satisfied and platform limitations have to be overcome. A definition of software architecture should consider the flexibility and extensibility of the software system.

Recently, more formal attempts have been made to derive a consensus on the definition of "architecture" as it applies to the software domain. For example, the Computer Society approved IEEE Standard 1471 which offers the following definition of an architecture [67]:

> "the fundamental organization of a system embodied in its components, their relative relations to each other and to the environment, and the principles guiding its design and evolution."

This definition is only a minor variation of the SEI definition offered in 1994. IEEE 1471 attempts to standardize neither processes nor architectural description languages. Instead, it attempts to build consensus regarding the definitions of various aspects associated with software architecture. In addition to the above definition, this IEEE Standard also provides a conceptual framework for software architecture which attempts to show architecture in the context of its entire environment. This framework is reproduced in Figure 2.1.

Important aspects of the conceptual framework are the notions of *view* and *viewpoints.* These attributes appear to be inspired by the scenario-based approach to software analysis and design. IEEE 1471 considers a *view* to be a collection of abstractions or representations (*i.e.*, *models*) that describe one particular aspect of a system. A *viewpoint* serves as a framework to establish common terminologies and

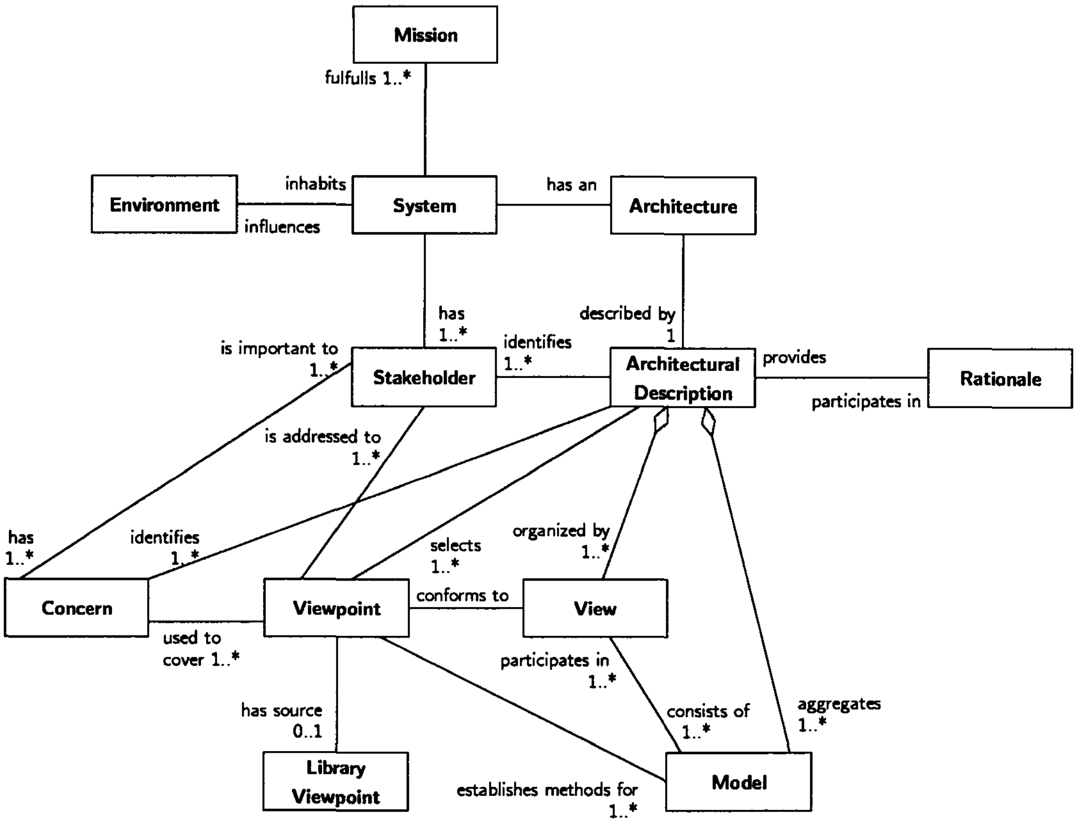

Figure 2.1: Conceptual framework for IEEE Standard 1471

notations upon which a view can be constructed.

Also of particular interest is the issue of *stakeholders* and *concerns*. Different stakeholders may have different requirements with respect to a software system. By enumerating the concerns of stakeholders and having them directly influence the architectural description, the system produced will more likely satisfy their demands. It should be noted that stakeholders not only include the end-users of a software system — developers and administrators of the system also have legitimate concerns relating to the extensibility and maintenance of the system.

One recurring theme that occurs in many definitions of software architecture is the concept of an entity, or, more commonly, a *component.* This concept will be elaborated upon in Chapter 3.

2.2.2 Architecture Representation

A software architecture can be described using an architecture description language (ADL) [39]. ADLs "usually provide both a conceptual framework and a concrete syntax for characterizing software architectures." As with software architecture, there is no formal definition as to what constitutes an ADL or what features a language must have in order to qualify as an ADL. Typically, however, ADLs must provide features to represent and analyze a software system at a high-level of abstraction.

Unfortunately, as we raise the level of abstraction, we cannot help but lose some of the precision afforded to us by more conventional and lower-level languages such as Java and C. Any tool capable of generating an implementation directly from an ADL must employ a certain degree of "intelligence" when translating a high-level abstraction to a low-level executable. Therefore, ADLs which allow for the automatic generation of a compliant implementation may choose to offer the designer access to lower-level abstractions so as to increase the viability of the generated code. Of course, having an unambiguous ADL whose semantics is well-defined can contribute significantly to the automatic generation of a software system.

A framework has been created for the classification and evaluation of ADLs [71]. In order to qualify as an ADL, a language should provide support for the specification of *components*, *connectors* and *configurations* (*i.e.*, topologies). With respect to components and connectors, an ADL would normally support the specification of attributes such as *interfaces*, *types*, *semantics*, *constraints*, and *evolution*. An ADL configuration should provide some degree of support for many different attributes, including *understandability*, *compositionality*, *constraints*, *evolution* and *dynamism*. Note that it is not necessary for a language to support all these attributes in order for it to be considered as an ADL. For example, *dynamism*, which allows for the insertion, removal and replication of architectural elements during run-time, is supported by relatively few ADLs. A toolset that supports the ADL can contribute significantly to the overall usefulness of the ADL. Such tools can work with the ADL to generate lower level code, provide architectural analysis and refinement, offer multiple architectural views and support dynamic execution or simulation of an architecture described by an ADL.

Note that architecture representations are not limited to ADLs. Indeed, some progress has been made in using the Unified Modelling Language (UML) to represent architectures [70]. Unfortunately, UML, which has traditionally been used in the design and analysis of object-oriented software systems, has not proven to be effective in modelling the nonfunctional aspects of an architecture. In particular, UML offers weak support for representing architectural constraints and explicit software connectors. Other nonfunctional aspects including portability, maintenance and configuration management can also be difficult to represent in UML. However, when used in conjunction with an existing ADL, UML diagrams may provide a more effective visual representation of a software architecture.

Attempts have also been made to create a mathematical basis for modelling large software systems [17]. Dynamic aspects of a software system can be modelled using heterogeneous algebras. Mathematical foundations offer a greater level consistency in architectural designs and may permit environments which are more amenable to simulation and verification strategies. However, the vocabulary of a rigidly formal mathematical model is often beyond that of a typical software architect, thereby discouraging strictly mathematical approaches to architecture description.

2.2.2.1 Overview of ADLs

Several ADLs have been described in the literature and each year, there are new developments in this area. Some of the more popular ADLs include Rapide, UniCon and Wright.

Rapide [64, 65] is an "event-based, concurrent, object-oriented language" for prototyping system architectures, particularly distributed systems. There are five major languages associated with a Rapide description. Interfaces to components are defined using a *types language*; the propagation of events throughout the collection of components is described by an *architecture language*; constraints on component behaviour are represented by the *specification language*; executable modules are described by the *executable language* and, finally, a *pattern language* is used to represent the various families of events. Rapide allows for the simulation of an architecture during which it generates a partially ordered set (poset) that enumerates the dependencies between events prior to execution. Rapide was influenced not only by software languages such as ML and C++, but also by hardware description languages such as VHDL and Verilog.

UniCon [90] uses two fundamental elements in its description of a software architecture: the component (players) and the connector (roles). Components represent the "locus of computation and state." The properties and specification of a component are determined by the component's interface. These properties represent both functional and nonfunctional aspects of the component's behaviour. UniCon defines a comprehensive collection of built-in component types. These built-in components include *Module* types, which are used to represent a single compilation unit and *Process* types which represent independently scheduled processes as defined by the underlying operating system. Connectors represent the relations among components; central to the specification of the connector is the interaction protocol. The connectors can be used to enforce type and sequence constraints amongst the components. As with components, there are numerous built-in connector types. For example, the connector type *Pipe* represents the conventional Unix pipe connector; the type *RemoteProcCall* provides a connector for making calls to procedures which may reside outside the address space of a given component. Ideally, both components and connectors can be hierarchical in nature and impose concepts of data abstraction and encapsulation upon its internal elements (early specifications of the UniCon language, however, did not provide a means to define compositional connectors).

The Wright [2] architecture description language employs formal abstractions for the definition and subsequent analysis of an architecture. As with most other ADLs, Wright employs the concepts of components and connectors. It also introduces the concept of a configuration, which is a collection of component instances combined by connectors. The configuration essentially gives rise to the topology of the architecture being described. Components are comprised of an *interface* and a *computation*. The interface, in turn, is comprised of an arbitrary number of *ports* which serve as the conduit through which components interact with their environment. Connectors, as expected, serve to define the communication between components. The connectors impose a set of requirements that must be met by a component in order for the connection to be deemed appropriate. If the component satisfies the requirements, the component is permitted to make certain assumptions about its operational context.

By employing formal representations of software architectures, Wright offers numerous advantages since formal models are suitable for mathematical analysis and manipulation through machine-driven techniques. However, formal representations can become very complicated and incomprehensible, especially if they employ notations which are unfamiliar to software architects.

The use of an existing, well-established language as the basis of an architectural description language has also been evaluated. For example, the use of Java and JavaBeans as a potential architectural description language has been studied [100]. Unfortunately, application builders, which are commonly used to interconnect JavaBeans, do not allow the semantics of components to be exposed and do not provide support for the evolution of components or connections. Also, the JavaBeans connection and configuration frameworks do not allow the specification of interaction protocols or global constraints, thereby limiting the usefulness of this language as a viable ADL.

2.3 Summary

Modern day large-scale software projects are rarely built in a monolithic fashion. Teams of developers working independently, in accordance to (hopefully) well-defined specifications, construct subunits of the final project which must then be subsequently assembled or integrated. In addition, software entities are being designed to be increasingly generic in nature and are intended to be used and reused in several independent projects.

Many of the ideas offered in the definitions of software architecture can be used as a foundation to systematically describe the structure and dynamics of software systems. Informally, a software architecture, at its highest level, can be represented by three major abstractions: components, which serve as the units of functionality for the architecture; interfaces, through which the components communicate with its external environment; and connections which dictate the topology of the architecture and provide context to the components.

Intuitively, a software architecture can be defined as a graph in which vertices represent components and their respective interfaces and edges are used to represent the connections between components and their interfaces and the connections between communicating interfaces.

As more emphasis is being placed on the system integration phase of the software engineering discipline, the notion of constructing fully functioning software systems from the composition of existing disparate entities is becoming increasingly important. In all recent development methodologies, there is always a need to "put the pieces together"; to assemble the individually designed and implemented entities into a fully functioning system. A formal means by which software can be automatically integrated would, therefore, be desirable.

This thesis uses many of the fundamental tenets of component-based software engineering in an attempt to lay down a framework upon which automated assembly of software systems may occur. The following chapter discusses the pragmatic developments behind the current state of the art in the field of component-based software engineering.

Chapter 3

Component-based Systems

Central to the construction of any large software project is the modularization and decomposition of large software entities into smaller units with a well-defined function. During this decomposition, there must exist well-defined boundaries between what an entity does and how it actually does it. Component-based software engineering has been proposed as a means to achieve these and other goals, so as to facilitate the software development process [15]. Although initially proposed over thirty-five years ago, it is only in the past decade that component-based software engineering has become viable as a means of constructing large-scale software systems.

3.1 Components

Components are the building blocks of software systems and hence comprise the fundamental elements of reuse in a software architecture. Informally, a component is considered to be the primary functional unit and the fundamental data type in an architectural design. The connections between the components serve to determine the flow of control and to provide a context or environment for the components. Components allow one to represent a high-level software model relatively faithfully by closely modelling entities that occur in the context and vocabulary of the problem space.

The idea of achieving conceptual integrity or so-called cohesion via concept analysis is an important factor when designing components. Indeed, ideas related to module decomposition and module restructuring [105] may prove to be helpful in attaining component conceptual integrity.

Components must be generic enough to work in a variety of contexts and in cooperation with other components [110]. At the same time, however, their functionality must not be excessively vague or generic so as to inhibit reuse. During the design of a software system, it is important to maintain a balance between the number of components and their respective functionality. Deploying too many components at the same level of abstraction may lead to an exponential explosion of contextual interdependencies amongst them, thereby dramatically increasing the complexity of the system. Too few components may discourage reuse as designers would be motivated to design their own smaller and more efficient components rather than deploy excessively large, monolithic, uncohesive components in a given architecture.

3.2 Component Definitions and Representation

As with the term *software architecture*, many definitions of *component* exist in the literature. In their attempt to convey the essence of what constitutes a component, these definitions tend to be vague and sometimes even arbitrary in nature. Some definitions found in the literature are presented and discussed below.

Szyperski provides the following definition [102]:

> "A software component is a unit of composition with contractually specified interfaces and explicit dependencies only. A software component can be deployed independently and is subject to composition by third parties."

This definition introduces the concept of an *interface* which represents the access point to a component from external sources. Interfaces represent the notions of data encapsulation and abstract data types, whereby access to a component's behaviour is restricted via the component's interface. Well-defined and comprehensive interfaces can also serve as a mechanism for reuse and substitutability. For example, if a software system accesses the services of a component only through a specified interface, then that component can be easily swapped out and replaced with another that supports the semantics of the original interface — the underlying implementation of the new component could be completely different. The interface serves to specify what services a component is able to provide.

Before a component can be deployed, it must be aware of all *contextual dependencies*, that is, what external services the component requires in order to behave correctly. In addition to requiring other components, a component may also require a specific deployment environment or container. This environment is dictated by the component world or *component model*, which is discussed later in this section. Note that in order to promote reuse, a component's contextual dependencies should not be excessive; components should be self-contained.

Also important in the above definition is the concept of independent component *deployment*. This feature allows a complete software architecture to be broken down into its constituent components during the design and development stage of the system. Then, instead of deploying all the components as a single monolithic executable, the components are deployed individually into an operating environment. This modular approach to software deployment allows subsystems to be replaced or upgraded relatively easily without having to terminate and restart the entire system.

With contractually specified interfaces and well documented contextual dependencies, a third party should be able to acquire components from two independent parties and compose them into a useful software system to perform some desired task. This would represent the ideal world of component programming where software entities can be traded as commodity items like resistors and capacitors in the hardware world. Composition of these separate components can then take place to produce a useful system.

The following definition reiterates the importance of component interfaces. However, it also emphasizes that a software component should aim for reusability by narrowing the scope of the component's behaviour [89]:

> "Reusable software components are self-contained, clearly identifiable artifacts that describe and/or perform specific functions and have clear interfaces, appropriate documentation and a defined reuse status"

By qualifying the behaviour and the context of a component, this definition suggests that components should limit their functionality so as to reduce the possibility of overlapping behaviours amongst different components. In a software system, this serves to make the deployed components more orthogonal, thereby reducing redundancy and enhancing efficiency. In the context of the above definition, the word *artifacts* implies that components themselves can take on many forms including source code, an executable or a shared library. Regardless of the form, the component itself should exist as a packaged unit, as opposed to being spread over several files, for example.

The documentation and reuse status attributes given in the above definition represent information related to the component that is necessary for effective usage and deployment of the component by end users. This documentation should consist not only of the component's requirements, services and deployment issues, but should also provide information regarding the component designer and maintainer.

Perhaps one of the most succinct component definitions originates from Brown [16]:

> "An independently deliverable piece of functionality providing access to its services through interfaces."

This definition highlights the relatively autonomous nature of a component and again stresses the requirement of a well-defined interface through which services are offered. As with the previous definitions, this definition offers little in the way of mechanisms to describe the formal semantics of a component and its interaction with its environment. The subsequent chapter will review some more formal definitions of components and propose an alternative method for formally representing components and their interaction with other components in the context of a software architecture.

3.3 Components and Objects

One of the issues raised in the context of components is that of "semantic overlap" between components and objects. This section provides a comparison of the concepts behind components and objects and also demonstrates some criteria to help distinguish between them. While components and objects have similarities, there are also important subtle differences.

The most obvious similarity between components and objects is that both support the notion of an *interface* through which the external world communicates with the component or object [110]. The concept of data encapsulation is important to both components and objects and both of them should indicate what services they require of and provide to the external environment. However, components, unlike objects, may not have a persistent state [101]. As such, components may lack the concept of identity which is integral to the Booch definition of an object [12]. Components are also used to represent larger, coarser grained entities than objects. As a result, it is not unusual for a component to actually be comprised of several classes, which are instantiated to objects when the component is actually deployed. Note that while object-oriented techniques are commonly used for component design and implementation, components can be implemented using any programming paradigm such as functional programming or even more conventional procedural-based programming. Component design and implementation are not restricted to object-oriented abstractions.

Component architectures, through the use of "intelligent" interconnections, are able to provide a richer set of possibilities for component interaction [39]. Object interaction, however, is restricted to method invocation only. While this method invocation may be determined at run-time through polymorphism, or even made over a network (using CORBA, for example), invocation is still relatively rudimentary when compared to more recent advances in component architectures. Indeed, in the context of component architectures, the connection mechanism may take a much more pro-active role in the underlying semantics of component interaction. From the perspective of object-oriented programming, connections between objects via method invocation are more passive. For example, in a traditional environment in which the connectors are passive, a server component would typically be responsible for prioritizing the requests it receives — the connector would simply provide the conduit through which the requests are delivered. In an environment in which the connectors are more active, the responsibility of prioritizing the requests could conceivably be handled directly by the connector itself. This has the added benefit of decoupling the prioritization of requests from the other responsibilities of the server.

Another difference between components and objects, is that an object is a unit of instantiation whereas a component is a unit of deployment. Because of this, object-oriented strategies usually lead to the creation of monolithic applications consisting of many objects which all must be deployed simultaneously as a single unit in order to be functional. By taking a component-based approach, a functioning system can be deployed in a more incremental fashion. This has the added benefit in that small changes to a deployed production system can be made simply by deploying the appropriate components rather than deploying the entire application. Indeed, if the component software system is well designed, it may not even be necessary to terminate and restart the entire system during minor system upgrades.

With respect to the deployment issue above, many components are usually distributed as dynamic link libraries, shared objects or Java archive files (in the context of Enterprise JavaBeans) whereas object-oriented programs are typically distributed as executables. Another important aspect is the execution environment of objects and components. Executables produced from object-oriented programming operate in the context of an operating system environment. Components, however, typically operate in the context of a container. This container, which acts as an intermediary between the component and the underlying operating system determines the lifetime of its components.

3.4 Component-Based Software Development

New software development disciplines are emerging to address the issues associated with component software systems. In particular, component-based development (CBD) and component-based software engineering (CBSE) have arisen to provide a systematic approach toward the analysis and construction of software systems by assembling prefabricated components [30, 48].

Amongst the advantages of CBD is the ability to rapidly construct and deploy software systems which have a high degree of complexity. By acquiring and integrating software components from different vendors, a software developer can rapidly construct a fully-functioning software system. If the software components used to build the system have been verified to be functionally correct and accurate, then the overall system should have a similar level of quality, provided that the components were integrated correctly.

CBD also allows software developers to substitute new components into a given architecture so as to meet various nonfunctional requirements (for example, memory usage). Ideally, components can be substituted for others that support an identical interface and compatible semantics. This allows the development process to quickly evaluate the merits of different components in the context of an existing architecture.

There are many factors, however, that are holding back component-based software development. For example, the lack of a viable component market limits the number of components publicly available for reuse. As the component market matures, however, the number of components as well as the domains over which the components operate will expand, making component-based development a more realistic option in the development of complex software systems.

One of the reasons for the limited number of components available for purchase is the high degree of difficulty in producing a component which is both *usable* and *reusable.* In order for a component to be usable, the user of the component must be able to integrate the component easily into an existing architecture. Because of this, a component's interface should be relatively simple and easy to understand. However, in order for a component to be reusable, the designer of the component must make the component as generic and as flexible as possible so as to allow the component to operate in a wide variety of environments. As such, a generic component will typically require a more complicated interface. A more complicated interface, while promoting reusability, inhibits usability. Naturally, a balance between reusability and usability must be achieved.

During component development, care must be taken to ensure that components are both reliable and resistant to change. The component developer must also be very clear in documenting the component's constraints and requirements. Because a collection of components can be deployed incrementally, the environment of a component may be constantly changing. As a result, components must be designed to be resistant to contextual change. With respect to the incremental component deployment mentioned in the previous section, components that are fragile in their deployment environment are more susceptible to reliability issues as adjacent components change around them. Component versioning and dependency strategies can help ensure inter-component compatibility [30].

3.5 Current Component Models

In industry, components were originally introduced to handle the construction of commonly used graphical user interface entities. However, as the component landscape matured, components have become more flexible to handle more general problem domains. Component models essentially provide the foundation upon which component deployment and communication take place. Component models provide the infrastructure through which components can identify each other and subsequently interact with one another. This section highlights some of the component models prevalent in the industry.

3.5.1 Common Object Request Broker Architecture (CORBA)

CORBA is a standard put forth by the Object Management Group (OMG) [80]. As such, the standard is platform and vendor neutral. CORBA basically allows for distributed objects to locate and interact with one another over an Object Request Broker (ORB). Method arguments are marshalled at the client end and transmitted over the ORB via a well-defined protocol, typically the Internet-InterORB Protocol (IIOP). They are subsequently unmarshalled at the server end, the method is invoked and any return values are similarly transmitted back to the client.

In order to locate objects, CORBA defines the *Naming Service* which allows objects to be located by name. The naming service is part of the *CORBAServices* package which also provides support for system-level services such as persistence, events, transactions and database queries. Higher level abstractions and constructs are defined by *CORBAFacilities* framework which addresses issues related to both the horizontal and vertical application markets.

One of the strengths of CORBA is the fact that it supports multiple languages through the use of an Interface Definition Language (IDL). This language allows the developer to define the method signatures and object hierarchy of all the distributed objects in a system. A translator is then used to map IDL to a conventional language, typically C++ or Java. Hence, libraries of objects written in different languages are able to interact with one another. Strictly speaking, because CORBA only provides an object-oriented approach to the conventional Remote Procedure Call (RPC), it could be argued that CORBA does not conform to the conventional definition of component as presented above. Attempts to rectify this have begun recently with the introduction of the CORBA Component Model (CCM) by the OMG [104].

3.5.2 J2EE / Enterprise JavaBeans

Enterprise JavaBeans (EJB) from Sun Microsystems is a more recent development in the component model industry [88]. This component model, which is part of the J2EE framework, offers a relatively mature platform for component deployment and interaction. Many attributes from CORBA have been borrowed and enhanced by EJB including the concept of a naming service and the communication protocol used by EJB components to communicate with one another (IIOP). The J2EE framework provides support for the 3-tier architecture in which clients (tier one) communicate indirectly with the EJBs residing on a server (tier two). The EJBs then interface with backend databases (tier three) in order to satisfy the clients' request.

EJBs reside in the context of a container on the server, therefore all communication to the EJB must take place through a remote interceptor object which links the client with the EJB. The container decides the life cycle of all the EJBs under its domain and can instantiate more EJB components as required. This contributes to the scalability of the J2EE architecture. The container can also take care of other responsibilities such as persistence and security, therefore allowing the EJB developer to concentrate solely on the functionality of the component without being distracted with ancillary tasks. This separation of responsibility between the EJB and the container allows for the construction of a more robust architecture. Parameterization of EJB components is made possible via a *deployment descriptor*. This XML file is placed on the server as part of the deployment of the component and offers a way to change the behaviour of a system without having to recompile its constituent components.

Unfortunately, EJB, by definition, is language dependent. However, because the EJB specification has adopted the IIOP remote communication protocol, it is possible for EJB to communicate with other CORBA objects on a network. The J2EE architecture also has the advantage of being vendor neutral as the specification produced by Sun may be implemented by other vendors. Indeed, other vendors such as IBM, IONA, and BEA Systems have implemented their own versions of the J2EE architecture in addition to Sun. JBoss, a freely available, open-source implementation of the J2EE architecture is also available.

3.5.3 .Net

In recent years, Microsoft has proposed the so-called .Net framework. Compared to other offerings, it is relatively immature and is intended to be platform neutral. Claims of language neutrality have been compromised by the promotion of Microsoft-controlled languages such as Visual Basic and C# as the basis for the .Net framework. Despite the level of univendor control, some attempts have been made to duplicate this framework outside the confines of the Microsoft platform. Recently, some efforts have also been made to make the platform amenable to vendor neutral languages. Such efforts may make this architecture worthy of further study in the future.

3.5.4 Summary

As the above discussion of component models demonstrates, there is no consistent approach to modelling components and their interactions. There are several competing approaches each with their own advantages and disadvantages. CORBA offers the most mature technology; however J2EE has been able to adopt many of the more successful concepts originated by CORBA to create a viable server-side component model.

Of particular importance is the fact that none of the pragmatic component development models described in this chapter provides a viable way of determining compatibility between software components. Apart from the very limited static checking of parameter and return types, none of the models makes any attempt to establish the dynamic consistency between components that must be present in order for components to successfully interact with one another.

Chapter 4

Formal Component Models

The previous chapter reviewed some popular definitions of components used in the literature. Since all these definitions are informal, they are not suitable to formal, automated analysis for the purposes of assessing component compatibility. Therefore, formal definitions are required which would allow one to determine if a component-based system satisfies the requirements, especially in terms of compatibility. Section 4.1 provides an overview of some formal definitions of components quoted from the literature. Section 4.2 provides formal definitions related to Petri nets. Section 4.3 and 4.4 proposes a formal model that uses Petri nets to represent the behaviour of component interfaces. Using the concepts in this chapter, a formal definition of component compatibility is given in the next chapter.

4.1 Formal Models of Components

Prior to laying down the foundation upon which component compatibility can be established, a formal model of a component, and in particular its interface, must be developed. The development of a formal model provides the possibility of automating the interconnection of software entities which leads to a software architecture that complies with a given software specification. A formal model for representing the static and dynamic attributes of component interfaces is presented in subsequent sections.

Apart from the informal definitions of components discussed earlier, more formal definitions of components have also been proposed in the literature. Often, a component is defined using a BNF formalism. For example, the following is a BNF description of a component in which the component's interface, parameters, methods, behaviour and context are enumerated [72]:

```
component ::=
   component component_name is
      interface component_message_interface
      parameters component_parameters
      methods component_methods
      [behaviour component_behaviour]
      [context component_context]
   end component_name
```

Definitions for other syntactical classes, including **component_message_interface**, **component_methods** and **component_behaviour** are also provided. Using formal grammars for component definitions facilitates the possibility of automatic code generation. Also, by having a comprehensive collection of well-defined software components, the likelihood of finding a component that accomplishes a required task is increased, thereby promoting the potential for reuse. However, such syntactic definitions cannot address the dynamic behavioural aspects of component descriptions.

Another avenue towards component specification and representation is to apply a more mathematical approach [8]. For example, if *Components*, *Interfaces* and *Con-*

nections denote the sets of all components, interfaces and connections, respectively, then the relationships between these entities can be formalized through functional descriptions. The association of an interface with a component can be described by the function *assigned*:

$$\mathit{assigned} : \mathit{Interfaces} \longrightarrow \mathit{Components}.$$

Similarly, the concept of a connection between the interfaces of two components is specified by a relation:

$$\mathit{connIfs} : \{\{i, j\} \mid i, j \in \mathit{Interface} \ \wedge \ i \neq j\}.$$

Many other aspects, such as interface and component behaviour, component composition and various constraints can also be defined similarly. This approach towards formal component specification is sufficiently generic and can be used to define component interaction in a variety of contexts. Unfortunately, this model results in a very static representation of the underlying architecture implemented by the deployed components. As a result, the model is insufficient for modelling the modification of a component's behaviour during runtime; the model is unable to distinguish between design time and execution time.

As with architecture description, there have been several attempts, recently, to use the Unified Modelling Language (UML) as a basis for component definition and representation [58]. Indeed, version 1.5 of the UML specification provides the following definition of a component [79]:

> "A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces. ... A component conforms to the interfaces that it exposes, where the interfaces

> represent services provided by elements that reside on the component. A component may be implemented by one or more artifacts, such as binary, executable, or script files. A component may be deployed on a node."

Unfortunately, the UML specification, which was primarily designed to model object systems, contains several semantic overlaps that make it less than ideal for component modelling. In addition, issues regarding connection mechanisms are not fully addressed by UML 1.5. Other problems include the inability to accurately model all the nuances of specific component technologies, such as EJB and CCM (described in Section 3.5). With the advent of UML 2.0 and the notion of "UML Profiles," many of these problems may be addressed, thereby making UML more amenable to the description of component-based systems and software architectures, in general [84].

4.2 Petri Net Component Models

As indicated earlier, several attempts have been made to define a component: many of these attempts have been summarized in [102]. 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 to formally define a component and its behaviour have made use of Petri nets [108]. Component composition and compatibility assessment using Petri net models are established in the literature [57, 92]. Related to this area is the composition and interoperability of web services [68] and verification of workflow composition [107]. While the method presented herein shares high-level concepts with those presented in the literature, this method of composition and compatibility assessment 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. This work is a further refinement of an earlier composition attempt presented in [29], in that this approach supports multi-requester and multi-provider composition scenarios.

For verification of component compatibility, the low-level, internal details of the component will be disregarded as they are not important in the formalism discussed below. The focus of attention is on the behaviour at the level 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 internal communications irrelevant unless they affect the interface behaviour.

The definitions and concepts in subsections 4.2.1 and 4.2.2 are taken from [33, 78, 86, 115]. The remaining subsections introduce new structural concepts and a deadlock detection strategy which are used in subsequent chapters to help simplify and analyze Petri net models.

4.2.1 Petri Nets

Petri nets [86] have been proposed, by Carl Adam Petri, as a simple and convenient formalism for modelling systems that exhibit concurrent actions. Traditional formalisms, developed for analysis of systems with sequential behaviour, are inadequate for representation of concurrent activities and their synchronization.

Petri nets are bipartite directed graphs, in which the two types of nodes, called *places* and *transitions*, represent conditions and events (Petri nets are sometimes called *condition-event* systems). An event can occur only when all conditions associated with it, and represented by arcs directed to the event node, are satisfied. An occurrence of an event usually satisfies some other conditions, indicated by arcs directed from an event node. In effect, an occurrence of an event causes some other event(s) to occur and so on.

Definition: A *place/transition Petri net* (sometimes also called a net structure) $\mathcal{N}$ is a triple $\mathcal{N} = (P, T, A)$, where P is a finite set of places (which represent conditions), T is a finite set of transitions (which represent events), and A is a set of directed arcs connecting places with transitions and transitions with places, $A \subseteq P \times T \cup T \times P$. ($A$ is sometimes called the flow relation or causality relation, and can also be represented in two parts, a subset of $P \times T$ and a subset of $T \times P$.) For each place $p \in P$ and each transition $t \in T$, the input and output sets are defined as:

$$\begin{aligned}
\mathrm{Inp}(p) = &\ \{ \, t \in T \mid (t, p) \in A \, \}, \\
\mathrm{Out}(p) = &\ \{ \, t \in T \mid (p, t) \in A \, \}, \\
\mathrm{Inp}(t) = &\ \{ \, p \in P \mid (p, t) \in A \, \}, \\
\mathrm{Out}(t) = &\ \{ \, p \in P \mid (t, p) \in A \, \}.
\end{aligned}$$

The structure of a net can be represented by a matrix which denotes the connectivity between the places and transitions of the net.

Definition: A *connectivity matrix* (or *incidence matrix*), **C**, of a net $\mathcal{N} = (P, T, A)$ is a matrix in which the rows correspond to places, the transitions correspond to columns, and the entries are defined by:

$$\forall p_i \in P\ \forall t_i \in T : \mathbf{C}[i,j] = \begin{cases} -1, & \text{if } p_i \in \mathrm{Inp}(t_j) - \mathrm{Out}(t_j), \\ +1, & \text{if } p_i \in \mathrm{Out}(t_j) - \mathrm{Inp}(t_j), \\ 0, & \text{otherwise.} \end{cases}$$

As will be shown later, the connectivity matrix can be used to determine various properties of nets.

The dynamic behaviour of a net is represented by *marking functions* which assign a non-negative number of *tokens* to each place of a net.

Definition: A *marked* net, $\mathcal{M}$, is a pair $\mathcal{M} = (\mathcal{N}, m_0)$, where $\mathcal{N}$ is a net structure, $\mathcal{N} = (P, T, A)$, and m_0 is the initial marking function, $m_0 : P \to \{\ 0, 1, \dots\ \}$. Marked nets are also defined as $\mathcal{M} = (P, T, A, m_0)$. A place which is assigned a nonzero number of tokens is called a *marked* place. Otherwise, it is called an unmarked place.

A marking function (or, more simply, a *marking*) is commonly represented as a (column) vector in which the number of elements is equal to the number of places in the net and each element represents the number of tokens in the corresponding place of the net.

Under certain conditions, the tokens can "move" in the net, changing one marking into another.

Definition: In a marked net $\mathcal{M} = (P, T, A, m_0)$, a transition $t \in T$ is *enabled* by the marking m_0 if all its input places are marked by m_0; the set of all transitions enabled by m_0 is denoted $E(m_0)$:

$$E(m_0) = \{\ t \in T \mid \forall p \in \mathrm{Inp}(t) : m_0(p) > 0\ \}.$$

Each transition, which is enabled by a marking, can *fire* (or, an event represented by this transition can occur). An occurrence of an event removes (simultaneously) a single token from all input places of the transition representing the occurring event, and (also simultaneously) adds a single token to all output places of this transition.

An occurrence of an event represented by transition t enabled by marking m creates a new marking m' which is directly reachable (*i.e.*, reachable in one step) from m.

