Formalisation of
Component-based Systems

Jiong Zhou

Submitted for the Degree of
Doctor of Philosophy
from the
University of Surrey

Centre for Software Systems Engineering
Department of Computing
School of Electronics, Computing and Mathematics
University of Surrey
Guildford, Surrey GU2 7XH, UK

January 2007

© J.Zhou 2007
Abstract

Current advances in software engineering practice involve the adoption of a component-based approach in developing large-scale, complex systems. The component-based paradigm provides better structuring of systems and facilitates systematic software reuse. However, complex interactions between components, especially in concurrent, real-time and embedded applications, pose greater challenges. This thesis proposes a formal language for this kind of systems, especially for managing the dependencies between components, in terms of their interactions in a concurrent setting.

In our model, components are autonomous elements encapsulating functionality and connectors exist only to serve the communication needs of others. Connectors further can initiate and govern component communications. This design takes communication and control out of components and encapsulates them into connectors, hence improves the reusability of components.

In our approach, each component is represented by a component signature, which identifies a component; and a time-slot language, which describes the behaviour of a component with timing sensitivity. This language-based representation of component behaviour makes it possible to capture concurrency at the individual interface level. The interpretation of concurrency is that of a non-interleaving model, with the notion of causal independence lifted to multi-threaded runs. Based on time-slot languages as an operational semantic domain, we introduce component protocols, a service-based expression language, serves as a syntactic behaviour description and which can be formally interpreted into time-slot languages via the initial algebra approach.

Component interoperability in this approach is a design time concept. It boils down to the properties of deadlock-freedom on glue, loyalty on roles with glue, compatibility of ports and roles, and substitutability between replacement ports and current ports. The well-formedness properties of components and connectors will be evaluated individually before being wired together. These mainly build on the concepts of well-definedness, well-behavedness and refinement relationship of component protocols. This approach follows the practical bottom-up approach from unit-testing to integration-testing.

Key words: components, component-based systems, Koala model, connectors, interactions, concurrency, formal languages, interoperability, well-formedness
Acknowledgements

I would like to express my gratitude to Professor Paul Krause, my supervisor, for offering me this opportunity in my life to finish this PhD study. I am grateful for his help and guidance with many discussions, including industrial case studies. I also very much appreciate Dr. Mike Shields, spending a large amount of time with me to formalise this work. At the last stage before he moved to Malta to enjoy his retired life, we had our meetings on a daily basis. I have also benefitted from discussions with Dr. David Pitt. Thanks are also due to the Department of Computing for financial support while working on this thesis.

I owe so much to my father 周亨利 (Heng Li Zhou), my mother 曹虹屏 (Hong Ping Cao), my uncle 周亨明 (Heng Ming Zhou) and other members of my family, for their love and support; this thesis is dedicated to them.

Finally, my warmest thanks go to 郑咏艳 (Yong Yan Zheng) for supporting me in many different ways. She deserves particular consideration for the selfless love and care shown to me.
Contents

1 Introduction ....................................................................................................................................... 1
  1.1 Setting the Context ...................................................................................................................... 1
  1.2 Main Objective of the Thesis .................................................................................................. 4
  1.3 Plan of the Thesis ..................................................................................................................... 5

2 Approaches to Modelling Component-based Systems ............................................................ 7
  2.1 Introduction ................................................................................................................................. 7
  2.2 Current Software Component Models ...................................................................................... 11
    2.2.1 The Koala Model for Components .................................................................................. 12
    2.2.2 The UML Model for Components .............................................................................. 19
    2.2.3 The SOFA Model for Components ................................................................................. 27
  2.3 Formal Approaches for Components ........................................................................................ 36
    2.3.1 Event-based Approaches ................................................................................................. 37
    2.3.2 State-based Approaches ................................................................................................... 43
  2.4 Conclusion and Goals of the Thesis ...................................................................................... 51

3 A Generic Component-based System Model ........................................................................... 56
  3.1 Primary Components ................................................................................................................ 56
  3.2 Connectors ................................................................................................................................. 58
  3.3 Composed Components .......................................................................................................... 61
  3.4 The Lifecycle of Component-based Systems .......................................................................... 63
  3.5 Conclusion ................................................................................................................................. 65

4 Formalisation of Component-based Systems ........................................................................... 69
  4.1 Introduction ............................................................................................................................... 69
  4.2 Component Signature ................................................................................................................ 72
  4.3 Time-slot Model ....................................................................................................................... 79
  4.4 Operations of Time-slot Model ............................................................................................... 88
  4.5 Component Protocol ................................................................................................................ 105
  4.6 Formalisation of Component-based Systems ........................................................................ 112
  4.7 Industrial Case Study .............................................................................................................. 114
  4.8 Conclusion ................................................................................................................................. 129

5 Properties of Component-based Systems ................................................................................ 137
List of Figures

Figure 2.1 The Koala Graphical Notation ................................................................. 13
Figure 2.2 The Koala Interface Description ............................................................... 14
Figure 2.3 The Koala Component Description ......................................................... 14
Figure 2.4 The Koala Configuration Description ....................................................... 15
Figure 2.5 The Use of Koala Switch ......................................................................... 16
Figure 2.6 The Scenario Specifications for Koala Switch .......................................... 17
Figure 2.7 A Component in UML 2.0 ..................................................................... 20
Figure 2.8 A Structural Diagram for Components .................................................. 21
Figure 2.9 A UML 2.0 Composite Structure Diagram for a Component Store .......... 21
Figure 2.10 The Parallel Construct in UML 2.0 and a Possible Interpretation ............ 22
Figure 2.11 The Package Structure of the UML Meta-model .................................. 26
Figure 2.12 A Database Component in SOFA ......................................................... 29
Figure 2.13 The CSProcCall Connector Type Architecture .................................... 30
Figure 2.14 Examples of Interface, Frame and Architecture Specification in SOFA CDL .............................................................................................................. 31
Figure 2.15 The Development Tree of Application and Assembled Application ........ 33
Figure 3.1 A Primary Component .......................................................................... 58
Figure 3.2 A Component with a Connector .............................................................. 60
Figure 3.3 A Composed Component ....................................................................... 62
Figure 4.1 The Vendor Component ........................................................................ 77
Figure 4.2 The Customer Component ..................................................................... 78
Figure 4.3 The Printer Component .......................................................................... 78
Figure 4.4 The Monitor Component ........................................................................ 78
Figure 4.5 The Conn Connector .............................................................................. 111
Figure 4.6 A Koala Fork with Synchronous Drop Requests and Synchronous Restores .............................................................................................................. 115
Figure 4.7 A Koala Fork with Asynchronous Drop Requests and Synchronous Restores .............................................................................................................. 116
Figure 4.8 The 3-ary Koala Fork Connector F .......................................................... 117
Figure 4.9 A Koala Fork with Asynchronous Drop Signal and Restore Commands in FSP .............................................................................................................. 122
Figure 4.10 A Koala Fork with Synchronous Drop Signal and Restore Commands in FSP .............................................................................................................. 123
Figure 4.11 The Component A Attached with Fork Connector F ............................... 125
Figure 4.12 The Component D ................................................................................ 128
Figure 5.1 A Port Attached with Two Roles ............................................................ 152
Figure 5.2 A Role Attached with Two Ports ............................................................ 153
Figure 5.3 The Interoperability of Components .......................................................... 168
Figure 6.1 The Vendor Component ................................................................. 171
Figure 6.2 The Printer Component ................................................................. 172
Figure 6.3 The Customer Component ............................................................. 172
Figure 6.4 The Monitor Component ............................................................... 173
Figure 6.5 The PScI Port ........................................................................... 175
Figure 6.6 The ComA Component ................................................................. 175
Figure 6.7 The ComA Component without PotG ........................................... 176
Figure 7.1 Binding with Adaptors ................................................................. 195
Chapter 1

Introduction

Recent years have seen the increasing use of software in the industrial marketplaces, ranging from consumer electronics products to telecommunications to biomedical devices. Modern software systems often comprise complex combinations of previously unrelated functions. By now, there is near-universal agreement amongst developers of large-scale software systems on the benefits that accrue from adopting a component-based approach to software engineering. The idea is that the development of software-intensive systems in a timely and affordable fashion can, potentially, be realised by assembling systems from prefabricated components. A software component can be seen as an encapsulated software entity which has an explicit interface that fully describes its externally visible properties and can be used in a variety of configurations.

1.1 Setting the Context

A component makes its functionality available to other components or the environment via a set of provided services and, possibly, requires services from other components, in order to deliver those promised, via a set of required services. A component supports a provided service if the component contains an implementation of that service. If the component requires access to a provided service defined on another component, it issues its request through a required service. Hence, the functionality of a component is made available to other components only through its services and which, on the other hand, hide the implementation details of that functionality.

Component interaction is understood as a sequence of service communications between components. It is recognised as one of the principal issues in component-based systems (CBS for short) development. A well-known approach is to specify the component interacting mechanisms separate from components themselves. The first-class connector concept in the Wright architecture definition language is introduced by Allen and Garlan [AG97, AG98], stating that "Whereas the implementation relationship is concerned with how a component achieves its computation, the interaction relationship is used to understand how that computation is combined with others in the overall system." Thus, the concepts of computations and communications of components are clearly being distinguished. That constitutes one of the starting points of our work.
In this way, the component-based paradigm offers promising solutions for software development. In addition to increasing the scope for reuse, a component-oriented approach allows for ease of maintenance and customisation of the pre-built components to allow for the incorporation of new functions and features. CBSs are more likely to be able to cope with the increasing demands for modifiability and evolvability of the constructed system to accommodate future demands.

The primary focus in building a component is that it must be replaceable. It should be possible to replace existing component in a system configuration either by a different implementation of the same functionality or by an upgraded version of the existing implementation in terms of added functionality. Inevitably, this places emphasis on component specification during design. This intrinsic aspect of the component-based approach becomes particularly relevant when considering that present-day software systems have to cater for ever changing requirements (even, as van Ommering points out, during the development of a single product [vO03, vO04]). Moreover, CBS development in principle results in reduced time-to-market since new instances of a product family can be developed with a short lead-time.

Not surprisingly, considering the above, components are often seen as panacea for solving the software productivity crisis. There are various potential benefits from moving towards the (re)use of pre-built components in constructing the final system, but, in practice, there are difficult technical issues that remain to be explored before this potential is realised in general.

Complex interactions between components pose greater challenges. Components may be developed at different times, by different developers with, possibly, even different uses in mind, and under different assumptions. Internal assumptions of a component about the order in which its operations will be called may no longer be valid when the component is placed in a different context. Differing assumptions between a set of components can be further exposed in a concurrent setting, especially in the case of reactive real-time embedded systems such as those developed for the customer electronic and telecommunication industries or other mission-critical software systems, which make strong demands in terms of asynchrony and concurrency. For instance, one component may have been designed under the assumption that it receives certain signals consecutively but, when placed in a different configuration, the other component is generating them concurrently.

Part of the problem seems to be that designers of such systems have no agreed way of expressing the behaviour of components; thus, where inconsistencies tend to occur. At a result, software systems built by assembling together prefabricated components sometimes exhibit the problem of interoperability. That is, behaviour inconsistencies occur when a situation is
encountered where components cannot work with each other although they may have matched service sets.

Many of the difficulties that arise during the integration of components into the final system could, in principle, be avoided by considering a detailed specification of the behaviour of components. In concurrent component technologies, the notion of a component is associated with the signatures of the services the component provides to its environment. This semantically thin component description does not offer enough information to be of value in reasoning about components beyond the signature of their services.

A precondition for managing reuse, replacement and composition of components is the ability to precisely describe their behaviour in terms of sequences of service executions occurred in the components, and also the ability to precisely describe the communications taking place at connectors [DR97]. A precise description of the communications between components boils down to capturing the dependencies between interacting components. In this context, a dependency is understood as the reliance of one component on another to support a specific chain of services. Such dependencies between components are often referred to as component contracts [Szy97].

Graphical descriptive techniques can be used to support component-based design in terms of visualisation and communication of ideas. Recently, diagrammatic notations such as the Unified Modelling Language (UML) [BRJ99] have been enhanced with component concepts and now include diagrams for representing the structural dependencies between components in a configuration (in which components rely on each other in terms of pairs of provided and required interfaces) and to a limited extent, their behavioural dependencies in terms of a causal ordering of interactions.