Definition: In a net $\mathcal{N} = (P, T, A)$, a marking m' is *directly reachable* from a marking m, $m \mapsto m'$, if there exists $t \in E(m)$ such that:

$$\forall p \in P : m'(p) = \begin{cases} m(p) - 1, & \text{if } p \in \mathrm{Inp}(t) - \mathrm{Out}(t), \\ m(p) + 1, & \text{if } p \in \mathrm{Out}(t) - \mathrm{Inp}(t), \\ m(p), & \text{otherwise.} \end{cases}$$

If an enabled transition t_k fires in marking m, then the new marking, m', can be determined using the connectivity matrix: $m' = m + \mathbf{C}[*, k]$, where m and m' are column vectors which represent the markings before and after the firing, respectively, and $\mathbf{C}[*, k]$ represents the k^{th} column of $\mathbf{C}$, *i.e.*, the column which corresponds to the transition t_k.

Definition: A marking m' is *generally reachable* from a marking m, $m \overset{*}{\mapsto} m'$, if there exists a sequence of (intermediate) markings $m_{i_0}, m_{i_1}, \ldots, m_{i_k}$ such that $m_{i_0} = m$, $m_{i_k} = m'$, and $m_{i_{\ell-1}} \mapsto m_{i_\ell}$ for $\ell = 1, \ldots, k$.

Definition: The set of *reachable markings* of a marked net $\mathcal{M} = (P, T, A, m_0)$, $M(\mathcal{M})$, is the set of all markings reachable from m_0 in $\mathcal{M}$:

$$M(\mathcal{M}) = \{ \ m : P \to \{ \ 0, 1, \dots \ \} \mid m_0 \overset{*}{\mapsto} m \ \}.$$

The set of reachable markings can be finite or infinite. If it is finite, the net is *bounded*, otherwise it is *unbounded*. If a marked net $\mathcal{M}$ is bounded, there exists a constant k (called the *bound*) such that:

$$\forall m \in M(\mathcal{M}) \ \forall p \in P : m(p) \leq k.$$

If this bound is equal to 1, the net is called *safe*.

Definition: A place is *shared* iff its output set contains more than one transition:

$$p \text{ is shared} \Leftrightarrow \text{card}(\text{Out}(p)) > 1.$$

A net which does not contain shared places is (structurally) *conflict-free*.

Definition: A place is (structurally) *free-choice* iff all transitions sharing it have identical input sets:

$$p \text{ is free-choice} \Leftrightarrow \forall t_i, t_j \in \text{Out}(p) : \text{Inp}(t_i) = \text{Inp}(t_j).$$

A net is *free-choice* if all its shared places are free-choice.

For a free choice place p in a net $\mathcal{N}$, a marking m either enables all transitions sharing p, or none of these transitions is enabled by m.

Definition: A marked net $\mathcal{M}$ is *dynamically conflict-free* iff for all reachable markings and for each place p, at most one transition in the output set of p is enabled by m:

$$\mathcal{M} \text{ is dynamically conflict-free} \Leftrightarrow \forall m \in M(\mathcal{M}) \ \forall p \in P : \text{card}(E(m) \cap \text{Out}(p)) \leq 1.$$

Conflict-free nets represent systems with deterministic behaviours *i.e.*, systems in which the "next actions" are always uniquely determined. There are several subclasses of Petri nets, such as state machines and marked graphs, for example. More information on these subclasses can be found in [78].

A sequence of transitions $(t_{i_1}, t_{i_2}, \ldots, t_{i_k})$ is a *firing sequence* at marking m if t_{i_1} is enabled by m, t_{i_2} is enabled by the marking obtained by firing t_{i_1}, and so on.

Definition: A *firing sequence* at marking m, $\sigma(m)$, in net $\mathcal{N}$ is defined as:

$$\begin{aligned}\sigma(m) = t_{i_1} t_{i_2} \ldots t_{i_k} \Leftrightarrow\ & \exists\, m_{i_0}, m_{i_1}, \ldots, m_{i_k} : m_{i_0} = m \ \wedge \\ & \forall\, 0 < j \leq k : t_{i_j} \in E(m_{i_{j-1}}) \ \wedge\ m_{i_{j-1}} \overset{t_{i_j}}{\longmapsto} m_{i_j},\end{aligned}$$

where $E(m)$ is the set of transitions enabled by m. The set of all firing sequences at the initial marking m_0 of $\mathcal{M}_i$ is denoted by $\mathcal{F}(\mathcal{M}_i)$.

Each firing sequence σ can be described by a firing vector which indicates the number of occurrences of each transition in the sequence σ.

Definition: The *firing vector*, f_σ, of a firing sequence σ is a mapping $f_\sigma : T \rightarrow \{0, 1, \ldots\}$, where $f_\sigma(t)$ is the number of occurrences of t in σ. The firing vector of σ is also known as a Parikh vector of σ.

It should be observed that a firing vector does not uniquely identify a firing sequence — a valid firing vector may correspond to many possible firing sequences in a marked net.

Definition: A net $\mathcal{M} = (P, T, A, m_0)$ is *live* if, for all transitions $t \in T$, and any marking m reachable from m_0, t can fire in m or some subsequent marking reachable from m:

$$\mathcal{M} \text{ is live} \Leftrightarrow \forall m \in M(\mathcal{M})\ \forall t \in T\ \exists m' \in M(\mathcal{M}) : m \overset{*}{\longmapsto} m' \ \wedge\ t \in E(m').$$

Live nets correspond to systems in which all events can occur (eventually). Absence of the liveness property may indicate some sort of "problem" in the system. A net which is not live contains a deadlock or a livelock.

Definition: A marking m in net $\mathcal{N} = (P, T, A)$ is *dead* if it does not enable any transition, *i.e.*,

$$m \text{ is dead} \Leftrightarrow E(m) = \emptyset.$$

If the set of reachable markings of $\mathcal{M} = (\mathcal{N}, m_0)$ contains a dead marking, then $\mathcal{M}$ contains a deadlock:

$$\text{deadlocked}(\mathcal{M}) = \exists m \in M(\mathcal{M}) : E(m) = \emptyset.$$

Deadlocks in Petri nets can be analyzed by checking the sets of reachable markings (for bounded nets) or by studying structural properties of nets. The concepts of siphons and traps [43] are commonly used in the structural analysis of nets.

Definition: A *siphon* in a net $\mathcal{N} = (P, T, A)$ is a subset of places $S \subseteq P$ such that $\text{Inp}(S) \subseteq \text{Out}(S)$, where $\text{Inp}(S) = \bigcup_{s \in S} \text{Inp}(s)$ and $\text{Out}(S) = \bigcup_{s \in S} \text{Out}(s)$. A *minimal siphon* is defined as a siphon which does not include any other siphon. A siphon is *proper* if $\text{Inp}(S) \subset \text{Out}(S)$. A *basis siphon* is a siphon that cannot be represented as a union of other siphons.

It can be observed that any union of siphons is also a siphon and that all minimal siphons are also basis siphons. Any siphon in a net contains one of the minimal siphons and any siphon in a net can be represented by the union of one or more basis siphons. In a siphon S, all input transitions are also output transitions of S, so if S becomes unmarked, it remains unmarked for all subsequent markings. It can be shown that for each dead marking m in a net $\mathcal{N}$, the set of unmarked places is a siphon [74]. Algorithms for extracting basis and minimal siphons in a net are well established in the literature [11, 52]. Unfortunately, in the general case, the running time of these algorithms is not polynomial [28, 103].

Definition: A *trap* in a net $\mathcal{N} = (P, T, A)$ is a subset of places $Q \subseteq P$ such that $\mathrm{Out}(Q) \subseteq \mathrm{Inp}(Q)$. A *minimal trap* is defined as a trap which does not include any other trap. A *marked trap* is defined as a trap which has at least one of its places marked.

If the input and output transitions of a subset of places are the same, then the subset of places represents both a trap and a siphon. This is known as a *siphon-trap*.

4.2.2 Siphons and Liveness

Siphons are an important concept in determining liveness of a net. If a marking reachable from the initial marking results in a siphon becoming unmarked, (*i.e.*, all places of the siphon are unmarked), then the net cannot be live. Deadlock-freeness can be asserted by ensuring that each minimal siphon in the marked net can never become unmarked [43].

Whether or not a minimal siphon, S, can become unmarked can be determined by minimizing the number of tokens in the siphon. This is typically formulated as a linear programming problem [25, 27, 74, 93, 94]:

$$\min\left(\sum_{p \in S} m(p)\right) \text{ subject to: } x \geq 0;\ m = m_0 + \mathbf{C}x \geq 0$$

where $\mathbf{C}$ is the connectivity matrix of $\mathcal{N}$, x is the yet unknown firing vector that minimizes the number of tokens in the siphon and m is the final marking obtained from the initial marking m_0 by the firing vector x. Because no place in any marking can ever have a negative number of tokens, all elements in the final marking vector must be greater than or equal to zero, which is indicated in the constraints above.

The objective function can also be expressed as:

$$\begin{aligned}\sum_{p\in S} m(p) &= \sum_{p\in S} (m_0(p) + \mathbf{C}[p,*]x) \\ &= \sum_{p\in S}\left(m_0(p) + \sum_{t\in \mathrm{Inp}(p)} x[t] - \sum_{t\in \mathrm{Out}(p)} x[t]\right),\end{aligned}$$

where $\mathbf{C}[p,*]$ is the row vector of $\mathbf{C}$ corresponding to place p. The constraints for the linear programming problem can be formulated as follows:

$$x(t) \geq 0,\ t \in T;$$

$$m_0(p) + \sum_{t\in \mathrm{Inp}(p)} x(t) - \sum_{t\in \mathrm{Out}(p)} x(t) \geq 0,\ p \in P.$$

It can be observed that the objective function is derived from rows in the connectivity matrix that correspond to the places in the siphon, whereas the constraints are derived from *all* rows (places) in the connectivity matrix, and not just those in the siphon S. If a solution of the linear programming problem exists, it is provided as a firing vector, which does not take into account the ordering of firing transitions. It is possible that the solution of the linear programming problem has no corresponding firing sequence; a simple example of such a situation is presented in the next section. Therefore, each solution of the linear programming problem is followed by a verification step which checks if the firing vector is feasible. This can be done by a recursive function which is analogous to backtracking in classic AI searches [13]. An example of such a function **feasible** is given in Figure 4.1. The function takes, as arguments, the net $\mathcal{N}$, the initial marking m_0 and the firing vector x which is the solution of the linear programming problem. The function returns a firing sequence that corresponds to the firing vector

```
func feasible($\mathcal{N}, m, x$) : sequence
  begin
    var $m'$,  /* the new marking */
        $x'$,  /* the new firing vector */
        $f$;   /* the (partial) firing sequence */

    if zero($x$) then
      return(<>) /* the empty sequence */
    endif;

    for each $t$ in enable($\mathcal{N}, m$) do
      if $x[t] > 0$ then
        $x' := x$;
        $x'[t] := x'[t] - 1$;
        $m' :=$ fire($\mathcal{N}, m, t$);
        $f :=$ feasible($\mathcal{N}, m', x'$);
        if $f \neq$ fail then
          return(<$t, f$>)
        endif
      endif
    endfor;
    return(fail)
  end
```

Figure 4.1: Function **feasible**

x, if one exists, or fail if one does not. The enable function in Figure 4.1 returns the set of transitions in the net that are enabled by the given marking. The fire function takes a net, its marking and a transition and returns the new marking that results from firing the transition. The zero function takes a vector and returns true if all the elements in the vector are zero.

If the firing vector is deemed feasible by the procedure, then it creates an unmarked siphon so the net cannot be live. As with many backtracking algorithms, this algorithm is exponential with respect to the magnitude of non-zero elements in the initial vector x. However, in practice, the magnitude of the non-zero elements returned by the linear programming minimization procedure is typically quite low. Indeed, in larger nets, the number of zero elements in the vector may be quite high, thereby mitigating the time required to test the feasibility of the vector.

In the general case, linear programming cannot be used directly to find deadlocks because the condition that all transitions are disabled (so there is a deadlock), can be a nonlinear one. However, if the net is an event graph (*i.e.*, each transition has exactly one input and one output place), the deadlock can be determined directly using linear programming as a solution of the following problem:

$$\min \left(\sum_{t \in T} m(\mathrm{inp}(t)) \right) \text{ subject to: } m_0 + \mathbf{C}x \geq 0, x \geq 0,$$

where $\mathbf{C}$ is the connectivity matrix of $\mathcal{N}$ and $\mathrm{inp} : T \rightarrow P$, *i.e.*, $\mathrm{inp}(t)$ determines the single input place for a transition, t.

This property forms the basis of yet another approach to deadlock detection [55, 56, 75], which first unfolds the (general) net to a simple occurrence net (reduced to a finite prefix which represents all important properties of the original net), and then uses linear programming to check if this finite prefix indicates a deadlock in the original net. It has been shown [69] that in some cases, the unfolding results in a very compact net, but in the general case, the unfolded net can be quite complex [56]. Unfolding strategies can also be employed for reachability analysis as well [35].

4.2.3 Example One

This section provides a comprehensive example of a Petri net to demonstrate many of the concepts introduced in the previous section. Figure 4.2 is a Petri net with five places (represented by circles), four transitions (represented by rectangles) and eleven arcs. Inp(t_3) = $\{p_4, p_5\}$ and Out(p_5) = $\{t_3, t_4\}$, for example. Its initial marking is represented by the column vector $m_0 = [1,0,1,0,0]^T$; the solid black circles in p_1 and p_3 represent the tokens of the initial marking.

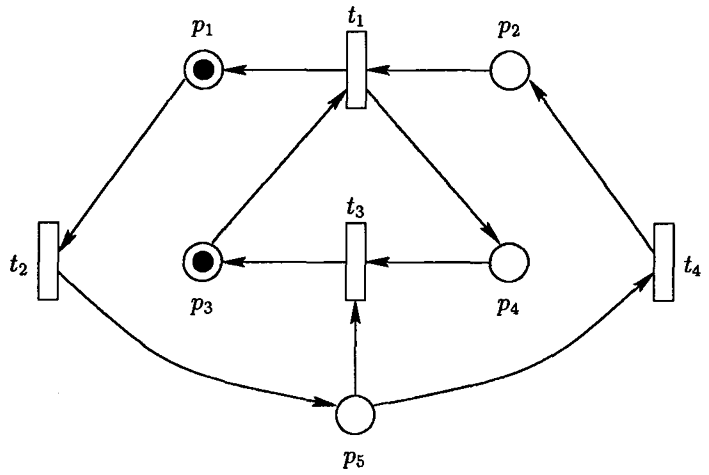

Figure 4.2: A Petri net

The connectivity matrix of this net is as follows:

$$\mathbf{C} = \begin{bmatrix} +1 & -1 & 0 & 0 \\ -1 & 0 & 0 & +1 \\ -1 & 0 & +1 & 0 \\ +1 & 0 & -1 & 0 \\ 0 & +1 & -1 & -1 \end{bmatrix}$$

Since p_5 is a shared place, the net is not conflict-free. The places p_1, p_2, p_3 and p_4 are all (trivially) free-choice. However, place p_5 is not free-choice since Inp(t_3) $\neq$ Inp(t_4). For the shown initial marking, the net is not dynamically conflict-free because the marking $[0,0,0,1,1]^T$, reachable from m_0, enables two transitions, t_3 and t_4.

The only transition initially enabled in the net is t_2. As it fires, the token is removed from p_1 and is added to p_5, creating marking, $[0,0,1,0,1]^T$ which enables t_4. t_3 is not enabled by this new marking since $p_4 \in$ Inp(t_3) is not marked.

When t_2 fires, the marking changes from $m_0 = [1,0,1,0,0]^T$ to $[1,0,1,0,0]^T + [-1,0,0,0,1]^T = [0,0,1,0,1]^T$. When t_4 then fires, the new marking is $[0,0,1,0,1]^T + [0,1,0,0,-1]^T = [0,1,1,0,0]^T$. The transition sequence (t_2, t_4, t_1, t_2) is a firing sequence for this Petri net. This firing sequence is represented by the firing vector $[1,2,0,1]$.

The list of all reachable markings for this net is given in Table 4.1, in which markings 5 and 6 constitute dead markings.

Table 4.1: Reachable markings of the net in Figure 4.2

Node	Marked Places	Firing Transition	Next Marking
0	$\{p_1, p_3\}$	t_2	1
1	$\{p_3, p_5\}$	t_4	2
2	$\{p_2, p_3\}$	t_1	3
3	$\{p_1, p_4\}$	t_2	4
4	$\{p_4, p_5\}$	t_3	5
		t_4	6
5	$\{p_3\}$	–	–
6	$\{p_2, p_4\}$	–	–

An alternative way to assess deadlock-freeness is to perform structural analysis and linear programming. The siphons and traps of this net, as well as their minimal counterparts and the basis siphons, are presented in Table 4.2. The subsets of places $\{p_1, p_2, p_3, p_5\}$ and $\{p_3, p_4\}$ are both siphon-traps.

The test for deadlock-freeness is initially based on the minimal siphons, $\{p_1, p_2, p_5\}$, $\{p_1, p_3, p_5\}$. For $S = \{p_1, p_2, p_5\}$, the minimization objective function is:

$$\begin{aligned}\sum_{p \in S} m(p) &= 1 + (x_{t_1} - x_{t_2}) + (-x_{t_1} + x_{t_4}) + (x_{t_2} - x_{t_3} - x_{t_4}) \\ &= 1 - x_{t_3}\end{aligned}$$

Table 4.2: Siphons/traps in Figure 4.2

Siphons:	$\{p_1, p_2, p_3, p_4, p_5\}$, $\{p_1, p_2, p_3, p_5\}$, $\{p_1, p_2, p_4, p_5\}$, $\{p_1, p_2, p_5\}$, $\{p_1, p_3, p_4\}$, $\{p_1, p_3, p_4, p_5\}$, $\{p_1, p_3, p_5\}$, $\{p_3, p_4\}$
Minimal Siphons:	$\{p_1, p_2, p_5\}$, $\{p_1, p_3, p_5\}$, $\{p_3, p_4\}$
Traps:	$\{p_1, p_2, p_3, p_4, p_5\}$, $\{p_1, p_2, p_3, p_5\}$, $\{p_2, p_3, p_4\}$, $\{p_2, p_3, p_4, p_5\}$, $\{p_3, p_4\}$
Minimal Traps:	$\{p_1, p_2, p_3, p_5\}$, $\{p_3, p_4\}$
Basis Siphons:	$\{p_1, p_2, p_4, p_5\}$, $\{p_1, p_2, p_5\}$, $\{p_1, p_3, p_4\}$, $\{p_1, p_3, p_5\}$, $\{p_3, p_4\}$

This objective function is derived by adding all rows of the connectivity matrix that correspond to each place in the siphon $\{p_1, p_2, p_5\}$. For example, the $(x_{t_1} - x_{t_2})$ term corresponds to the first row (p_1) of the matrix and the subsequent terms correspond to the second and fifth rows (p_2 and p_5). The number of tokens in the siphon at the initial marking is added to the objective function.

Hence, the linear programming problem is to minimize $1 - x_{t_3}$, subject to the constraints:

$$\begin{aligned}
p_1: &\quad x_{t_1} - x_{t_2} + 1 &\geq 0,\\
p_2: &\quad -x_{t_1} + x_{t_4} &\geq 0,\\
p_3: &\quad -x_{t_1} + x_{t_3} + 1 &\geq 0,\\
p_4: &\quad x_{t_1} - x_{t_3} &\geq 0,\\
p_5: &\quad x_{t_2} - x_{t_3} - x_{t_4} &\geq 0,\\
&x_{t_1} \geq 0,\ x_{t_2} \geq 0,\ x_{t_3} \geq 0,\ x_{t_4} \geq 0.
\end{aligned}$$

The first five constraints correspond to places of the net and specify that the number of tokens in each place cannot be negative. The final group of constraints simply states that all transitions cannot fire a negative number of times.

Solving this problem gives the vector $[1, 2, 1, 1]$, which can be verified by the feasible function to correspond to the firing sequence: $(t_2, t_4, t_1, t_2, t_3)$ which empties the siphon and also results in the net becoming deadlocked. Using the same technique on the other siphon, $\{p_1, p_3, p_5\}$, results in the firing vector $[1, 2, 0, 2]$, which corresponds to the firing sequence $(t_2, t_4, t_1, t_2, t_4)$. This provides another means of creating a deadlock. In this particular example, it is not necessary to examine the basis siphons in order to determine if a deadlock is present. However, as will be shown in a subsequent example, examination of the basis siphons may be needed to obtain a deadlock in the net. If a deadlock cannot be obtained by minimizing the number of tokens in any sequence of basis siphons, then the net is deadlock-free.

To illustrate an example in which linear programming can yield an infeasible vector, consider the simple net given in Figure 4.3. The objective function associated

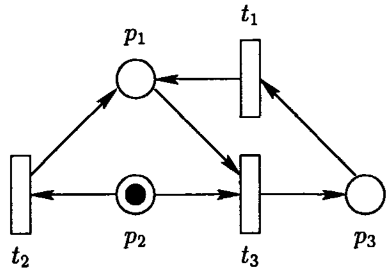

Figure 4.3: A Petri net with an infeasible firing vector

with the minimal siphon $\{p_2\}$ is $1 - x_{t_2} - x_{t_3}$. The constraints obtained from the connectivity matrix are:

$$\begin{aligned}
p_1: &\quad x_{t_1} + x_{t_2} - x_{t_3} \geq 0,\\
p_2: &\quad 1 - x_{t_2} - x_{t_3} \geq 0,\\
p_3: &\quad -x_{t_1} + x_{t_3} \geq 0,\\
&x_{t_1} \geq 0,\ x_{t_2} \geq 0,\ x_{t_3} \geq 0.
\end{aligned}$$

The firing vector $[1, 0, 1]$ satisfies the constraints while minimizing the objective function to zero. However, this vector violates the feasibility test (because in the initial marking, neither t_1 nor t_3 can fire) and this vector must therefore be rejected during deadlock analysis.

For larger net models, the extraction of the minimal and basis siphons can become more troublesome because of the time complexity. This can be mitigated by eliminating "similar" siphons of the net while still preserving the underlying structural properties of the net that may generate a deadlock. This is the topic of the next section.

4.2.4 Similar and Essential Siphons

In many net models, the number of siphons increases very quickly with the size of the model. It appears, however, that for deadlock detection, only a small number of siphons is needed. The concepts of *essential siphons* and *siphon similarity* are introduced to determine which siphons are important and which are not when determining deadlock in a Petri net.

Definition: Two siphons S_1 and S_2 in a net $\mathcal{M}$ are similar, $S_1 \sim S_2$, if for all reachable markings either both are marked or both are unmarked:

$$S_1 \sim S_2 \Leftrightarrow \forall m \in M(\mathcal{M}) : \text{mark}(S_1, m) = 0 \Leftrightarrow \text{mark}(S_2, m) = 0$$

where $\text{mark}(S, m) = \sum_{p \in S} m(p)$.

Corollary 4.1 *The relation of siphon similarity is an equivalence relation on the set of siphons of a marked net $\mathcal{M}$, so it implies a partition of this set into classes of similar siphons.*

The corollary is a straightforward consequence of the definition of similar siphons. □

Definition: Set $S = \{S_1, S_2, \ldots, S_n\}$ is the set of *essential siphons* for $\mathcal{M}$ if no two siphons in S are similar and if any other siphon of $\mathcal{M}$ is similar to one of the siphons in S.

Corollary 4.2 *The set of essential siphons of a net $\mathcal{M}$ contains one siphon from each equivalence class of the siphon similarity relation.*

The corollary is a straightforward consequence of the definition of essential siphons.

□

As a consequence of this corollary, non-essential siphons of $\mathcal{M}$ can be eliminated by removing from $\mathcal{M}$ elements which create similar siphons.

Definition: A *simple path* in a net $\mathcal{N}$ is a sequence of transitions and places $t_{i_0} p_{i_1} t_{i_1} p_{i_2} \ldots p_{i_k} t_{i_k}$, such that:

$$(\forall 1 \le j \le k \;\; : \text{Inp}(p_{i_j}) = \{t_{i_{j-1}}\} \;\wedge\; \text{Out}(p_{i_j}) = \{t_{i_j}\}) \;\wedge$$
$$(\forall 1 \le j < k \;\; : \text{Inp}(t_{i_j}) = \{p_{i_j}\} \;\wedge\; \text{Out}(t_{i_j}) = \{p_{i_{j+1}}\}).$$

Each simple path is denoted $path(t_{i_0}, t_{i_k})$, though there can be several simple paths connecting t_{i_0} and t_{i_k}. In order to represent the places along a path, the *places* function can be used to "extract" the places for a given path, π:

$$places(\pi) = \begin{cases} \{p\} \cup places(\sigma), & \text{if } \pi = p\sigma, \\ places(\sigma), & \text{if } \pi = t\sigma, \\ \emptyset, & \text{if } \pi = \varepsilon. \end{cases}$$

There are two classes of paths in a net that can lead to simplifications that do not adversely affect the net's behavioural properties with respect to deadlock analysis. These paths are called *parallel* paths and *alternate* paths and are defined in the following subsections.

4.2.4.1 Parallel Paths

Informally, parallel paths are represented by a pair of delimiting transitions that encompass two (or more) simple paths, as illustrated in Figure 4.4.

Definition: *Parallel paths* are any two simple paths which connect the same transitions t_i and t_j.

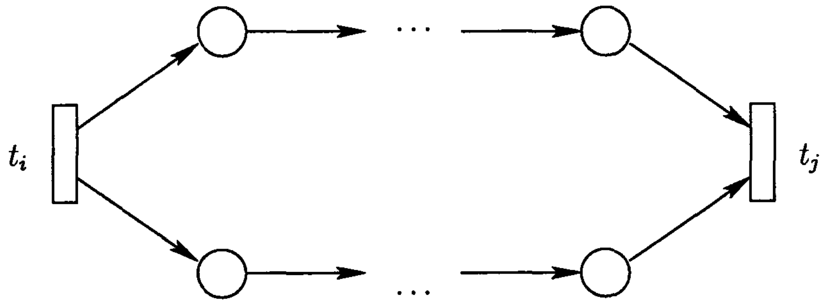

Figure 4.4: Parallel paths in a Petri net

Definition: Two or more simple paths are *equally marked* if they all possess at least one token or if they all possess no tokens in the initial marking.

It can be observed that if two or more parallel paths are equally marked, then if a siphon contains the set of places in one of the parallel paths, then similar siphons exist that contain places in each of the other parallel paths.

Corollary 4.3 *For equally marked parallel paths π_1 and π_2, if* places(π_1) *is a subset of siphon S_i in $\mathcal{M}$, then* places(π_2) *is a subset of another siphon S_j which is similar to S_i, $S_i \sim S_j$.*

This corollary follows as a consequence of the definition of siphons and similar siphons.

□

Corollary 4.4 *If a net $\mathcal{M}$ has parallel paths, then the set of essential siphons for $\mathcal{M}'$, a net obtained from $\mathcal{M}$ by removing one of the parallel paths, is also a set of essential siphons for $\mathcal{M}$.*

This corollary is a straightforward consequence of Corollary 4.3 and the definition of essential siphons. □

Eliminating nonessential siphons for $\mathcal{M}$ can thus be performed by first reducing $\mathcal{M}$ until it has no parallel paths, and then finding minimal/basis siphons in the reduced net. A procedure that identifies parallel paths in a given net is shown in Appendix C.

4.2.4.2 Alternate Paths

Informally, alternate paths are delimited by two or more pairs of transitions which envelop simple paths. In addition, all of the transition pairs share a common simple path, known as the *base*, as illustrated in Figure 4.5.

Definition: An *alternate path* in a net $\mathcal{N} = (P, T, A)$ is a collection of disjoint, simple paths $path(t_{i_1}, t_{j_1})$, $path(t_{i_2}, t_{j_2}), \ldots, path(t_{i_n}, t_{j_n})$, $t_{i_\ell} \neq t_{i_k}$, $t_{j_\ell} \neq t_{j_k}$, for $1 \leq \ell < k \leq n$, with an additional simple path (called the *base*), $path(p_i, p_j)$ connected to all t_{i_ℓ} and t_{j_ℓ}, $(t_{i_\ell}, p_i) \in A, (p_j, t_{j_\ell}) \in A, 1 \leq \ell \leq n$.

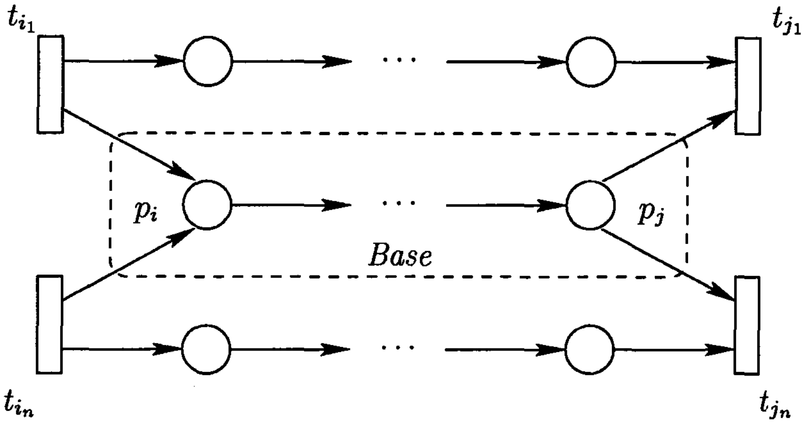

Figure 4.5: Alternate paths in a Petri net

Corollary 4.5 *For alternate paths $\pi_1, \ldots, \pi_k$ with base π_0, if* places(π_i) *is a subset of a siphon S_j in $\mathcal{M}$, then* places(π_0) *is a subset of another siphon S_ℓ which is similar to S_j, $S_j \sim S_\ell$.*

This corollary is a straightforward consequence of the definition of essential siphons and alternate paths. □

An algorithm to identify all alternate paths in a net $\mathcal{N}$ is described in detail in Appendix C.

Corollary 4.6 *If a net $\mathcal{M}$ has alternate paths, then the set of essential siphons for $\mathcal{M}'$, a net obtained from $\mathcal{M}$ by removing the base of alternate paths, is also a set of essential siphons for $\mathcal{M}$.*

This corollary is a straightforward consequence of the definition of essential siphons and alternate paths. □

By removing the bases of all alternate paths in a net, siphon extraction becomes less troublesome since the number of (inessential) siphons is reduced.

4.2.5 Deadlock Checking

The reduction of parallel and alternate paths preserves the deadlocks of the original model, so a simpler, reduced net can be examined for deadlock. If none of the minimal siphons can be emptied of their tokens, then the net is deadlock-free. If, however, (some) minimal siphons can be emptied and the resulting marking is not dead, then further analysis is required to determine whether or not a deadlock actually exists in the net. Therefore, a systematic, siphon-based verification of deadlocks in marked Petri nets recursively tries to empty as many siphons as possible. The performance of this procedure is improved if only essential siphons are considered, but this essentiality of siphons is not necessary.

The recursive algorithm to detect the presence of a deadlock is presented in Figure 4.6. While this deadlock detection algorithm is believed to be original, other techniques related to deadlock prevention and avoidance are also available [63]. The algorithm takes a marked net $(\mathcal{N}, m)$ and the set of minimal and basis siphons, S and S_b, respectively, and determines if there is a sequence in which the siphons can be emptied to produce a deadlock.

The deadlock function initially tests the marking of the net to determine if it is dead. If so, the function returns immediately, terminating any recursion. If not, the

```
func deadlock($\mathcal{N}, m, S, S_b$) : *boolean*
  begin
    var $m'$,   /* the new marking */
        $S'$,   /* the new siphon set */
        $n$,    /* the minimum number of tokens in a siphon */
        $v$;    /* the minimizing firing vector */

    if enable($\mathcal{N}, m$) $= \emptyset$ then
      return true
    endif;
    if $S \neq \emptyset$ then
      for each $s$ in $S$ do
        $v, n :=$ lp_minimize($\mathcal{N}, m, s$);
        if not zero($v$) and $n = 0$ and feasible($\mathcal{N}, m, v$) then
          $m' := m + \mathbf{C} \times v$;
          $S' :=$ marked($m', S$);
          if deadlock($\mathcal{N}, m', S', S_b$) then
            return true
          endif
        endif
      endfor
    endif;
    if $S_b \neq \emptyset$ then
      return deadlock($\mathcal{N}, m, S_b, \emptyset$)
    endif;
    return false
  end
```

Figure 4.6: Function deadlock

function iterates over each marked siphon, testing if the siphon can become empty. The lp_minimize function takes a marked net and a siphon and attempts to minimize the number of tokens in the siphon using linear programming. This function returns the minimum number of tokens that the siphon can possess (n) as well as a firing vector which minimized the tokens (v). If the siphon can be emptied and the vector is feasible, the marking of the net is updated and the marked function is used to determine the subset of siphons that still possess tokens under the new marking. The **deadlock** function is then called recursively to try to empty other siphons, if necessary.

Because the number of minimal siphons is typically quite small, the **deadlock** function is initially run on the set of minimal siphons to determine if they can be emptied to produce a deadlock. The initial invocation of the function is as follows:

$$\textsf{deadlock}(\mathcal{N}, m_0, S_m, S_b)$$

where m_0 is the initial marking S_m is the set of minimal siphons and S_b is the set of basis siphons. If a deadlock is not obtained by analyzing minimal siphons, the function undergoes a second round of recursion, as shown near the bottom of Figure 4.6:

$$\textsf{deadlock}(\mathcal{N}, m, S_b, \emptyset)$$

The same algorithm is used to identify a deadlock, but this time the basis siphons, S_b are used instead of the minimal siphons. If no sequence of basis siphons can be found which, when emptied, results in a deadlock, then the net is deadlock-free. In the worst case, due to the implicit backtracking, the complexity of the **deadlock** function is exponential with respect to the number of siphons. But because a deadlock can usually be reached by several paths, even large net models can be analyzed quite efficiently as illustrated in Chapter 6.