It transpires that graphical descriptive approaches alone may not adequately describe component contracts with sufficient precision. In order to capture the behavioural aspect of dependencies, a component contract needs to be formalised in terms of the sequences of requests the component is capable of servicing, and the respective sequences of requests the component makes through its required services while making its services available. This additional behavioural information can be exploited in reasoning about component behaviour and can be ‘reused’ when the component is placed in a different context.

A long-standing concern of work in formal methods is the precise description of the behaviour of communicating systems. Prime examples are Petri nets as developed by Petri [Pet79b], process algebras as established by Miller [Mil80] and Hoare [Hoa85], Mazurkiewicz’s trace languages [Maz88], asynchronous transition systems as proposed by Shields [Shi85] and Bednareczk [Bed88], and event structures as devised by Nielsen, Plotkin and Winskel [NPW81]. A common denominator of these theoretical models is the treatment of the phenomenon of concurrency.
Process algebra, such as CSP and CCS, identify concurrency with nondeterministic *interleaving* of events. In event structure and partial order models, concurrency is derived from causal dependency and conflict between events. This notion of concurrency is often referred to as *true concurrency*. A generalisation of the event structures model, the so-called *behaviour presentations* as created by Shields [Shi88], introduces, in addition, the concept of *simultaneity* which can be understood as a refinement of true concurrency. In trace languages, and asynchronous transition systems, concurrency is captured through a notion of independence between events. In these models, often referred to as *independence models*, two events are concurrent if they are independent, i.e. no relative ordering is imposed between them, and they could be simultaneous.

It can be seen that concurrency comes in different flavours in the various formal models and it is the target application for the formal semantics that determine the choice between them. In the context of Component-Based Software Engineering (CBSE for short), it can be argued that different aspects of software design can be covered, and different classes of software design errors be uncovered, by considering different notions of concurrency.

**1.2 Main Objective of the Thesis**

The objective of this thesis is to provide a formal framework for rigorous analysis and reasoning about CBSs, in terms of their composition. We shall attempt to formulate a model for components which is expressive enough to capture subtle issues like concurrency, nondeterminism and simultaneity, whilst providing a formal underpinning to graphical notations used in more conventional approaches to component-based software engineering, such as Koala and so forth.

The formal framework is enlightened by a range of concurrency theories, though we have opted for a non-interleaving semantics model which draws upon early ideas about vector languages [Shi79, MS03]. This representation of component behaviour allows for expressing concurrency at the level of individual interfaces of components in addition to the usual concurrency arising through composition, and recently through distinct interfaces with the same component.

Further, a service-oriented description of behaviour is introduced to make the behaviour description simple enough to be easily applied in any component architectural definition language (ADL) specifications. The description specifies component behaviour in a protocol pattern, and can be uniquely translated into a language-based description by means of the initial algebra approach.
Chapter 1. Introduction

The notions of interactability and well-formedness within the CBS model allows for reasoning about properties of the composite, based on properties of the individual components and connectors.

The contribution of this work lies in enriching component-based design practices with mathematical methods which can be used to faithfully describe component behaviours and determine whether component can be fitted together in a useful way, taking into account concurrency range from individual interface level to composition level. In doing so, it also provides a formal operational semantics to the Koala component model, and identifies component interoperability and well-formedness properties. In addition, it should also contribute towards cementing a relationship between formal foundational research and software system engineering whilst advancing the field of component-based software design.

1.3 Plan of the Thesis

This thesis is organised as follows.

In Chapter 2, we review state-of-the-art techniques for modelling CBS. After an overview of component concepts for specification and analysis, we describe three software component models: Koala [vOvdL'00, vO02b], UML 2.0 [OMG04a] and SOFA [PBJ98] and discuss formal approaches tailored to the specification and analysis of components, followed by addressing the motivation and goal of this research work.

Chapter 3 presents a generic CBS model inspired by the Koala component model. Based on Allen and Garlan's first-class connector model which separates communication mechanisms from components, it increases the reusability of components (the same component can be used in a variety of environments with different communication primitives), our motivation for using our connectors is to further encapsulate control out of components, in order to minimize coupling between components. This refinement improves the reusability of components in a number of different contexts and makes the development process of CBS focus more on designing the component interacting mechanism, so that moves the CBS development activities towards a higher level of design abstraction. The hierarchical design of components is facilitated by introducing proxy ports, which provide a medium for the internal structure of the component to exchange information with external entities from higher levels in the hierarchy.

In Chapter 4, we introduce a formal description of a component in terms of a signature, which describes the static information of a component, and a time-slot language formed over the signature, which describes the behaviour of a component. The language part of a component comprises a set of runs. Each run records a finite set of observable services occurred over an
infinite sequence of time slots. In component-based software design, we are interested in the intended behaviour rather than all possible behaviours of a component. Therefore, we advocate restricting an appropriate subset of all possible runs formed over a given signature. The resulting set of runs comprises the so-called time-slot language. An enhanced regular language – component protocol, is proposed to serve as a syntactic notation for component behaviours. By applying the initial algebra approach, component protocol expressions can be interpreted in terms of time-slot languages. The ideas are applied to a case study example from the retail industry.

Chapter 5 discusses the architectural properties of CBS, especially for the interoperability of components. The notions of well-definedness and well-behavedness of the component protocol are presented to facilitate the evaluations of connectors, in terms of the deadlock-freedom of the glue and the loyalty of roles with the glue. The nature of language-based description in the operational semantic domain enables us to define the behaviour refinement relationship by the subset relationship in set theory. This relationship assists in determining the compatibility and substitutability properties of components.

By using the unit-testing approach, Chapter 6 aims to evaluate the well-formedness of each entity prior to integration testing for the system under question. In this context, our work concentrates on identifying the well-formedness property of components as well as connectors before wiring them together for interoperability checking. A well-formed component requires that the service unmatchability holds in its signature, and that its behaviour respects the design from all individual parts. Applying hierarchical design to components, the well-formed composite components have, in addition, to keep certain consistency between the descriptions at different hierarchical levels. Further, the well-formedness of connector requires the deadlock-free interaction workflow specified in its glue and requires all its roles are in loyalty to the glue.

Chapter 7 contains evaluations, overall summaries and future developments and directions of the works of this thesis.
Chapter 2

Approaches to Modelling Component-based Systems

In the Oxford Dictionary of Computing [Oxf96], a model is defined as a representation of something. The representation may be physical or abstract and may be restricted to certain properties of the referent. Since we are interested in software systems, the referent in this case is a software system. The process of modelling involves deciding which assumptions, simplifications, idealisations or abstractions to make in representing a software system and expressing its properties. In fact, a model is an abstraction at some given level. It captures the essential aspects of a system and ignores others. As Booch, Rumbaugh and Jacobson [BRJ99] state that a model is characterised as being a 'good' model if it includes those elements that are broadly positive effect and omits those minor elements that are not relevant, at least not at the given level of abstraction.

In the context of CBS design, we are interested in models that facilitate the communication of ideas but also the specification of components for rigorous analysis and reasoning about their behaviour. In this chapter, we present an overview of the basic background concepts related to components, a brief survey of the existing component models together with a review of state-of-the-art formal approaches for modelling CBSs.

The term component seems to be overloaded in software engineering, and thus we start with a brief overview of basic component concepts that underlie the use of the term in the thesis in Section 2.1. Next, we outline the current existing approaches to software design and examine their suitability for the component-based paradigm in Section 2.2. We then turn our attention to formal methods for describing the behaviour of software systems and discuss formal approaches tailored to the specification and analysis of CBS in Section 2.3. Finally, in Section 2.4, we conclude this chapter by addressing the open issues of the current CBS modelling approaches. Those issues draw the motivation of our work.

2.1 Introduction

Unfortunately most software engineering concepts encompass various aspects and this often comes with difficulties when formulating intuitive definitions. The notion of a component is by its
very nature difficult to define precisely. In fact, there are noticeable variations in definitions of a component within the CBSE community. The root of this variation seems to be that components can be seen from different angles: a component as a unit of implementation and a component as an architectural abstraction. Our understanding of the term is consistent with the overall thrust of these definitions, though we are mostly interested in components at the specification level and thus inclined to view components as design abstractions. A number of experts in the field have defined a component by enumerating its characteristic properties. According to Szyperski [Szy97, SGM02] the properties that characterise a component are that it:

1. is a unit of independent deployment
2. is a unit of third-party composition
3. has no (externally) observable state

The first property implies that a component encapsulates its constituent features and is well separated from its environment and other components. Third-party composition entails that a component should have a precise description of the services it provides and those it requires. Being an encapsulated software unit, it interacts with its environment or other components by means of well-defined interfaces. Thus, a clear specification of its provided/requires dependencies is essential. These two properties underline the component concept considered in this thesis.

The third property, related to the (lack of) state of a component is not equally well-received. In fact, whether a component has state or not has been a subject of heated debate, since the specification of operations in an interface requires partial knowledge of the state of the components. As we understand it, Szyperski is talking about a component as a template, thus requiring no observable state, whereas instances of that component template are indeed associated with state during their lifecycle. It is worth mentioning that this understanding of the issue is closer to Szyperski’s thinking in his recent book [SGM02] on components as compared to his first book [Szy97].

In similar vein, D’Souza and Wills [DW99] define a component as a coherent package of software implementation that:

1. can be independently developed and delivered
2. has explicit and well-specified interfaces for services it provides
3. has explicit and well-specified interfaces for services it requires from others
4. can be composed with other components, perhaps customising some of their properties, without modifying the component themselves
It can be seen that the properties identified by D'Souza and Wills can be reasonably equated to those of Szyperski and are rather close to our understanding of a component. However, with regard to the point 4 of their component definition we would like to see a more precise description of what aspects of a component are considered to be customisable and what aspects are related to components themselves.

The Software Engineering Institute (SEI) definition [BBB'00] is consistent with the above concepts, but in addition considers a component in the context of a component model. A component model in [BBB'00] is defined to be a set of standards or conventions that describe how components interact and therefore expresses global or architectural design constraints. According to SEI [BBB'00], a component is:

1. an opaque implementation of functionality
2. subject to third-party composition
3. conformant with a component model

By considering a component model, this definition attempts to bring together the view of components as implementations and components as architecture abstractions. The idea is that components as implementations can be deployed and assembled into larger systems and components as architectural abstractions express design rules to which components must conform.

Consideration of a component model is also central to the definition of a component given by Heineman and Councill in [HC01]. They define a component as “… a software element that conforms to a component model, and can be independently deployed and composed without modification according to a composition standard”. It can be seen that the definition presupposes the existence of a component model and the authors discuss the relationship between a component infrastructure, components, and a component model. In particular, a component model is intended to enforce global behaviour on how components in a CBS communicate and interact with each other. A composition standard, according to [HC01], defines how components can be composed and how a component that already exists in the system can be replaced by another component with the same or upgraded functionality while ensuring that its substitution has minimal impact on the composite system. Although it is not obvious in their definition, the authors go on to state that a component must clearly define its explicit context dependencies whether these are on the operating system, on another component (or some other software element) or on performance and hardware related features.

The Unified Modelling Language (UML) specification [OMG03] defines a component as “[…] a physical, deployable and replaceable part of a system that encapsulates implementation and provides the realisation of a set of interfaces”. Although the UML component concept takes a
black box view and encompasses the notion of provided interfaces, the spirit of the definition is somewhat different to those discussed so far. It defers from the ‘give-and-take’ concept of provides/requires interfaces and places emphasis on software elements that reside on the component such as binary files, libraries, executables, scripts or command files, which implement the services provided by the component. It is important to note that in the most recent update to the specification of the language, UML2.0 [OMG04a], a component can also be viewed at the specification level and is additionally understood as a modular part of the system that encapsulates its contents with well-defined interfaces. This view is discussed in greater detail in Section 2.2.2.

[Si98] discuss components as viewed in standard UML component diagrams and states that “[…] components represent distributable physical units, including source code, object code and executable code”. Likewise, [BRJ99] considers a component to be “[…] a physical implementation of a set of other logical elements such as classes and collaborations”. These definitions are not in conflict with our view of a component. Being primarily interested in specification and analysis of CBS though, we regard a component as a more coarse-grained concept than just a collection of libraries and other executable files. While libraries allow for low-level code reuse they are not useful for managing similarities and differences in the structure and behaviour of software systems. We return to the issue of components in UML in Section 2.2.2, where we discuss the application of UML in modelling component-based software.