4.2.6 Example Two

To demonstrate other features of the deadlock algorithm, the unbounded net presented in Figure 4.7 is analyzed. This net has several parallel paths and an alternate path. The eliminated parallel paths and the base of the alternate path are denoted

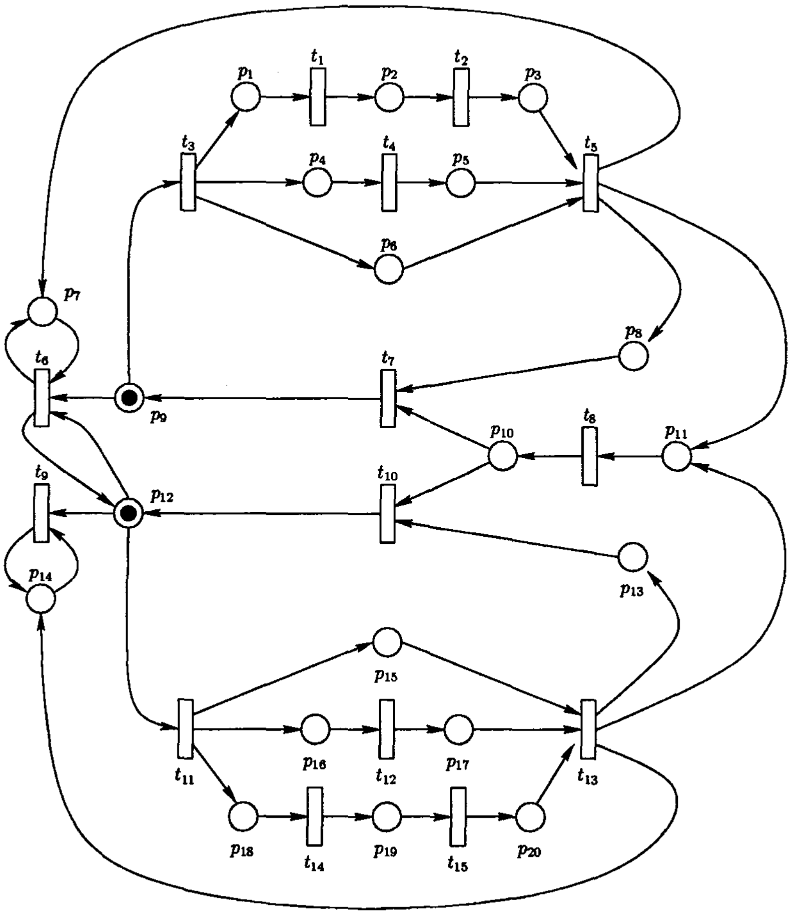

Figure 4.7: Petri net for example two

by dashed and dotted lines, respectively, in Figure 4.8. Elimination of these paths does not adversely affect the deadlock analysis of the net.

The original net has a total of 91 basis siphons, 15 of which are minimal siphons, and another 15 of which are siphon-traps. Although the number of siphons is not particularly large, removing the parallel and alternate paths can dramatically reduce the number of siphons in the net that need to be analyzed. After simplification, the reduced net has just four basis siphons, two of which are minimal. The other two siphons are actually (marked) siphon-traps, so they can be disregarded for the

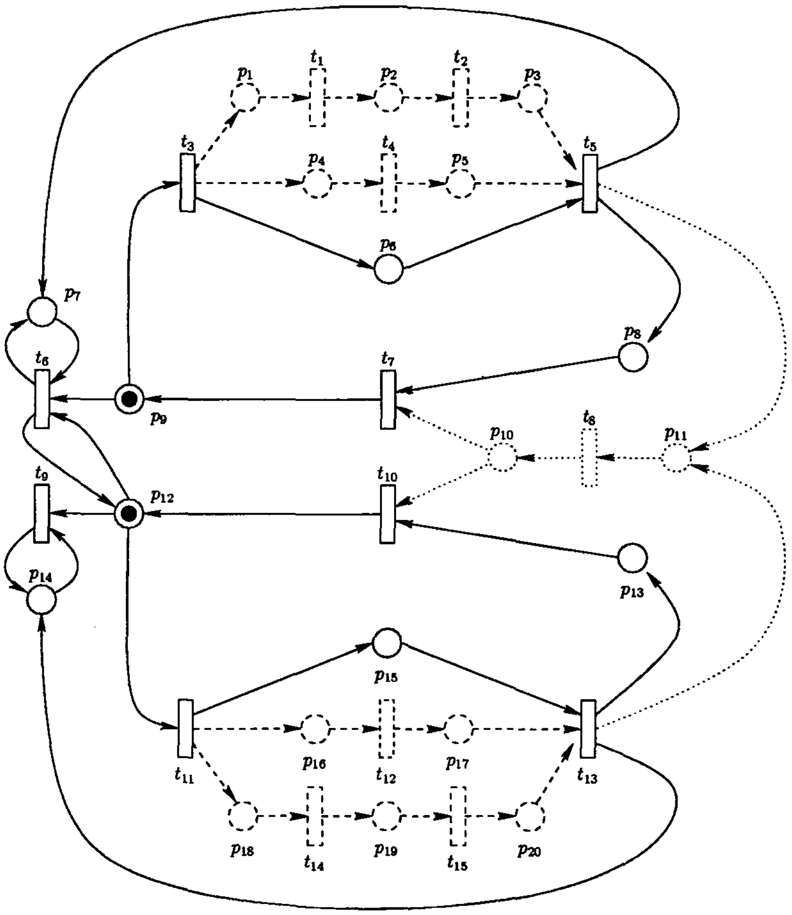

Figure 4.8: A Petri net with parallel and alternate paths identified

purposes of deadlock analysis as they cannot become empty. As a result, the relevant basis siphons and the minimal siphons are identical in this case. The basis and minimal siphons are shown in Table 4.3 and the minimal siphons are illustrated in Figure 4.9. The constraints, as deduced from the connectivity matrix of the reduced net are presented in Table 4.4. (The constraint that each transition must fire a non-negative number of times is not explicitly given in the table.) Note that the "self-loops" between the place/transition pairs p_7/t_6, p_{12}/t_6 and p_{14}/t_9 result in extra constraints that must be satisfied by the linear programming minimization.

Table 4.3: Siphons in Figure 4.9

Minimal Siphons:	$S_1 = \{p_6, p_8, p_9\}$, $S_2 = \{p_{12}, p_{13}, p_{15}\}$
Basis Siphons:	$S_3 = \{p_6, p_8, p_9\}$, $S_4 = \{p_6, p_7, p_8, p_9\}$, $S_5 = \{p_{12}, p_{13}, p_{15}\}$, $S_6 = \{p_{12}, p_{13}, p_{14}, p_{15}\}$

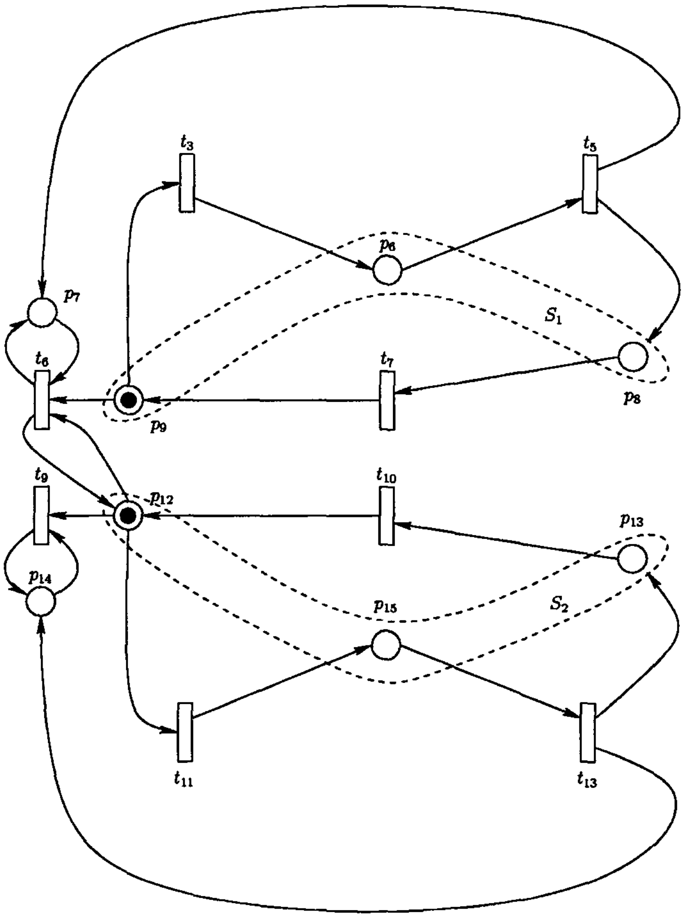

Figure 4.9: A Petri net with parallel and alternate paths removed

The marked net and the minimal siphons (S_1, S_2), both of which are marked, are passed into the deadlock function. The objective function $1 - x_{t_6}$, corresponding to S_1, can be minimized to zero by the firing vector $[1, 1, 1, 1, 0, 0, 0, 0]$ which corresponds

Table 4.4: Constraints for the Petri net of Figure 4.9

Place	Constraint
p_6	$x_{t_3} - x_{t_5} \geq 0$
p_7	$x_{t_5} + x_{t_6} \geq 0$
p_7	$x_{t_5} - x_{t_6} \geq 0$
p_8	$x_{t_5} - x_{t_7} \geq 0$
p_9	$-x_{t_3} - x_{t_6} + x_{t_7} + 1 \geq 0$
p_{12}	$x_{t_6} - x_{t_9} + x_{t_{10}} - x_{t_{11}} + 1 \geq 0$
p_{12}	$-x_{t_6} - x_{t_9} + x_{t_{10}} - x_{t_{11}} + 1 \geq 0$
p_{13}	$-x_{t_{10}} + x_{t_{13}} \geq 0$
p_{14}	$x_{t_9} + x_{t_{13}} \geq 0$
p_{14}	$-x_{t_9} + x_{t_{13}} \geq 0$
p_{15}	$x_{t_{11}} - x_{t_{13}} \geq 0$

to the feasible firing sequence (t_3, t_5, t_7, t_6). This firing sequence marks places p_7 and p_{12}. The function then recurses, checking the updated siphon set $\{S_2\}$ and the new marking. The objective function corresponding to S_2, *i.e.*, $1 - x_{t_9}$, is then minimized to zero by the firing vector $[0, 0, 0, 0, 1, 1, 1, 1]$. The corresponding feasible firing sequence is $(t_{11}, t_{13}, t_{10}, t_9)$, which marks p_7 and p_{14}. This is a dead marking which causes the recursion to unfold.

This example also shows that the ordering of siphons can influence the behaviour of the deadlock algorithm. If the siphons are analyzed by the deadlock function in reverse order (*i.e.*, first S_2 then S_1) then S_2 would become empty by firing $(t_{11}, t_{13}, t_{10}, t_9)$ resulting in p_9 and p_{14} becoming marked. However, at this point, S_1 cannot be emptied of its token since t_6 can never fire. Therefore, the function would return `false`, causing the recursion to unfold and an attempt would then be made to empty the next siphon, S_1, in the original marking. This attempt would be successful and the function would recurse with the new siphon set $\{S_2\}$. The remaining siphon in this set, S_2, could also be emptied, producing the same deadlock as demonstrated earlier.

A further example in which the set of relevant basis siphons is different than the set of minimal siphons is presented in Chapter 6.

The existence of deadlock is used to assess component compatibility, as will be described in Chapter 5. Prior to this, the fundamental notions of *interface models* and *interface languages* are needed, which are described in the next section.

4.3 Interface Models

Component interfaces are represented by cyclic Petri nets in which labels are associated with transitions.

Definition: A model of a component's interface is a labelled Petri net:

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

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 labelling function $\ell_i : T_i \to L_i \cup \{\varepsilon\}$, where ε is the empty label, $\varepsilon \notin L_i$, and F_i is a set of *final markings*, $F_i \subseteq M(\mathcal{M}_i)$. Final markings are used to indicate sequences of firings in cyclic nets. This is somewhat similar to the concept of final states in finite automata.

It is believed that requiring an interface net to be live is overly restrictive, hence only the weaker condition of deadlock-freeness is imposed upon the net. While this may mean that some of the services may become disabled, this may have been the intention of the original interface designer, particularly if some services can be used just once or a limited number of times. A component interface is usually represented by a net in which $F_i = \{m_i\}$, *i.e.*, the set of final markings contains just the initial marking. A simple example of an appropriately marked and labelled interface is presented in Figure 4.10.

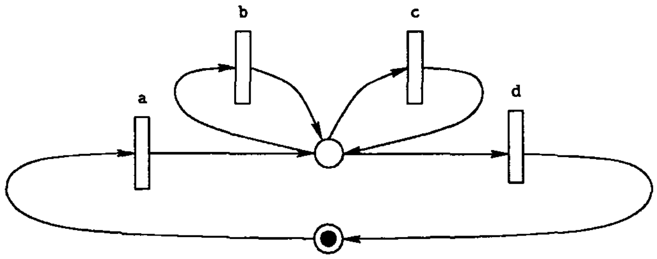

Figure 4.10: A component interface with services a,b,c and d

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*) [77].[1]

In the context of a provider interface, a labelled transition can be thought of as a *service* provided by that component. Each transition provides only one service. Labelled transitions on the provider essentially denote entry points into the component. It should be noted that it is possible to have unlabelled transitions on an interface (*i.e.*, labelled by ε). Such transitions may be needed to implement behavioural logic of the interface and do not actually constitute a service.

Since the services provided by a component need to be uniquely identified, it is

[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.

required that each service in each p-interface has exactly one labelled representation:

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

In addition to the uniqueness of the labelled transitions in each p-interface, all providers must be ε−conflict-free:

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

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 is assumed that if the p-interface requires parameters from the r-interface, then the appropriate number and types parameters are delivered by the r-interface. Similarly, it is assumed that the p-interface generates an appropriate return value to the r-interface, if required.

Another assumption is that if an r-interface requests an 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).

4.4 Interface Languages

Some proposals have restricted interface behaviour to regular languages, or modest variations thereof [85, 87]. However, by employing Petri nets, this model allows for significant flexibility in the protocol language between components [78]. 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. It is known that Petri net languages include all regular languages, a subset of context-free languages and a subset of context-sensitive languages [78].

Possible sequences of services provided by a p-interface are determined by the transition labels of all possible firing sequences in the Petri net model of an interface.

Definition: 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 labelling firing sequences which begin with m_i and end at one of the final markings:

$$\mathcal{L}(\mathcal{M}_i) = \{\ \ell(\sigma) \mid \sigma \in \mathcal{F}(\mathcal{M}_i)\ \wedge\ m_i \xrightarrow{\sigma} m\ \wedge\ m \in F_i\}$$

where $\ell(t_{i_1} \dots t_{i_k}) = \ell(t_{i_1}) \dots \ell(t_{i_k})$.

As an example, the language describing the behaviour of the interface presented in Figure 4.10 with $F = \{m_i\}$ is defined by the regular expression (a(b|c)*d)*.

4.5 Summary

A formal model representing the interface of a component by a labelled Petri net has been introduced. This model captures the behavioural properties of a component's interface which can also be characterized as the language generated by the model. In the next chapter, this model is used to assess the compatibility between a provider and requester component by studying the structural and linguistic properties of the respective interface models.

Chapter 5

Component Composition and Compatibility

As described earlier, in the context of software architectures and component-based programming, there is increasing emphasis on the integration phase of the software development process. This chapter uses the concept of Petri nets (introduced in the previous chapter) to propose a foundation upon which the composition of two or more components can be analyzed.

Such a composition must enforce a compatible sequence of operations between components providing services (*provider components*) and components that request them (*requester components*). For the purposes of this chapter, the component that initiates the interaction and issues the operations will be deemed the requester and the other component will become the provider. The structural properties of the resulting composition can be analyzed to verify the compatibility of the component integration. With a formal method of establishing component compatibility, it may be possible to provide some level of automation to the tedious but important process of system integration.

5.1 Component Compatibility

Compatibility of two components is determined by the behaviour at their respective interfaces. For two components to interact, the provided services must be compatible with requested ones. This means that not only must all the services required by the requester be made available by the provider, but also that any sequence of services demanded by the requester must be satisfied by the provider.

Definition: A requester interface $\mathcal{M}_i$ and a provider interface $\mathcal{M}_j$ are *compatible* iff $\mathcal{L}(\mathcal{M}_i) \subseteq \mathcal{L}(\mathcal{M}_j)$.

This definition implies that the provider's alphabet L_j must be a superset of the requester's alphabet L_i, $L_i \subseteq L_j$, although usually it will be assumed that $L_i = L_j$ because the symbols in $L_i - L_j$ obviously have no influence on the compatibility of the components. If the nets representing the requester and provider interfaces are bounded, and a provider interface is interacting with a single requester interface, the compatibility can be verified directly on the basis of the definition of interface compatibility.

Corollary 5.1 *The language of a bounded interface $\mathcal{M}_i$ is regular, so it can be represented by a deterministic finite automaton.*

Proof: A nondeterministic finite automaton, $\mathcal{A}$, is usually defined as $\mathcal{A} = (S, A, \Delta, s_0, F)$ where S is a set of states, A is the alphabet, Δ is the transition relation and is a subset of $S \times (A \cup \varepsilon) \times S$, $s_0 \in S$ is the initial state and $F \subseteq S$ is the set of final, or accepting, states.

If the interface $\mathcal{M}_i = (P_i, T_i, A_i, L_i, \ell_i, m_i, F_i)$ is bounded, its reachability set, $\mathcal{R}(\mathcal{M}_i) = (M(\mathcal{M}_i), A, m_0)$, is finite, so it can be used as the set of states of a finite automaton defining the language $\mathcal{L}(\mathcal{M}_i)$:

$$\mathcal{A}(\mathcal{M}_i) = (M(\mathcal{M}_i), L_i, \Delta, m_0, F_i)$$

where:

$$\Delta \subseteq M(\mathcal{M}_i) \times (A \cup \{\varepsilon\}) \times M(\mathcal{M}_i) \text{ and}$$
$$(m, a, m') \in \Delta \Leftrightarrow \exists t \in T : m \overset{t}{\mapsto} m' \ \wedge \ \ell_i(t) = a \text{ and}$$
$$(m, \varepsilon, m') \in \Delta \Leftrightarrow \exists t \in T : m \overset{t}{\mapsto} m' \ \wedge \ \ell_i(t) = \varepsilon.$$

The construction of $\mathcal{A}(\mathcal{M}_i)$ guarantees that $\mathcal{L}(\mathcal{M}_i) = \mathcal{L}(\mathcal{A}(\mathcal{M}_i))$. □

$\mathcal{A}(\mathcal{M}_i)$ is, in general, a nondeterministic automaton with ε-transitions, which, however, can be converted to an equivalent deterministic finite automaton [62].

In many cases, the finite automaton defining the language of an interface can be derived directly from the net representing the interface. For bounded nets, the compatibility can be verified by simple operations on the interface languages.

Corollary 5.2 *Bounded requester and provider interfaces $\mathcal{M}_i$ and $\mathcal{M}_j$, respectively, are compatible iff*

$$\mathcal{L}(\mathcal{M}_i) \cap \overline{\mathcal{L}(\mathcal{M}_j)} = \emptyset,$$

where $\overline{\mathbf{L}}$ is the complement of $\mathbf{L}$.

Proof: $\mathbf{L}_1 \subseteq \mathbf{L}_2 \Leftrightarrow \mathbf{L}_1 \cap \overline{\mathbf{L}_2} = \emptyset$. □

For regular languages $\mathbf{L}_1$ and $\mathbf{L}_2$, the condition $\mathbf{L}_1 \cap \overline{\mathbf{L}_2} = \emptyset$ can be checked because regular languages are closed under complementation and intersection, so checking the emptiness of a language is equivalent to checking if the set of accepting (or final) states in the finite automaton defining the language is empty.

The states that are present in an automaton that accepts the intersection of the two languages can be determined by the algorithm shown in Figure 5.1. The function takes two automata $\mathcal{A}_1 = (S_1, A, \delta_1, s_1, F_1)$ and $\mathcal{A}_2 = (S_2, A, \delta_2, s_2, F_2)$ with common alphabets and disjoint sets of states ($S_1 \cap S_2 = \emptyset$). In the worst case, this function will return a list containing $|S_1| \times |S_2|$ states. However, pragmatically, the number of states will be less, depending upon the number of transitions in each automaton.

```
func product($\mathcal{A}_1, \mathcal{A}_2$) : $state\_list$
   begin
      var  $states := \{(s_1, s_2)\}$;
           $new :=< (s_1, s_2) >$;

      while $new \neq<>$ do
         $s :=$ head($new$);
         $new :=$ tail($new$);
         for each $a$ in $A$ do
            $s' := (\delta_1(s.one, a), \delta_2(s.two, a))$;
            if $s' \notin states$ then
               $states := states \bigcup \{s'\}$;
               $new :=$ append($new, s'$)
            endif
         endfor
      endwhile;
      return $states$
   end;
```

Figure 5.1: Function product

If the interfaces are unbounded or if a provider interface interacts with several requester interfaces, a different approach to verifying the compatibility is needed, in which the inclusion of requester and provider languages is checked indirectly, by checking properties of the composition of requester and provider interfaces.

5.2 Component Composition

This section provides an overview of attempts to compose two or more interface nets together under a variety of circumstances. Some of the advantages and disadvantages of these strategies are discussed. A new model of composition is then proposed which addresses the issues that arise from the discussed composition attempts.

5.2.1 Simple Composition Models

The composition of a requester and provider nets, $\mathcal{M}_i$ and $\mathcal{M}_j$, respectively, can be defined in many ways, and several versions of composition have been proposed in the literature [57, 92]. In its simplest form, composition can be performed by "fusing" (some) transitions in the two nets; this fusion performs the synchronization of the corresponding operations.

For the purpose of component composition, the fusion of transitions with the same labels is possible in simple cases, as outlined in Figure 5.2 [24]. This technique is very straightforward and can be useful in a wide variety of circumstances. This strategy can also be used to compose components when the requester interface uses the same operation more than once as described by the COSY approach [51]. Unfortunately, this results in a proliferation of labelled transitions, which may make subsequent analysis of the composed net challenging. This can be especially true when a requester uses a provider's service many times or when repetition occurs over two or more components. Also, if two or more requesters are involved in the com-

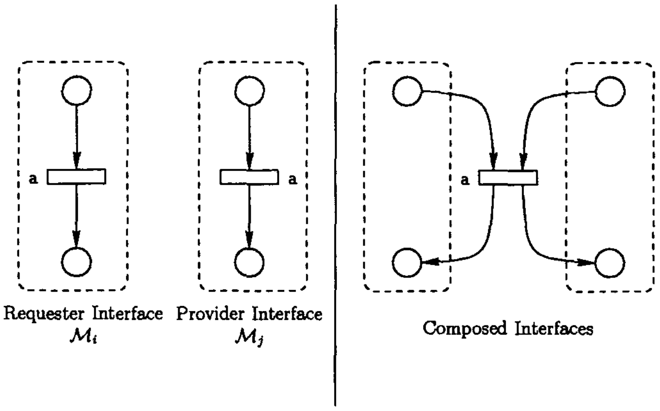

Figure 5.2: Fusion of a requester and provider service

position, the requesters themselves would have to be connected together. This may be problematic in cases in which the requesters are to maintain independence from one another. To alleviate these issues, it is desirable to have each provider's service appear only once in the composed net. This can be done by extending the composition model slightly, as illustrated in Figure 5.3. Each transition representing a service operation to be employed during the composition is "extracted" from the provider and the corresponding transitions in the requester's interface are replaced with synchronizing transitions to coordinate their interaction with the shared transition of the provider. A very similar strategy can be used when fusing multiple requester interfaces to a provider, as illustrated in Figure 5.4. Moreover, the operation a may be composed of some other operations, as in a hierarchical approach [37]. For example, Figure 5.5 shows the operation a implemented as a simple sequence of a_1 and a_2 in a requester. This requester can be hierarchically decomposed into its underlying net prior to composition with a provider.

More complex hierarchical constructs are possible. For example, if the hierarchical

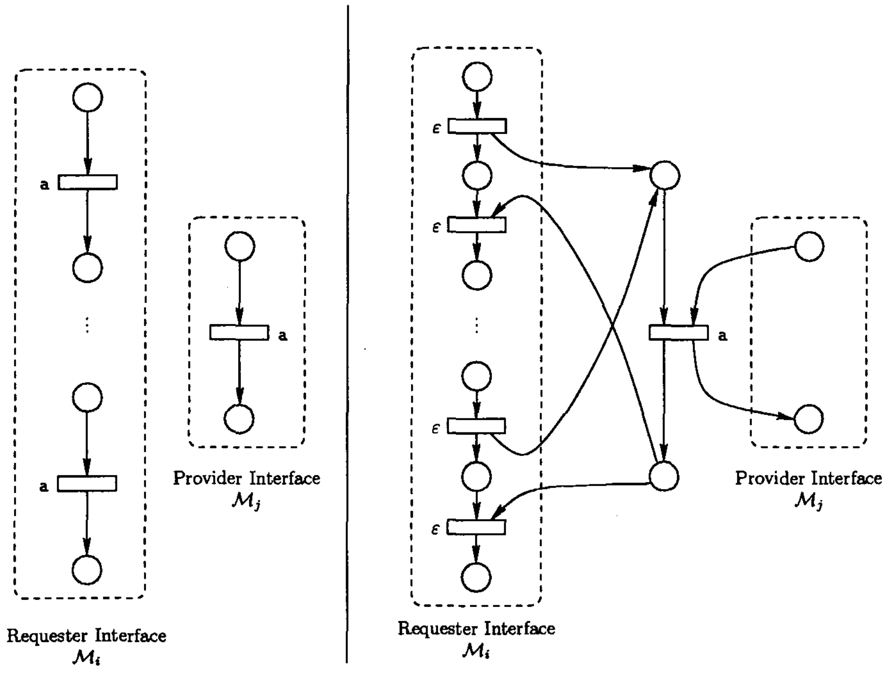

Figure 5.3: Fusion with the same operation requested two times

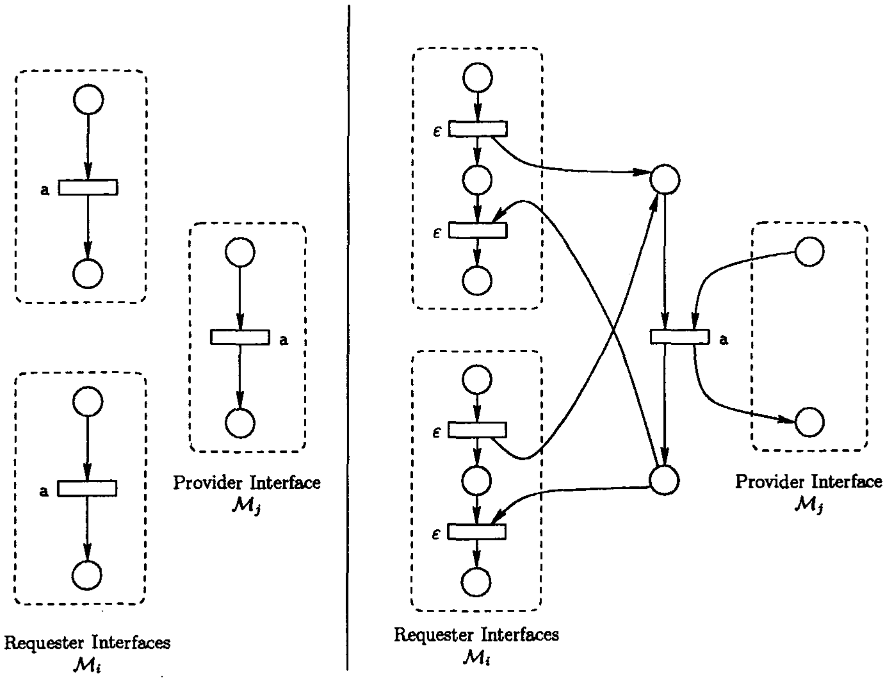

Figure 5.4: Fusion with multiple requesters

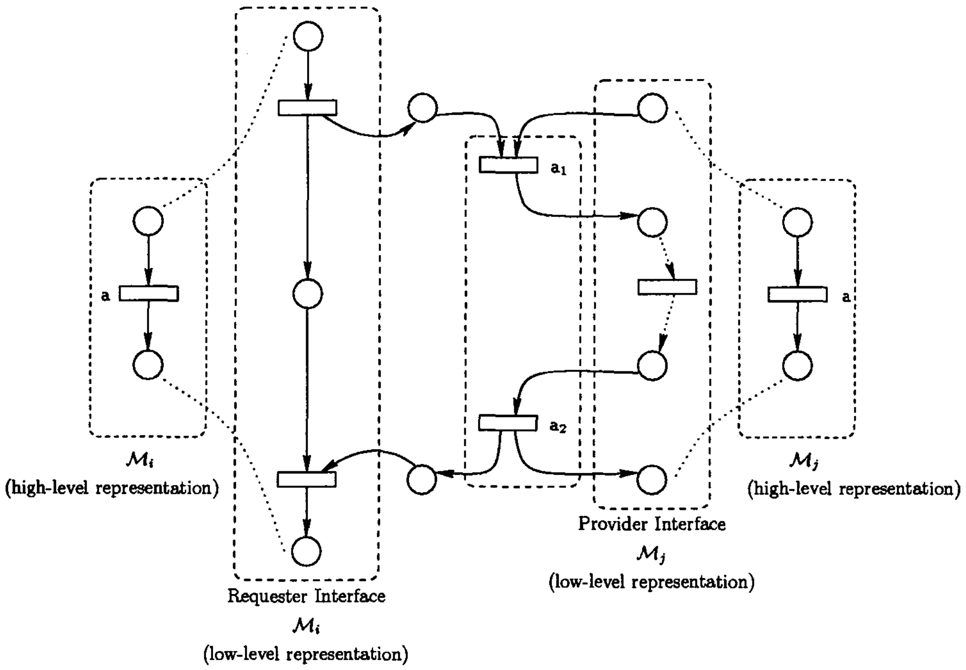

Figure 5.5: Elementary hierarchical composition

sequence is used more than once in the composition, then this can be implemented by factoring the high-level operation a (*i.e.*, the transition labelled by a) from all participants involved in the composition. The composition is shown in Figure 5.6. Similarity with Figure 5.3 should be observed.

Performing composition in this manner allows for several different modes of interaction to occur between interfaces. In addition, this composition model resolves some ambiguity with respect to the semantics of the composed net since there is only one instance of the transition representing the service operation after the composition has taken place. Indeed, this method of composition is structurally similar to that described by Lauer and Campbell [60] which uses Petri nets to represent path programs. This method is also described further in the COSY-style approach [51]. While

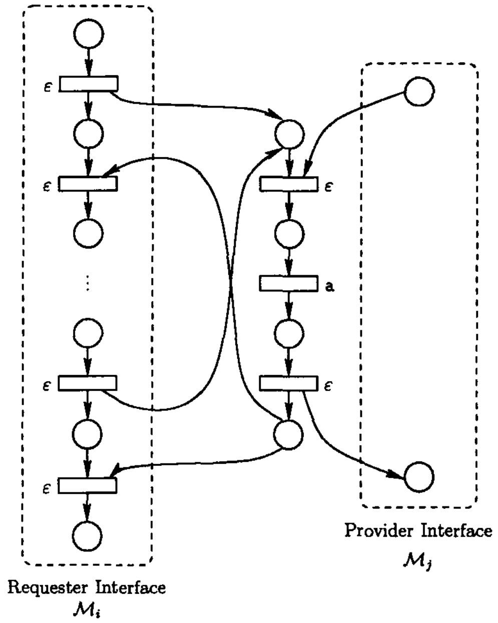

Figure 5.6: Hierarchical composition with the same operation requested twice.

this strategy of composition is reasonably flexible, it suffers from some drawbacks. This method appears to assume a semantic symmetry between programs which may not be strictly true in the context of the composition of provider and requester software components. For example, in the case when a free-choice structure exists in the requester, this model of composition can cause the provider to artificially impose its sequence of operations upon any requesters. For example, Figure 5.7 demonstrates a simple example in which a requester has a free-choice structure that allows it to invoke a or b operations in any order. However, after composition, it is denied the ability to invoke the b operation before the a operation due to the structure of the provider. Consequently, this composition model is inadequate, since the sequencing constraint of the provider is incompatible with that of the requester. A model of composition must ensure that the sequence of operations of the provider cannot affect the

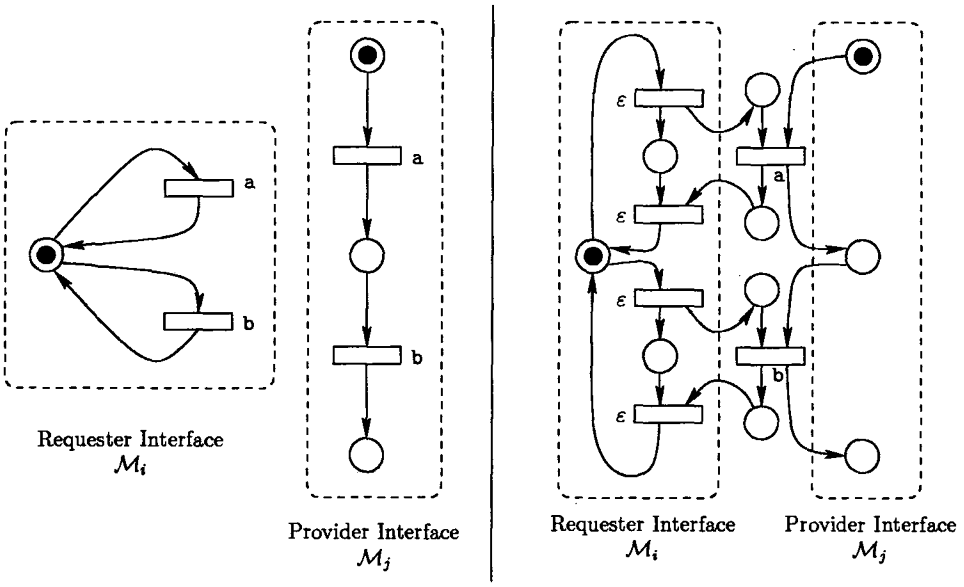

Figure 5.7: Provider imposing sequence order on a requester

sequence of requested operations.

5.2.2 Proposed Composition Model

In order to address the issues presented in the previous section, a new model of interface composition is proposed and formalized. In this model, the composition is performed by "melding" an r-interface $\mathcal{M}_i = (P_i, T_i, A_i, L_i, \ell_i, m_i, F_i)$ and a corresponding p-interface $\mathcal{M}_j = (P_j, T_j, A_j, L_j, \ell_j, m_j, F_j)$ into a single labelled Petri net $\mathcal{M}_{ij} = (P_{ij}, T_{ij}, A_{ij}, L_i, \ell_{ij}, m_{ij}, F_{ij})$, assuming $P_i \cap P_j = T_i \cap T_j = \emptyset$. While the composition strategy defined below addresses the issues described earlier, other possible approaches for composing interface nets, with possibly different properties, may exist.

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

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

The composition strategy is visually demonstrated by Figures 5.8 and 5.9 which show a requester and provider interface before and after composition, respectively.