Current component technologies such as Microsoft COM/.NET [Cor], Sun’s Enterprise JavaBeans (EJB) [Mic03] and the CORBA Component Model (CCM) [OMG02] support the assembly of systems from pre-compiled parts. However, components in these technologies are not adequately treated at the specification level. As a result, there is little support for reasoning about the final system until the parts have been composed, executed and tested as a whole. This is partly due to lack of behavioural information about the individual components which could be subsequently used to guide their composition. Today’s component technologies offer an infrastructure of services to create, assemble and execute components. Their focus is on providing mechanisms for solving problems related to component interoperability. The specification of components is restricted to an informal description of the services provided, together with the signature of the methods that invoke those services. This signature-based form of specification, although popular, does not provide the necessary information for reasoning about component dependencies beyond the compatibility of their interface.

It is not feasible to manage change, replacement and the composition of component successfully if components have not been specified properly. A predicate for successful component trading, as asserted in [SGM02, BBB'00], is the ability to precisely describe the behaviour of components at their interfaces, what is often referred to as the observable behaviour of a component. In order to
manage dependencies between interacting components [KC00], emphasis should be placed during design on specification and analysis of the observable behaviour of components. In this context, a dependency is understood as the reliance of a component on others to support a specific functionality.

Design by contract (DbC) [Mey92] is a fundamental approach to software design for improved correctness and robustness of software systems. The idea is that an interface specification contains assertions which define a contract between the client and the supplier of a service provided by that interface. Three different kinds of assertions are used: pre-condition, post-condition and invariants. A pre-condition states the properties that must hold before an operation is called. A post-condition describes the properties that are guaranteed to hold after the operation is executed. An invariant states a condition that must be preserved by all operations of a certain instance.

The concept of a contract in DbC, which has been considered in view of object-oriented development by Meyer [Mey97], is restricted to explicitly stating a supplier's 'offer' to potential clients but the supplier's 'needs' are hidden in the implementation. Components, unlike objects, are a unit of composition and thus it is important to specify what a component needs in order to deliver the service it provides. Approaches to adopting DbC for component-based design include [Rau02] which proposes signed contracts in the form of templates as a means of mapping the services required by a component to service provided by other components.

It transpires that the main challenge in CBS design is to analyse and specify dependencies between components in such a way that the component can be treated as independently as possible. Additional behavioural information is needed on interfaces in order to be able to describe the respective component contracts precisely. Various approaches and methodologies have been devised with varying levels of precision, ranging from pure diagrammatic and architectural definition language (ADL)-based component models to formal component models.

2.2 Current Software Component Models

Modelling a software system is a central part of the overall activities involved in producing efficient software systems. Efficient, in the wider sense of possessing all '-ilities' of software engineering state of practice. A model of the intended system provides a medium for communicating the desired structure and behaviour as it allows for the visualisation of the system's architecture. Further, a model is typically a simplification of reality and in that respect it abstracts from details for different purposes and different audiences, and helps focus the discussion on the key aspects of the software system under development.
In CBSE, models are usually abstract and are increasingly represented using diagrammatic notations with natural language semantics. The prevalent component diagrammatic descriptive model nowadays is defined in the UML 2.0 [OMG04a] which is becoming an industrially well-known standard. Other well-known component models exist, such as Koala model [vOvdL'00, vO02b] widely used in Philips Electronics [Phi] to design real-time and embedded systems; and the SOFA model [PBJ98] with the original aim of providing Czech banks a practical CBS framework which can clearly separate the business logics of an application from any concrete communication middleware. In this section, we briefly introduce these software component models and highlight their strengths and weaknesses in software engineering practices.

2.2.1 The Koala Model for Components

The consumer electronics (CE) industry has been keen to adopt a component-based approach to developing embedded software for its products. The size and complexity of the software in individual products are increasing rapidly, especially in recent years, so that software is expected to combine previously unrelated functions [KFP'04]. CE products are no longer isolated entities but have become members of complex product-family structures. The structures exhibit a large degree of diversity in product features. Moreover, today’s dynamic CE market is anxious to capitalise on advances in hardware technology in order to provide new product features. Strong competition in the market also dictates that development time (and effort) must be significantly reduced.

The above factors suggest that the diversity and complexity of embedded software, at an increasing product-development speed, cannot be handled without employing reusable components. The Koala component model and language [vOvdL'00, vO02b] was developed in response to this challenge and is currently being used in Philips Electronics [Phi] for developing software embedded in audio-video product families. Its primary objective is to facilitate building a large population of products [vO02a] by reinforcing connection technology between components.

In the Koala model, an interface is a small set of semantically related functions as in COM and EJB. Components access all external functionalities through required interfaces and offer functionalities through provided interfaces. This approach provides the architects with a clear view of the system’s resource use.

Koala’s graphical representation strongly resembles hardware design, in this reuse has taken place for some years. Components are rendered as rectangles, reminiscent of IC chips, configurations look like electronic circuits, and interfaces are drawn as small squares containing triangles, as if pins of the chip. Triangles designate the direction of function calls; when the tip of
the triangle is outwards of the component it is associated with outgoing calls, while a base outwards of the component is associated with incoming calls. In other words, a small square box whose triangle has its base outwards of the component denotes a provided interface while one whose triangle has its tip outward of the component is used to denote a required interface.

For example, in a TV, a tuner is a hardware device that accepts an antenna signal as input, filters a particular station, and outputs the signal at an intermediate frequency. This signal is fed to a high-end output processor (HOP) that drives the TV screen. Each of these devices is controlled by a software driver that can access hardware through a serial \textit{I2C} bus. Therefore each driver requires an \textit{I2C} interface, which must be bound to an \textit{I2C} service in a configuration.

As can be seen from the TV software platform depicted in Figure 2.1, when components are interconnected to form a product, the resulting configuration looks like an electronic circuit. People in the CE domain will readily understand such pictures.

Koala's graphical notation was slightly extended by van Ommering [vO03, vO04] where connections between provided and required interfaces carry an additional meaning. Vertical connections represent basic control activities, whereas horizontal connections represent coordination of downstream devices. Upstream devices are those closer to the source of the signal and downstream devices are those further away from it (see [vO03, vO04] for details). Many control tasks in a TV, for instance, coordinate devices in the same signal path. This implies a strong dependency upon the topology of the hardware, which is subject to change in new products but also for the same product during its development. In light of such problems the approach taken in [vO03, vO04] is to allow components to communicate using horizontal communication interfaces in addition to the vertical control interfaces. The idea is that components controlling individual hardware devices have input and output ports that mirror the hardware and communicate through those.
In the configuration of Figure 2.1, the horizontal connection between interfaces of \textit{CTunerDriver} and \textit{CHipDriver} allow for direct communication between devices along the same signal path while vertical connections are used for connecting components \textit{CTunerDriver} and \textit{CHipDriver}, which are controlled by component \textit{CFrontEnd}.

A Koala \textit{interface} is defined by a simple interface definition language (IDL), which essentially lists the function prototyping in C syntax. For instance, this is the \textit{ITuner} interface definition.

\begin{verbatim}
interface ITuner
{
    void SetFrequency(int f);
    int GetFrequency(void);
}
\end{verbatim}

\textbf{Figure 2.2 The Koala Interface Description}

\textit{ITuner} is an example of a specific interface type, which will be provided or required by only a few different components. The \textit{IInit} interface, also presents in Figure 2.1, exemplifies a more generic interface: It contains functions for initialising a component, and most components will provide this interface.

A Koala interface is immutable in the sense that it cannot be changed once it has been published. In case the interface needs to be changed, to handle diversity for instance, it is possible to create a new interface in its place so long as the new interface is backward compatible, i.e., contains all functions of the preceding interface plus some additional features.

To maximise the potential for reuse, configuration-specific information is moved out of the component by parameterisation, in general. The services the component requires from the configuration, are requested through the standard interface notion, i.e., through required interfaces which are also called diversity interface in this case.

A Koala component is described in a component description language (CDL). The tuner driver is defined as follows.

\begin{verbatim}
component CTunerDriver
{
    provides ITuner ptun;
    IInit pini;
    requires II2c ri2c;
}
\end{verbatim}

\textbf{Figure 2.3 The Koala Component Description}

Each interface is labelled with two names – for example, \textit{ITuner} – is the interface type name. This globally unique name refers to a particular description in our interface repository. The other name – for example, \textit{ptun} – is a local name to refer to the particular interface instance. This
convention allows component designers to have two interfaces on the border of a component with
the same interface type – for instance, a volume control for the speakers and one for the
headphones – as long as the instance names are different.

A Koala configuration is actually a composed component which comprises a set of components
connected together to form a product. Hence it is not convenient to define system configurations
directly in terms of basic components. Therefore, an architectural description language (ADL),
Darwin\(^1\) [MDE'95], is employed, to provide an explicit hierarchical structure in terms of
components with provides and requires interfaces, and connectors. Here is an incomplete
definition of the TV platform in Figure 2.1.

```plaintext
component CTvPlatform {
    provides IProgram pprg;
    requires II2c slow, fast;
    contains
        component CFrontEnd cfre;
        component CTunerDriver ctun;
    connects
        pprg = cfre.pprg;
        cfre.rtun = ctun.ptun;
        ctun.ri2c = fast;
}
```

Figure 2.4 The Koala Configuration Description

Each composed component has a type name – for example, \textit{CTunerDriver} – and an instance
name – for example, \textit{ctun}. The globally unique type name refers to the reusable component
definition in the component repository. The instance name is local to the configuration.

Koala components are constructed as component definition files and deposited into a web-based
repository called \textit{KoalaModel Workspace} (that is in fact a flat file system) in the design phase.
Koala components can also be retrieved from \textit{KoalaModel Workspace} and composed with other
components to a composite component that is then deposited back to the repository.

To facilitate reuse, the Koala model takes the binding knowledge out of the components. At
configuration time, components are coordinated by a software control layer, called connectors, as
the only components with knowledge of the system topology (a by-product of this feature is
otherwise reduced the portability of connectors). Whereas components are concerned with how to
achieve its computation, connectors determine how that computation is combined with others at
the product level. Changes to the system topology are therefore isolated in components. This
significantly improves the reusability and simplicity of components.

\(^{1}\) It is an enhanced version of Darwin, which further supports the easy addition of glue code between components
and a diversity parameter mechanism.
Two kinds of connectors are defined in the Koala model: static-binding connector and dynamic-binding connector. The static-binding connector, such as a module (an interface-less component), is used for those connections already known at configuration time. Static bindings are performed via the glue code, which is a set of naming conventions and renaming macros that bind functions of a required interface to those of the corresponding provided interface. Dynamic-binding connectors are designed to meet the needs from high-end products, which allow for the upgrading of components in the near future. For instance, Koala uses a switch, in combination with diversity interfaces, for dynamic (runtime) binding. [vO03, vO04] provides a richer set of connectors to handle structural diversity within a configuration in terms of forks, switches, matrixes and source selection connectors.

![Figure 2.5 The Use of Koala Switch](image)

Figure 2.5 demonstrates a typical use of a switch when building a TV platform. Prior to describing the operation of switch, the following paragraphs first introduce product requirements that have to be satisfied.

The first task concerns tuning. When the frequency of a tuner is changed, the tuner temporarily produces noise. This leads to undesired artefacts on screen and in the speakers. Therefore, the screen should be blanked and the sound muted before the frequency is changed, and they should be restored only after the tuner’s output is stable again. Activities such as Teletext decoding should also be stopped during the turning activity.

The presence of a switch adds complexity. The blanking, muting must only occur for the devices that are actually connected to the tuner being tuned. For devices connected to the other tuner, no action is required.

Going back to Figure 2.5, the front end connects to the first or second tuner driver depending on the switch’s setting. Suppose the switch is currently in position CTunerDriver, we shall discuss
the result of two actions. The first is a Tune operation on CTunerDriver. Since the switch is in position CTunerDriver, it passes the drop request of CTunerDriver to CFrontEnd and returns its answer to CTunerDriver. The restore command of CTunerDriver is handled similarly. The second action is a Tune operation on CTuner2Driver. Since the switch does not connect CTuner2Driver to component CFrontEnd, it can handle the drop request and the restore by itself. In fact, switch serves as an output stub here, answering any drop request with true, and returning any restore command immediately.