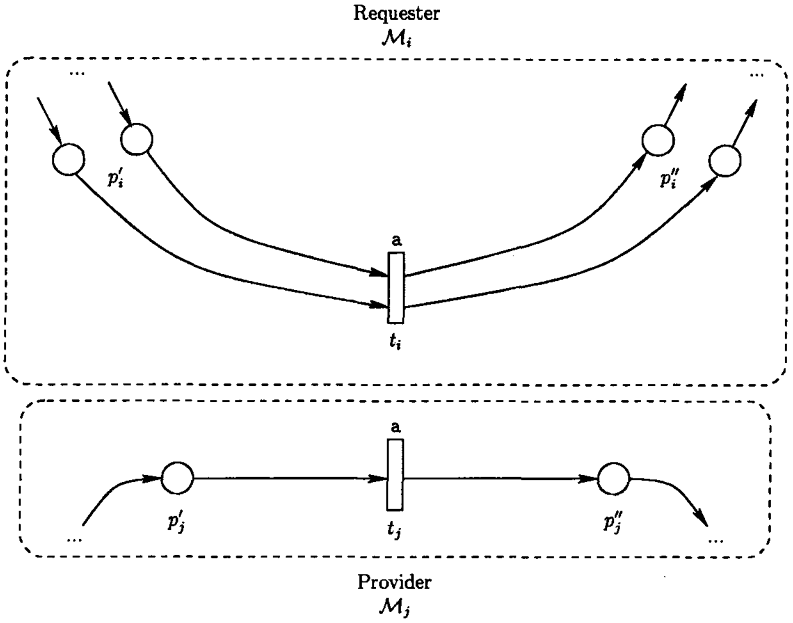

Figure 5.8: A requester and provider interface before composition

Overall, the composition of a requester and a provider interface introduces four new places and three new transitions for each common service transition, while the requester's corresponding service transitions are removed. Two of the new places (p_{t_i} and p'_{t_i} in Figure 5.9) and the three new transitions (t_i', t_i'' and t_i''' in Figure 5.9) are created for each service request in the r-interface, and the transition/place pair t_i''' and p'_{t_i} allows the requester to initiate the interaction with the provider and to direct

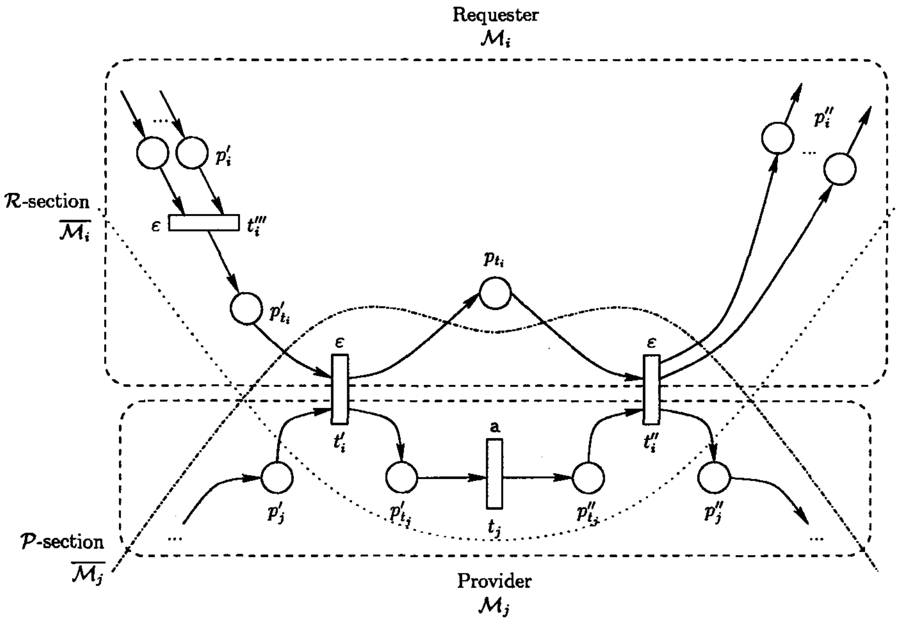

Figure 5.9: A requester and provider interface after composition

the ensuing sequence of operations. This prevents the "requester free-choice" problem described earlier. The other place (p_{t_i}) and transitions (t'_i and t''_i) serve to coordinate and serialize the requesters' interaction with the provider at the service point. The remaining pair of new places are introduced for each service of the provider interface (p'_{t_j} and p''_{t_j} in Figure 5.9) are situated on either side of the service transition and serve to coordinate the access to the service itself.

The composition of a single requester with a single provider can be formally defined as follows:

Definition: Let $P_i \cap P_j = T_i \cap T_j = \emptyset$. A composition of an r-interface $\mathcal{M}_i = (P_i, T_i, A_i, L, \ell_i, m_i, F_i)$ and a p-interface $\mathcal{M}_j = (P_j, T_j, A_j, L, \ell_j, m_j, F_j)$, denoted

$\mathcal{M}_i \rhd \mathcal{M}_j$, is a net $\mathcal{M}_{ij} = (P_{ij}, T_{ij}, A_{ij}, L, \ell_{ij}, m_{ij}, F_{ij})$ where:

$$\begin{aligned}
P_{ij} &= P_i \cup P_j \cup \{\, p_{t_i}, p'_{t_i} : t_i \in \hat{T}_i \,\} \cup \{\, p'_{t_j}, p''_{t_j} : t_j \in \hat{T}_j \,\}; \\
T_{ij} &= T_i \cup T_j - \hat{T}_i \cup \{\, t'_i, t''_i, t'''_i : t_i \in \hat{T}_i \,\}; \\
A_{ij} &= A_i \cup A_j - P_i \times \hat{T}_i - \hat{T}_i \times P_i - P_j \times \hat{T}_j - \hat{T}_j \times P_j \cup \\
&\quad \{\, (p'_i, t'''_i), (t'''_i, p'_{t_i}), (p'_{t_i}, t'_i), (t'_i, p_{t_i}), (p_{t_i}, t''_i), (t''_i, p''_i) : \\
&\qquad t_i \in \hat{T}_i \ \wedge \ p'_i \in \mathrm{Inp}(t_i) \ \wedge \ p''_i \in \mathrm{Out}(t_i) \,\} \cup \\
&\quad \{\, (p'_j, t'_i), (t'_i, p'_{t_j}), (p'_{t_j}, t_j), (t_j, p''_{t_j}), (p''_{t_j}, t''_i), (t''_i, p''_j) : \\
&\qquad t_i \in \hat{T}_i \ \wedge \ t_j \in \hat{T}_j \ \wedge \ \ell_j(t_j) = \ell_i(t_i) \ \wedge \\
&\qquad p'_j \in \mathrm{Inp}(t_j) \ \wedge p''_j \in \mathrm{Out}(t_j) \,\}; \\
\forall t \in T_{ij} : \ell_{ij}(t) &= \begin{cases} \ell_i(t), & \text{if } t \in T_i, \\ \ell_j(t), & \text{if } t \in T_j, \\ \varepsilon, & \text{otherwise;} \end{cases} \\
\forall p \in P_{ij} : m_{ij}(p) &= \begin{cases} m_i(p), & \text{if } p \in P_i, \\ m_j(p), & \text{if } p \in P_j, \\ 0, & \text{otherwise;} \end{cases}
\end{aligned}$$

$$\begin{aligned}
F_{ij} &= \{m_{ij} : P_{ij} \to \{0, 1, \ldots\} \mid \\
&\qquad m_{ij} \upharpoonright P_i \in F_i \ \wedge \ m_{ij} \upharpoonright P_j \in F_j \ \wedge \\
&\qquad \forall p \in P_{ij} - P_i - P_j : m_{ij}(p) = 0\}.
\end{aligned}$$

All new transitions introduced by the composition are assigned empty labels, and all labelled transitions of the requester are merged with the corresponding transitions of the provider. Consequently, there is no duplication of service names in the composed model. The marking function of the composition is based upon the markings of the interface nets of the underlying pair of interacting components — the new places introduced by the composition do not have any tokens initially. The set of final markings, F_{ij}, of the composed net is obtained from the final markings of the component nets. The symbol $\restriction$ is used as the "restriction" operator (of the marking functions) in this context.

The discussion of some properties of the composition uses the concepts of $\mathcal{R}$-section and $\mathcal{P}$-section of the composition, which correspond to slightly modified r-interface and p-interface models. The $\mathcal{R}$-section of this composition is defined to be $\overline{\mathcal{M}_i} = (P_{\mathcal{R}}, T_{\mathcal{R}}, A_{\mathcal{R}}, L, \ell_{\mathcal{R}}, m_{\mathcal{R}}, F_i)$, where:

$$\begin{aligned}
P_{\mathcal{R}} &= P_i \cup \{\ p_{t_i}, p'_{t_i} : t_i \in \hat{T}_i\ \} \cup \{\ p'_{t_j}, p''_{t_j} : t_j \in \hat{T}_j\ \}; \\
T_{\mathcal{R}} &= T_i - \hat{T}_i \cup \{\ t'_i, t''_i, t'''_i : t_i \in \hat{T}_i\ \} \cup \hat{T}_j; \\
A_{\mathcal{R}} &= A_i - P_i \times \hat{T}_i - \hat{T}_i \times P_i \cup \\
&\quad \{\ (p'_i, t'''_i), (t'''_i, p'_{t_i}), (p'_{t_i}, t'_i), (t'_i, p_{t_i}), (t'_i, p'_{t_j}), \\
&\quad (p'_{t_j}, t_j), (t_j, p''_{t_j}), (p''_{t_j}, t''_i), (p_{t_i}, t''_i), (t''_i, p''_i) : \\
&\quad t_i \in \hat{T}_i\ \wedge\ (p'_i, t_i) \in A_i\ \wedge\ (t_i, p''_i) \in A_i\ \wedge \\
&\quad t_j \in \hat{T}_j\ \wedge\ (p'_j, t_j) \in A_j\ \wedge\ (t_j, p''_j) \in A_j\ \};
\end{aligned}$$

$$\forall t \in T_{\mathcal{R}} : \ell_{\mathcal{R}}(t) = \begin{cases} \ell_i(t), & \text{if } t \in T_i, \\ \ell_j(t), & \text{if } t \in \hat{T}_j, \\ \varepsilon, & \text{otherwise;} \end{cases}$$

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

and the set F_i does not change.

The $\mathcal{P}$-section of this composition is $\overline{\mathcal{M}_j} = (P_\mathcal{P}, T_\mathcal{P}, A_\mathcal{P}, L, \ell_\mathcal{P}, m_\mathcal{P}, F_j)$, where

$$\begin{aligned}
P_\mathcal{P} &= P_j \cup \{\, p'_{t_j}, p''_{t_j} : t_j \in \hat{T}_j \,\}; \\
T_\mathcal{P} &= T_j \cup \{\, t'_i, t''_i : t_i \in \hat{T}_i \,\}; \\
A_\mathcal{P} &= A_j - P_j \times \hat{T}_j - \hat{T}_j \times P_j \cup \\
&\quad \{\, (p'_j, t'_i), (t'_i, p'_{t_j}), (p'_{t_j}, t_j), (t_j, p''_{t_j}), (p''_{t_j}, t''_i), (t''_i, p''_j) : \\
&\quad t_j \in \hat{T}_j \ \wedge \ (p'_j, t_j) \in A_j \ \wedge \ (t_j, p''_j) \in A_j \,\};
\end{aligned}$$

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

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

and the set F_j does not change.

The $\mathcal{R}$-section and $\mathcal{P}$-section are shown in Figure 5.9. Some elements near the boundary of the two interfaces are common to the $\mathcal{R}$-section and $\mathcal{P}$-section.

5.3 Compatibility Verification

Compatibility of an r-interface and a p-interface is verified by checking properties of their composition. In particular, as will be shown later in this section, if the net resulting from the composition of two component interfaces is deadlock-free, then the two components are indeed compatible with one another. The compatibility of an r-interface and a p-interface can also be described in terms of their languages.

Corollary 5.3 *For composition of an r-interface $\mathcal{M}_i$ and a p-interface $\mathcal{M}_j$, the language of the $\mathcal{R}$-section $\overline{\mathcal{M}_i}$ is the same as that of $\mathcal{M}_i$ and the language of the $\mathcal{P}$-section*

$\overline{\mathcal{M}_j}$ *is the same as that of* $\mathcal{M}_j$*:*

$$\begin{aligned}\mathcal{L}(\overline{\mathcal{M}_i}) &= \mathcal{L}(\mathcal{M}_i),\\ \mathcal{L}(\overline{\mathcal{M}_j}) &= \mathcal{L}(\mathcal{M}_j).\end{aligned}$$

The corollary follows from the observation that the structures of $\mathcal{M}_i$ and $\overline{\mathcal{M}_i}$ (and of $\mathcal{M}_j$ and $\overline{\mathcal{M}_j}$) are the same (*i.e.*, the additional elements are introduced as simple paths replacing single transitions), so for each firing sequence σ in $\mathcal{M}_i$, there exists a firing sequence $\overline{\sigma}$ in $\overline{\mathcal{M}_i}$ such that $\ell_i(\sigma) = \ell_i(\overline{\sigma})$. Similarly for $\mathcal{M}_j$ and $\overline{\mathcal{M}_j}$. □

Corollary 5.4 *For composition of an r-interface* $\mathcal{M}_i$ *and a p-interface* $\mathcal{M}_j$*, the language of the composed model* $\mathcal{M}_{ij} = \mathcal{M}_i \rhd \mathcal{M}_j$ *is a subset of the language of the requester,* $\mathcal{L}(\mathcal{M}_{ij}) \subseteq \mathcal{L}(\mathcal{M}_i)$*.*

Proof by contradiction: The corollary is not true, so there is a string $x \in \mathcal{L}(\mathcal{M}_{ij})$ such that $x \notin \mathcal{L}(\overline{\mathcal{M}_i})$. Let a be the first symbol in x which is not generated by $\overline{\mathcal{M}_i}$ and let a be the label of t_i, $a = \ell_{ij}(t_i)$. Since t_i is enabled in $\mathcal{M}_{ij}$, and it also is an element of $\overline{\mathcal{M}_i}$, it must also be enabled in $\overline{\mathcal{M}_i}$ by the same firing sequence as in $\mathcal{M}_{ij}$ (restricted to $\overline{\mathcal{M}_i}$) which contradicts the assumption. □

Corollary 5.5 *For composition of an r-interface* $\mathcal{M}_i$ *and a p-interface* $\mathcal{M}_j$*, the language of the composed model* $\mathcal{M}_{ij} = \mathcal{M}_i \rhd \mathcal{M}_j$ *is a subset of the language of the provider,* $\mathcal{L}(\mathcal{M}_{ij}) \subseteq \mathcal{L}(\mathcal{M}_j)$*.*

The justification is the same as for Corollary 5.4. □

Before introducing the next result, the notion of *merging* (or interleaving) sequences of symbols is introduced. Let $\mathsf{symb}(x)$ represent the set of symbols in sequence x.

Definition: If $\mathsf{symb}(x) \cap \mathsf{symb}(y) = \emptyset$, then the merging of strings x and y is a set of strings $\mathsf{merge}(x, y)$ defined as follows:

$$\mathsf{merge}(x, y) = \{z \in (\mathsf{symb}(x) \cup \mathsf{symb}(y))^* \mid h_x(z) = x \ \wedge \ h_y(z) = y\}$$

where

$$h(a_1 a_2 \ldots a_n) = h(a_1) h(a_2) \ldots h(a_n)$$

and

$$\forall a \in \mathsf{symb}(x) \cup \mathsf{symb}(y) : h_x(a) = \begin{cases} a, & \text{if } a \in \mathsf{symb}(x), \\ \varepsilon, & \text{otherwise,} \end{cases}$$

and

$$\forall a \in \mathsf{symb}(x) \cup \mathsf{symb}(y) : h_y(a) = \begin{cases} a, & \text{if } a \in \mathsf{symb}(y), \\ \varepsilon, & \text{otherwise.} \end{cases}$$

The operation $\mathsf{merge}(x, y)$ is sometimes called the *shuffle* of sequences x and y.

If $\mathsf{symb}(x) \cap \mathsf{symb}(y) = A \neq \emptyset$, the strings x and y can be merged only if their substrings composed of common symbols are identical:

$$\mathsf{merge}(x, y) = \{z \in (\mathsf{symb}(x) \cup \mathsf{symb}(y))^* \mid h_x(z) = x \ \wedge \ h_y(z) = y \ \wedge \ h_A(x) = h_A(y)\}$$

where

$$h_A(a) = \begin{cases} a, & \text{if } a \in A, \\ \varepsilon, & \text{otherwise.} \end{cases}$$

For example, merge(“abc”, “12”) = { “abc12”, “ab1c2”, “ab12c”, “a1bc2”, “a1b2c”, “a12bc”, “1abc2”, “1ab2c”, “1a2bc”, “12abc” } and merge(“1ab23c”, “a45b6c”) = { “1a45b623c”, “1a45b263c”, “1a45b236c” }.

However, the second definition of merge is equivalent to the previous one, as illustrated by the following proof:[1] If $A = \mathsf{symb}(x) \cap \mathsf{symb}(y) \neq \emptyset$, then for each

[1] Proof courtesy of Dr. R. Janicki.

merged string z, $h_A(z) = h_x(h_y(z)) = h_y(h_x(z))$, so ($h_x(z) = x$ and $h_y(z) = y$) imply $h_y(h_x(z)) = h_y(x)$, which is equivalent to $h_x(h_y(z)) = h_y(x)$. Since $h_y(z) = y$, we have $h_x(y) = h_y(x)$. Obviously, $h_x(x) = x$ and $h_y(y) = y$, so $h_x(y) = h_y(x)$ implies $h_x(h_y(y)) = h_y(h_x(x))$, *i.e.*, $h_A(y) = h_A(x)$. Consequently, $h_x(z) = x$ and $h_y(z) = y$ imply that $h_A(y) = h_A(x)$, making the latter equality unnecessary in the second definition. The merge function is a special case of the "restriction" or "concurrency" operator ($||$) which has been used in the past by Hoare to denote two processes interacting in lock-step synchronization with one another [47]. This operation has also been used by Janicki and Lauer in the preliminary development of COSY systems [51] and in the context of Petri net languages by Hack and Starke [44, 97].

Theorem 5.1 *The language of the composition of two interfaces with the same alphabet L, an r-interface $\mathcal{M}_i$ and a p-interface $\mathcal{M}_j$ is the intersection of $\mathcal{L}(\mathcal{M}_i)$ and $\mathcal{L}(\mathcal{M}_j)$:*

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

By Corollaries 5.4 and 5.5, $\mathcal{L}(\mathcal{M}_i \rhd \mathcal{M}_j) \subseteq \mathcal{L}(\mathcal{M}_i) \cap \mathcal{L}(\mathcal{M}_j)$. What remains to be shown is that $\mathcal{L}(\mathcal{M}_i) \cap \mathcal{L}(\mathcal{M}_j) \subseteq \mathcal{L}(\mathcal{M}_i \rhd \mathcal{M}_j)$.

Proof by contradiction: The theorem is not true, so there exists a string x such that $x \in \mathcal{L}(\mathcal{M}_i) \cap \mathcal{L}(\mathcal{M}_j)$ and $x \notin \mathcal{L}(\mathcal{M}_i \rhd \mathcal{M}_j)$. Let a be the first symbol in x which is not generated by $\mathcal{M}_i \rhd \mathcal{M}_j$, and let a be the label of t, $a = \ell_{ij}(t)$. Since t is an element of $\overline{\mathcal{M}_i}$ and is enabled in $\overline{\mathcal{M}_i}$ by an initial firing sequence σ_i (such a sequence must exist since $x \in \mathcal{L}(\mathcal{M}_i)$), and since t is also an element of $\overline{\mathcal{M}_j}$ and is enabled in $\overline{\mathcal{M}_j}$ by an initial firing sequence σ_j (such a sequence must exist since $x \in \mathcal{L}(\mathcal{M}_j)$), any sequence $\sigma_{ij} \in \mathsf{merge}(\sigma_i, \sigma_j)$, enables t in $\mathcal{M}_i \rhd \mathcal{M}_j$, which contradicts the assumption. □

Theorem 5.2 *Two interfaces with the same alphabet L, an r-interface $\mathcal{M}_i$ and a p-interface $\mathcal{M}_j$, are compatible iff the language of the composition, $\mathcal{L}(\mathcal{M}_i \rhd \mathcal{M}_j)$, is equal to the language of the r-interface $\mathcal{L}(\mathcal{M}_i)$.*

The theorem is a consequence of Theorem 5.1 and the definition of interface compatibility:

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

The first equivalence above follows from set theory. □

Theorem 5.3 *Two interfaces with the same alphabet L, 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.*

An r-interface $\mathcal{M}_i$ is incompatible with a p-interface $\mathcal{M}_j$ if $\mathcal{L}(\mathcal{M}_i) \not\subseteq \mathcal{L}(\mathcal{M}_j)$. It needs to be shown that such incompatibility is represented by a deadlock in $\mathcal{M}_{ij}$.

1. $\mathcal{L}(\mathcal{M}_i) \not\subseteq \mathcal{L}(\mathcal{M}_j) \Rightarrow \mathcal{M}_{ij}$ contains a deadlock.

   If $\mathcal{L}(\mathcal{M}_i) \not\subseteq \mathcal{L}(\mathcal{M}_j)$, there exists a string $x \in \mathcal{L}(\mathcal{M}_i)$, such that $x \notin \mathcal{L}(\mathcal{M}_{ij})$. Let a be the first symbol of x which is not generated by $\mathcal{M}_{ij}$, and let $a = \ell(t_k)$, $t_k \in T_{ij}$. Since t_k is enabled in $\overline{\mathcal{M}_i}$ but is not enabled in $\mathcal{M}_{ij}$, the requested service a cannot be satisfied by $\overline{\mathcal{M}_j}$, so $\mathcal{M}_j$ must be waiting for some other requested service and this creates a deadlock in $\mathcal{M}_{ij}$

2. $\mathcal{M}_{ij}$ contains a deadlock $\Rightarrow \mathcal{L}(\mathcal{M}_i) \not\subseteq \mathcal{L}(\mathcal{M}_j)$.

   Proof 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 an initial finite firing sequence $\sigma = t_{i_1}t_{i_2}\ldots t_{i_k}$ such that $E(m_k) = \emptyset$. However, in $\mathcal{M}_i$, σ_i, the firing sequence obtained by restricting σ to T_i, 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.*, $\ell(\sigma) \notin \mathcal{L}(\mathcal{M}_j)$, which 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. This model is extended (in subsequent sections) to handle several requesters interacting with a single provider as well as several requesters interacting with several providers.

5.3.1 Compatibility and Deadlock Detection

The most straightforward and most robust approach to deadlock detection is based on exhaustive exploration of the marking space of a net $\mathcal{M}$ (*i.e.*, the exploration of the set of reachable markings, $M(\mathcal{M})$), and checking if it contains any dead marking (which represents a deadlock). However, such an approach can be used only for bounded models and even for bounded models, this marking space can be huge due to the so called *state space explosion* [106]. An alternative approach can be based on structural properties of net models, and in particular, on siphons (Section 4.2). Since an unmarked siphon is a necessary condition for a deadlock (and a sufficient condition for non-liveness), verification of component compatibility can be performed by checking if any combination of the minimal and/or basis siphons of the composed net can be emptied so as to produce a deadlock. Of all such siphons, only essential siphons, as defined in Section 4.2.4, should be checked. Although the number of essential siphons depends upon the structure of the net, practical experience indicates that usually there are just a few essential siphons, which makes compatibility checking more efficient.

5.3.2 Requester and Provider Alphabets

In the definition of component composition, it is assumed that the alphabets of composed interfaces are the same or that the requester alphabet is a subset of the provider alphabet, $L_i \subseteq L_j$. If $L_j \supset L_i$, the operations which are provided but not requested have no effect on the composition, so they can all be replaced by ε. On the other hand, if the alphabet of the requester L_i is a superset of the provider L_j, $L_i \supset L_j$, then the interfaces cannot be compatible because all requested operations in the subset $L_i \dot{-} L_j$ cannot be satisfied. Consequently, if a component has several requester interfaces with different sets of requested services, each such interface is considered separately, with its set of services.

5.4 Multicomponent Composition

The previous section described the composition of a single requester interface with a single provider interface. In practice, however, a provider may have several requesters demanding its services concurrently; or a requester may demand the services of several providers. This section describes how the previous composition model can be extended to describe a variety of multicomponent interactions and compositions.

5.4.1 Multirequester Composition

In multirequester composition, several requester interfaces interact with the same provider interface. For example, multiple web clients connecting to a web service would constitute a multirequester composition. Multirequester composition is a straightforward generalization of the approach presented in Section 5.2. Figures 5.10 and 5.11 show a simple example of two requesters composed with a single provider. For clarity, the multiple arcs to and from the transition in the requesters have been removed. The composition of multiple requesters $\mathcal{M}_I = \{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_k\}$, with

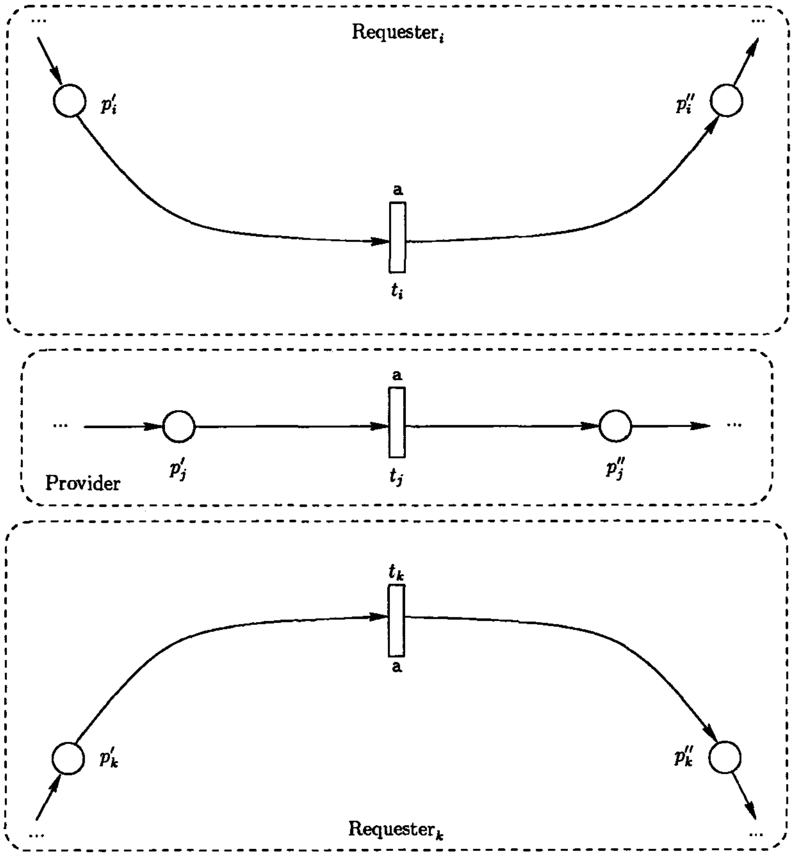

Figure 5.10: Multirequester interaction (before composition)

a single provider $\mathcal{M}_j$ is denoted by $\mathcal{M}_I \rhd \mathcal{M}_j$. Moreover, for each requester $\mathcal{M}_i$, $i = 1, 2, \ldots, k$, let $\hat{T}_i$ denote the set of labelled transitions of a single requester:

$$\hat{T}_i = \{\ t \in T_i : \ell(t) \neq \varepsilon\ \}, i \in I.$$

The set of all labelled transitions of all the requesters involved in the composition is:

$$\hat{T}_I = \bigcup_{i \in I} \hat{T}_i.$$

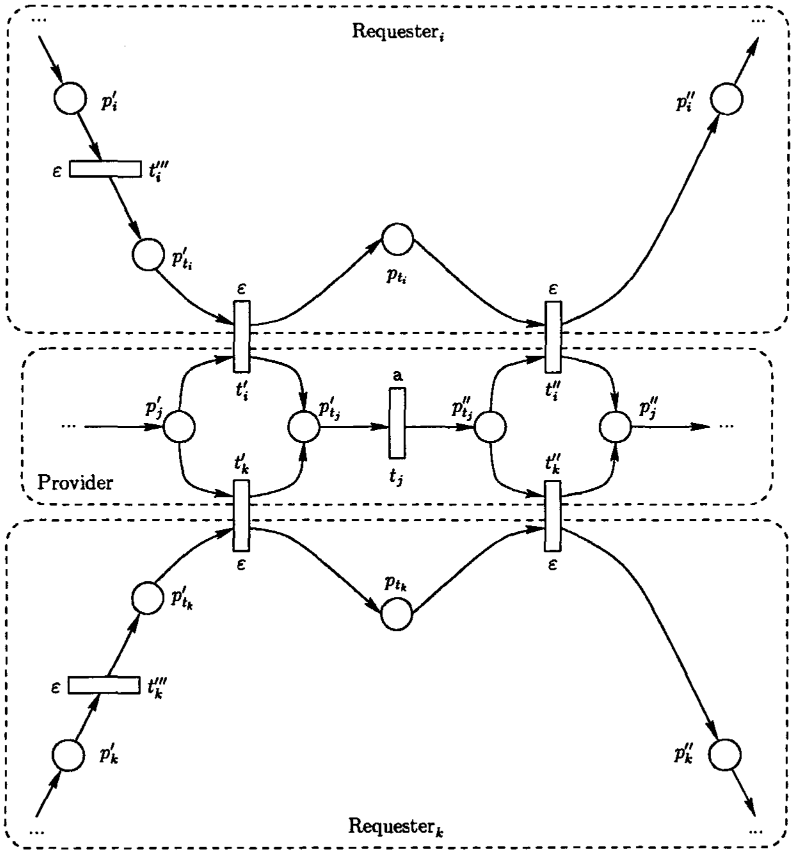

Figure 5.11: Multirequester interaction (after composition)

Finally, let the set of all the requesters' transitions (both labelled and unlabelled), all places, all arcs and all final markings be denoted, respectively, as:

$$T_I = \bigcup_{i \in I} T_i, \quad P_I = \bigcup_{i \in I} P_i, \quad A_I = \bigcup_{i \in I} A_i, \quad F_I = \bigcup_{i \in I} F_i.$$

Definition: Let $\mathcal{M}_I = \{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_k\}$ be a family of r-interfaces with the same alphabet L and with disjoint sets of places and transitions, and let $\mathcal{M}_j$ be a p-interface also with the same alphabet L. The composition of $\mathcal{M}_I$ with $\mathcal{M}_j$, denoted by $\mathcal{M}_I \rhd \mathcal{M}_j$, is a net $\mathcal{M}_{Ij} = (P_{Ij}, T_{Ij}, A_{Ij}, L, \ell_{Ij}, m_{Ij}, F_{Ij})$ where:

$$\begin{aligned}
P_{Ij} &= P_I \cup P_j \cup \{\, p_{t_i}, p'_{t_i} : t_i \in \hat{T}_i \ \wedge \ i \in I \,\} \cup \{\, p'_{t_j}, p''_{t_j} : t_j \in \hat{T}_j \,\}; \\
T_{Ij} &= T_I \cup T_j - \hat{T}_I \cup \{\, t'_i, t''_i, t'''_i : t_i \in \hat{T}_i \ \wedge \ i \in I \,\}; \\
A_{Ij} &= A_I \cup A_j - P_I \times \hat{T}_I - \hat{T}_I \times P_I - P_j \times \hat{T}_j - \hat{T}_j \times P_j \cup \\
&\quad \{\, (p'_i, t'''_i), (t'''_i, p'_{t_i}), (p'_{t_i}, t'_i), (t'_i, p_{t_i}), (p_{t_i}, t''_i), (t''_i, p''_i) : \\
&\quad\ \ t_i \in \hat{T}_i \ \wedge \ i \in I \ \wedge \ p'_i \in \mathrm{Inp}(t_i) \ \wedge \ p''_i \in \mathrm{Out}(t_i) \,\} \cup \\
&\quad \{\, (p'_j, t'_i), (t'_i, p'_{t_j}), (p'_{t_j}, t_j), (t_j, p''_{t_j}), (p''_{t_j}, t''_i), (t''_i, p''_j) : \\
&\quad\ \ t_j \in \hat{T}_j \ \wedge \ t_i \in \hat{T}_i \ \wedge \ i \in I \ \wedge \ \ell_j(t_j) = \ell_i(t_i) \ \wedge \\
&\quad\ \ p'_j \in \mathrm{Inp}(t_j) \ \wedge \ p''_j \in \mathrm{Out}(t_j) \,\};
\end{aligned}$$

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

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

$$\begin{aligned}
F_{Ij} &= \{ m_{Ij} : P_{Ij} \to \{0, 1, \ldots\} \mid m_{Ij} \upharpoonright P_I \in F_I \ \wedge \ m_{Ij} \upharpoonright P_j \in F_j \ \wedge \\
&\qquad \forall p \in P_{Ij} - P_I - P_j : m_{Ij}(p) = 0 \}.
\end{aligned}$$

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

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

The multirequester composition must take into account concurrency of requests from different r-interfaces. Therefore, the compatibility is checked for the worst case scenario, *i.e.*, the composition of all r-interfaces with the p-interface.

Definition: A family of r-interfaces $\mathcal{M}_I = \{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_k\}$, is compatible with a p-interface $\mathcal{M}_j$ iff any sequence of requests that can be issued by $\mathcal{M}_I$ can be provided by $\mathcal{M}_j$.

Theorem 5.4 *If a family of r-interfaces $\mathcal{M}_I = \{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_k\}$ is compatible with a p-interface, $\mathcal{M}_j$, then each r-interface $\mathcal{M}_i, i = 1, 2, \ldots, k$, is also compatible with $\mathcal{M}_j$.*

Proof by contradiction: The theorem is not true, so there is an r-interface $\mathcal{M}_i$ which is incompatible with $\mathcal{M}_j$ if $\mathcal{M}_I$ is compatible with $\mathcal{M}_j$. Consequently, there exists a sequence of service requests $x \in \mathcal{L}(\mathcal{M}_i)$ such that $x \notin \mathcal{L}(\mathcal{M}_j)$. However, the compatibility of $\mathcal{M}_I$ with $\mathcal{M}_j$ means that an arbitrary sequence of requests of $\mathcal{M}_I$ is satisfied by $\mathcal{M}_j$, so, in particular, a sequence z of requests starting with the requests of $\mathcal{M}_i$, $z = xy$, is satisfied by $\mathcal{M}_j$, which contradicts the assumption that $x \notin \mathcal{L}(\mathcal{M}_j)$.