The switch has an extra complexity: it can also change positions. Before it does so, it must request permission from connecting devices using the drop request protocol as defined above. The top-level control software calls the Switch(i) command to select input i. The switch requests component CFrontEnd for permission to drop the signal. In this case, CFrontEnd accepts the request immediately. The switch can then change position, after which it sends a restore command to CFrontEnd to indicate that the signal is valid again (assuming that CTunerDriver’s output is still valid). The subtleties of the switch protocol can be further explored when component CFrontEnd delays the approval of the drop request, and the switch permits multiple interfaces to be switched simultaneously and between more than two targets.

The above mentioned protocols of drop request and switch manipulation are in-depth discussed in the horizontal communication protocol proposed by van Ommering [vO03, vO04], it aims to standardise the way of describing device communications in the TV platform. The author illustrated component interactions in a syntactic subset of Message Sequence Charts (MSC) of the ITU standard [ITU00]. For example, two scenarios for switch are illustrated in Figure 2.6.

A scenario most generally, is understood as a unit of behaviour that focuses on the observable exchange of information between participating entities with the objective of performing a specific task. Scenario-based descriptions are a common mechanism for modelling systems and are popularly used in design where the precise inter-process communication must be set up according
to specified protocols. The wide acceptance of MSCs leads to their standardisation by the telecommunication industry with the ITU standard. Yet, their interpretation can be ambiguous; for instance, does a MSC describe how the system will always behave or does it give a possible behaviour of the system? According to the ITU standard MSCs only do the later. But then, virtually nothing can be said in MSCs about what the system will do when the described scenario actually occurs. Being motivated by the MSC's limitation in expressing necessity, Live Sequence Charts (LSC) introduced by Damm and Harel [DH01], regarded as an extension to MSCs, address this issue as they can explicitly distinguish between mandatory, possible and forbidden behaviour.

The ambiguous semantics of MSC may also encounter the race condition issue. For example, the \textit{DropReq} (3) and \textit{Restore} (4) operations as experienced by component \textit{Switch} in Figure 2.6 are in a race condition. The vertical dimension of the diagram suggests that \textit{Tune(f)} (1) happens first, and then \textit{DropReq} (2), then \textit{DropReq} (3), followed by \textit{Restore} (4) and so on, finally reaches \textit{Restore} (8). However, this is not necessary the case. This is because the event of sending \textit{DropReq} (3) is ordered to occur before the receiving of \textit{Restore} (4) (the ordering is imposed along the lifetime of component \textit{Switch}), but the sending of \textit{DropReq} (3) and the sending of \textit{Restore} (4) are not ordered (they belong to different lifelines). Therefore, \textit{Restore} (4) may be sent before \textit{DropReq} (3) or even at the same time as \textit{DropReq} (3) (since \textit{CTunerDriver} who is responsible for sending \textit{Restore} (4) after \textit{DropReq} (2) does not know whether \textit{Switch} has already sent \textit{DropReq} (3) or not). As a result, there is no way to ensure that \textit{DropReq} (3) will occur before \textit{Restore} (4). Such a limitation results from the fact that the partial order induced by a MSC imposes an ordering on events appearing along a particular lifeline, but events on distinct lifelines can only be ordered as a consequence of inter-lifeline communication. This subtlety in the semantics often gives rise to inconsistencies between the specified ordering of events and the order in which events can occur in practice.

In summary, being inspired by COM/ActiveX [Cor], Koala offers a component technology that stimulates the development of largely independent components, including their evolution and code generation. Koala was established to achieve a strict separation between component and configuration development. Component builders make no assumptions about the configuration in which their component is to be used and they construct components and store them in a global component repository. Similarly, configuration designers are not permitted to change the internal of a component to suit their configurations; they retrieve desired components from the repository, design the system configuration by composing components with connectors, and finally compile the system into a programming language, such as C.

Within this architecture, Koala components basically define its interfaces in an interface definition language (IDL) and itself in a component description language (CDL), and the product-level compositions of components are designed by Darwin language. However, the notion of IDL,
CDL and Darwin are semantically thin in that they comprise no more than the signatures of the operations the component offers to and requires from its environment and their bindings. These notations, although popular, yield little information of value in a more rigorous approach to the composition of products from components. Additional information about components, possibly in terms of the ordering of associated signalled events, is captured as scenarios in Message Sequence Charts (MSC). However, the ambiguous semantic issues of MSCs highlight the fact that, although useful for informal documentation and triggering discussion during design, the Koala component model lacks a formal operational semantics to underpin rigorous analysis and formal verification on its model.

2.2.2 The UML Model for Components

UML has been standardised by the Object Management Group (OMG) since 1997, in a series of specification documents starting from UML 1.1 [OMG97] to UML 1.3 [OMG99] to UML 1.4 [OMG01] to UML 1.5 [OMG03b] and most recently with the adoption of the final specification for UML 2.0 [OMG04a]. All previous versions include minor updates and refinements, with the exception perhaps of the move from UML 1.4 to UML 1.5 which was concerned with the inclusion of actions in an attempt to accommodate the idea of an action language [WKC+03] leading to executable UML models [MB02, RFW+04]. The move to UML 2.0 includes a significant update to preceding versions and offers interesting perspectives with regard to specification and analysis of CBS. We discuss these features explicitly in the sequel. In shorthand, we shall refer to the previous version of UML as UML 1.x.

In an attempt to support the notion of components throughout the modelling lifecycle, UML 2.0 adopts a component concept at the specification level on top of the implementation focus of UML 1.x. A component in UML 2.0 (see Chapter 8 in [OMG04a]) is understood as a modular part of a system that encapsulates its contents with well-defined interfaces, and is replaceable within its environment. It has one or more provided and required interfaces (potentially exposed via ports) and its internals are hidden and inaccessible other than as provided through its interface. A component may be dependent on its environment and these dependencies are expressed in terms of its required interfaces. The challenge is to analyse and specify dependencies in such a way that the component can be treated as independently as possible. In this respect, we believe that a formal description of component contracts is needed as it can provide the necessary level of precision.

As a result of their interface notion, components are encapsulated and can be reused and replaced by connecting them together via matching provided and required interfaces. A component is given a semantics in UML 2.0 in terms of a formal contract of the services it
provides to its clients and those it requires from other components through its provided and required interfaces. It is noteworthy that the UML 2.0 semantics hints towards a formal contract, but this is not provided or prescribed in the specification.

Graphically, a component is represented in UML 2.0 as a Classifier (for example class) rectangle with the standard stereotype ‹‹component››. Optionally, a component icon can be displayed on its top right-hand corner – this is a rectangle with two small rectangles protruding from its left-hand side just as in component and deployment diagrams of UML 1.x. The interfaces of the component are represented as symbols sticking out of the rectangle; provided interfaces are denoted by a ‘ball’ or ‘lollipop’, while a ‘socket’ is used to denote required interfaces. Figure 2.7 depicts an Order component in UML 2.0 notation with a provided interface OrderEntry, and a required interface OrderableItem.

A component can be embedded into any environment (or system) that satisfies the constraint expressed by the provided and required interfaces of the component. This is done by connecting (wiring, in UML dialect) components via their provided and required interfaces. Interfaces allow for the specification of both structural (for instance, attributes, association ends) and behavioural features (for example operations and events). The provided and required interfaces may optionally be organised through ports which enable the definition of named sets of provided and required interfaces.

Putting components together to form a system is structurally defined in UML 2.0 by using dependencies between interfaces. This is typically done in structure diagrams. These diagrams show components and connections between them in terms of marking provided and required interfaces. (Note that interface compatibility is not defined in UML, and rightly so since it depends on the underlying interface model being used.)

An assembly connector is used for the matching. This is a connector between two components that defines that one component provides services the other component requires. In this case, an assembly connector is used from a required interface or port of one component to the provided interface or port of the other component.

The semantics of the assembly connector, given in UML 2.0 is that signals or operation calls or events originate in the required interface and are delivered to the provided interface, by travelling along an instance of the connector.
The structure diagram of Figure 2.8 shows a component architecture where an assembly connector is used to connect the required interface `OrderableItem` of `Order` component to the provided interface `OrderableItem` of the `Product` component.

Furthermore, UML 2.0 introduces composite structure diagrams which can be used when more detail is required about the internal structure of a component. The internal structure in UML 2.0 refers to interconnected elements within the containing classifier that collaborate to achieve some common objectives. This is relevant to CBS when considering compound components - components which contain other components whose collaborations provide the overall functionality, as promised in the compound component’s contract.

In addition to the assembly connector, composite structure diagrams use a delegation connector which links the external contract of a component to the internal realisation of that behaviour by the contained components. The delegation connector is used to model the decomposition of behaviour in the sense that behaviour that is available on a component may not actually be realised by that component itself, but by another component that has compatible capabilities. The use of a delegation connector represents the forwarding of events (operation calls, signals) from one interface to the other for actual handing.
Figure 2.9 shows a Store component which relies on the collaboration between the contained components Order, Product and Customer for fulfilling its component contract in terms of its provided interface OrderEntry and its required interface Account.

UML 2.0 has been significantly improved with respect to modelling of behaviour and composition of CBS. In general, like MSCs, sequence diagrams are used to illustrate the global interactions between actors and components in a CBS in the runtime, whereas state diagrams are mainly used to encapsulate the local activities of modelled components. The improvements of these two diagrams in UML 2.0 are discussed accordingly in the following.

Compared with those of UML 1.x, sequence diagrams in UML 2.0 have been considerably enriched its expressive power to capture and simulate system interaction logics. In brief, a sequence diagram can contain sub-interactions, called interaction fragments, which can be combined through interaction operators. Such as seq operator for describing sequential behaviour, alt for alternative behaviour, par for parallel behaviour, neg for forbidden behaviour, loop for iteration and so forth. Hence, the semantics of the resulting sequence diagram depends upon the operator used and is described informally in the UML 2.0 superstructure specification document (see [OMG04a], Chapter 14).

However, the lack of precise behavioural semantics in the specification for interaction operators sometimes allows for varying interpretations of the behaviour prescribed in the resulting interaction fragments. For instance, the par operator describes a set of concurrent event occurrences. The informal semantics of par hints towards considering all possible interleavings in the resulting sequences of events (see [OMG04a], page 403, 410). According to this semantics, it is not possible to differentiate between the behaviour described in the following sequence diagrams.

![Sequence Diagrams](image)
Following an interleaving interpretation, the sequences of event occurrences in the diagram v1, pictured in Figure 2.10 (i), says that either $b_1$ occurs before $b_2$, or, $b_2$ occurs before $b_1$. This is precisely the behaviour described in Figure 2.10 (ii) even though the alt operator is used this time. This may not be an issue when all we are interested in is that both $b_1$ and $b_2$ have occurred at the end of the interaction – this implies an implicit synchronisation point at the end of the diagram v1. If, however, we want to include the case that $b_2$ and $b_1$ occur at exactly the same time then, diagram v2 no longer describes this intended behaviour. The situation gets even more complicated if we were to insist on $b_1$ and $b_2$ occurring at exactly the same time (for example, consider $CTunerDriver$ blocking the audio output on the speaker and the video output on the screen before changing the frequency). Hence, the use of UML models that have informal semantics can make it difficult for consistency checking and the quality control of specifications. Generally speaking, this limitation originates from the objective of UML. We will return this point in the late of this section.

Another major change in UML 2.0 has to do with state diagrams. We have seen that UML features State Machine diagrams, essentially, an object-based variant of the well-known Harel statecharts [Har87], for modelling behaviour through finite state transition systems. A state machine describes the behaviour of a part of the system observed in terms of events accepted and actions executed resulting in a change of state. Such state machines are termed behavioural state machines in UML 2.0. In addition, UML 2.0 introduces protocol state machines for expressing the usage protocol of a part of the system. This enhancement can prove useful in describing the behaviour of components at their interfaces.

A protocol state machine (PSM) specifies which operations of a Classifier (typically, a component in this context) can be called in which state and under which condition. Thus, it can be used to specify the allowed sequences of events on an interface.

The states of a PSM present an external view of the Classifier that is exposed to its clients. The transitions of a PSM specify the legal changes between states and, in contrast to the behavioural state machine, cannot have associated actions. PSM transitions carry the following information: a pre-condition, a trigger and a post-condition. The pre-condition (or guard in this context) specifies the condition that must be true before triggering the transition. The post-condition specifies the condition that should hold once the transition is triggered. Either or both can be omitted.

The PSM modelling construct can be useful for component-based design considering that a PSM can be attached to each interface. Since a PSM expresses the legal transitions the interface can trigger, it may be used to enforce legal usage scenarios for the component on that interface. Further, there may be some potential for determining compatible interfaces. UML 2.0 explicitly considers a notion of conformance of PSM in terms of the $ProtocolConformance$ model element.
The semantics of this relationship is limited to declaring that a behavioural state machine complies with the structure and constraints on the PSM. Both state machines refer to the same Classifier, and the behavioural state machine is understood to implement the PSM. Conformance is also defined between a specific PSM and a general PSM, in which case the former is understood as a specification of the latter.

Interestingly, one of the constraints specified on PSMs (see [OMG04a], page 584) states that if two interfaces are connected, then the PSM of the required interface must be conformant to the PSM of the provided interfaces. This is certainly in the spirit of a component-based approach, but the lack of a precise semantics for protocol conformance hinders the use of PSMs for more rigorous reasoning on the compatibility relation between the corresponding interfaces of components.

The main problem with designing state machines in UML 1.x is that it lacks a precise and formal foundation for specifying transition guards or transition activities, for which it resorts to semantic loopholes in the form of ‘uninterpreted’ expressions. The Action Semantics, currently being standardised in UML 2.0, aims at filling this gap by providing both a meta-model integrated into the UML meta-model, and a model of execution for these statements. The adoption of the precise Action Semantics for UML by OMG supports the viability of eXecutable UML, which is one of the cornerstones, on which rests the OMG’s new initiative, Model-Driven Architecture (MDA).

The Action Semantics provides modellers with a complete, software-independent specification for actions in their models. The goal is to make UML modelling executable modelling, i.e., to allow the designer to test and verify early and to generate 100% of the code if desired. It builds on the foundations of existing industrial practices such as SDL, Kennedy Carter [WKC’03] or BridgePoint [Pro] action languages.

Action Semantics seems to become a breakthrough for tool vendors to develop highly automated and optimised code generators for UML CASE tools with an executable action specification language. However, Action Semantics does not help UML to gain formal semantics. That is because the Action Semantics (even UML itself) is (semi-) formally defined using a meta-model. The meta-model itself is expressed using constructs in the UML, thus implying a meta-circular interpreter approach; the language itself is defined in terms of itself. In fact, a small subset of UML is used in defining the meta-model.

The UML meta-model is a logical model and not a physical or implementation model. As such, the meta-model emphasises declarative semantics and abstracts away from implementation details. Various UML tool vendors may implement the logical model in different ways, thereby allowing
for the custom tuning of their implementations, for reliability and/or performance so long as these implementations conform to the semantics of the meta-model.

The UML meta-model [OMG03b] is described in a semi-formal manner according to the following views.

- **Abstract Syntax**: provided as a model described using a UML class diagram and a supporting natural language description;
- **Well-formedness rules**: a list of constraints on elements expressed in natural language (text description) and the Object Constraint Language (OCL) (see [WK99] for an introduction to OCL and [OMG04a] for its latest specification);
- **Semantics**: described primarily in natural language, but may include some additional notation based on the part of the model being described.

The complexity of the UML is managed by decomposing it into three main local packages, namely the **Foundation**, **Behavioural Elements** and **Model Management** packages. The idea is that these packages group meta-classes that show strong cohesion with each other and loose coupling with meta-classes in other packages. Each package is briefly described below. More details can be found in [OMG03b].

The **Foundation** package defines the static structure of UML and contains three sub-packages, namely **Core**, **Extension Mechanisms** and **Data Types**, for describing the main constructs in UML, the mechanisms for customising and extending these constructs, and the basic data structures for the language. The **Behavioural Elements** package defines the dynamic structure of UML in that it specifies the basic concepts required for the behavioural elements of the language. The **Model Management** package defines packages, models and sub-systems, which serve as grouping units for UML model elements.

The three top-level packages of the UML meta-model, together with their sub-packages, are shown in Figure 2.11 which has been taken from the specification documents of the UML (see [OMG03b]) issued by OMG.

It can be seen from this brief presentation of the UML semantic model that it is a combination of graphical notation, natural language (English), and formal language (OCL). There are inevitable theoretical limits to what can be expressed about a meta-model using the meta-model itself. This is counterbalanced though by the fact that a satisfactory trade-off between expressiveness and readability can be achieved using such a combination. In other words, the primary objective for UML seems to be an accessible and easily comprehended modelling language, even if this entails (some) sacrifice of formal rigor. As a result, a wide circle of developers can quickly get a reasonable understanding of UML as the language is described at
present. Considering that a standard interpretation of UML constructs and resulting diagrams is needed for applying the language in more rigorous approaches to software engineering however, the question arises as to whether they (developers) understand in the same way.

Further, the UML meta-model seems to focus on defining the relationship between groups of UML concepts, in the form of packages and their sub-packages, rather giving a semantics to the various diagrams and the graphical constructs used therein. This is partly done in the specification documents using natural language (English text description). Still, this does not guarantee the unambiguous interpretation of UML diagrams in some cases.

UML also has an extension concept called profiles. Profiles are used to define meta-classes in UML to tailor a UML model to a specific platform. UML Profile for Real-Time [SR98] is a UML profile based on ROOM modelling language [Sel93] and it targets modelling of real-time and embedded applications. Curiously, this profile matches closely the concepts introduced in most CBS models.

In principle, the Profile for Real-Time supports two views of a model: the structure of the system and its behaviour. In the system, a capsule (a component) communicates with other capsules through ports. A port is an access point for the event-based communication. The behaviour of a capsule is described by a statechart, or (for composed capsules) by a statechart combined with sub-capsules. The ports of a composed capsule can be delegated to ports of sub-
capsules and the statechart of the capsule can control creation and destruction of sub-capsules. In this sense, it models dynamic changes of capsule architecture.

Each port plays a specific role in a protocol. A protocol is a specification of desired behaviour, that can take place over (typically two) connected ports representing a contractual agreement. Each role lists a set of sent and received signals. Optionally, a protocol role can specify the sequences of signals by using a statechart and/or a set of prototypical sequence diagrams. The sequence diagrams must conform to the statechart. In [Sel98], the role substitutability is shortly outlined with the introduction of multi-role protocols. For a runtime verification, the exact definition of the event execution model in [Sel93] allows the use of the behaviour specification for prototyping the software system.

To conclude, in this section we have outlined the UML 2.0 component model and its behaviour specifications, thereafter we discussed the core set of concepts and constructs that underlie UML, and given a short introduction of UML’s profile for real-time. UML 2.0 components are represented in UML 2.0 notation that is used as a kind of ADL. There is no repository in UML 2.0. In the design phase, components can be constructed in a visual builder tool such as Rational Rose [Ibm]. Components are composed by UML connectors: delegation connectors and assembly connectors. In the deployment phase, no new composition is possible, so there is no assembler, the implementation of components and connectors can be done in various programming languages, and so the runtime environment in the deployment phase is that for the chosen programming language platform.

We have seen that UML is essentially a diagrammatic language which provides a wide range of notations and support for techniques that can be used to capture different aspects of a software system. By and large, it has become the standard practice for software modelling. However, the fundamental problem of UML is it still lacks a precise (i.e., formal) semantics domain to underpin its model. This limitation makes software engineers difficult to uncover design faults in UML models.

2.2.3 The SOFA Model for Components

The SOFA (SOFTware Appliances) project [PBJ98] targets the issue of composing applications from components which can be deployed over a network. In the SOFA component model, an application is viewed as a hierarchy of nested software components. Analogously with the classical concept of object being an instance of a class, SOFA introduces software component as an instance of a component template. In principle, ‘template’ can be interpreted as ‘component type’.
A template $T$ is a pair $<F, A>$ where $F$ is a template frame, and $A$ is a template architecture. The frame $F$ defines the set of individual interfaces any component which is an instance of $T$ will possess. The interfaces are instances of interface types. In $F$, an interface can be instantiated as a provides-interface or a requires-interface. Basically, the frame $F$ reflects the black-box view on $T$. To support versioning, the frame $F$ can be implemented by more than one architecture. An architecture $A$ describes the structure of an implementation version of $F$ by

1. instantiating direct subcomponents of $A$ (those on the adjacent level of component nesting, and by

2. specifying the subcomponents' interconnections via interface ties using connectors (will be introduced shortly). Basically, the architecture $A$ reflects a particular grey-box view on the template $T$. The ties itself contain a specification of connector type (see Section 2.1.2) to be used for the tie.

Basically, the architecture $A$ reflects a particular grey-box view on the template $T$.

There are four kinds of interface ties:

- **binding** of a requires-interface to a provides-interface between two subcomponents
- **delegating** from a provides-interface of $F$ to a subcomponent's provides-interface
- **subsuming** from a subcomponent’s requires-interface to a requires-interface of $F$
- **exempting** an interface of a subcomponent from any ties (the interface is not employed in $A$)

An architecture can also be specified as primitive, which means that it does not contain subcomponents and its structure/implementation will be provided in an underlying implementation language, out of the scope of the component model.

For example a simple database sever depicted in Figure 2.12, is designed as a $DB$ component, an instance of a $<Database, DatabaseV2>$ template. $DB$ provides the Insert, Delete, and Query operations for inserting, removing and querying records in the database. In support of its functionality, $DB$ employs a lower-level database – $Data$ component (instance of $DBAccess$) and a $Logm$ component (instance of $LogMan$) allowing for logging. These components publish their services by means of provides-interfaces; $Data$ provides the interface access of the $IDatabaseAccess$ type, and $Logm$ provides the log interface of the $ILogging$ type. In a similar vein, the $DB$ component provides its services via $dbSrv$, an instance of the $IDBServer$ interface type. $DB$ is composed of the $Transm$ and $Local$ subcomponents. The $Local$ component provides the $d$ and $ds$ interfaces while requiring $lg$, $da$ and $tr$ interfaces. The $dbSrv$ interface of $DB$ is
delegated to the $d$ interface of $Local$, while the $lg$ interface of $Local$ is subsumed to $dbLog$ of $Local$. The $tr$ requires-interface is bound to the $trans$ provides-interface of the $Transm$ component.

A connector is an abstraction capturing communication/interaction between components, clearly separating communication from the business logic of the components. As described in [Bal02], by the separation, connectors can address several software development issues ranging from application distribution including data transfer, conversion and support for various middleware, to interface adaptation and access coordination. Therefore, a connector implementation can be fully devised typically right before the application start, although large parts of the implementation can be created beforehand. Specifically to SOFA, a connector implementation is semi-automatically generated, since most of the connector code is generic and can be reused.

Connectors represent all communication channels between two or more components/interfaces. A connector is an instance of a connector type. The connector types are either predefined (for instance $CSProcCall$ representing a client/server RPC call as shown in Figure 2.13) or user-defined.
Chapter 2 Approaches to Modelling Component-based Systems

The role of a connector type represents an access point of a connector. Each role is supposed to be tied ("plugged") to a component interface in a connector instance. The methods provided by a role are determined after the role is tied. This results in a generic nature of roles in connector types. Similarly to the component templates, a connector type is specified as a pair of a connector frame and a connector architecture. A connector frame specifies a black-box view of the connector type specifying the roles as provides-roles or requires-roles. Roles in a frame have a cardinality, i.e., the number of entities that can be simultaneously tied to a connector instance role. There are four types of cardinality: 1, 0..1, 1..*, 0..*. A connector architecture specifies the internal structure of the connector using predefined primitive connector elements, instances of other connector types and even component instances. For example, the CSProcCall connector frame in Figure 2.13 provides multiple cRole roles and requires a single sRole role. The architecture of the connector type contains a graph of primitive connector elements such as cInterceptor, stub, and skeleton.

Based on CORBA IDL [OMG02], SOFA component definition language (CDL) is the means to specify the static aspect of components in SOFA. The syntax of CDL is provided in full in [Men98]. Here, we just demonstrate the use of CDL through an incomplete definition of the database example in Figure 2.12.