□

Corollary 5.6 *If $\mathcal{M}_I = \{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_k\}$ is compatible with $\mathcal{M}_j$, then any subset $\mathcal{M}_{I'} \subset \mathcal{M}_I$ is also compatible with $\mathcal{M}_j$.*

The proof is a straightforward adaptation of the previous proof. □

Theorem 5.5 *For a family $\mathcal{M}_I$ if r-interfaces, $\mathcal{M}_I = \{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_k\}$, the compatibility of each r-interface $\mathcal{M}_i, i = 1, 2, \ldots, k$, with the same p-interface $\mathcal{M}_j$ is not a sufficient condition for compatibility of $\mathcal{M}_I$ with $\mathcal{M}_j$:*

$$(\forall \mathcal{M}_i \in \mathcal{M}_I : \mathcal{M}_i \rhd \mathcal{M}_j) \not\Rightarrow \mathcal{M}_I \rhd \mathcal{M}_j$$

Proof: Let $\mathcal{M}_I = \{\mathcal{M}_1, \mathcal{M}_2\}$ and $\mathcal{L}(\mathcal{M}_1) = \mathcal{L}((aa)^*)$, $\mathcal{L}(\mathcal{M}_2) = \mathcal{L}((bb)^*)$. Then $\mathcal{L}((abab|baba)) \subset \mathcal{L}(\mathcal{M}_I)$. Let $\mathcal{L}(\mathcal{M}_j) = \mathcal{L}((aa|bb)^*)$. $\mathcal{M}_j$ is compatible with $\mathcal{M}_1$ as well as with $\mathcal{M}_2$, but is incompatible with $\mathcal{M}_I$. □

Consequently, for multirequester composition, the maximum configuration of requesters needs to be verified for compatibility with the provider interface. Incompatibility arises if this maximum configuration introduces a deadlock in the composed net, as described in Section 5.3.

5.4.2 Multiprovider Composition

The composition discussed in the previous sections can be easily applied to the case when several providers interact with a single requester (or multiple requesters). This could take the form, for example, of a requester component that queries several databases simultaneously. In the case when a single requester interacts with several providers, the model can be split into a collection of independent requester-provider pairs in which each provider has its unique alphabet, and each such pair can then be analyzed independently.

In formal terms, it is possible that the same r-interface, $\mathcal{M}_i$, is composed with two p-interfaces $\mathcal{M}'_j$ and $\mathcal{M}''_j$, such that $L'_j \cap L''_j = \emptyset$. In such cases, $\mathcal{M}_i$ could be split into two r-interfaces, $\mathcal{M}'_i$ interacting with $\mathcal{M}'_j$, and $\mathcal{M}''_i$ interacting with $\mathcal{M}''_j$. Alternatively, $\mathcal{M}_i$ can be directly composed with both $\mathcal{M}'_j$ and $\mathcal{M}''_j$, provided that the composition uses a relevant subset of L_i, *i.e.*, $L_i \cap L'_j$ for composition with $\mathcal{M}'_j$ and $L_i \cap L''_j$ for composition with $\mathcal{M}''_j$. If $\mathcal{M}'_i$ is compatible with $\mathcal{M}'_j$ and $\mathcal{M}''_i$ with $\mathcal{M}''_j$, then $\mathcal{M}_i$ is compatible with $\mathcal{M}'_j$ and $\mathcal{M}''_j$ and vice versa, so the two approaches are equivalent.

Theorem 5.6 *Let $\mathcal{M}_J = \{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_k\}$ be a family of p-interfaces and $\mathcal{M}_i$ be an r-interface. If $\mathcal{M}_i$ is compatible with each $\mathcal{M}_j$ p-interface, $j = 1, 2, \ldots, k$, then $\mathcal{M}_i$ is compatible with $\mathcal{M}_J$.*

Proof by contradiction: If each p-interface, $\mathcal{M}_j$, $j = 1, \ldots, k$ is compatible with $\mathcal{M}_i$, then $\mathcal{M}_J$ is not compatible with $\mathcal{M}_i$, so there exists an interface, say $\mathcal{M}_j$, and a sequence of requests $s \in \mathcal{L}(\mathcal{M}_i)$ such that $\mathcal{M}_j$ is deadlocked for s. Let $\sigma_j = h(s)$, where

$$\forall a \in L_i : h(a) = \begin{cases} a, & \text{if } a \in L_j, \\ \varepsilon, & \text{otherwise.} \end{cases}$$

Since $\mathcal{M}_j$ is compatible with $\mathcal{M}_i$, σ_j must be accepted by $\mathcal{M}_i \triangleright \mathcal{M}_j$, which contradicts that $\mathcal{M}_j$ is deadlocked on σ_j. □

5.4.3 Multiprovider/Multirequester Composition

The case of multiple requesters interacting with several providers (many-to-many) can be converted to a family of multiple requester/single provider cases, and analyzed as discussed earlier. The following theorem and its proof are essentially an amalgamation of the two earlier theorems.

Theorem 5.7 *Let $\mathcal{M}_I = \{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_k\}$ be a family of r-interfaces, and $\mathcal{M}_J = \{\mathcal{M}'_1, \mathcal{M}'_2, \ldots, \mathcal{M}'_\ell\}$ be a family of p-interfaces. If $\mathcal{M}_I$ is compatible with each $\mathcal{M}'_j, j = 1, 2, \ldots, \ell$, then $\mathcal{M}_I$ is compatible with $\mathcal{M}_J$.*

Proof: The theorem is a straightforward extension of Theorems 5.4 and 5.6. □

The strategy discussed earlier can be used for compositions that involve multiple requesters and multiple providers. For example, Figure 5.12 shows two simple providers and two requesters that require the services of each provider. The composition of all four interfaces is shown in Figure 5.13. The same model of composition can be used to compose several interfaces into a single net. Because the net is bounded, reachability analysis can be used to test the net for deadlocks. Reachability analysis shows that the bottom-most requester causes the net to deadlock, for example, when it attempts to invoke operation `b` two consecutive times from the provider on the

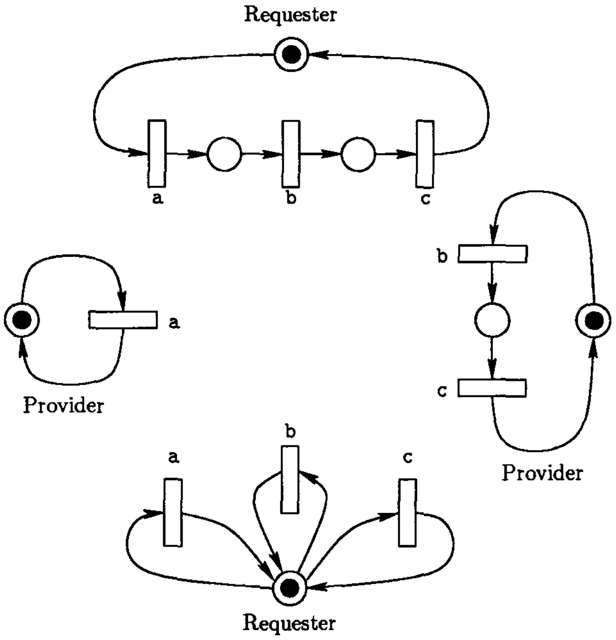

Figure 5.12: Two requester and two provider interfaces

right. This is a violation of the provider which imposes a strict ordering on the services that can be demanded of it by a requester. The deadlock can also be identified through structural analysis.

A more sophisticated example demonstrating multiprovider/multirequester interaction is presented in the following chapter.

5.5 Mixed Requester-Provider Interfaces

Normally, requester and provider interfaces are disjoint because of different logical requirements on "requested" and "provided" operations. However, if an interface $\mathcal{M}_0 = (P_0, T_0, A_0, L_0, \ell_0, m_0, F_0)$ contains some "request" operations, $L_{or} \subset L_0$, and some "provider" operations, $L_{op} \subset L_0$, $L_{or} \cap L_{op} = \emptyset$, it can be split into two interfaces

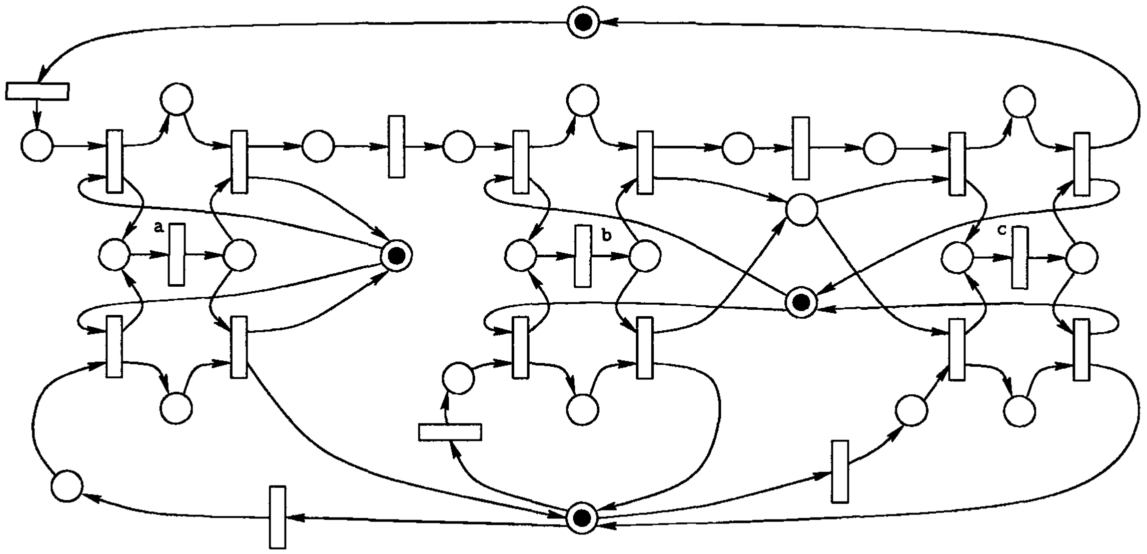

Figure 5.13: Composition of two requester and two provider interfaces

$\mathcal{M}_{\text{or}} = (P_0, T_0, A_0, L_{\text{or}}, \ell_{\text{or}}, m_0, F_0)$ and $\mathcal{M}_{\text{op}} = (P_0, T_0, A_0, L_{\text{op}}, \ell_{\text{op}}, m_0, F_0)$, where

$$\forall t \in T : \ell_{\text{or}}(t) = \begin{cases} \ell_0(t), & \text{if } \ell_0(t) \in L_{\text{or}}, \\ \varepsilon, & \text{otherwise,} \end{cases}$$

and

$$\forall t \in T : \ell_{\text{op}}(t) = \begin{cases} \ell_0(t), & \text{if } \ell_0(t) \in L_{\text{op}}, \\ \varepsilon, & \text{otherwise.} \end{cases}$$

Theorem 5.8 *If an interface $\mathcal{M}_0$ is compatible with an r-interface $\mathcal{M}_i$ and a p-interface $\mathcal{M}_j$, then $\mathcal{M}_{op}$ is compatible with $\mathcal{M}_i$ and $\mathcal{M}_{or}$ is compatible with $\mathcal{M}_j$.*

Proof by contradiction: The theorem is not true, so if an interface $\mathcal{M}_0$ is compatible with an r-interface $\mathcal{M}_i$ and a p-interface $\mathcal{M}_j$, then $\mathcal{M}_{\text{op}}$ is incompatible with $\mathcal{M}_i$ or $\mathcal{M}_{\text{or}}$ is incompatible with $\mathcal{M}_j$ and there exists a sequence $\sigma \in \mathcal{L}(\mathcal{M}_i \rhd \mathcal{M}_{\text{op}}) \cup \mathcal{L}(\mathcal{M}_{\text{or}} \rhd \mathcal{M}_j)$ which deadlocks $\mathcal{M}_{\text{or}}$ or $\mathcal{M}_{\text{op}}$. If $\mathcal{M}_{\text{op}}$ is deadlocked on σ, then there must exist a sequence $\sigma' \in L_0^*$ such that $\sigma = h'(\sigma')$ where:

$$\forall a \in L_0 : h'(a) = \begin{cases} a, & \text{if } a \in L_{\text{op}}, \\ \varepsilon, & \text{otherwise,} \end{cases}$$

and σ' deadlocks $\mathcal{M}_0$, which contradicts compatibility of $\mathcal{M}_i$ and $\mathcal{M}_0$. A similar argument can be used for a deadlocked $\mathcal{M}_{\text{or}}$. □

5.6 Examples

This section provides examples which illustrate 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.

5.6.1 Database Transactions

As a simple example of the composition of a requester and provider interface modelled as Petri nets, Figure 5.14 represents a simple database client (requester) and a database server (provider).

The first requested service is denoted by a which could represent an operation 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

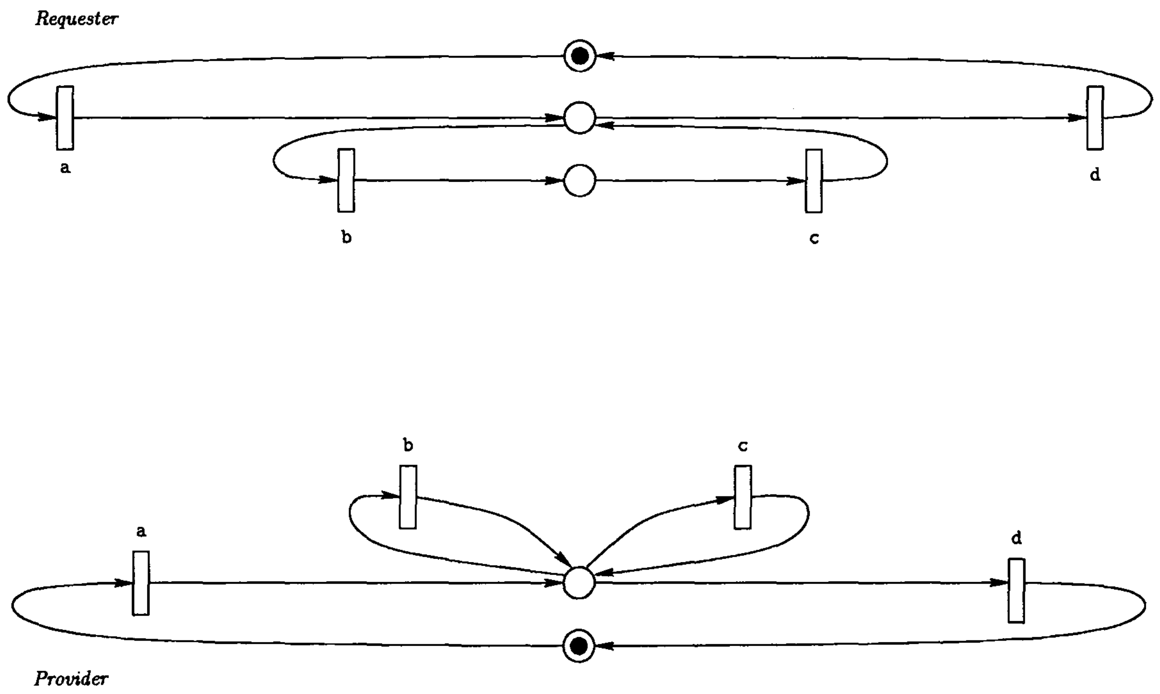

Figure 5.14: Database requester and provider interfaces

the closing of the database. If there is only one final marking which is the initial marking, *i.e.*, $F = \{m_0\}$, the cyclic behaviour of the model is represented by the regular expression (a(bc)*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. Again, if the initial marking is the only final marking, the behaviour of the database server is described by the expression (a(b|c)*d)*.

The composition of interfaces shown in Figure 5.14 creates the net shown in Figure 5.15. This composition is achieved by using the construction technique presented in Section 5.2. The composed net can be simplified as described in Section 4.2.4, so as to facilitate structural analysis. The reduced net is shown in Figure 5.16.

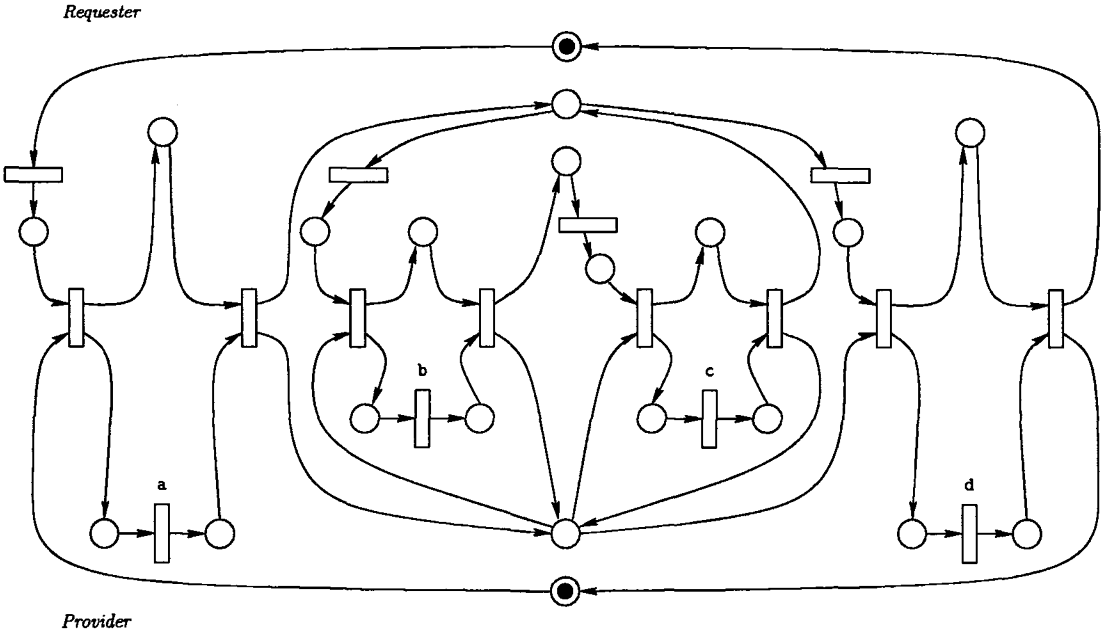

Figure 5.15: Composition of compatible database requester and provider interfaces

It should be noted that the composition given in Figure 5.15 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.

The condition $\mathcal{L}(\mathcal{M}_{\text{Requester}}) \subseteq \mathcal{L}(\mathcal{M}_{\text{Provider}})$ is obviously satisfied in this case and it can be checked that the model shown in Figure 5.15 is deadlock-free (the unreduced model is bounded and its marking space contains fifteen markings, none of which is dead). From a structural perspective, when the composed net is simplified, the reduced net contains just two minimal siphons, as denoted by the places containing the diagonal line patterns in Figure 5.16; five of the places belong to both siphons and therefore contain a crosshatch pattern. Further analysis reveals that these siphons are actually marked siphon-traps, so the net is deadlock-free since neither of the minimal

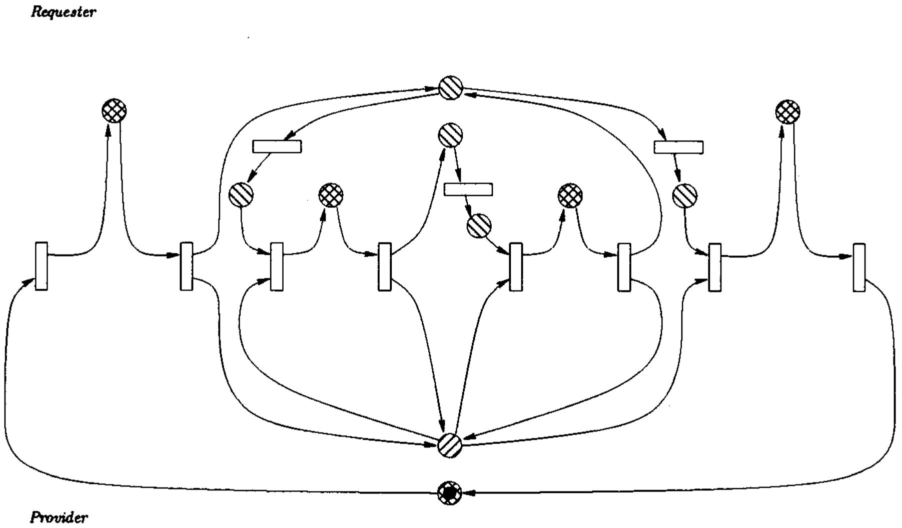

Figure 5.16: Composition of compatible database requester and provider interfaces after simplification

siphons can become empty.

As an example of incompatibility, consider the case where the roles of the p-interface and r-interface from the previous example are swapped (so the requester's language is described by (a(b|c)*d)*, and the provider's language by (a(bc)*d)*) and the interfaces recomposed. The resulting net exhibits deadlock as demonstrated by the composition shown in Figure 5.17. The dotted arcs, lines and transitions represented elements of the net which can be eliminated so as to simplify structural analysis. A deadlock situation occurs 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. Hence the resulting net is deadlocked, demonstrating incompatibility between the two interfaces. In this case, the language of the requester is a superset of the language of the provider. Other deadlocks result if the requester attempts to perform operations b or d immediately after performing operation b.

This is contrary to the provider which insists that the invocation of a b operation be followed by a c operation.

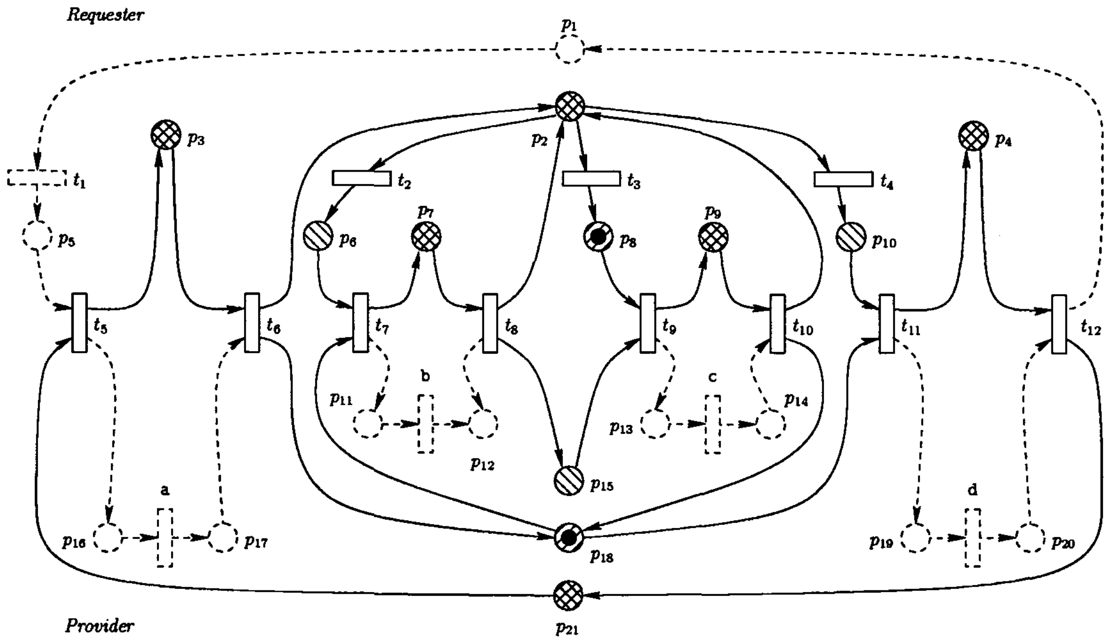

Figure 5.17: Composition of incompatible database requester and provider interfaces

In this particular case, the deadlock can be easily detected by reachability analysis since, in the original, unreduced composition, there are a total of only 18 possible markings, three of which result in deadlock. Alternatively, siphons can be used to identify the deadlock. The reduced net contains two minimal siphons that are not marked traps: $S_1 = \{p_2, p_3, p_4, p_6, p_7, p_9, p_{10}, p_{15}, p_{21}\}$ and $S_2 = \{p_2, p_3, p_4, p_7, p_8, p_9, p_{18}, p_{21}\}$. The objective functions corresponding to these siphons are $-x_2 - x_4 + x_6 + x_{10} + 1$ and $-x_3 + x_8 + 1$, respectively. To empty the siphons, linear programming is used to determine if there is a way to minimize the number of tokens in a siphon to zero subject to the constraints in Table 5.1, as obtained from the connectivity matrix of the reduced net.

Table 5.1: Constraints for the Petri net of Figure 5.17

Place	Constraint
p_2	$-x_{t_2} - x_{t_3} - x_{t_4} + x_{t_6} + x_{t_8} + x_{t_{10}} \geq 0$
p_3	$x_{t_5} - x_{t_6} \geq 0$
p_4	$x_{t_{11}} - x_{t_{12}} \geq 0$
p_6	$x_{t_2} - x_{t_7} \geq 0$
p_7	$x_{t_7} - x_{t_8} \geq 0$
p_8	$x_{t_3} - x_{t_9} \geq 0$
p_9	$x_{t_9} - x_{t_{10}} \geq 0$
p_{10}	$x_{t_4} - x_{t_{11}} \geq 0$
p_{15}	$x_{t_8} - x_{t_9} \geq 0$
p_{18}	$x_{t_6} - x_{t_7} + x_{t_{10}} - x_{t_{11}} \geq 0$
p_{21}	$-x_{t_5} + x_{t_{12}} + 1 \geq 0$

Attempting to drain S_1 is successful and results in deadlock. The resulting firing vector is $[0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0]$. (In the context of the reduced net, each element in the firing vector corresponds to transitions $t_2, \ldots, t_{12}$.) This firing vector corresponds to the firing sequence (t_5, t_6, t_3) in the reduced net. The final marking of this sequence is shown in Figure 5.17. S_2 can also be emptied to produce a deadlock. The firing vector that minimizes this siphon's objective function to zero is $[2, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0]$ which corresponds to the firing sequence $(t_5, t_6, t_2, t_7, t_8, t_2)$. Because a deadlock was identified by analyzing the minimal siphons, there is no need the analyze the basis siphons of the reduced net in this particular case.

5.6.2 Database with Nested Transactions

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

A requester and provider interface that employ nested database transactions can be represented by the Petri nets given in Figure 5.18.[2] The provider interface keeps track of the number of opened transactions by accumulating a corresponding number

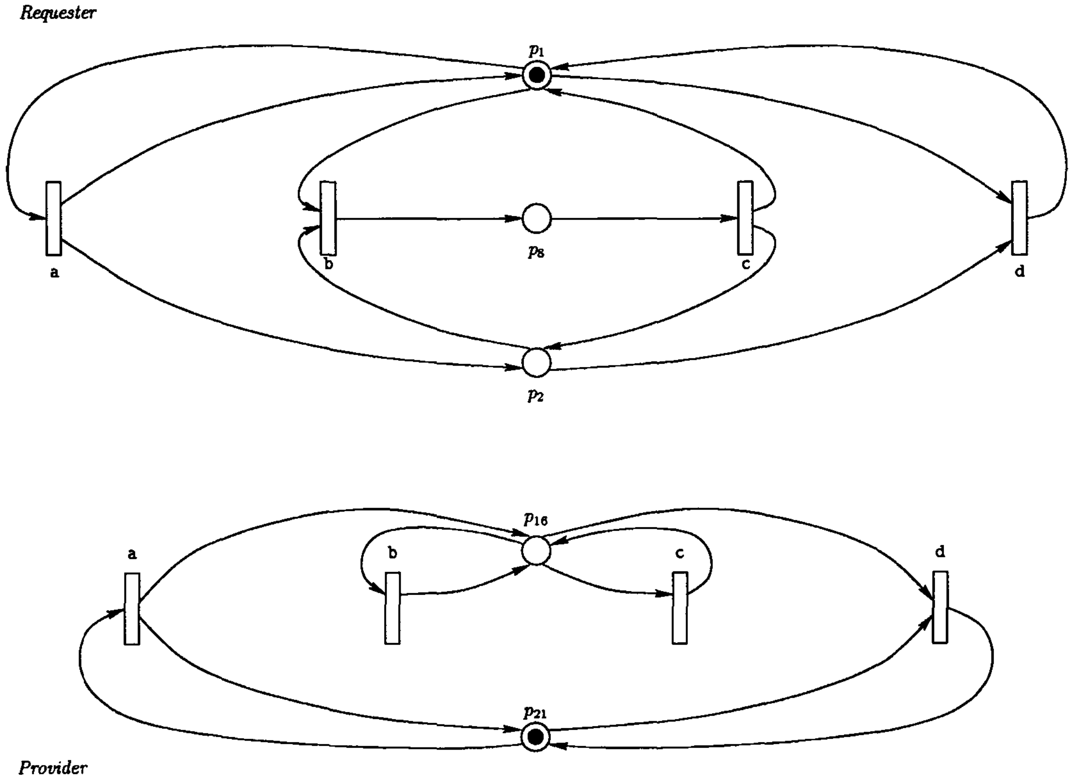

Figure 5.18: Database requester and provider interfaces using nested transactions

[2] 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 changed by introducing a new arc from the b transition to the top-most place in the requester.

of tokens in its top-most place (p_{16}) . Similarly, the number of tokens in the bottom-most place of the requester (p_2) indicates how many transactions have been opened.

The composed net, shown in Figure 5.19, does not exhibit any deadlock, implying

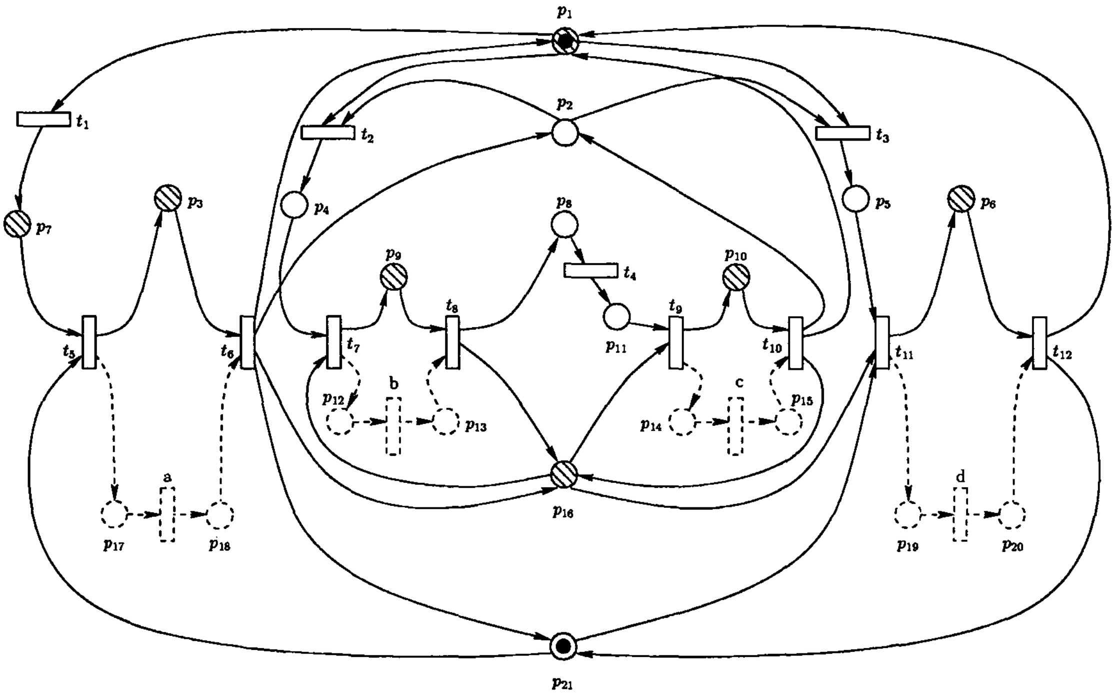

Figure 5.19: Composition of requester and provider interfaces using nested transactions

that the interfaces are indeed compatible. Because the net is unbounded, reachability analysis is not a viable method to show deadlock-freeness. Instead, deadlock-freeness can be verified by using structural analysis. After reducing the net, the only proper minimal siphon is $\{p_1, p_3, p_6, p_7, p_9, p_{10}, p_{16}\}$, which corresponds to the objective function $1-x_2-x_3+x_6+x_{10}$. The constraints, obtained from the connectivity matrix, are shown in Table 5.2 Linear programming shows that the minimum number of tokens

Table 5.2: Constraints for the Petri net of Figure 5.19

Place	Constraint
p_1	$-x_{t_1} - x_{t_2} - x_{t_3} + x_{t_6} + x_{t_{10}} + x_{t_{12}} + 1 \geq 0$
p_3	$x_{t_5} - x_{t_6} \geq 0$
p_2	$-x_{t_2} - x_{t_3} + x_{t_6} + x_{t_{10}} \geq 0$
p_4	$x_{t_2} - x_{t_7} \geq 0$
p_5	$x_{t_3} - x_{t_{11}} \geq 0$
p_6	$x_{t_{11}} - x_{t_{12}} \geq 0$
p_7	$x_{t_1} - x_{t_5} \geq 0$
p_8	$-x_{t_4} + x_{t_8} \geq 0$
p_9	$x_{t_7} - x_{t_8} \geq 0$
p_{10}	$x_{t_9} - x_{t_{10}} \geq 0$
p_{11}	$x_{t_4} - x_{t_9} \geq 0$
p_{16}	$x_{t_6} - x_{t_7} + x_{t_8} - x_{t_9} + x_{t_{10}} - x_{t_{11}} \geq 0$
p_{21}	$-x_{t_5} + x_{t_6} - x_{t_{11}} + x_{t_{12}} + 1 \geq 0$

in the minimal siphon cannot be reduced to zero thereby showing that the composed net is deadlock-free. The two nets are indeed compatible.

5.7 Summary

This chapter has presented a formal model for the composition of software components and the verification of their compatibility. Various strategies for the composition of multiple components in a software architecture have also been addressed. The examples in this chapter have demonstrated the viability of the presented compatibility verification technique. More advanced examples which demonstrate multicomponent composition, compatibility checking and multiple interfaces are presented in the next chapter.

Chapter 6

Example of Application

To demonstrate the practicality of the approach described earlier, an extended, non-trivial example involving an electronic prescribing system is presented in this chapter. The purpose of this example is to demonstrate the composition of more substantial components. To this effect, this chapter first provides the system model describing how the components interact at a high-level. Each of the nets representing the components' interfaces are then described in detail and finally, their composition is provided and analyzed.

It should be noted that many of the lower-level details regarding the interface behaviours are not fully specified. They include details related to aspects specific to a particular implementation of the application and are not pertinent to interface compatibility. For example, whether the notion of authentication requires a swipe card or a password (or both) is irrelevant to the actual interaction between the components. The same applies to the user interface issues and the location of the data repositories accessed by each component. Also, some of the interfaces are simplified for pedagogical purposes. It should be noted that the composition strategy proposed in the previous chapter can be used to integrate components of arbitrary complexity; however, the net resulting from complex compositions can become quite complicated.

6.1 System Model and Events

There are three primary components involved in the system: the physician, patient and prescription server. An ancillary authentication component may also be present to verify the identities of the system participants, but is not explicitly included in the example. As can be seen from Figure 6.1, all three components interact with one another at some point during their respective lifetimes.