In SOFA CDL, the interface type definitions are expressed via the interface construct specifying an interface type as a set of method signatures (such as ICfgDatabase in Figure 2.14). After the necessary interface types have been specified, the black-box view of the proposed component can be designed. In CDL, this is done by means of the frame construct that encapsulates instances of
the provides-interface and requires-interfaces in the way illustrated in the Database frame in Figure 2.14.

```plaintext
interface ICfgDatabase {
    int GetTrModel();
    void SetTrModel(int model);
};

frame DatabaseBody {
    provides:
        IDBServer d;
        ICfgDatabase ds;
    requires:
        IDatabaseAccess da;
        ILogging lg;
    ITransaction tr;
};

frame Database {
    provides:
        IDBServer dbSrv;
    requires:
        IDatabaseAccess dbAcc;
        ILogging dbLog;
};

architecture DatabaseV2 implements Database {
    inst TransactionManager Transm;
    inst DatabaseBody Local;
    bind Local:tr to Transm:tran using CSProcCall;
    exempt Local:ds;
    subsume Local:lg to dbLog using CSProcCall;
    subsume Local:da to dbAcc using CSProcCall;
    delegate dbSrv to Local:d using CSProcCall;
};
```

Figure 2.14 Examples of Interface, Frame and Architecture Specification in SOFA CDL

The internals of the proposed components are specified via the CDL architecture construct. In Figure 2.14, the DatabaseV2 architecture illustrates how subcomponents are instantiated and how their ties are specified (distinguishing bind, subsume, delegate, and exempt ties). Here, two subcomponents Transm and Local are instantiated, each of them being specified at the abstraction level of its frame (the respective architectures of these subcomponents will be specified at the application assembly time). Notice how Local's interfaces are tied to the interfaces of the Database frame and to the Transm subcomponent using the predefined procedure call connector CSProcCall. Moreover, the architecture specification reveals that ds, one of the DatabaseBody's interfaces, is not bound to any subcomponent nor the DatabaseBody frame interface; this means that ds will never be engaged in component communication.

SOFA provides a formal model to specify the behaviour aspect of software components. The behaviour language abstracts from a particular component model and most of the ADL-dependent details such as name spaces, typing rules, and so on. Being focused on fundamental principles, the language is based on the abstract component – 'agent' concept, where interface ties of components become connections among agents, method calls on interfaces turn into events on connections, and a component's behaviour is modelled via the event sequences (traces) on the connections of the agent representing the component. The behaviour can be approximated and represented by regular expression-like behaviour protocol (see Section 2.3.1 for more details). Relations defined upon these protocols will allow component engineers to reason about component cooperation statically at assembly time and dynamically at runtime.
To illustrate how a component’s behaviour can be specified, assume that the component Local in Figure 2.12 is an agent and its ties are the agent’s connections. Let us limit ourselves to the set of events \{trans.Begin ↑, trans.Commit ↑, trans.Abort ↑, dbAcc.Insert ↑\}. By the following protocol, we express that agent Local does two sequential successful transactions

\[
\text{!trans.Begin ↑;!dbAcc.Insert ↑;!trans.Commit ↑;}
\]
\[
\text{!trans.Begin ↑;!dbAcc.Insert ↑;!trans.Commit ↑;}
\]

The composition operator \( \Pi_X \) is used to express the joint behaviour of components communicating via bound interfaces. For example, should there be a transaction manager component (agent) Transm with the behaviour specified as

\[
\]

communicating with Local via the trans interface, their joint behaviour can be described using the composition operator as

\[
(!\text{trans.Begin ↑;!dbAcc.Insert ↑;!trans.Commit ↑;})
\]
\[
(!\text{trans.Begin ↑;!dbAcc.Insert ↑;})
\]
\[
\]

where \( X \) is composed of the events on the trans connection, i.e., \{trans.Begin ↑, trans.Commit ↑, trans.Abort ↑\}. Since the events on trans will be exhibited as internal events in the only trace generated by this protocol, they will be prefixed by \( \tau \), so that the trace takes the form

\[
<\tau\text{trans.Begin ↑;!dbAcc.Insert ↑;},\tau\text{trans.Commit ↑;},
\]
\[
\tau\text{trans.Begin ↑;!dbAcc.Insert ↑;},\tau\text{trans.Commit ↑;}>
\]

More discussion and comment on the behaviour protocol will be given in Section 2.3.1.

A component’s lifecycle is characterised by (potentially repeated) sequence of design time and runtime phases. In a more detailed view, a design time phase composes the following design stages: development and provision, assembly, and deployment.

At the development and provision stage, a component is specified by its frame and potentially several architectures, each of them being a design version of the frame as illustrated in Figure 2.15. For instance, the frame \( F_{\text{Main}} \) is implemented by three different architectures: \( A_1, A_2, \) and \( A_3 \). While \( A_1 \) and \( A_2 \) are primitive, \( A_3 \) is composed of two subcomponents \( \text{Sub1} \) and \( \text{Sub2} \); these subcomponents are visible in \( A_3 \) solely at the level of their frames \( F_{\text{Sub1}} \) and \( F_{\text{Sub2}} \). It is important to emphasise that the actual specification of an architecture \( A \) is always based on the frames of \( A \)’s
subcomponents (and not on the architecture of those subcomponents). Reflecting top-down
design, the specification of an application is factored this way into alternating layers frame –
architecture – frame – ..., forming a tree with nodes alternately of the 'frame' and 'architecture'
types.

At the assembly stage of design time, the executable form of an application/component is
determined by selecting an implementation architecture for each frame. In Figure 2.15 it means
reducing the tree in such a way that each frame node has only one child architecture node as
presented on the right hand side of Figure 2.15. This process starts at $F_{\text{Main}}$ by choosing one
particular template $<F_{\text{Main}}, A_2>$. If $A_2$ is not primitive, the selection is applied recursively to all
frames involved in $A_2$. Consequently, the executable form of the application/component is
primarily based on all the primitive architectures involved recursively in the reduced subtree of
$A_2$.

![Figure 2.15 The Development Tree of Application and Assembled Application](image)

The executable form is completed at the deployment stage, when the component runtime
configuration is devised. Connectors allow a component to be divided into several deployment
units, which can be distributed to multiple hosts (called deployment docks). Therefore, the
deployment stage includes generating connector instances, distribution over computer network
nodes and setting parameters for component properties. At the end of the deployment, the
component is ready to run.

A connector’s lifecycle substantially differs from the component lifecycle. It can be viewed as a
sequence of the design time, instantiation time, deployment and generation time, and runtime
phases.
To define a connector type, it is necessary to specify its frame and architecture. The frame specification involves specifying all provides and requires roles and their generic parameters. As to the connector architecture, specifying compound connector architecture is similar to specifying compound component architecture – the connector internals are described in terms of its nested component and connector instances and their interconnections. A simple connector architecture is based on primitive element types. For each of the primitive element types, its description in plain English must be provided together with a definition of its mappings to concrete underlying environments (at least one mapping must be provided). Moreover, the connector internal architecture is to be described in terms of its primitive elements instances and their interconnections. Note that most of the primitive elements present in the connector architecture are usually generic (employing both interface type and property parameters).

Since connectors are inherently distributed entities, the last step of the development process of a connector type is the specification of potential distribution boundaries. This is done by dividing the connector architecture into a number of disjointed deployment units. A deployment unit is formed by the roles and those internal elements designed to share the same deployment dock.

The second stage of the connector lifecycle consists in instantiating the connector types within an application. For every connector instance, since the actual interface types of the components connected by the connector instance are known at this stage, the interface type parameters of the roles of the connector can be resolved. Also the need for certain primitive elements (such as interface adaptors) to be present within the connector architecture arises. Nevertheless, a part of the connector instance remains generic – due to the unresolved property parameters related to a future deployment of the connector.

It is natural that connectors are deployed at the same time as those components of which they represent the interactions. During the deployment phase, each of the deployment units of the connector is assigned a specific deployment dock to be deployed into. The actual deployment dock of the deployment units of the connector can be inferred from the locations of the components interconnected by the connector. However, this is true for connectors with simple architectures only. As connectors with compound architectures may contain component instances as their internal elements, the deployment of these internal components is usually specified separately.

Once the deployment of a connector is known, the implementation code of the connector is (semi-automatically) generated according to the communication primitives offered by the underlying environments of the deployment docks. Note that the generated code of the primitive elements either follows their mapping to the underlying programming environment, or it can be
null (for example there is no need for an adaptor if the interfaces of the connected components are functionally match).

To summaries, in SOFA, a component is a unit of design which has a specification and an implementation. A SOFA component is specified by its frame and architecture. The frame defines provides and requires interfaces, and properties of the component. The frame can be implemented by more than one architecture. The architecture describes the structure of the component. Semantically, components in SOFA are units of computation and control (and data) connected together in an architecture.

SOFA components are constructed as component definition files using the builder tool SOFAnode and deposited into the Template Repository, which is the repository of SOFA components. In brief, in the design phase, SOFA components are composed by method calls through connectors. SOFA components can also be retrieved from template repository and composed with other components to a composite component which is then deposited back to the repository in the design phase. In the deployment phase, no new composition is possible, so there is no assembler; SOFAnode provides the runtime environment for SOFA components.

SOFA components are statically defined in SOFA CDL that is similar to ADLs, and are dynamically defined in the behaviour protocol which is an extended regular expression language underpinned by a variant of Hoare's trace model [Hoa85]. The downside of this event-based protocol language is that it makes difficult for the users to distinguish synchronous operation calls from asynchronous events in its process expressions, but this distinction is fundamental in most of interface description languages (such as CORBA IDL [OMG02]). Moreover, its interleaving-based concurrency semantics cannot precisely distinguish nondeterminism and concurrency (see Figure 2.10) and is not powerful enough to faithfully capture the scenario where a component is serving a number of services at the same time.

In the current CBSE community, there are many existing component models designed for different purposes and different users, such as COM/.Net [Core], EJB [Mic03], CCM [OMG02] and etc. A detailed survey of state-of-the-state software component models can be found in [LW06]. In this section, we mainly discussed three well-known component models: Koala [vOvdL'00, vO02b], UML 2.0 [OMG04a] and SOFA [PBJ98]. Our description of each model mainly focuses on the aspects of component architecture, static and dynamic specifications, component composition (via connectors) and development lifecycle. In each model, we also give an example to illustrate the characteristics of component descriptions. We have shown that both Koala and UML 2.0 describe their component models in graphic notations and natural languages, but lack of formal underpinnings for rigorous analysis and reasoning about component behaviour.
relations and architectural properties (such as component compatibility and substitutability). In the following section, we will turn our focus on the formal approaches for specifying CBSs.

2.3 Formal Approaches for Components

In the previous section, we illustrated current component models, identified their key characteristics and discussed their strength and weakness. Certainly, diagrammatic notations are useful for visualising and communicating ideas, but they cannot support more rigorous approaches to software engineering, unless they are equipped with a precise formal semantics. In this section, we discuss approaches that provide a more comprehensive, formal framework to support the engineering task involved in developing software systems. Particular emphasis is placed on formal approaches for the specification and analysis of systems whose architecture comprises a group of interacting components.

General-purpose formal methods such as Z [Spi92] or VDM [Jon90] could be useful in specifying the behaviour of CBS. However, these well-established formal methods were introduced before the advent of object-oriented programming. As a consequence, they do not explicitly consider a semantic characterisation of object, components, frameworks or other high-level software concepts, and therefore cannot describe component contracts in a straightforward manner. Object-oriented extensions of these traditional methods have been developed such as Object-Z [DRS95] but they are not regarded as mainstream in CBS design.

In addition, components are increasingly expected to operate in a distributed and concurrent setting [KFP+04]. This makes strong demands in terms of component interactions and parallel behaviour. Therefore, the study of a suitable formal model for components points towards methods introduced for describing concurrent computations such as Petri nets [Pet79b, Pet79a], CCS [Mil80], CSP [Hoa85], event structures [NPW81, Win88], asynchronous transition systems [Shi85, Bed88], \( \pi \)-calculus [Mil99] among others.

Other formal notations arise as a combination of approaches in an attempt to build on the strength of each, for instance a combination of CSP and B [Tre00, ST02, ST04]. The main feature of this approach is that it provides a way of describing systems involving both event-oriented and state-oriented aspects of behaviour. It stands out because it makes it possible to exploit existing tool support for both CSP and B.