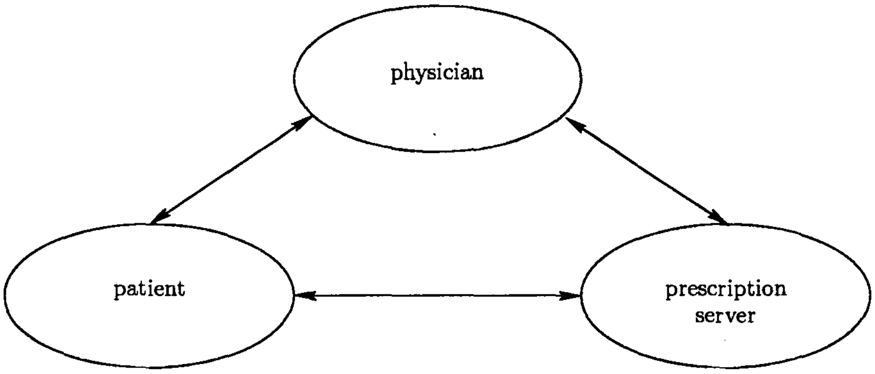

Figure 6.1: Component model of an e-prescription system

The basic sequence of events, or use-case, in which a physician creates a prescription and a patient subsequently fills the prescription can be outlined as follows:

1. A patient visits a physician; during this visit all relevant medical data is exchanged electronically. From an implementation perspective, the patient com-

ponent could take the form of a *smart card* with embedded software and hardware that allow access to all information relevant to the patient's medical record. Since the patient initiates the interaction, he/she is the requester and the physician is the provider.

2. The physician component interacts with the prescription server in order to obtain (or verify) the patient's drug history. During this phase, the role of the physician changes to that of a requester and the prescription server becomes the provider.

3. Once the physician has all the pertinent information, a diagnosis is made. In the context of this example, the diagnosis may be computer-assisted but the final diagnosis would have to be approved by the physician. If a prescription is necessary, then the physician relays details of the prescription to the patient and the prescription server (potential drug interaction difficulties could also be detected at this point). If no prescription is necessary, then the patient is so advised and no further interaction with the prescription server is necessary.

4. Finally, if a prescription was granted, the patient interacts (indirectly via the pharmacy) with the prescription server to fill and pay for the prescription.

The interfaces implemented by each of these components are described in the following subsections.

6.1.1 Patient Component Interfaces

Each patient component has two interfaces — one for interacting with a physician and another for communicating with a prescription server. A model of a patient interface for interaction with a physician is shown in Figure 6.2. In practice, this is the patient interface that would be used by a computer in the physician's office to access the relevant data on the card, and to relay information related to the final diagnosis. Naturally, the information obtained from the interaction between the two electronic components would be complemented by a more thorough examination of the patient by the physician. Such data could be provided to the system manually prior to the diagnosis.

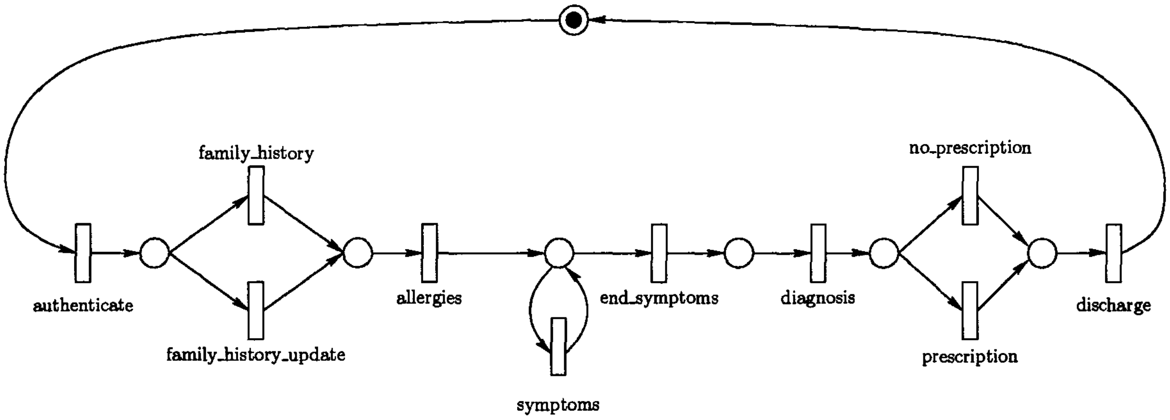

Figure 6.2: Patient component interface for physician components

The "standard" first step in the patient-physician interaction is authentication. As mentioned earlier, a separate authentication component can be used, but for the purposes of this example, it is assumed that the authentication mechanism is self-contained within the domain of the interacting components.

After authentication, the patient interface may transmit the patient's entire family history if this is the first time the physician is examining the patient. Alternatively, if the physician already has the patients family history on record, then only updates are sent, for efficiency. Services for the transmission of *allergies* and *symptoms* are then invoked. While both services can be implemented iteratively, the *allergies* service will transmit the allergies to the physician in bulk, whereas the *symptoms* service will deliver the symptoms incrementally, for demonstration purposes. Because symptoms are transmitted in an iterative manner, an *end_symptoms* service is used so as to ensure that both the provider and requester exit the iteration synchronously. (The *end_symptoms* service could also have be named *no_more_symptoms.*) Upon transmission of all the relevant medical data, the physician component has all the needed information, at which point the *diagnosis* service is invoked. The resulting diagnosis may or may not result in a prescription — the interface handles both cases. Finally, the communication between the patient and physician ends with the *discharge* operation which effectively terminates the authenticated session. A more complicated implementation could allow an arbitrary ordering of the *allergies* and *symptoms* services.

A patient component has a second interface to interact with the prescription server. This interface would have services for authenticating, filling a prescription and one or more payment methods. If the patient component supports only one payment option, for example, a debit account which would be maintained on the smart card, then the service invocation could be very linear in nature, as shown in Figure 6.3.

Alternatively, a patient component may support more than one payment option

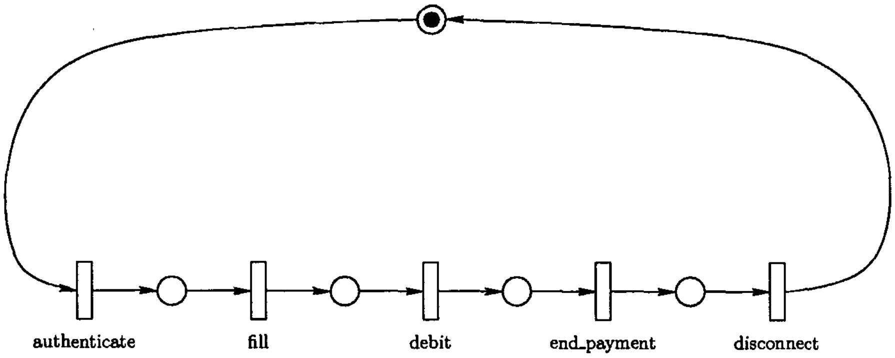

Figure 6.3: Patient component interface for prescription server (one payment option)

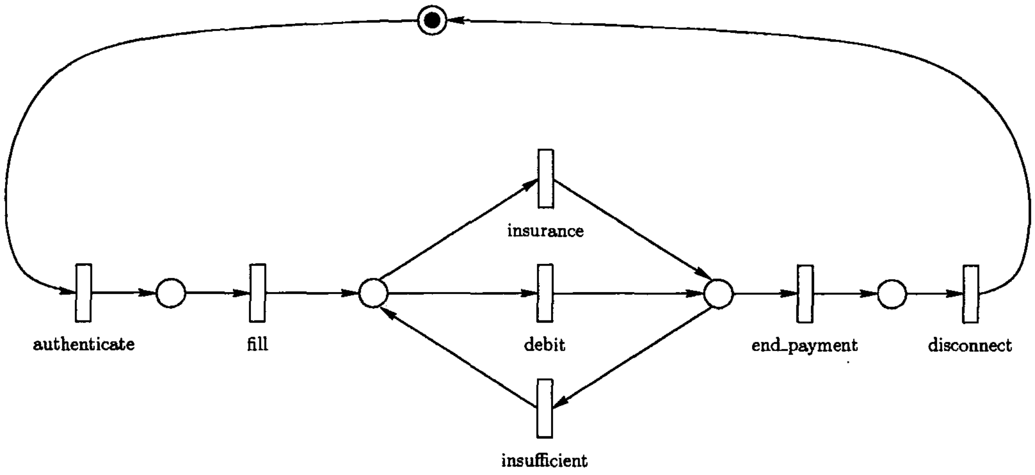

Figure 6.4: Patient component interface for prescription server (multiple payment methods)

to cover the cost of the prescription, as shown in Figure 6.4. For example, a patient could cover some or all of the cost with an insurance plan. Also, the payment portion of the interface allows for a loop to consider the possibility that the insurance may not cover the entire cost of the drug (the *insufficient* service would be used at this point), thereby requiring the rest of the cost to be made up for via (one or more) invocations of the *debit* service.

Additional payment methods, such as credit card, or borrowing, could also be implemented by the interface.

6.1.2 Physician Component Interface

Unlike the patient interface, the physician interface shown in Figure 6.5 does not impose any constraints on the order in which the allergies and symptoms are given by the patient component, but requires that all three steps, *symptoms*, *allergies* and *drug_history*, are performed in some order. As long as the *diagnosis* service is not activated prior to providing all relevant details, the component interfaces have the potential to be compatible. Another major difference between the patient and physician interfaces is that the physician interface requires an ability to obtain the drug history of the patient prior to the diagnosis and prescription services so as to prevent a patient from receiving two identical prescriptions from two different doctors. The drug history could conceivably be supplied by the patient component. However, to reduce the possibility of tampering, the physician interface should satisfy this service by consulting an external source, such as the prescription server itself.

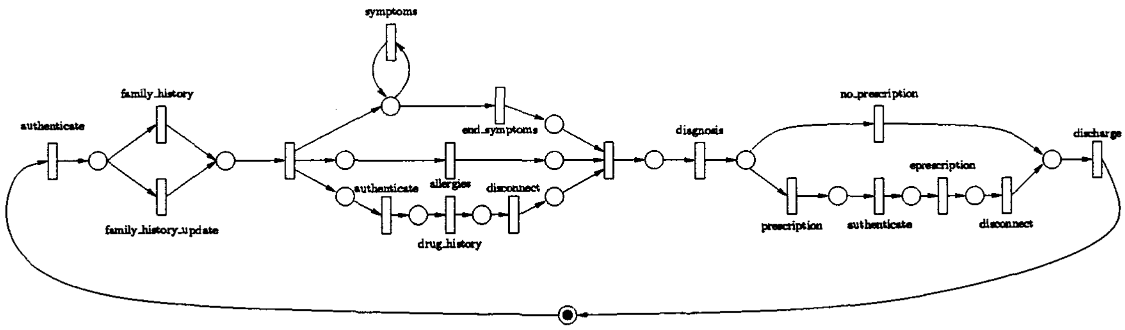

Figure 6.5: Physician component interface

Once all the relevant information is available, the diagnosis service can be activated and a prescription, if necessary, can be created by the physician interface reconnecting with the prescription server. If no prescription is necessary, then no further interaction with prescription server is required.

It should be observed that this particular physician interface may require two separate connections with the prescription server. This could be alleviated by having one authentication/disconnect pair instead of two, essentially leaving the connection to the prescription server interface "open" for the entire duration of the interaction between the physician and patient. The prescription server, described in the next section, does not allow for multiple operations to take place in a single session; however, this could be changed, if necessary.

6.1.3 Prescription Server Component Interface

The provider interface of the prescription server, presented in Figure 6.6, is relatively simple when compared with the patient and physician interfaces. After authentication, three primary services may be activated by the requester: with the appropriate credentials, a drug history can be requested, an e-prescription can be requested, or an e-prescription can be filled by paying via debit card and/or insurance. Only one of these three choices may be used during any single interaction.

With respect to authentication, it can be common to all subsequent services offered by the prescription server. The operations that a requester can perform on the prescription server depend upon the permissions level of the requester itself. For example, a component that is authenticated as a patient would have the ability to request that

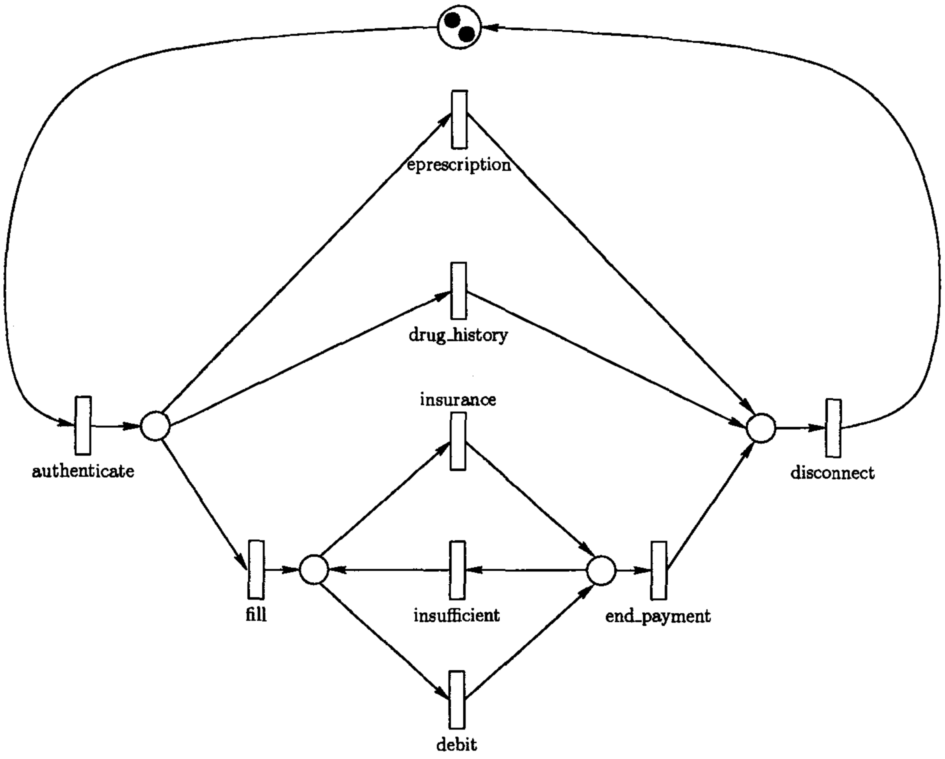

Figure 6.6: Prescription server interface

a prescription be filled. It would not have permission to make e-prescriptions, but it should have the ability to request its own drug history and not the history of others. The implementation would ultimately have to confirm that the requester has the appropriate credentials to carry out each service that it requests of the prescription server. However, if it is required that different types of authentication be used for each branch of services offered by the prescription server, then the server interface could be separated into three separate interfaces, if desired, each one with its unique authentication mechanism. The compositions described below will be the same. Also, the authentication required by a patient to debit his or her bank account would likely be different than the authentication required to access the prescription server to fill a prescription. The authentication to access the bank account would be encompassed by the *debit* transition itself.

In order to illustrate some elements of concurrency, the prescription server shown in Figure 6.6 can handle two requests simultaneously, which is represented by two tokens in the place of the prescription server that leads to the *authenticate* service transition.

Compatibility testing between a patient and prescription server is possible only when a patient has acquired a prescription from a physician. The validation of the prescription could be done as part of the authentication service. Upon completion of the interaction, the prescription server can then disconnect from the requester, allowing for any resources employed during the interaction to be appropriately deallocated, and used for serving subsequent requests.

6.2 Composition of Interfaces

This section describes the composition of the various interfaces. The resulting composed models are analyzed for compatibility using reachability analysis and linear programming.

6.2.1 Patient-Physician-Prescription Server Composition

An overview of the composition of one patient requester, one physician and two prescription server interfaces is outlined in Figure 6.7. The left and right portions of the composed net, as delimited by the dashed boxes, are shown magnified, with corresponding transition labels, in Figures 6.8 and 6.9, respectively.

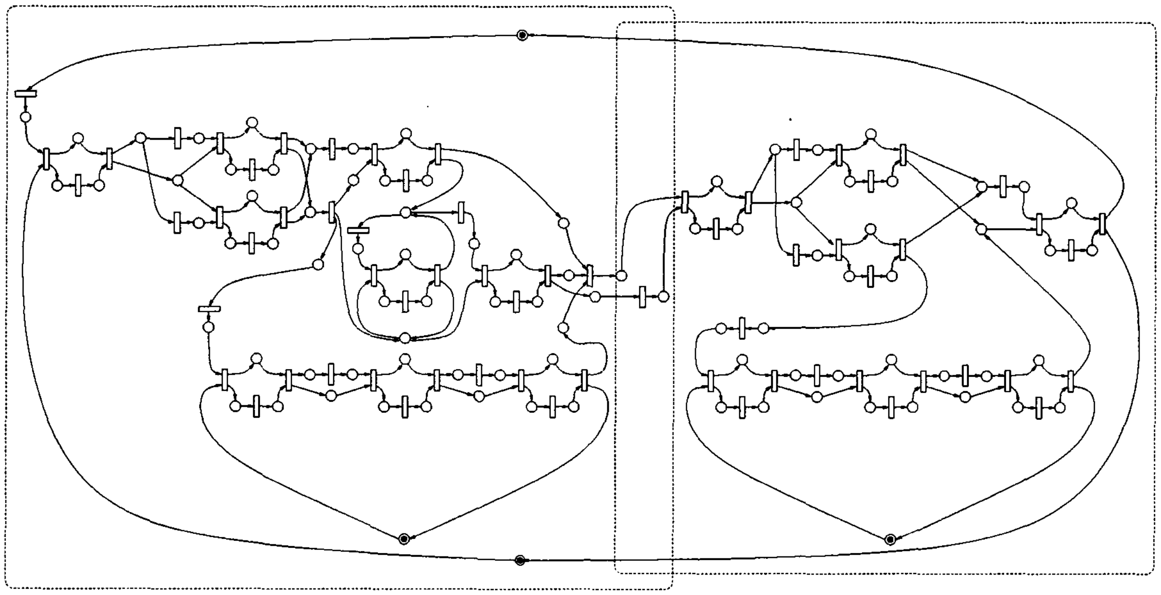

Figure 6.7: Overview of patient-physician-prescription server composition

In the context of the patient-physician composition, the patient (Figure 6.2) is the requester and the physician (Figure 6.5) is the provider since it is the patient that initiates the interaction. The composition of the *authentication* and the *family_history/family_history_update* free choice structure is a direct application of the rules presented earlier. After this point, the patient requester interface imposes an order on its two subsequent services (*symptoms* and *allergies*), which the physician provider interface is able to accommodate since it places no restriction on the relative order of these two service invocations. The *drug_history* service is not available in the patient interface but does exist in the prescription server interface. In order to satisfy this service, the physician provider interface becomes a requester interface when demanding the *drug_history* service of the prescription server provider interface.

Even though there are two prescription server interfaces in the composition, they would likely represent the same prescription server with the same data repository. As

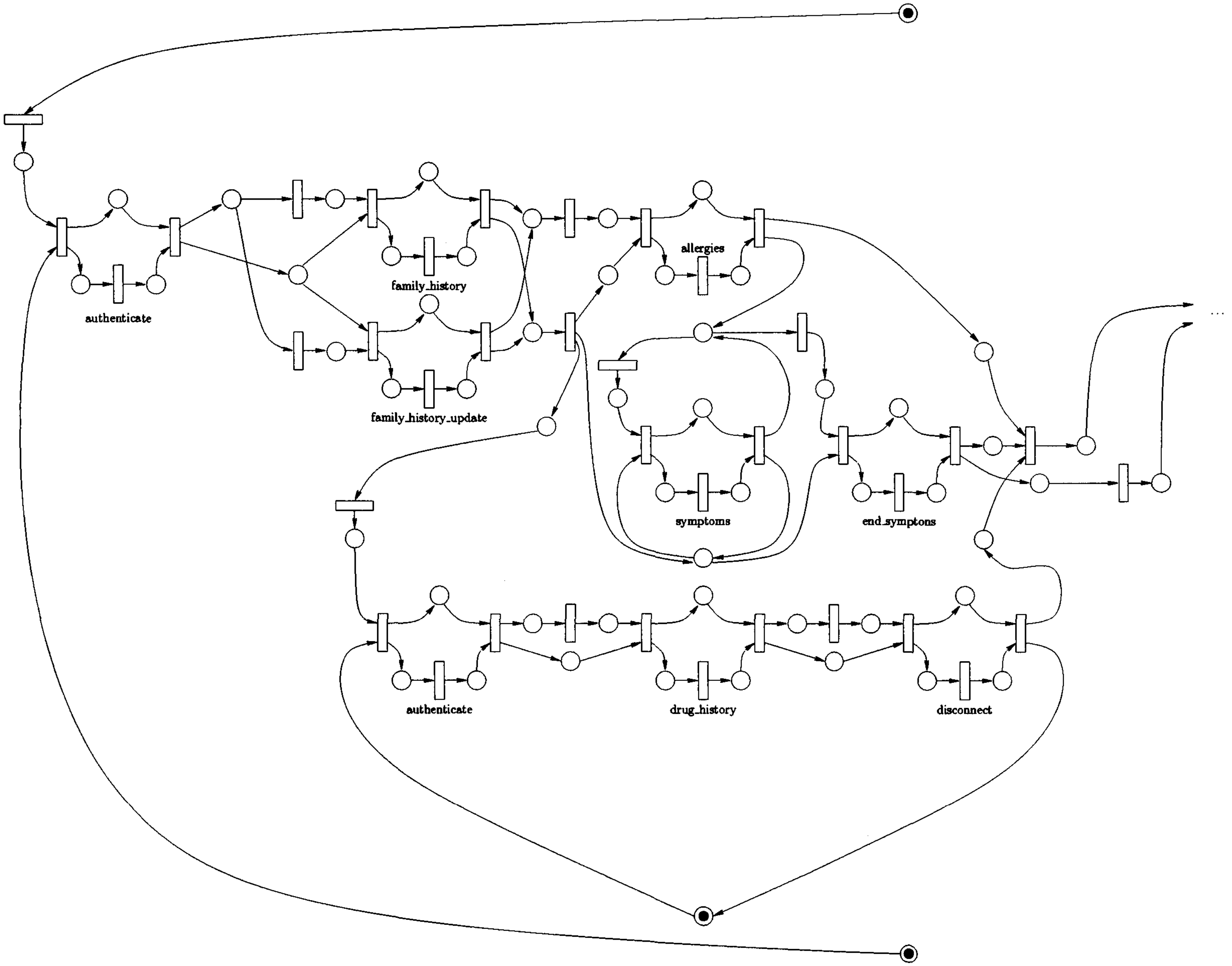

Figure 6.8: Left part of Figure 6.7

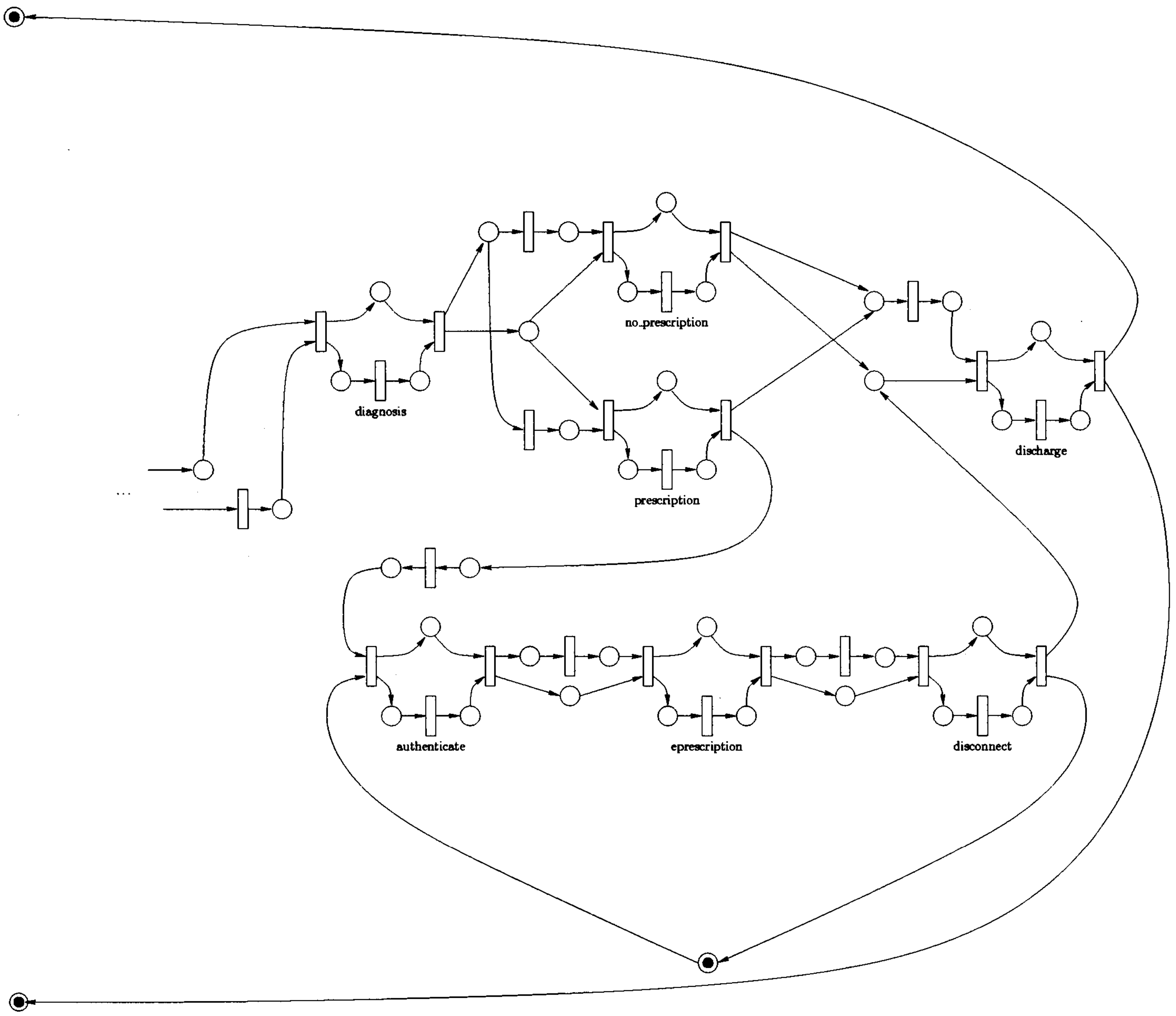

Figure 6.9: Right part of Figure 6.7

mentioned above, the composition could have been accomplished using a single prescription server interface if the server permitted more than one prescription-related service to be performed during a single session. Also note that there is no direct interaction between the prescription server interface and the patient interface — the patient interface used in this composition is intended to interact only with the physician interface. As a result, the *fill* and *debit* service transitions of the prescription server are unused in the composition. Interaction between patient interfaces and the prescription server interface is described in the next subsection.

Once all the components have their services satisfied by the composition, the resulting net can be analyzed. Although the model is quite complex, it is sequential, so a small number of reachable markings is expected. Indeed, reachability analysis reveals that there are 221 reachable markings of the net, none of which is dead. The absence of deadlocks demonstrates the compatibility of the four interfaces.

The model shown in Figure 6.7 has several parallel and alternate paths as described in Section 4.2.4; the redundant paths can be removed without adversely affecting subsequent deadlock analysis of the model. The simplified net, after removing all appropriate elements associated with the parallel and alternate paths, is shown in Figure 6.10; it has 74 minimal siphons, all of which are also marked traps. Consequently, the net is deadlock-free since none of the minimal siphons can ever become empty. It can be shown that the number of minimal siphons appears to roughly double each time an alternate or parallel path is re-introduced, so elimination of parallel and alternate paths can substantially reduce the number of minimal siphons.

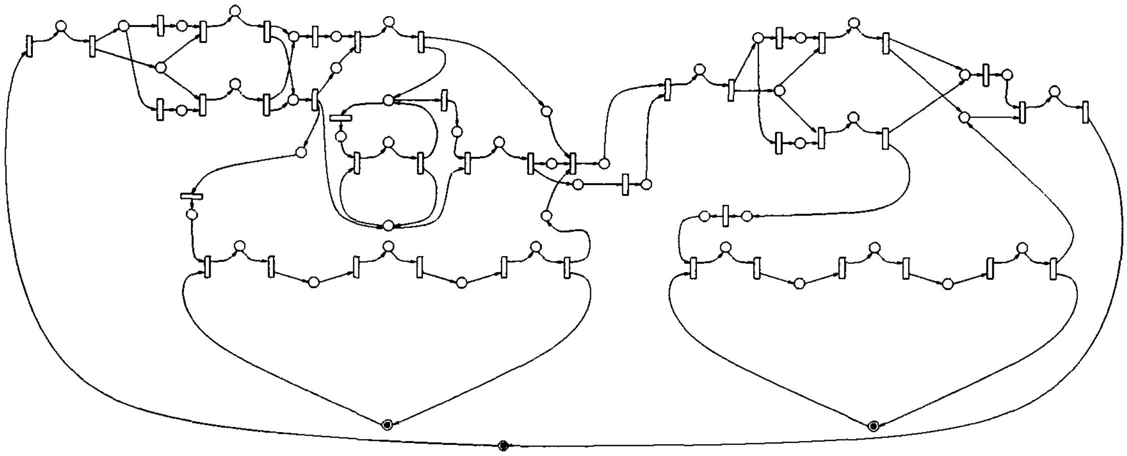

Figure 6.10: Simplified patient-physician-prescription server composition

6.2.2 Patient-Prescription Server Composition

To demonstrate multi-requester composition, the two patient interfaces (from Figures 6.3 and 6.4) and prescription server interface (Figure 6.6) are composed. One of the patient interfaces provides for only one payment option, while the other allows for two such options (*debit* and *insurance*). Combining these nets results in the net shown in Figure 6.11.

Reachability analysis of this composed net reveals that there are 515 distinct markings. Although the number of markings is higher than for the previous composition, it is still relatively small considering the complexity of the composed net. Again, the resulting net is bounded (the bound is equal to two) thereby making reachability analysis straightforward in this case.

As with the previous example, siphon extraction can be simplified by removing the elements associated with numerous parallel and alternate paths, creating the net shown in Figure 6.12. Analysis of this simplified net reveals five minimal siphons, all

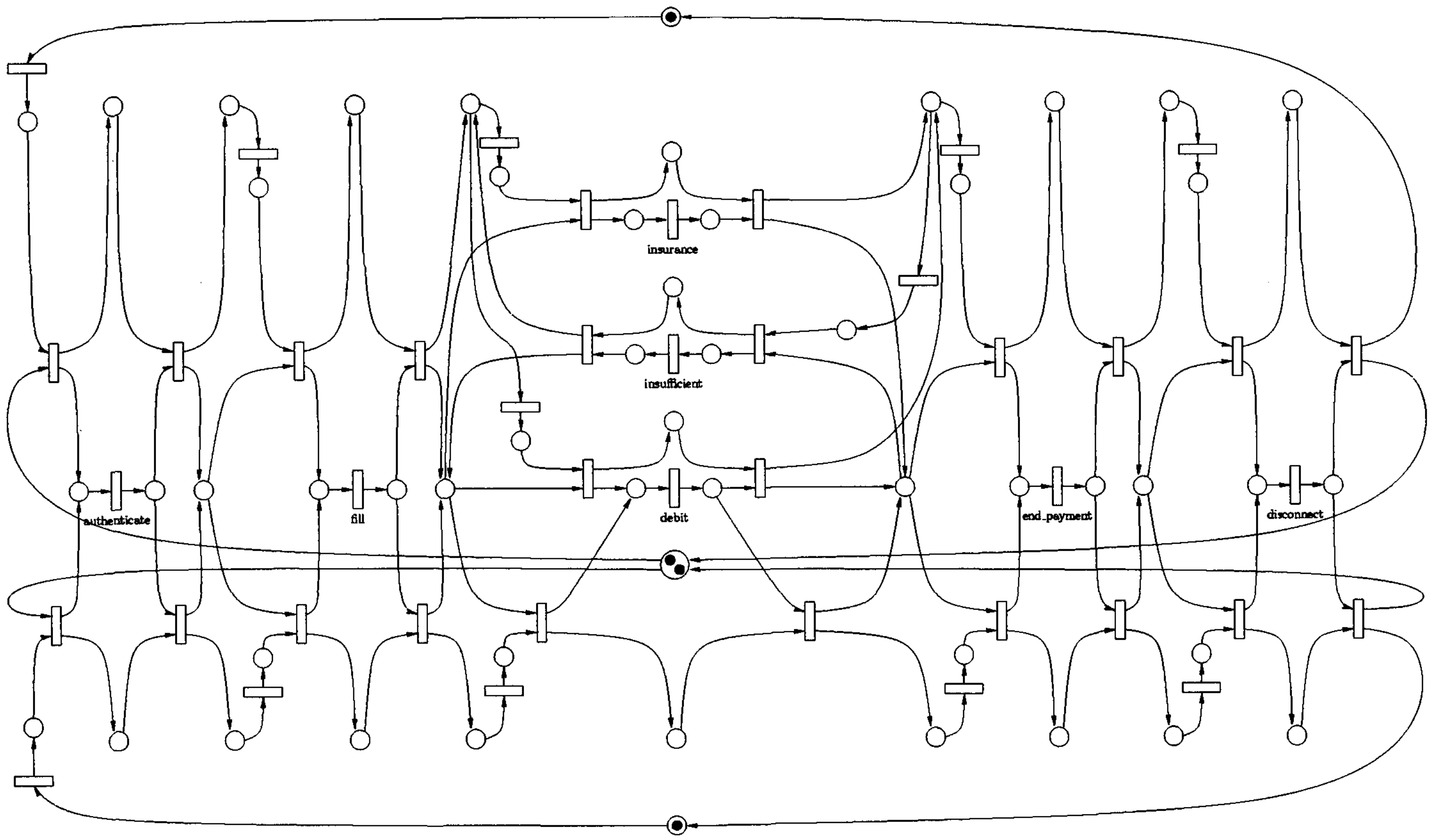

Figure 6.11: Patient-prescription server composition

of which are also marked traps. The resulting composed net is thus deadlock-free and the composition of the prescription server with the two different patient requesters is indeed compatible.

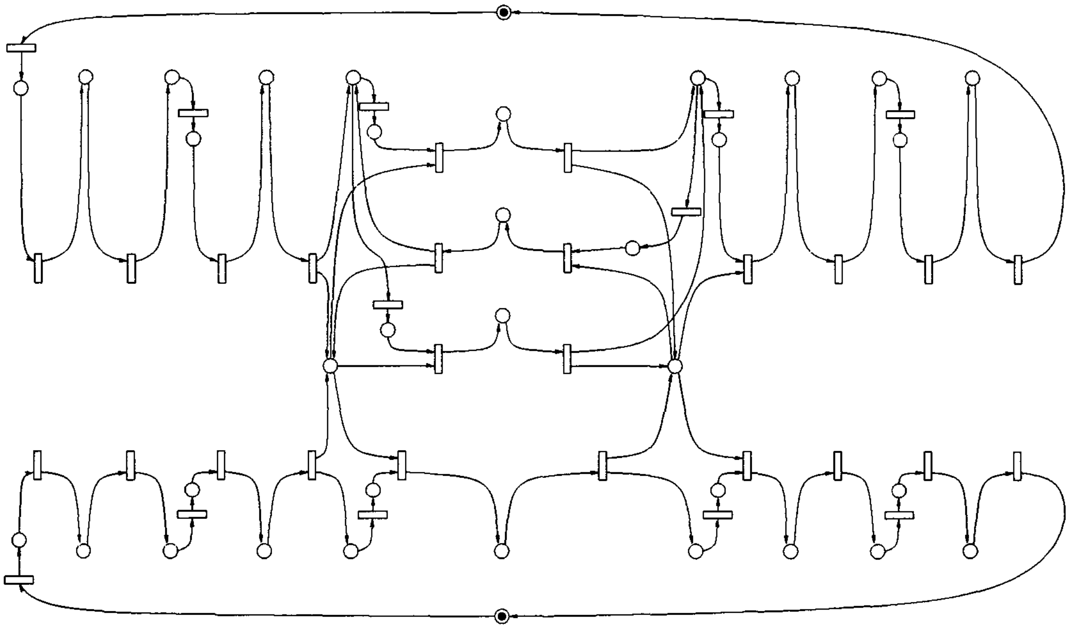

Figure 6.12: Simplified patient-prescription server composition

6.2.3 Incompatible Composition

To demonstrate a case where deadlock arises, a patient requester component interface similar to Figure 6.4 is modified by reversing the arcs connected to the *insufficient* transition. Intuitively, this new patient requester interface cannot be compatible with the prescription server provider since the latter interface requires that either an *insurance* or *debit* service be invoked prior to using the *insufficient* operation. This restriction is not obeyed by the modified patient interface. Figure 6.13 shows the composition of this modified interface with the other patient and prescription server interfaces.

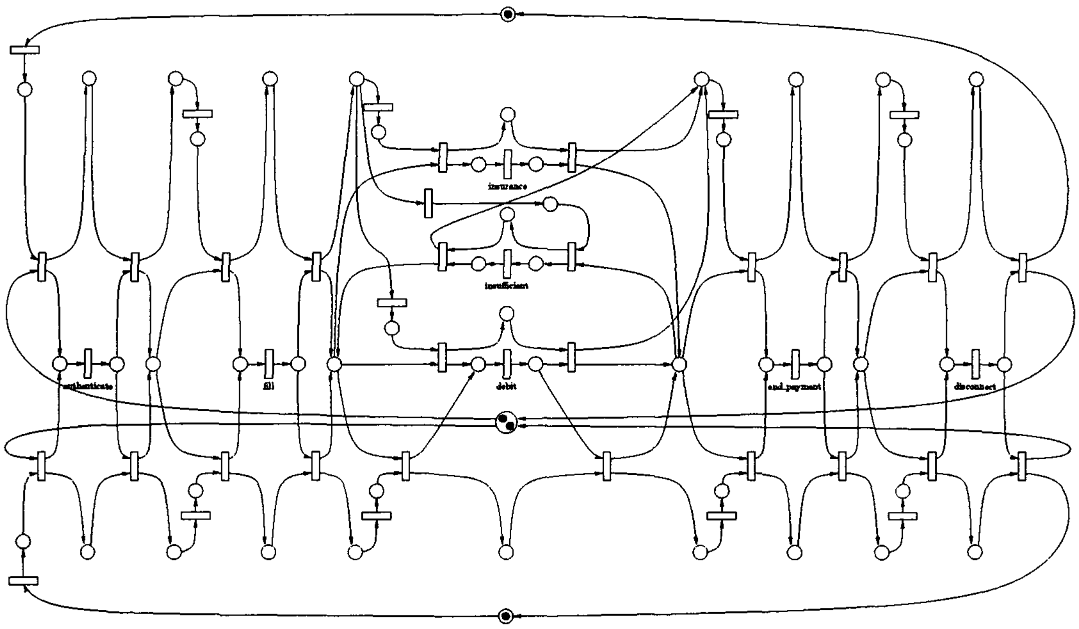

Figure 6.13: Incompatible composition between patient-prescription server interfaces

The incompatibility that results from the new component can be demonstrated using reachability analysis. There are 483 unique markings, one of which is dead, as shown in Figure 6.14. Hence the resulting composition is incompatible.

Using siphons extraction and linear programming to isolate the deadlock is also possible. Deadlock does not result by emptying either of the two minimal siphons present in the simplified net. However, when the algorithm of Figure 4.6 analyzes the 17 basis siphons, a siphon sequence can be identified which, when followed in the minimization process, does result in a deadlock situation. A detailed analysis and some further discussion of deadlock determination in this net is presented in

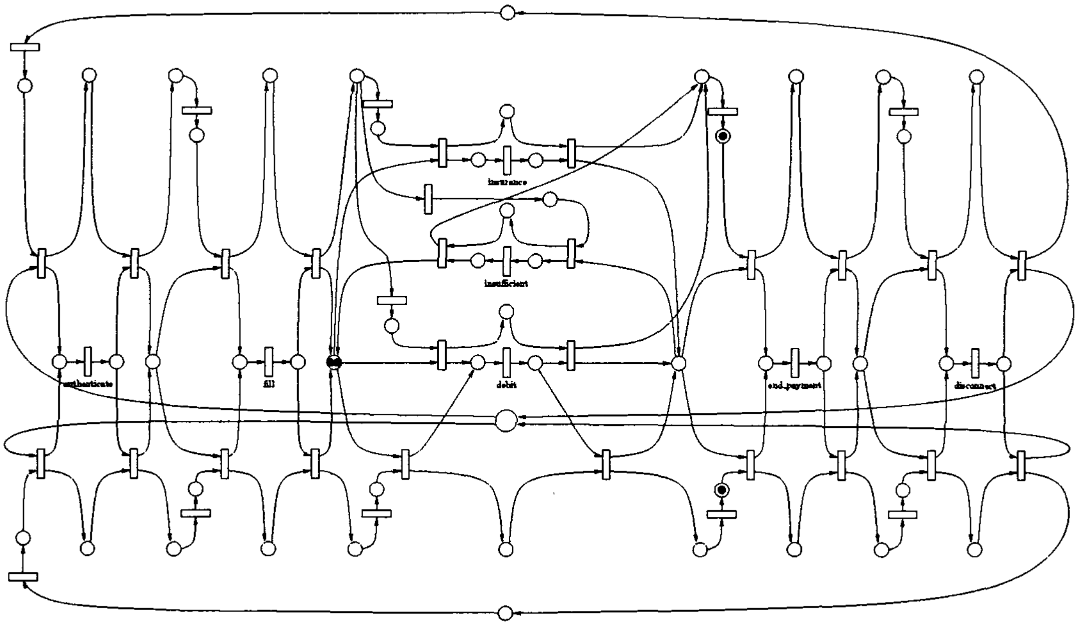

Figure 6.14: Dead marking of patient-prescription server composition

Appendix A.

6.2.4 Analysis of Unbounded Component Compositions

As a final example of compatibility assessment, consider the case in which a physician may issue a patient more than one prescription. The physician and patient interfaces can be modified as shown in Figures 6.15 and 6.16, respectively. (These figures show only the changes necessary to the interfaces in Figures 6.2 and 6.5 — the remainder of the interfaces are unchanged.)

The patient interface keeps track of the number of prescriptions assigned to it by the shaded place in Figure 6.15. This place is shared between the patient interface that communicates with the physician and the patient interface that communicates with the prescription server. The modifications required in the latter patient interface to

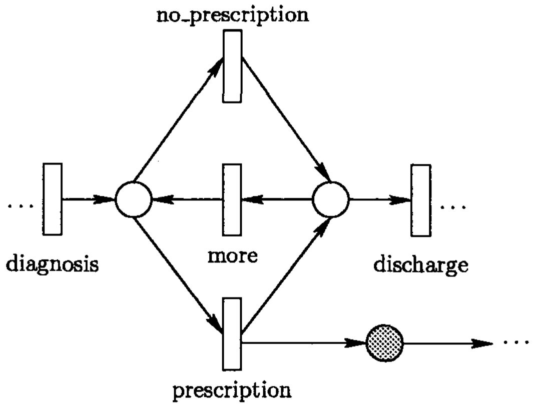

Figure 6.15: Modifications to the patient interface of Figure 6.2

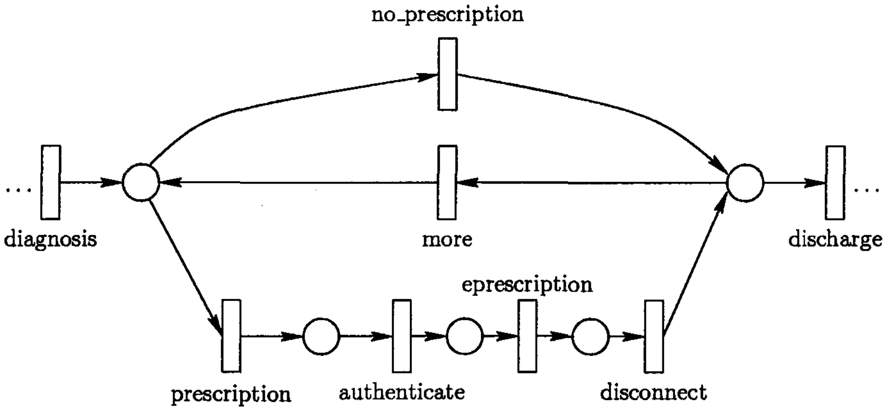

Figure 6.16: Modifications to the physician interface of Figure 6.5

handle multiple prescriptions is presented in Figure 6.17. To keep the example simple, only the patient interface of Figure 6.3 will be used in the composition. The addition of a *more* service can also be made to the prescription server interface of Figure 6.6 in a similar manner.

To test for compatibility, all interfaces are composed into one net using the composition strategy presented in the previous chapter. For the purposes of this example, the prescription server will be limited to servicing the requests of only one patient.

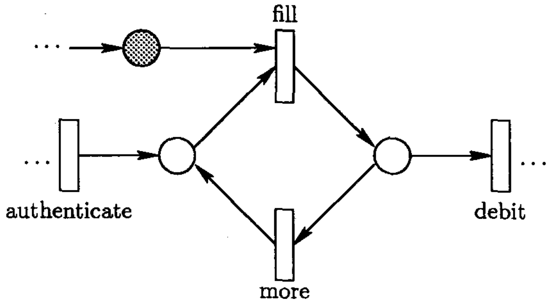

Figure 6.17: Modifications to the patient interface of Figure 6.3

The entire composition is shown in Figure 6.18. The place that is shared between the two patient interfaces is shaded in the figure. The fact that this place can accumulate an arbitrary number of tokens implies that the markings for this net are unlimited, thereby making the net unbounded — an arbitrary number of prescriptions may be assigned to a patient. Consequently, reachability analysis cannot be used to test deadlock-freeness; structural analysis can be performed instead. After removing the parallel and alternate paths the reduced net has 78 minimal siphons. All of these siphons are also marked traps so the composition of the modified interfaces is indeed deadlock-free.

In addition to demonstrating how compositions resulting in unbounded nets can be analyzed, this example also demonstrates how one interface can influence the interactions of another via a shared place. Other extensions to the model involve allowing multiple patients to visit a physician. Because a physician can service only one patient at a time, extra transitions and places are necessary in the respective interfaces to simulate the notion of a "waiting room," and to coordinate the appropriate "prescription counter" places. While these extra net elements would obviously make the composed net more complicated, the same analysis strategy can be employed as

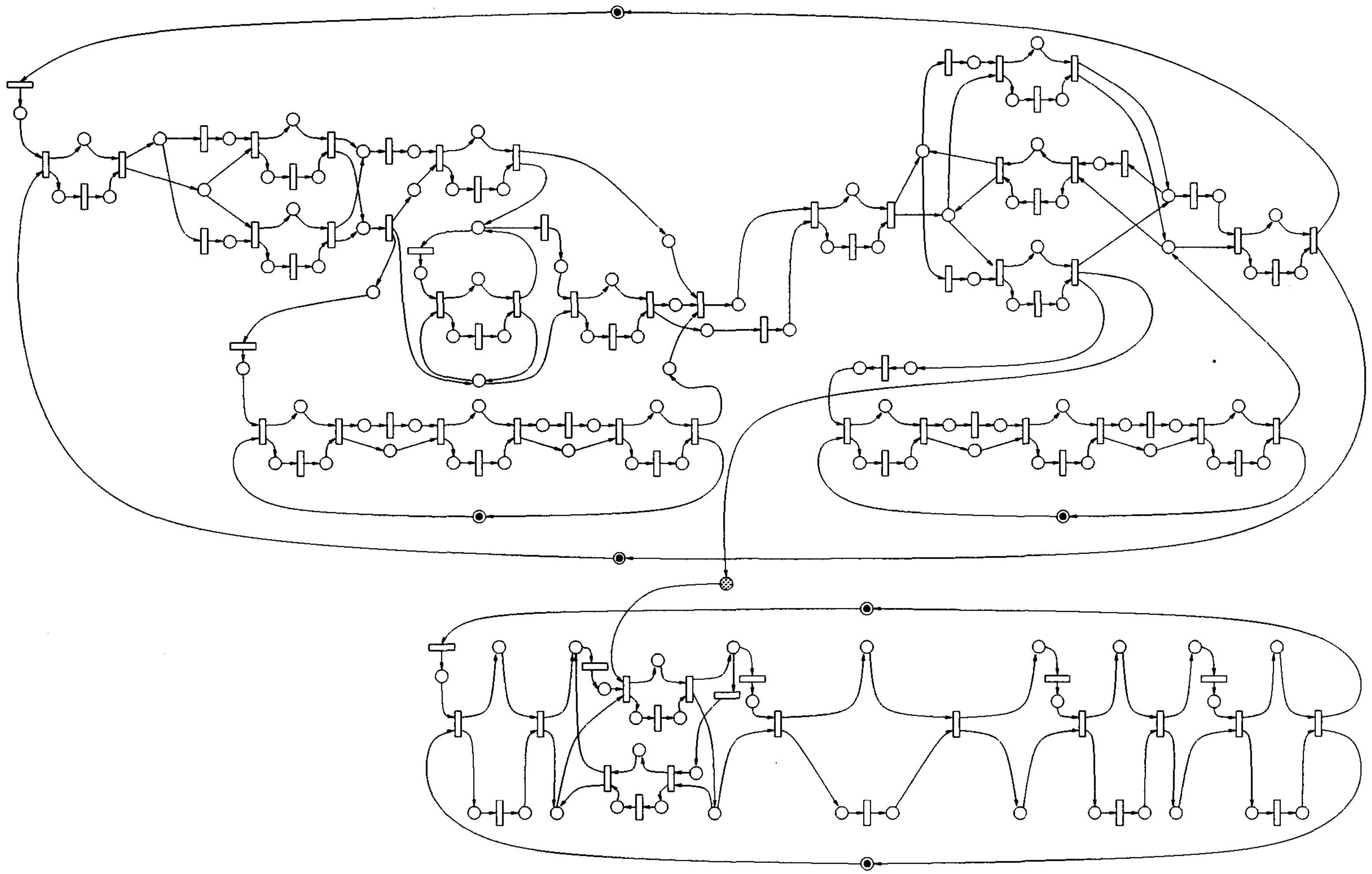

Figure 6.18: Composition of physician, patient and prescription server interfaces

demonstrated by this chapter.

6.3 Summary

This chapter provided a non-trivial example to demonstrate how compatibility between two or more interfaces can be determined formally by representing the dynamic behaviour of the interfaces as Petri nets and studying the resulting composition for deadlock. While reachability analysis may appear to be the most straightforward option for the analysis, siphon extraction and linear programming can be used when the model is unbounded or the space of reachable markings becomes excessively large. In order to simplify siphon extraction, the composed net can be reduced by eliminating inessential siphons introduced by parallel and alternate paths.

Chapter 7

Concluding Remarks

Determining the degree to which components are compatible with one another in a software architecture is a multi-faceted issue that, in the general case, requires a comprehensive understanding of both the static and dynamic nature of the components involved. By abstracting away the internal, low-level behaviour of components and concentrating upon the dynamic nature exhibited at the respective components' interfaces, one can establish whether or not the components are compatible, *i.e.*, whether or not they can communicate effectively and reliably.

This work presented a formal approach for composing two or more components by integrating the Petri nets that represent their interfaces into a single net. The dynamic compatibility is assessed by determining if the resulting net exhibits deadlock. In order to identify deadlocks, two methods were discussed, one based on analysis of reachable states and the second based upon analysis of net structural properties (combined with linear programming). The relative strengths and weaknesses of these methods were also discussed.

7.1 Potential Applications

This work has several possible applications in the areas of construction and deployment of software systems. It is also believed that the compatibility assessment strategy may help promote the reuse of existing software components and may provide a metric to measure whether one component can be substituted for another in a deployed system. The composition and compatibility techniques may also have applications in the development of large-scale, distributed software systems.

7.1.1 Software Development and Deployment

The composition strategy described in this work has potential applications in the software development life-cycle. In particular, during the design and implementation phase, this composition strategy may serve to encourage component reuse by providing designers and developers a formal means of identifying external components which can successfully satisfy the requirements of the components that they build. Also, the compatibility of components imply the (minimal) requirements on those new components which are supposed to interact with the existing components.

In addition to the design and implementation phase, the composition technique may also be used during the deployment phase to allow a component to dynamically discover compatible components. If a component determines that it is compatible with more than one component, other discriminating factors, such as latency or memory requirements, can be used to choose the preferred component for interaction.

7.1.2 Substitutability and Reuse

All practical software systems are in a constant state of change as functionality is enhanced, errors are corrected and efficiency is improved. As part of the evolution of a software system, it is sometimes necessary to replace existing subsystems with newer subsystems; usually these replacements are intended to improve the behaviour of the entire system in some way. Two criteria have been identified which define the problem of substitutability [23]: (1) the component that is being used for replacement must provide all the services that its previous counterpart provided (it may also provide more services, if desired) and (2) any correctness properties that existed in the previous system must still be preserved after the substitution has taken place.

In this context, the model presented in this work can assist in determining the feasibility of replacing an existing component with a new component in a software architecture during maintenance cycles. In particular, the notion of provider services are central to the interface model described by this work and any additions or deletions to these services are easily identified. While this model does not ensure that all low-level semantic behaviours related to a component's state are retained, correctness criteria with respect to the preservation of service invocations can be evaluated by the compatibility checking procedure prior to a component substitution actually taking place. A very simple substitutability criterion can be formulated on the basis of interface languages (introduced in Section 4.4): Component A can be substituted by component B, if for all corresponding interfaces $I_j^{(A)}$ and $I_j^{(B)}, j = 1, \ldots, k$: $\mathcal{L}(I_j^{(A)}) \subseteq \mathcal{L}(I_j^{(B)})$.

This substitutability aspect can serve to make the upgrading and subsequent maintenance of existing systems easier and may also help promote reuse in a software architecture [41, 42, 81]. When new provider components are deployed in an environment, requester components can evaluate them for compatibility and abandon their previous providers if the new providers offer better performance. This would naturally require a means for components to evaluate the performance of others.

7.1.3 Web Services

Related to this work is the increasingly important areas of web-services [9, 19, 50] and the more general areas of Service-Oriented Architectures (SOA) [49] and agent-oriented methodologies [32]. Much research is currently being done in these two fields, both of which involve the exchange of messages between machines in large-scale, distributed software architectures. The need to assess compatibility between client and provider services is important to the success of these two development models and it is believed that the model presented in this dissertation can be adapted for these methodologies.

7.2 Future Work

While the applications of the proposed technique for compatibility verification are apparent in the software development process, further work and study are necessary to enhance the model so as to make it more robust in the context of challenges that can be present in the construction of "real-world" software architectures.

7.2.1 Deadlock Detection

The primary challenges facing this model are the aspects related to the deadlock detection of the composed net. Reachability analysis, has, surprisingly, been very effective at identifying the presence of deadlocks in the composed (bounded) nets of the previous chapter. Unfortunately, assessing the reachability of unbounded nets is more challenging, but there may be ways to address this problem. In particular, the concept of *modified reachability trees* (MRT) [109] may be helpful in dealing with this issue. Alternatively, unbounded nets exhibit a "pumping" effect which can be represented by vector addition systems [73].

7.2.2 Siphon Extraction

Structural analysis and linear programming are ideally suited for unbounded nets in which the number of places in the composed net is not too large. As the number of places in a composed net increases, the extraction of basis and minimal siphons becomes much more challenging using currently known algorithms. Additionally, the deadlock algorithm and the feasibility check for firing vectors both employ backtracking strategies which increase the complexity of the proposed compatibility verification technique. Further empirical evidence is required to fully understand the practical complexity of the algorithms proposed in this work. It has been demonstrated that the aforementioned complexity concerns can be mitigated by removing parallel and alternate paths from the composed net; most likely, the number of siphons in a net can be reduced by identifying additional redundant structures in the net and removing the corresponding net elements. Alternative algorithms to finding basis and minimal siphons and/or to identify deadlock should also be investigated [28, 75, 103]. Some research has recently been published that describes a strategy to assesses component behavioural compatibility without encountering the state explosion problem [3]. Determining if this technique can be adapted to the composition model presented in this work may be beneficial.

Ideally, siphon extraction and linear programming should only be attempted on composed nets when reachability analysis is unreasonable, for example, in the context of an unbounded net.

7.2.3 Semantic Compatibility

The presented research addresses only one aspect in the broad area of component compatibility. There are many other issues related to compatibility which are not fully addressed. The notion of compatibility proposed by the model in this work provides a means to determine the *potential* for two or more components to interact successfully. In particular, this model provides a viable method to assess the *structural* compatibility of two different interfaces but does not provide a means to determine the *semantic* compatibility. Returning to the example given in Section 6.2.2, if the actual amount of money from the both debit card and insurance combined were insufficient to pay for the prescription, then this could be regarded as a form of semantic incompatibility which cannot be detected by this model. Another example would be a component's failure to authenticate because of an invalid password. If authentication fails during runtime for example, this does not mean that the two components were incompatible. These failures relate to the actual values interchanged between components and are therefore below the level of abstraction of the compatibility model proposed in this work. Their effect, however, could be included in the models by introducing free-choice structures for more than one outcome of an operation.

7.2.4 Asynchronous Interaction

While the composition model does allow for concurrency when multiple requesters are simultaneously communicating with a single provider, there is no support for spontaneous events or asynchronous behaviours in the composition. Therefore, this model may not be particularly amenable to implementing events such as interrupts and exceptions arising from serious software faults, such as overflows of arithmetic operations. Some representation of such effects could be introduced by additional free-choice structures, as indicated above, but a more systematic approach is needed in the general case.

Interfaces whose behaviours are much more complex than that presented in the previous chapter can also be represented by the model. Indeed, with the introduction of inhibitor arcs in the Petri net model of an interface, an arbitrary protocol can be modelled (the modelling power of Petri nets with inhibitor arcs is equivalent to Turing machines). Unfortunately, as the number of elements in the composed net increases, detecting deadlock becomes more computationally intensive. Moreover, the introduction of inhibitor arcs renders the compatibility assessment based on structural properties inapplicable (except for some special cases).

7.2.5 Temporal Aspects

Many conventional applications involving interactions amongst components are not time dependent 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 proposed model does not address temporal compatibility of two components. Fortunately, through the use of extensions such as timed Petri nets [114, 115, 116], such timing aspects may be added to the model and may serve as a means to assess compatibility based upon various performance evaluation metrics.

7.2.6 Model Building

Another issue not fully addressed by this work is how one can construct the Petri net for an interface when given the corresponding requester client code and library that implements the interface in the provider. 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 can be taken during which all branches of execution are exercised during a run-time simulation so as to deduce the complete Petri net structure for an interface. An alternative method for determining the nets is through the use of formal specification languages such as JML [20] which can be used to specify various pre- and post-conditions in the code via well-defined, structured comments prefacing each service definition. By chaining services with corresponding pre- and post-conditions, it may be possible to deduce the net using a code base whose services have been appropriately adorned with a complete set of specifications. For example, in the extended example of the previous chapter, all services after the authentication would require that an *authenticated* attribute be set to true. A post-condition of the *authenticate* service would be to set this attribute to true in the specification, therefore dictating that the authentication service must be performed before any other services are invoked.

7.2.7 Component Discovery

This model does not provide a formal mechanism for components to discover one another in a pragmatic context. Other technologies exist to accomplish this such as UDDI, CORBA Naming services and JNDI. The question remains, however, as to how an initially unconnected component can attempt to intelligently query the interfaces of thousands of other interfaces to determine compatibility and to interact successfully with other components in a deployment environment. The labels specifying the services could be used for preliminary discovery of potentially compatible components. Additionally, a hierarchical decomposition of the deployed component space on the basis of the linguistic properties of their interfaces may help to alleviate some of the complexity of discovery.

While verifying compatibility during the deployment phase poses some challenges, it may also lead to some benefits. For example, it may allow for the possibility of a software architecture that can dynamically reconfigure itself [6], potentially giving rise to autonomous, self-assembling software systems. It may also lead to the possibility of removing a component from a running system and plugging in a compatible component without adversely affecting the operation of the system. 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.

7.2.8 Measures of Compatibility

Currently, the existing model of compatibility amounts to a decision problem — either two or more components are compatible, or they are not. The model does not allow for a continuum of compatibility measures [112]. Such a measure may be helpful especially during maintenance of software systems, when components and their interfaces are subject to modifications to improve performance or to provide additional functionality. Changes to a component may result in an interface which is, for example, 99% compatible with its peer components instead of 100% (which may mean that some infrequently used service is not being used in a compatible context). If the compatibility measure is relatively high, then it may be possible to introduce a *facade* interface that maintains backwards compatibility with the old interface. This would allow for interactions between components that "almost" successfully interact with one another. The facade could later be dropped as other components are updated. Some work in this general area, using interface automata, has already been performed [59].

7.2.9 Design Quality

The proposed model does not describe the attributes of a well-designed interface or, indeed, how to build such an interface. However, there are some qualitative guidelines which can be followed. It is known that components with complex interactions are difficult to design and implement, especially when not all requirements are known. Therefore, interface development may be performed in an incremental or step-wise fashion for subsets of related operations. These subsets may then be clustered into a single "super-service," thereby facilitating the reachability or structural analysis. It is also the duty of the interface designer to ensure that the interface is "well-behaved," for example, it must not contain deadlocks (non-liveness, however, is acceptable). If two poorly designed interfaces are created and successfully composed in accordance with the model proposed in this work, this does not necessarily mean that the composition will result in the composed net exhibiting desirable or correct behaviour.

7.3 Epilogue

The research described in this thesis represents an important step in the continuing evolution of the design and construction of 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 the integration of software components in a given software architecture. Ultimately, this may contribute to the reliable evolution of a deployed component-based software system.

In addition, the formal models proposed in this work constitute a foundation which may enable the automation of some of the tedious aspects of component integration and deployment. This work could lead to the formation of autonomous software systems in which each component possesses the knowledge of the services it requires and provides via each of its interfaces. By giving components the ability to independently determine other compatible components in the context of its deployment, we allow for the potential of self-assembling software systems, thereby allowing for an increasing degree of automation in the field of software development. Although many practical aspects of such systems must be studied extensively, the work presented in this thesis provides a possible foundation for such research.

Bibliography

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

[2] R. Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon University, May 1997.

[3] P. Attie, D. Lorenz, A. Portnova, and H. Chockler. Behavioral compatibility without state explosion: Design and verification of a component-based elevator control system. In *CBSE 2006*, volume 4063 of *Lecture Notes in Computer Science*, pages 33–49. Springer-Verlag, 2006.

[4] J. Baragry and K. Reed. Why we need a different view of software architecture. In R. Kazman, P. Kruchten, C. Verhoef, and H. van Vliet, editors, *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, pages 125–134. IEEE Computer Society, Aug. 2001.

[5] L. Barroca, J. Hall, and P. Hall, editors. *Software Architectures.* Springer, 2000.

[6] T. Batista, A. Joolia, and G. Coulson. Managing dynamic reconfiguration in component-based systems. In R. Morrison and F. Oquendo, editors, *Software*

Architecture 2 European Workshop, EWSA 2005, volume 3527 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 2005.

[7] K. Beck. *eXtreme Programming eXplained : Embrace Change.* Addison-Wesley, 2000.

[8] K. Bergner, A. Rausch, M. Sihling, A. Vilbig, and M. Broy. A formal model for componentware. In Leavens and Sitaraman [61], chapter 9, pages 189–210.

[9] D. Beyer, A. Chakrabarti, and T. Henzinger. Web service interfaces. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 148–159, New York, NY, USA, 2005. ACM Press.

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

[11] 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.

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

[13] I. Bratko. *Prolog Programming for Artificial Intelligence.* Addison-Wesley, 1986.

[14] F. Brooks, Jr. No silver bullet — essence and accident in software engineering. In *The Mythical Man-Month: Essays on Software Engineering*, chapter 16. Addison-Wesley, Anniversary edition, 1995.

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

[16] A. Brown. An overview of components and component-based development. In M. Zelkowitz, editor, *Advances in Computers*, volume 54, pages 1–34. Academic Press, 2001.

[17] M. Broy. Toward a mathematical foundation of software engineering methods. *IEEE Transactions on Software Engineering*, 27(1):42–57, Jan. 2001.

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

[19] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW '03: Proceedings of the 12th International Conference on World Wide Web*, pages 403–410, New York, NY, USA, 2003. ACM Press.

[20] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. R. M. Leino, and E. Poll. An overview of jml tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, Jun. 2005.

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

[22] G. Canfora. Software evolution in the era of software services. In K. Inoue, T. Ajisaka, and H. Gall, editors, 7th *International Workshop on Principles of Software Evolution (IWPSE 2004)*, pages 9–18. IEEE Computer Society, Sep. 2004.

[23] S. Chaki, E. Clarke, N. Sharygina, and N. Sinha. Dynamic component substitutability analysis. In M. Bravetti, L. Kloul, and G. Zavattaro, editors, 13$^{\text{th}}$ *International Symposium of Formal Methods (FM 2005)*, volume 3582 of *Lecture Notes in Computer Science*, pages 512–528. Springer-Verlag, 2005.

[24] S. Christensen and L. Petrucci. Modular analysis of Petri nets. *The Computer Journal*, 43(3):224–242, 2000.

[25] 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.

[26] S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design: The Theme Approach.* Addison-Wesley, Inc., 2005.

[27] 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.

[28] 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.

[29] D. Craig and W. Zuberek. Modelling and verification of compatibility of component composition. In *Third Workshop on Modelling of Objects, Components, and Agents*, pages 117–130, Oct. 2004.

[30] I. Crnkovic. Component-based software engineering — new challenges in software development. *Software Focus*, 2(4):127–133, 2002.

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

[32] K. Dam and M. Winikoff. Comparing agent-oriented methodologies. In P. Giorgini, B. Henderson-Sellers, and M. Winikoff, editors, 5th *International Bi-Conference Workshop, AOIS 2003*, volume 3030 of *Lecture Notes in Computer Science*, pages 78–93. Springer-Verlag, 2003.

[33] J. Desel and J.Esparza. *Free Choice Petri Nets.* Cambridge University Press, 1995.

[34] G. Di Marzo Serugendo, N. Foukia, S. Hassas, A. Karageorgos, S. Mostéfaoui, O. Rana, M. Ulieru, P. Valckenaers, and C. Van Aart. Self-organization: Paradigms and applications. In G. Di Marzo Serugendo, A. Karageorgos, O. Rana, and F. Zambonelli, editors, *Engineering Self-Organizing Systems: Nature Inspired Approaches to Software Engineering*, volume 2977 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 2004.

[35] J. Esparza and C. Schröter. Unfolding based algorithms for the reachability problem. In *Fundamenta Informaticae*, volume 46, pages 1–17. IOS Press, 2001.

[36] K. Ewusi-Mensah. *Software Development Failures: Anatomy of Abandoned Projects.* MIT Press, 2003.

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

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

[39] D. Garlan, R. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In Leavens and Sitaraman [61], chapter 3, pages 47–67.

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

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

[42] B. George, R. Fleurquin, and S. Sadou. A component-oriented substitution model. In M. Morisio, editor, *Reuse of Off-the-Shelf Components:* 9th *International Conference on Software Reuse*, volume 4039 of *Lecture Notes in Computer Science*, pages 340–353. Springer-Verlag, 2006.

[43] M. Hack. Analysis of production schemata by petri nets. Technical Report Project MAC TR-94, MIT, 1972.

[44] M. Hack. Petri net languages. Technical Report Project MAC TR-159, MIT, 1975.

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

[46] J. Highsmith III. *Adaptive software development : a collaborative approach to managing complex systems.* Dorset House Pub., 2000.

[47] C. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[48] J. Hopkins. Component primer. *Communications of the ACM*, 43(10):27–30, Oct. 2000.

[49] M. Huhns and M. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, 2005.

[50] R. Hull, M. Benedikt, V. Christophides, and J. Su. E-services: a look behind the curtain. In *PODS '03: Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–14, New York, NY, USA, 2003. ACM Press.

[51] R. Janicki and P. Lauer. *Specification and Analysis of Concurrent Systems: The COSY Approach.* Springer-Verlag, 1992.

[52] M. Jeng and M. Peng. Generating minimal siphons and traps for Petri nets. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, volume 4, pages 2996–2999. IEEE Press, Oct. 1996.

[53] C. Kapps and R. Stafford. *Assembly language for the PDP-11 RTS-RSX-UNIX.* PWS Computer Science, second edition, 1987.

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

[55] V. Khomenko and M. Koutny. LP deadlock checking using partial order dependencies. In C. Palamidessi, editor, 11th *International Conference on Concurrency Theory (CONCUR 2000)*, volume 1877 of *Lecture Notes in Computer Science*, pages 410–425. Springer-Verlag, 2000.

[56] V. Khomenko and M. Koutny. Towards an efficient algorithm for unfolding Petri nets. In K. Larsen and M. Nielsen, editors, 12th *International Conference on Concurrency Theory (CONCUR 2001)*, volume 2154 of *Lecture Notes in Computer Science*, pages 366–380. Springer-Verlag, 2001.

[57] 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.

[58] C. Kobryn. Modelling components and frameworks with UML. *Communications of the ACM*, 43(10):31–38, Oct. 2000.

[59] P. Krishnan and L. Wang. Supporting partial component matching. In R. Ghosh and H. Mohanty, editors, *First international conference on distributed computing and internet technology (ICDCIT 2004)*, volume 3347 of *Lecture Notes in Computer Science*, pages 294–303. Springer-Verlag, 2004.

[60] P. Lauer and R. Campbell. Formal semantics of a class of high-level primitives for coordinating concurrent processes. *Acta Informatica*, 5(4):297–332, Dec. 1975.

[61] G. Leavens and M. Sitaraman, editors. *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[62] H. Lewis and C. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, 1998.

[63] 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.

[64] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Munn. Specification and analysis of system architecture using rapide. *IEEE Transactions of Software Engineering*, 21(4):336–355, Apr. 1995.

[65] D. Luckham, J. Vera, and S. Meldal. Key concepts in architecture definition languages. In Leavens and Sitaraman [61], chapter 2, pages 23–45.