In this section, we discuss formal approaches for specification and analysis of CBS. In particular, we describe representatives of approaches originating from various branches of mathematics such as algebra, logics, regular expressions, transition systems and automata.
2.3.1 Event-based Approaches

Come back to the SOFA model, the approach to formalising the behaviour of components at their interfaces is that of [PV02, Pla05, AP04a, AP04b, AP04c, AP05]. The authors propose a formal framework for describing the ordering of events on interfaces based on the use of behaviour protocols [PV02] which take a form similar to regular expressions (see for example [Coh97]).

The notion of behaviour protocols originates in objects and can be understood as consisting of sequences of requests (calls to operations) that an object is capable of servicing. An object’s protocol can be modelled as a finite state machine which can be specified as a regular-like expression generalising the valid request sequences [vdB’91]. The approach of [PV02] is based on applying the idea of object protocols to components. Since components provide a higher level of design abstraction than objects [Szy97], this approach specifies components within an ADL and in particular, using the SOFA architecture [PBJ98] in which an application is seen as a hierarchy of nested components (components inside other components). Within this architecture a component is considered to be an instance of a component template, similarly to an object being an instance of a class. A component template in [PV02] is a pair of \(<F, A>\) where \(F\) is a template frame defining the set of interfaces (provided and required interfaces) of the component and \(A\) is a template architecture which describes the structure of an implementation version of \(F\) by instantiating the sub-components of \(A\) as well as specifying their interconnections.

A component in this approach has provided interfaces which offer access to the services it provides by listing methods/operations that can be called by clients of the component having reference to the interface, and required interfaces which capture references to other components’ interfaces and list methods that are supposed to be called by the component on the target of the reference represented by this interface.

Components are put together by connecting (or binding) suitable required and provided interfaces from each. In the case of nested components – usually, the result of composition – a connection may exist between a provided interface of the nested component and a provided interface of a sub-component (this is termed delegation) and between a required interface of the nested component and a required interface of a sub-component (termed subsuming).

In terms of a formal description, a component \(c\) in [PV02] is considered within its environment (a collection of other components) and is assumed to have a set of connector \(V\) (to interfaces). The set of all events processed by a component \(c\) on its interfaces forms its alphabet \(A_c\). A trace of \(c\) on \(V\) is defined in [PV02] as a sequence of events handled during a period of activation. Hence, the traces of \(c\) on \(V\) are words over the component’s alphabet \(A_c\), i.e., \(\text{words} \in A_c^*\). For an event \(a\)
in $A_c$, a request and response associated with $a$ is denoted by $a \uparrow$ and $a \downarrow$, respectively. If a request and a response occur inside a nested component (between its sub-components) then the corresponding events are prefixed by $\tau$, such as $\tau a$.

The behaviour of a component $c$ is the set of all possible traces produced by $c$, forming a language $L_c \subseteq A_c^*$. This is called the language of $c$ on $V$ in [PV02]. It can be approximated by a behaviour protocol which is essentially a type of regular expression that generates a set of event traces over $A_c$.

The regular-like expression, behaviour protocol, is defined by a set of classic regular operators such as sequencing (;), alternative (+) and repetition (*), also a set of enhanced operators such as interleaving/shuffling (|), restriction (/) and composition ($\Pi_X$). Here, we find it is sufficient to discuss the composition operator in more detail since it is used in connecting components through binding provided and required interfaces.

The composition operator $\Pi_X$, inspired by the CCS parallel composition operator [Mil80], is used for expressing the behaviour of a component communicating via connected interfaces (these should be a provided interface of one component and a required interface of the another, though not explicitly stated in [PV02] or [AP04a]). For languages $L_1, L_2$ and a set of events $X = A_1 \cap A_2$, the composition operator $\Pi_X$ is defined to be the set of traces where each is formed as an arbitrary interface of a pair of traces $\alpha \in L_1$ and $\beta \in L_2$ such that for every event $e \in X$, if $e$ is prefixed by $?\ $ in $\alpha$ and by ! in $\beta$ (or vice versa), any appearance of $?e;!e \ $ (or $!e;?e \ $) as a product of the interleaving is merged into $\tau e$ in the resulting trace. In other words, it behaves as an internal event. Any event $?e' \ $ or $!e' \ $, for $e' \in X$, which remains unmerged in a product trace $t$ results in the trace $t$ being excluded from the result.

When the composition operator $\Pi_X$ is applied to protocols, the resulting composite protocol $P_1 \Pi_X P_2$ gives the product traces which describe the cases where two components are behaving correctly, but omits any traces that describe potentially faulty behaviour. The authors claim that the problem is rooted in the CCS parallel composition operator, where the originator of complementary events cannot be determined. In subsequent work [AP04a], they attempt to address aspects of the problem by way of including error events and erroneous traces.

The potential pitfalls are experienced for instance when investigating the use of UML 2.0 Protocol State Machine (PSM) for generating behaviour protocols in [Men04]. One of the reasons these cannot be adopted in a straightforward manner has to do with the fact that an operation call in [PV02] is viewed as a pair of consecutive atomic events representing the start of the call (request) and the end of the call (response). This leads the authors to propose a variant of PSMs,
the so-called Port State Machines, which generate the communication language of a behaviour protocol. Not surprisingly, the variation points are mostly related to attributes of transitions. The proposed state machines are in part motivated by the postulate that PSMs cannot capture the interleaving of operation calls on interfaces, which seems to be unsubstantiated given that, in principle, a PSM may comprise sub-PSMs in orthogonal regions (see [OMG04a], page 585) and this provides a means of expressing parallel behaviour.

Further, components in this approach are understood as having a number of provided and required interfaces but events occur sequentially and there is no provision for parallel behaviour (such as on the same interfaces of the same component, not on connected interfaces of different components). Thus, an interface may emit two events concurrently in response to the requests sent from two different required interfaces, cannot be modelled.

As a comment on this approach, we note the following: the behaviour protocol used to approximate the language for each component comprises (a set of) sequences of events occurring on all interfaces of the component. Assuming that the sets of events associated with each interface are disjoint, it is possible to determine what events occur on each interface and derive the orderings between events of different interfaces – in particular, between provided and required interfaces – in this representation. It can be argued however that this representation can be counterintuitive, especially when considering reusing the component in different contexts. In such cases, some rather than all of the component’s interfaces are involved and it is for those interfaces that the ordering relationship to new events (due to the different configuration) needs to be specified. In this respect, a notation for the language that expresses events on each interface separately would appear to be more suitable.

In contemporary component model, components communicate with each other asynchronously and synchronously. SOFA CDL uses the key word ‘oneway’ to differentiates asynchronous services from synchronous services in its interface definitions. In the behaviour protocol, asynchronous communications can be directly expressed in the atomic entity – events. For synchronous communications, the behaviour protocol denotes \(?m \uparrow; !m \downarrow\) as a provided operation call and \(!m \uparrow; ?m \downarrow\) as a required operation call, further abbreviates them as \(?m\) and \(!m\), respectively. Nesting calls are modelled as \(?m \uparrow; \alpha; !m \downarrow\) and \(?m(\alpha)\) in short. Note that here, \(?m \uparrow\) stands for an absorbing request event while \(?m\) represents a provided operation call (a pair of \(m\) with opposite prefixes and suffixes). It seems the behaviour protocol restricts any operation call to be a pair of events with the same names. However, it is unclear how to model an operation call made up of two distinct events (such as \(!m \uparrow; ?n \downarrow\) and, if so, how to model nesting calls then? We believe in order to elegantly differentiate synchronous communications from asynchronous communications in the behaviour representation, it is necessary to formally
introduce the concept of operation calls and distinguish them from a sequence of events in the
behaviour expression. One subsequent benefit is that it makes the behaviour description more
clear, concise and readable and more suitable for component modelling. Further comments
regarding the syntax aspect of the behaviour protocol can be found in Section 4.7.

Another approach of Broy [Bro00, Bro95] presents an algebraic model for components. The
basic idea behind this approach to formal specification of reactive CBS originates in the
functional approach to the formal description of communicating systems in [Bro93]. The
input/output behaviour is described in [Bro93] by predicates which characterise sets of
deterministic behaviours. A deterministic behaviour is represented by a stream processing
function. This functional approach is extended in [Bro00] by algebraic specification concepts. The
motivation is to provide an algebraic technique for writing specifications for CBS in a fashion
similar to the use of algebraic specification techniques for data structures and information flow.

The algebraic approach proposed by Broy [Bro00, Bro95] advocates the use of *streams* to
describe communications over the channels of a component. Given a set of messages \( M \), a stream
over \( M \) is a finite or infinite sequence of elements from \( M \). The set of all streams over \( M \) is
denoted by \( M^* \). Hence, \( M^* = M^* \cup M^0 \) where \( M^* \) denotes the finite sequences over \( M \),
including the empty sequence, and \( M^0 \) denotes the infinite sequences over \( M \).

Concatenation and prefix ordering operations are introduced to streams. The set of streams \( M^* \)
equipped with the prefix ordering relation is complete in the sense that every directed set \( S \subseteq M^* \)
has a least upper bound. Recall (for example, [DP90]) that a directed set \( X \) is defined as a
nonempty subset of a partially ordered set (poset) \( D \) if any two elements in \( X \) are bounded above
by a third element also in \( X \). Least upper bounds of directed sets of finite streams can be used to
describe infinite streams. Further, [Bro95] defines functions for selecting the first element of a
stream and removing the first element of a stream, providing the stream is nonempty.

Based on the concepts introduced in [Bro95], a mathematical concept of a component is
subsequently given in [Bro00]. Syntactically, a component is described by a set \( I \) of input channel
identifiers and a set of \( O \) of output channel identifiers. Each channel is associated with a sort,
which is essentially a data set indicating the messages communicated along this channel.
Semantically, a component is described by a predicate defining a set of deterministic behaviours.
A deterministic behaviour of a component is represented by a stream processing function
\( f : (I \rightarrow M^*) \rightarrow (O \rightarrow M^*) \) that maps every input history onto output history. An input (resp.
output) history is obtained by a valuation of the input (resp. output) channels by streams. The
formal description of a component \( C \) in [Bro00] is given in terms of a predicate \( \mathbb{B} \) (true or false)
on the stream processing functions of \( C \). This defines a set of deterministic behaviours \( Q \) as
Chapter 2 Approaches to Modelling Component-based Systems

\( Q : (I \rightarrow M^w) \rightarrow (O \rightarrow M^w) \rightarrow \mathbb{B} \). Thus, given a component \( C \) and a predicate \( B \) on its behaviours, the corresponding set \( Q \) comprises all mappings between legal input and output histories of the channels of \( C \).

The functions for selecting and removing the first element of a stream introduced in [Bro95] are lifted to stream processing functions in [Bro00] by way of input and output transitions. An input transition is defined as follows. Given a stream processing function \( f \), a message \( m \in M \) and input channel \( c \in I \), the expression \( f < c : m \) is defined to be the stream processing function that behaves like the function \( f \) on the communication history \( x \in (I \rightarrow M^w) \) after message \( m \) is added as the first message on channel \( c \) to the input \( x \). An output transition is defined for every output channel \( c \in O \) in a similar fashion. The expression \( c : m < f \) is defined as the stream processing function that behaves like function \( f \) but always adds the message \( m \) as its first output on channel \( c \) to the output produced by function \( f \).

All operations on function can be extended to sets of functions and thus to specifications by applying them pointwise to the elements in the set described by the specification \( Q \). Therefore, the expression \( Q < c : m \) characterises all behaviours \( f \) for which there exists a behaviour \( f' \) with \( Q(f') \) such that \( f \) behaves like \( f' \) after it has received the message \( m \) on its input channel \( c \). Likewise, the expression \( c : m < Q \) characterises all behaviours \( f \) for which there exists a behaviour \( f' \) with \( Q(f') \) such that \( f \) behaves like \( f' \) after producing the message \( m \) on its output channel \( c \).

With regard to composition of components [Bro00] considers two basic operations, namely parallel composition and feedback. These operations are described by logical connectives on the predicates representing the specification. [Bro95] also defines sequential composition, denoted by \( C_1 ; C_2 \), which is in fact functional composition of the stream processing functions describing the behaviour of each component.