[66] G. Luger and W. Stubblefield. *Artificial Intelligence and the Design of Expert Systems*. Benjamin/Cummings Publishing Company Inc., 1989.

[67] M. Maier, D. Emery, and R. Hillard. Software architecture: Introducing IEEE Standard 1471. *IEEE Computer*, 33(4):107–109, Apr. 2001.

[68] 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.

[69] K. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Fourth International Workshop on Computer-Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177. Springer-Verlag, 1992.

[70] N. Medvidovic, D. Rosenblum, D. Redmiles, and J. Robbins. Modeling software architectures in the unified modeling language. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(1):2–57, 2002.

[71] 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.

[72] N. Medvidovic, R. Taylor, and E. W. Jr. Formal modelling of software architectures at multiple levels of abstraction. In *Proceedings of the California Software Symposium*, pages 28–40, Apr. 1996.

[73] R. Melinte. Coverability structure based analysis. Technical report, Universitatea "Alexandru Ioan Cuza" Iaşi, Facultatea de Informatică, Sep. 2002. TR 02-05.

[74] 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.

[75] S. Melzer and S. Römer. Deadlock checking using net unfoldings. In O. Grumberg, editor, 9th *International Workshop on Computer-Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 352–363. Springer-Verlag, 1997.

[76] R. Morrison, G. Kirby, D. Balasubramaniam, K. Mickan, F. Oquendo, S. Cîmpan, B. Warboys, B. Snowdon, and R. Greenwood. Support for evolving software architectures in the ArchWare ADL. In J. Magee, C. Szyperski, and J. Bosch, editors, *WICSA: 4th Working IEEE/IFIP Conference on Software Architecture*, pages 61–78. IEEE Computer Society, Jun. 2004.

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

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

[79] Object Management Group. OMG Unified Modeling Language Specification Version 1.5, March 2003.

[80] R. Orfali and D. Harkey. *Client/Server Programming with Java and CORBA*. John Wiley & Sons, 1997.

[81] E. Ostertag, J. Hendler, R. P. Díaz, and C. Braun. Computing similarity in a reuse library system: An AI-based approach. *ACM Transactions of Software Engineering and Methodology*, 1(3):205–228, Jul. 1992.

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

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

[84] J. Pérez-Martínez and A. Sierra-Alonso. UML 1.4 versus UML 2.0 as languages to describe software architectures. In F. Oquendo, B. Warboys, and R. Morrison, editors, *First European Workshop, EWSA 2004*, volume 3047 of *Lecture Notes in Computer Science*, pages 88–102. Springer-Verlag, 2004.

[85] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, Nov. 2002.

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

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

[88] E. Roman, S. Ambler, and T. Jewell. *Mastering Enterprise JavaBeans.* John Wiley & Sons, second edition, 2002.

[89] J. Sametinger. *Software Engineering with Reusable Components.* Springer-Verlag, 1997.

[90] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions of Software Engineering*, 21(4):314–335, Apr. 1995.

[91] M. Shaw and D. Garlan. *Software Architecture: Perspectives on and Emerging Discipline.* Prentice Hall, 1996.

[92] C. Sibertin-Blanc. The client-server protocol for communication of Petri nets. 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.

[93] M. Silva, J. Colom, and J. Campos. Linear algebraic techniques for the analysis of Petri nets. *Recent Advances in Mathematical Theory of Systems, Control, Networks, and Signal Processing II, Mita Press*, pages 35–42, 1992.

[94] M. Silva, E. Teruel, and J. Couvreur. Linear algebraic and linear programming techniques for the analysis and place/transition net systems. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 309–373. Springer-Verlag, 1998.

[95] M. Smith and M. Munro. Runtime visualisation of object-oriented software. In *First IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 81–89. IEEE Computer Society, Jun. 2002.

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

[97] P. Stark. Free petri net languages. In G. Goos, J. Hartmanis, and J. Winkowski, editors, *Proceedings of the* 7th *Symposium: Mathematical Foundations of Computer Science 1978*, volume 64 of *Lecture Notes in Computer Science*, pages 508–515–. Springer-Verlag, 1978.

[98] D. Steinberg and D. Palmer. *Extreme Software Engineering: A Hands-on Approach.* Peason/Prentice Hall, 2004.

[99] D. Stubbs and N. Webre. *Data Structures with Abstract Data Types and Pascal.* Brooks/Cole Publishing Company, 1985.

[100] S. Stuurman. Software architecture and JavaBeans. In P. Donohoe, editor, *Software architecture : TC2 first Working IFIP Conference on Software Architecture (WICSA1)*, pages 183–199, San Antonio, Texas, USA, February 1999. Kluwer Academic Press.

[101] C. Szyperski. Component software and the way ahead. In Leavens and Sitaraman [61], chapter 1, pages 1–20.

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

[103] S. Tanimoto, M. Yamauchi, and T. Watanabe. Finding minimal siphons in general Petri nets. *IEICE Transactions Fundamentals*, E79-A(11):1817–1824, 1996.

[104] E. Teiniker, S. Mitterdorfer, C. Kreiner, Z. Kovás, and R. Weiss. Local components and reuse of legacy code in the CORBA component model. In M. Fer-

nadez, editor, *Proceedings of the 28th EUROMICRO Conference*, pages 4–9. IEEE Computer Society, Sep. 2002.

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

[106] 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.

[107] 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.

[108] 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.

[109] F. Wang, Y. Gao, and M. Zhou. A modified reachability tree approach to analysis of unbounded Petri nets. *IEEE Transactions on Systems, Man, and Cybernetics — Part B: Cybernetics*, 34(1):303–308, Feb. 2004.

[110] A. Wills. Designing component kits and architecture with *Catalysis*. In L. .Barroca, J. Hall, and P. Hall, editors, *Software Architecture: Advances and Applications*, chapter 4, pages 61–85. Springer-Verlag, 2000.

[111] E. Yourdon. *Death March.* Prentice Hall, 2004.

[112] A. M. Zaremski and J. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(4):333–369, 1997.

[113] K. Zhang, D. Zhang, and J. Cao. Design, construction, and application of a generic visual language generation environment. *IEEE Transactions on Software Engineering*, 27(4):289–307, Apr. 2001.

[114] W. Zuberek. M-timed Petri nets, priorities, preemptions, and performance evaluation of systems. In G. Rozenberg, editor, *Advances in Petri Nets 1985*, volume 222 of *Lecture Notes in Computer Science*, pages 478–498. Springer-Verlag, 1985.

[115] W. Zuberek. Petri nets and timed Petri nets: basic concepts and properties. Technical Report 2000-01, Memorial University of Newfoundland, Department of Computer Science, Memorial University, St. John's, Newfoundland and Labrador, Canada A1B 3X5, Dec. 2000.

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

Appendix A

Deadlock Detection

This appendix provides a detailed demonstration of how structural properties and linear programming can be used to analyze the Petri net in Figure 6.13 presented in Chapter 6. The net exhibits a deadlock, as shown by the reachability analysis in Section 6.2.3. The deadlock can also be determined by simplifying the composed net and using linear programming, as will be demonstrated in this Appendix.

To employ linear programming, the basis and minimal siphons must be extracted so that objective functions can be determined. Since the original net has a large number of siphons, the concepts of similar and essential siphons, as well as parallel and alternate paths can be used to reduce the number of siphons (and to simplify their extraction).

Removing parallel paths and the bases of alternate paths from the net in Figure 6.13 results in the net shown in Figure A.1. The reduced net still preserves the deadlock properties of the original net (as discussed in Section 4.2.4); the reduced net will deadlock if and only if the original net deadlocks. This simplified model has

23 basis siphons, all of which are marked. These basis siphons include five minimal siphons and six marked traps, as shown in Table A.1. (The minimal siphons are S_1, S_2, S_{18}, S_{19} and S_{20}.) The marked traps can be disregarded because they can never become empty to create a deadlock. Therefore, only 17 siphons must be examined using linear programming ($S_1, \ldots, S_{17}$), the first two of which are minimal siphons.

The deadlock detection algorithm presented in Figure 4.6 of Chapter 4 first attempts to determine if there is a firing sequence which removes tokens from one of the minimal siphons (in order to produce a deadlock). The objective function for each minimal siphon is determined from the places constituting the siphon and their marking, while linear programming is used to determine if a firing vector exists which minimizes the objective function to zero while observing the constraints presented in Table A.2. These constraints are derived from the structure of the analyzed net and represent the requirement that the number of tokens assigned to the place cannot be negative. It should be noted that as siphons are emptied, new markings are obtained and the constraints can change accordingly. In particular, the constant in each constraint can change as siphons are emptied of their tokens. (This constant describes the marking of a particular place.) Additional constraints include the fact that all transitions must fire zero or more times (*i.e.* $x_{t_i}, 1 \leq i \leq 36$, must all be greater than or equal to zero).

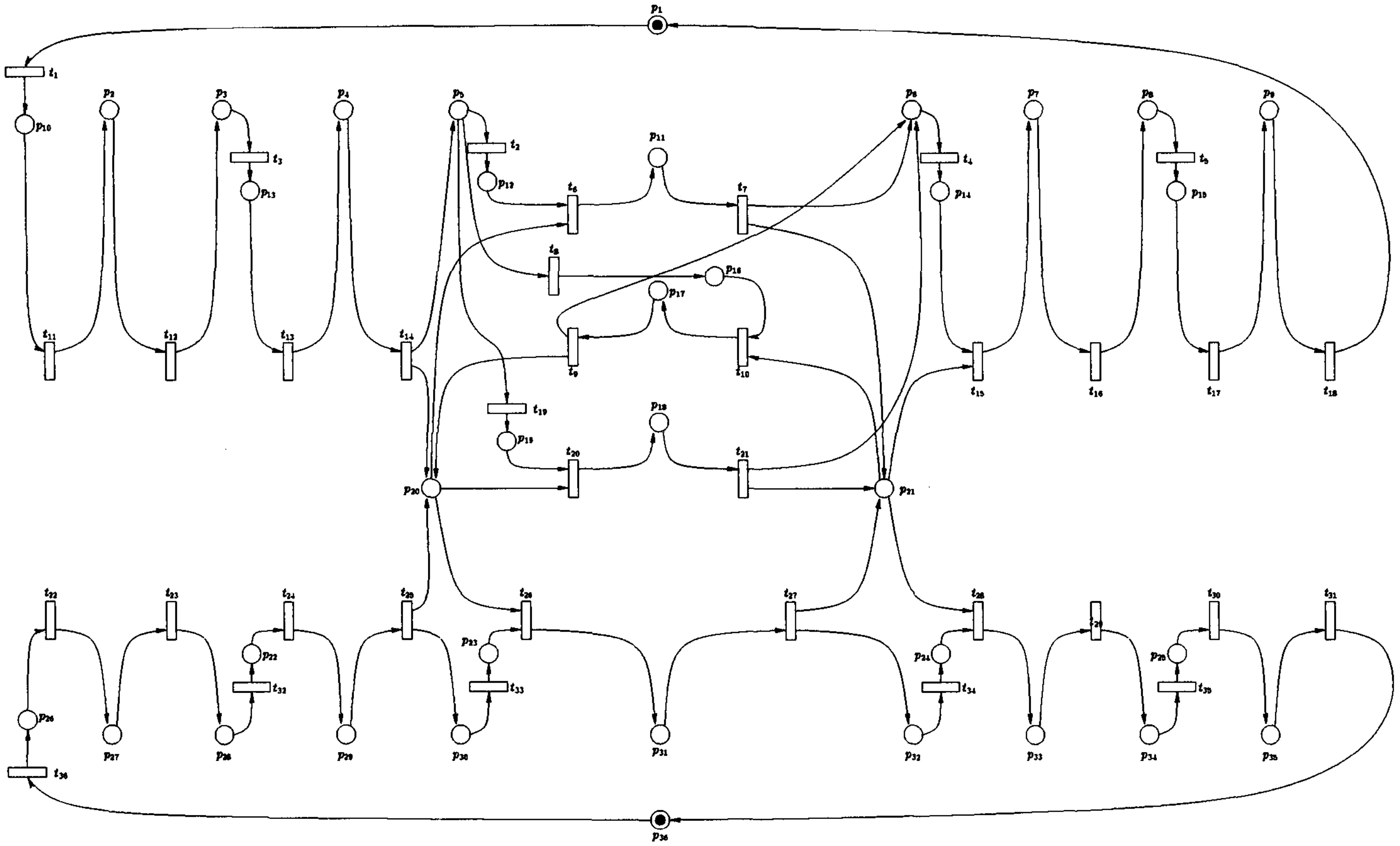

Figure A.1: Simplified incompatible composition between patient-prescription server interfaces

Table A.1: Marked basis siphons in Figure A.1

$S_1 = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{13}, p_{14}, p_{15}, p_{16}, p_{17}, p_{18}, p_{20}, p_{22}, p_{24}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{31}, p_{32}, p_{33}, p_{34}, p_{35}, p_{36}\}$
$S_2 = \{p_1, p_2, p_3, p_4, p_5, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{15}, p_{18}, p_{19}, p_{21}, p_{22}, p_{23}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{30}, p_{31}, p_{33}, p_{34}, p_{35}, p_{36}\}$
$S_3 = \{p_1, p_2, p_3, p_4, p_5, p_7, p_8, p_9, p_{10}, p_{11}, p_{13}, p_{15}, p_{17}, p_{18}, p_{20}, p_{21}, p_{22}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{31}, p_{33}, p_{34}, p_{35}, p_{36}\}$
$S_4 = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{15}, p_{17}, p_{18}, p_{19}, p_{21}, p_{22}, p_{23}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{30}, p_{31}, p_{33}, p_{34}, p_{35}, p_{36}\}$
$S_5 = \{p_1, p_2, p_3, p_4, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{13}, p_{15}, p_{17}, p_{18}, p_{20}, p_{21}, p_{22}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{31}, p_{33}, p_{34}, p_{35}, p_{36}\}$
$S_6 = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}, p_{15}, p_{16}, p_{17}, p_{18}, p_{20}, p_{22}, p_{24}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{31}, p_{32}, p_{33}, p_{34}, p_{35}, p_{36}\}$
$S_7 = \{p_1, p_2, p_3, p_4, p_5, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{15}, p_{17}, p_{18}, p_{20}, p_{21}, p_{22}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{31}, p_{33}, p_{34}, p_{35}, p_{36}\}$
$S_8 = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}, p_{15}, p_{17}, p_{18}, p_{19}, p_{21}, p_{22}, p_{23}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{30}, p_{31}, p_{33}, p_{34}, p_{35}, p_{36}\}$
$S_9 = \{p_1, p_2, p_3, p_4, p_5, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{15}, p_{16}, p_{18}, p_{19}, p_{21}, p_{22}, p_{23}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{30}, p_{31}, p_{33}, p_{34}, p_{35}, p_{36}\}$
$S_{10} = \{p_1, p_2, p_3, p_4, p_5, p_7, p_8, p_9, p_{10}, p_{11}, p_{13}, p_{15}, p_{16}, p_{17}, p_{18}, p_{20}, p_{21}, p_{22}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{31}, p_{33}, p_{34}, p_{35}, p_{36}\}$
$S_{11} = \{p_1, p_2, p_3, p_4, p_5, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{15}, p_{17}, p_{18}, p_{19}, p_{21}, p_{22}, p_{23}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{30}, p_{31}, p_{33}, p_{34}, p_{35}, p_{36}\}$
$S_{12} = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{13}, p_{14}, p_{15}, p_{16}, p_{17}, p_{18}, p_{19}, p_{20}, p_{22}, p_{24}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{31}, p_{32}, p_{33}, p_{34}, p_{35}, p_{36}\}$
$S_{13} = \{p_1, p_2, p_3, p_4, p_5, p_7, p_8, p_9, p_{10}, p_{11}, p_{13}, p_{15}, p_{17}, p_{18}, p_{19}, p_{20}, p_{21}, p_{22}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{31}, p_{33}, p_{34}, p_{35}, p_{36}\}$
$S_{14} = \{p_1, p_2, p_3, p_4, p_7, p_8, p_9, p_{10}, p_{11}, p_{13}, p_{15}, p_{17}, p_{18}, p_{20}, p_{21}, p_{22}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{30}, p_{31}, p_{33}, p_{34}, p_{35}, p_{36}\}$
$S_{15} = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{13}, p_{14}, p_{15}, p_{16}, p_{17}, p_{18}, p_{20}, p_{22}, p_{24}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{30}, p_{31}, p_{32}, p_{33}, p_{34}, p_{35}, p_{36}\}$
$S_{16} = \{p_1, p_2, p_3, p_4, p_7, p_8, p_9, p_{10}, p_{11}, p_{13}, p_{15}, p_{17}, p_{18}, p_{20}, p_{21}, p_{22}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{31}, p_{32}, p_{33}, p_{34}, p_{35}, p_{36}\}$
$S_{17} = \{p_1, p_2, p_3, p_4, p_5, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{15}, p_{18}, p_{19}, p_{21}, p_{22}, p_{23}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{30}, p_{31}, p_{32}, p_{33}, p_{34}, p_{35}, p_{36}\}$

Siphon-traps

$S_{18} = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}, p_{14}, p_{15}, p_{16}, p_{17}, p_{18}, p_{19}\}$
$S_{19} = \{p_1, p_2, p_3, p_4, p_7, p_8, p_9, p_{10}, p_{11}, p_{13}, p_{15}, p_{17}, p_{18}, p_{20}, p_{21}, p_{22}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{31}, p_{33}, p_{34}, p_{35}, p_{36}\}$
$S_{20} = \{p_{22}, p_{23}, p_{24}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{30}, p_{31}, p_{32}, p_{33}, p_{34}, p_{35}, p_{36}\}$
$S_{21} = \{p_1, p_2, p_3, p_4, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{13}, p_{14}, p_{15}, p_{17}, p_{18}, p_{20}, p_{21}, p_{22}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{31}, p_{33}, p_{34}, p_{35}, p_{36}\}$
$S_{22} = \{p_1, p_2, p_3, p_4, p_7, p_8, p_9, p_{10}, p_{11}, p_{13}, p_{15}, p_{17}, p_{18}, p_{20}, p_{21}, p_{22}, p_{23}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{30}, p_{31}, p_{33}, p_{34}, p_{35}, p_{36}\}$
$S_{23} = \{p_1, p_2, p_3, p_4, p_7, p_8, p_9, p_{10}, p_{11}, p_{13}, p_{15}, p_{17}, p_{18}, p_{20}, p_{21}, p_{22}, p_{24}, p_{25}, p_{26}, p_{27}, p_{28}, p_{29}, p_{31}, p_{32}, p_{33}, p_{34}, p_{35}, p_{36}\}$

Table A.2: Constraints for Figure A.1

Place	Constraint	Place	Constraint
p_1	$-x_{t_1} + x_{t_{18}} + 1 \geq 0$	p_{19}	$x_{t_{19}} - x_{t_{20}} \geq 0$
p_2	$x_{t_{11}} - x_{t_{12}} \geq 0$	p_{20}	$-x_{t_6} + x_{t_9} + x_{t_{14}} - x_{t_{20}} + x_{t_{25}} - x_{t_{26}} \geq 0$
p_3	$-x_{t_3} + x_{t_{12}} \geq 0$	p_{21}	$x_{t_7} - x_{t_{10}} - x_{t_{15}} + x_{t_{21}} + x_{t_{27}} - x_{t_{28}} \geq 0$
p_4	$x_{t_{13}} - x_{t_{14}} \geq 0$	p_{22}	$-x_{t_{24}} + x_{t_{32}} \geq 0$
p_5	$-x_{t_2} - x_{t_8} + x_{t_{14}} - x_{t_{19}} \geq 0$	p_{23}	$-x_{t_{26}} + x_{t_{33}} \geq 0$
p_6	$-x_{t_4} + x_{t_7} + x_{t_9} + x_{t_{21}} \geq 0$	p_{24}	$-x_{t_{28}} + x_{t_{34}} \geq 0$
p_7	$x_{t_{15}} - x_{t_{16}} \geq 0$	p_{25}	$-x_{t_{30}} + x_{t_{35}} \geq 0$
p_8	$-x_{t_5} + x_{t_{16}} \geq 0$	p_{26}	$-x_{t_{22}} + x_{t_{36}} \geq 0$
p_9	$x_{t_{17}} - x_{t_{18}} \geq 0$	p_{27}	$x_{t_{22}} - x_{t_{23}} \geq 0$
p_{10}	$x_{t_1} - x_{t_{11}} \geq 0$	p_{28}	$x_{t_{23}} - x_{t_{32}} \geq 0$
p_{11}	$x_{t_6} - x_{t_7} \geq 0$	p_{29}	$x_{t_{24}} - x_{t_{25}} \geq 0$
p_{12}	$x_{t_2} - x_{t_6} \geq 0$	p_{30}	$x_{t_{25}} - x_{t_{33}} \geq 0$
p_{13}	$x_{t_3} - x_{t_{13}} \geq 0$	p_{31}	$x_{t_{26}} - x_{t_{27}} \geq 0$
p_{14}	$x_{t_4} - x_{t_{15}} \geq 0$	p_{32}	$x_{t_{27}} - x_{t_{34}} \geq 0$
p_{15}	$x_{t_5} - x_{t_{17}} \geq 0$	p_{33}	$x_{t_{28}} - x_{t_{29}} \geq 0$
p_{16}	$x_{t_8} - x_{t_{10}} \geq 0$	p_{34}	$x_{t_{29}} - x_{t_{35}} \geq 0$
p_{17}	$-x_{t_9} + x_{t_{10}} \geq 0$	p_{35}	$x_{t_{30}} - x_{t_{31}} \geq 0$
p_{18}	$x_{t_{20}} - x_{t_{21}} \geq 0$	p_{36}	$x_{t_{31}} - x_{t_{36}} + 1 \geq 0$

Linear programming shows that the numbers of tokens in S_2 (in Table A.1), which is represented by the objective function $2 - x_{t_8} - x_{t_{10}}$, can be minimized to zero with the firing vector:

$$[1,0,1,0,0,0,0,1,0,1,1,1,1,1,0,0,0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,1,1,0,0,1].$$

This vector is feasible and corresponds to the firing sequence

$$(t_1, t_{11}, t_{12}, t_3, t_{13}, t_{14}, t_8, t_{36}, t_{22}, t_{23}, t_{32}, t_{24}, t_{25}, t_{33}, t_{26}, t_{27}, t_{10}).$$

However, emptying this siphon does not result in a deadlock. Furthermore, the first minimal siphon, represented by the objective function $2 - x_{t_2} + x_{t_9} + x_{t_{14}} - x_{t_{19}}$, cannot be emptied. Therefore, the algorithm must continue to test all 17 basis siphons to (recursively) determine a siphon draining sequence that results in deadlock.

Using the **deadlock** algorithm, it can be determined that draining the basis siphons in the sequence of (S_2, S_4, S_{17}) results in the dead marking shown in Figure 6.14. The corresponding objective function for each siphon, the firing vector that minimizes each objective function to zero and the firing sequence that corresponds to the firing vector are presented in Table A.3.

Table A.3: Siphons, objective functions and firing vectors/sequences

Siphon	Objective function, firing vector and firing sequence
S_2	$2 - x_{t_8} - x_{t_{10}}$, [1,0,1,0,0,0,0,1,0,1,1,1,1,1,0,0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,1,1,0,0,1], $(t_1, t_{11}, t_{12}, t_3, t_{13}, t_{14}, t_8, t_{36}, t_{22}, t_{23}, t_{32}, t_{24}, t_{25}, t_{33}, t_{26}, t_{27}, t_{10})$
S_4	$1 - x_{t_4} + x_{t_7} - x_{t_8} + x_{t_{21}}$, [0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0], (t_9, t_4)
S_{17}	$1 - x_{t_8} - x_{t_{10}} + x_{t_{27}} - x_{t_{34}}$, [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0], (t_{34})

The siphon sequence (S_2, S_4, S_{17}) is not unique — other sequences exist which also result in the same deadlock. Some of these siphon sequences and the corresponding transition firing sequences that result in a deadlock, are shown in Table A.4. These siphon sequences were obtained by randomizing the order of the basis siphons and running the same algorithm as above.

The presence of several siphon sequences may help to lessen the time complexity of the algorithm in pragmatic cases — the siphons can be ordered in such a way that the most likely candidates are analyzed first. Also helping to mitigate the complexity of the algorithm is the fact that when a siphon's tokens are fully drained, other "overlapping" siphons could also have become empty. These additional empty

Table A.4: Other siphons emptying sequences and their corresponding firing sequences that result in deadlock

Siphon	Firing sequence
S_{11}	$(t_1, t_{11}, t_{12}, t_3, t_{13}, t_{14}, t_8, t_{36}, t_{22}, t_{23}, t_{32}, t_{24}, t_{25}, t_{33}, t_{26}, t_{27}, t_{10}, t_9)$
S_{17}	(t_{34})
S_4	(t_4)
S_2	$(t_1, t_{11}, t_{12}, t_3, t_{13}, t_{14}, t_8, t_{36}, t_{22}, t_{23}, t_{32}, t_{24}, t_{25}, t_{33}, t_{26}, t_{27}, t_{10})$
S_{11}	(t_9)
S_4	(t_4)
S_{17}	(t_{34})
S_9	$(t_1, t_{11}, t_{12}, t_3, t_{13}, t_{14}, t_8, t_{36}, t_{22}, t_{23}, t_{32}, t_{24}, t_{25}, t_{33}, t_{26}, t_{27}, t_{10})$
S_{17}	(t_{34})
S_4	(t_9, t_4)
S_{17}	$(t_1, t_{11}, t_{12}, t_3, t_{13}, t_{14}, t_8, t_{36}, t_{22}, t_{23}, t_{32}, t_{24}, t_{25}, t_{33}, t_{26}, t_{27}, t_{10}, t_{34})$
S_{11}	(t_9)
S_4	(t_4)
S_{17}	$(t_1, t_{11}, t_{12}, t_3, t_{13}, t_{14}, t_8, t_{36}, t_{22}, t_{23}, t_{32}, t_{24}, t_{25}, t_{33}, t_{26}, t_{27}, t_{10}, t_{34})$
S_4	(t_9, t_4)

siphons effectively prune the recursive search space since the algorithm does not explore siphons that have become emptied as it progresses through its analysis of the set of basis siphons.

Appendix B

Petri Net File Format

This appendix describes a file format that can be used to textually describe the nets presented in this thesis. The file structure consists of individual sections listing the places, the transitions/connectivity, and finally the initial marking. Each of these three sections are described in more detail in the following sections. For convenience, software has been written which can convert graphical descriptions of the nets (in `xfig` format) into the format described by this Appendix.

B.1 Places

The places are specified by a comma-separated list of the place names. Each place name has the syntax of a traditional identifier (an alphabetic character followed by zero or more alphanumeric characters) and each name must be unique. The list of places is prefixed by `pdef=`. The resulting string is then enclosed by `net[` and `]`.

B.2 Transitions/Connectivity

The transitions are then provided in a semi-colon separated list in which each list item consists of the transition identifier (preceded by '#') followed by a comma-separated list of the identifiers representing the input places of the transition followed by a comma-separated list of identifiers representing the output places of the transition. The transition identifier is separated from the two lists of places by '=' and the input/output places are separated from each other by '/'.

B.3 Initial Marking

The initial marking is specified by providing all the places that have tokens in a comma-separated list. By default, places in this list will have exactly one token in the initial marking. More than one token can be assigned to a place by specifying the number of tokens after the place name. A colon separates the place name from the number of tokens.

B.4 Grammar

A grammar for the textual description of nets is given below:

```
Net          ::=  net[pdef= Places ]
                  ( Transitions );
                  mark( Markings );
Places       ::=  Id-List
Transitions  ::=  Trans {; Trans}*
Trans        ::=  #id = Id-List/Id-List
Markings     ::=  mark(id[:num]{,id[:num]}*)
Id-List      ::=  id {, id}*
```

B.5 Example

Using the above grammar, the Petri net given in Figure A.1 has the following textual description:

```
net[pdef=p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,
p16,p17,p18,p19,p20,p21,p22,p23,p24,p25,p26,p27,p28,p29,p30,
p31,p32,p33,p34,p35,p36]
   (#t1=p1/p10;
    #t2=p5/p12;
    #t3=p3/p13;
    #t4=p6/p14;
    #t5=p8/p15;
    #t6=p20,p12/p11;
    #t7=p11/p6,p21;
    #t8=p5/p16;
    #t9=p17/p20,p6;
    #t10=p21,p16/p17;
    #t11=p10/p2;
    #t12=p2/p3;
    #t13=p13/p4;
    #t14=p4/p20,p5;
    #t15=p14,p21/p7;
    #t16=p7/p8;
    #t17=p15/p9;
    #t18=p9/p1;
    #t19=p5/p19;
    #t20=p20,p19/p18;
    #t21=p18/p21,p6;
    #t22=p26/p27;
    #t23=p27/p28;
    #t24=p22/p29;
    #t25=p29/p30,p20;
    #t26=p23,p20/p31;
    #t27=p31/p32,p21;
    #t28=p24,p21/p33;
    #t29=p33/p34;
    #t30=p25/p35;
    #t31=p35/p36;
    #t32=p28/p22;
    #t33=p30/p23;
    #t34=p32/p24;
    #t35=p34/p25;
    #t36=p36/p26);
mark(p1,p36);
```

Appendix C

Parallel/Alternate Path Algorithms

This appendix presents algorithms used in Section 4.2.4 (Chapter 4) in the structural analysis of Petri nets. In particular, the algorithms identify parallel and alternate paths which can be used to reduce the number of inessential siphons and to simplify deadlock detection.

C.1 Parallel Paths

The procedure to identify the transitions that delimit parallel paths in a net is presented in Figure C.1. The procedure examines all transitions in the given net that have more than one output place (by definition, a transition that has less than two output places cannot act as the origin of a parallel path). Each of the output places is then analyzed to determine if there exists more than one simple path that extends from the originating transition to the same terminating transition. Each output of the same pair of transitions denotes another parallel path between the two transitions (which can be removed from the net).

```
proc parallel-paths($\mathcal{N}$)
  begin
    for each $t$ in $\mathit{Transitions}(\mathcal{N})$ do
      if $|\mathsf{Out}(t)| > 1$ then
        $T := \{ \}$;
        for each $p$ in $\mathsf{Out}(t)$ do
          $t' :=$ endtpath($\mathcal{N}, p$);
          if $t' \neq$ nil then
            if $t' \in T$ then
              output($t, t'$)
            else
              $T := T \cup t'$
            endif
          endif
        endfor
      endif
    endfor
  end
```

Figure C.1: Procedure parallel-paths

The parallel-paths algorithm makes use of an auxiliary function called endtpath, presented in Figure C.2, which identifies the terminating transition at the end of a simple path that starts from the place given as its argument. If the place is not part of a simple path, then the function returns nil. The function steps through each place and transition along the path ensuring that their respective input and output sets are all singletons. The assignment of $cont := p \neq first$ prevents infinite looping which may occur if the net contains a cycle. In this function, $\mathsf{Out}(p)$ is the set of all transitions in the output set of p, whereas $\mathsf{out}(p)$ represents the single output transition of the given place. The same idea applies to the difference between $\mathsf{Out}(t)$

and out(t). The complexity of this algorithm is linear with respect to the sum of the number places and transitions.

```
func endtpath($\mathcal{N}, p$) : transition
  begin
    $t :=$ nil;
    $first := p$;
    $cont :=$ true;
    while $cont$ do
      $cont :=$ false;
      if $|\mathsf{Inp}(p)| = 1$ and $|\mathsf{Out}(p)| = 1$ then
        $t := \mathsf{out}(p)$;
        if $|\mathsf{Inp}(t)| = 1$ and $|\mathsf{Out}(t)| = 1$ then
          $p := \mathsf{out}(t)$;
          $cont := p \neq first$
        endif
      else
        $t :=$ nil
      endif
    endwhile;
    return $t$
  end
```

Figure C.2: Function **endtpath**

C.2 Alternate Paths

A procedure that identifies the base portion of alternate paths is presented in Figure C.3. The procedure iterates over the places attempting to identify a place which is the starting point of the base of an alternate path. Places whose input sets contain only one transition are rejected as they cannot be part of a base path. For all other places, the **endppath** function, described below, is then used to determine if the place is the start of a simple path (with the result from **endppath** being the last place of the simple path). If a place is the start of a simple path, then the sets of transitions that lead into and out of the base path are identified. A check is then made (using the **endtpath** function of Figure C.2) to ensure that the two sets of transitions can be paired off completely by identifying simple paths between them. If this is successful, the two places identified earlier constitute the starting and ending points of a base path.

```
proc alternate-paths($\mathcal{N}$)
  begin
    for each $p$ in $Places(\mathcal{N})$ do
      if $|\mathsf{Inp}(p)| > 1$ then
        $base := \texttt{nil}$;
        $p' := \mathsf{endppath}(\mathcal{N}, p)$;
        if $p' \neq \texttt{nil}$ then
          $T_1 := \mathsf{Inp}(p)$;
          $T_2 := \mathsf{Out}(p')$;
          for each $t$ in $T_1$ do
            for each $p''$ in $\mathsf{Out}(t)$ do
              $t' := \mathsf{endtpath}(p'')$;
              if $t' \neq \texttt{nil}$ and $t' \in T_2$ then
                $T_2 := T_2 - \{t'\}$;
                $T_1 := T_1 - \{t\}$;
                $base := (p, p')$
              endif
            endfor
          endfor;
          if $T_1 = \emptyset$ and $T_2 = \emptyset$ and $base \neq \texttt{nil}$ then
            output($base$)
          endif
        endif
      endif
    endfor
  end
```

Figure C.3: Procedure **alternate-paths**

The algorithm for the **endppath** function, used by the **alternate-paths** function above, is shown in Figure C.4. This function works similarly to **endtpath**, but instead of returning the last transition of a simple path, it returns the last place. The only other major difference is that the **endppath** algorithm must take into account the fact that the input set of the starting place of the base and the output set of the final place are not necessarily singleton sets. As with **endtpath**, the complexity of this algorithm is linear with respect to the sum of the number places and transitions.

```
func endppath($\mathcal{N}$, $p$) : place
   begin
      $p' := p$;
      $first := p$;
      $cont :=$ true;
      while $cont$ do
         $cont :=$ false;
         if $|\mathrm{Out}(p')| = 1$ then
            $t := \mathrm{out}(p')$;
            if $|\mathrm{Inp}(t)| = 1$ and $|\mathrm{Out}(t)| = 1$ then
               $p' := \mathrm{out}(t)$;
               if $|\mathrm{Inp}(p')| = 1$ then
                  $cont := p' \neq first$
               else
                  $p' :=$ nil
               endif
            else
               $p' :=$ nil
            endif
         endif
      endwhile;
      return $p'$
   end
```

Figure C.4: Function **endppath**