The parallel composition of two components \( C_1 \) and \( C_2 \) with disjoint sets of output channels is denoted by \( C_1 \parallel C_2 \) and the channels of the composite are given by

\[
\text{Out}(C_1 \parallel C_2) = \text{Out}(C_1) \cup \text{Out}(C_2) \quad \text{and} \quad \text{In}(C_1 \parallel C_2) = \text{In}(C_1) \cup \text{In}(C_2)
\]

where \( \text{Out}(C_i), \text{In}(C_i) \), for \( i = 1,2 \), denote the output resp. input, channels of the components \( C_i \). The actual sequences of messages on channels of the resulting composite component are represented by a tuple of streams which is formed by elementwise concatenation of the streams corresponding to (the channels of) each component. This form of parallel composition tends to
focus on describing the I/O behaviour at the system (or composite) level but does not seem to involve interaction between the participating components.

Besides parallel composition, [Bro00] also works with feedback. For channels \( x \in \text{In}(C) \) and \( y \in \text{Out}(C) \) the feedback operator \( \mu^y_x C \) describes the feedback of the stream output from channel \( y \) to the input channel \( x \). It is defined by \( \text{In}(\mu^y_x) = \text{In}(C) \setminus \{x\} \) and \( \text{Out}(\mu^y_x) = \text{Out}(C) \), where \( \text{In}(C) \setminus \{x\} \) denotes the hiding of channel \( x \) – it is no longer considered in the set of input channels of the component with feedback.

The algebraic model for components of [Bro00, Bro95] described so far, makes use of a rich set of standard mathematical notions and provides a theoretical framework for the engineering aspect of software development. It seems to be geared towards modelling data flows between components, however. This is also reflected in the composed system which is modelled by data flow nets. In the context of CBS design however, it is also important to describe the dependencies between provided and required services. Further, the algebraic model of [Bro00, Bro95] describes the input/output history of a component (or system of components) but does not explicitly relate input events to output events during the course of the behaviour described by the I/O relation.

In respect to this issue, [Bro03] introduces a timing property on stream processing functions describing components as a causality requirement between input and output histories together with a notion of a service. The timing property says that whenever two input histories are the same at time \( t \), then their corresponding output histories at time \( t + 1 \) shall also be the same.

It might be worth noting here that this builds on earlier work on introducing time to the model through timed streams in [BRS*00]. These are essentially streams with discrete time, assuming a global time scale that is valid for all parts of the system. Each time interval is mapped onto a stream over \( M^n \). This allows isolating the stream containing the elements of the first \( t \) time intervals. Another discussion of the explicit use of the notion of time in this model can be found in Section 4.7.

A service in [Bro03] has a syntactic interface of a component but the stream processing function describing its behaviour is partial. In contrast to a component where the causality requirement implies that for all input histories the corresponding output histories are either all empty or none of them is, a service is defined only for a subset of its input histories (for instance certain access conventions must hold before the service is available).

[Bro04] elaborates on this notion of causality in relation to time. The idea is that if the time granularity of the system is taken to be fine enough then the corresponding time model can separate between causally related events. The argument goes that if the time scale is fine enough then causally related events can be associated with different time units. Then the causality
requirement simply says that output which depends causally on certain input cannot be generated until this input has been received and hence the component does not react to input received at time $t$ before the $t + 1$ time unit.

This seems to address dependencies between provided and required interfaces at the level of a single component where causality between events is known in advance. When components are put together however, it is common that some events generated by one component only in response (reaction) to events generated by another component. For example, it is not entirely clear how choosing the time granularity appropriately can exclude independent events (generated by different components and thus initially not causally related) from occurring within the same time unit.

Such situations may cause a (potentially concurrent) series of other causally related events and lead to a slightly different behaviour, or even result in so-called causal loops. It is not entirely clear in the approach of [Bro00, Bro03, Bro04] how causality at the individual component level is interpreted in the composed system. Further, the potential for concurrency at the composition level is not addressed.

### 2.3.2 State-based Approaches

The approach of [SR02a, SR00, Reu00] advocates the use of automata for capturing the dependencies between provided and required interfaces of components, building on early ideas in [Reu99]. In addition, this work proposes the use of *adaptors* for the purpose of reusing components in different configurations.

This approach takes a black-box view of components in which communication with the environment is exercised by so-called *gates*. Provided gates are used to describe possible connections to the external world for the purpose of providing services while required gates are used to represent possible connections to other components required to perform the services provided. The set of provided gates postulates the provided interfaces of the component while the required gates define its required interfaces.

Interfaces in [SR02a] are modelled by a type of finite state machine (FSM). In particular, a FSM in [SR02a] consists of: (i) a finite set of states $S$. This includes an *initial* state, a set of *final* states and an *error* state which designates a system failure (once the system enters this state it cannot leave); (ii) a finite set of inputs $I$; this comprises a set of events and a set of actions, where each event is accepted in at least one state and actions are triggered by incoming events but are regarded themselves as inputs for transitions too; (iii) a transition relation $t$ given by $t: S \times I \rightarrow S$. Transitions are regarded as instantaneous and deterministic FSMs are only considered in this approach, in the sense that there is at most one transition for each source state and input event.
Different specifications of each FSMs are used to model the observable behaviour of a component in terms of change of state due to methods or operation calls. FSMs without actions, denoted by P-FSM (for provides-FSM), are used to model the behaviour of a provided interface. FSMs without events on transitions, denoted by M-FSM (for method FSM), are used to model required interfaces. Each operation call on the provided interface gives rise to a sequence of operation calls through some required interface. Such invocations are modelled using M-FSMs. Hybrid forms denoted by C-FSM (for component FSM), including both the P-FSM and the corresponding M-FSMs are used to model the behaviour of the component as a whole on its interfaces. Notice that in this approach a component has a single provided interface and (possibly) multiple required interfaces.

The C-FSM for a component is constructed by taking the P-FSM and after every transition (method invocation or operation call) inserting a copy of the M-FSM corresponding to the respective method/operation appearing on that transition. A transition labelled by 'return' is drawn from the final state of each inserted M-FSM \textit{op} to the target state of \textit{op} in the initial P-FSM. A detail algorithm for the construction of the C-FSM is given in [SR00]. Further, [Reu00] describes the reconstruction of the provided interface out of the C-FSM for a component. Note the use of the UML notation \textit{e/a} for labels on transitions though event/action pairs are essentially modelled by two distinct transitions in the formal approach of [SR02a, SR00].

For the purpose of combining components in a configuration, [SR02a] introduces two adaptors, namely the split-operator and the join-operator.

The split-operator is used to model the situation where one component uses two other components. Hence, it takes a required interface and splits the corresponding outgoing methods/operation calls to two provided interfaces. This comes down to merging the sequences of operations of two provided interfaces to a single provided interface. The basic idea in [SR02a] is that behaviours from each P-FSM can be merged by considering all possible interleavings of the corresponding sequences. This is possible since the P-FSMs belong to different components and can change states independently. The resulting interleaving generates the language accepted by the P-FSM of the combined provided interface, which is termed shuffle-FSM in [SR02a]. A detailed algorithm for constructing the shuffle-FSM is given in [SR00]. To our understanding, the split-operator essentially amounts to composition of FSMs when no communication is involved.

The join-operator is used to model the situation where a component is used by two other components. In short, it takes two required interfaces and joins their outgoing sequences of operation calls so that they can be serviced by a single provided interface. Since they belong to different components the two required interfaces can potentially call the same operation of the provided interface at the same time; this comes down to ensuring that the sequences of both M-
FSMs are reflected in the resulting joined outgoing sequences of operation calls. The basic idea in [SR02a] is that the behaviours of the two required interfaces can be merged so long as conflicting calls are excluded. Two calls are understood to be conflicting when they both call the same operation of a provided interface. They are excluded by imposing a form of synchronisation that ensures only one call can be performed. This is applied to consecutive calls when the first call is made by one required interface and the second by the other. These situations are detected by traversing all paths in the intersection of the shuffle-FSM of the two required interfaces and the P-PSM of the provided interface. A detailed description of the algorithm for the join-operator is given in [SR00]. To our understanding, the join-operator is not performing composition where communication is involved. The two FSMs cannot change state independently but this is due to competing for access to the same resource (i.e. calling the same operation) rather than a result of change in state in one machine.

The works of [SR02a, SR00, Reu00, Reu99] are directed at capturing dependencies between provided and required interfaces of components, a central issue in component-based design. A component in this approach is understood as having a single provided interface through which it makes its services available and a number of required interfaces through which it states its requirements. This is not in line with the view taken of a component in UML 2.0 where a component has a number of provided and required interfaces, as discussed before. The graphical notation may not be a major concern but restricting components to only have a single provided interface does not allow for parallel behaviours of the component. This limits a component to servicing requests sequentially only when concurrency could be realised (for instance through replication of objects, codes or even resources). Also, an operation call on the provided interface can cause a sequence of calls to be made by the component but this must be done through one required interface (exclusively).

For example, in reactive systems, upon receiving an operation call on one of its interfaces, a component might have to respond by making operation calls through its required interfaces concurrently. To model such a situation in the approach being discussed, would entail inserting a copy of each M-FSM corresponding to the respective operations and in a way that the transitions leading to each can be fired concurrently. It appears that the use of P-FSMs and M-FSMs in constructing the C-FSM for the component as a whole does not have the expressive power to capture concurrency between operation calls occurring on interfaces.

This is also manifested in the algorithm described for the join-operator between FSMs corresponding to different interfaces where synchronisation points have to be used. Synchronisation points may be suitable for the purpose of accessing the same interface, as prescribed in [SR02a], but they would not be adequate for a more general form of composition.
where communication is involved, since they impose a specific sequence of operation calls and exclude others (that could also be allowed in principle).

[SR02a] uses a FSM to address an interesting case of incompatibility between otherwise compatible components. This is the case where one component requires a service from another component and while the other component does offer the service it is not available at the time the request is being made. In other words, the C-FSM\textsubscript{A} of one component, say \textit{A}, makes a call to operation \textit{op} which exists in the C-FSM\textsubscript{B} of the other component, say \textit{B}, but is not yet ready at the current state of component \textit{B}. The basic idea is to prefix all calls to the P-FSM by a sequence of operation calls. This sequence is used to bring the P-FSM to a state in which the operation in question can be called.

Additionally, an appropriate postfix must also be considered such that after the call, the P-FSM can move to a final state. Such prefixes and postfixes are computed via the so-called asymmetric shuffle-FSM whose states are a subset of the Cartesian product of the states of the C-FSM and the P-FSM. See [SR02a] for a detailed algorithm for constructing this FSM. In fact, the asymmetric shuffle-FSM contains two kinds of transitions: marked, where the input \textit{i} is handled by both the C-FSM and the P-FSM, and unmarked transitions, where the input \textit{i} is handled in the P-FSM but not in the C-FSM. The prefix is determined as a path in the asymmetric shuffle-FSM from a state pair \((s_c, s_p)\) to a marked transition \(i\). The postfixes are determined as paths from \((s_c, s_p)\) to a final state of the asymmetric shuffle-FSM.

With regard to the more general problem of component interoperability, [SR02b] takes a view of components in which their interfaces are not fixed. This is particularly relevant for component-based design because it is often the case in architecting components that the full set of provided and required services do not map exactly, yet there is a meaningful subset on which they do agree. The idea is to consider the provided services (post-condition) as a parameter for computing the required services (pre-conditions), and vice versa, in defining the component contract. In contrast to class contracts, parametric contracts link the provided and required interfaces of the same component and allow for new interfaces to emerge which are tailored to the specific context or configuration the component is placed in. Subsequent extensions of this work have considered parametric contracts in the context of component composition [RBF04] and their effect on system reliability [RPS03].

A parametric contract is determined in [RBF04] by considering a function \(\rho\) from the set of all possible provided interfaces of component \(c\) to the set of all possible required interfaces of component \(c\). A possible interface is any interface offering (resp. requesting) a subset of the functionality offered (resp. required) by \(c\). The function \(\rho\) maps each possible provided interface to one or more possible required interfaces (\(\rho\) is not injective). Thus the inverse mapping \(\rho^{-1}\)
associates each possible required interface with a set of possible provided interfaces of component $c$. To obtain a single provided interface, [RBF04] considers the least upper bound of the set returned by $\rho^{-1}$.

The actual parameter contract specification, i.e. the nature of the function $\rho$, is not given in [RBF04]. This is not surprising since it depends on the interface model used, just like in classic component contracts. Nevertheless, the component designer does not need to foresee possible reuse contexts and instead only needs to specify the bidirectional mapping between possible provided and required interfaces of the component in hand. Hence, the parametric contract is part of the component specification. If the required sequences of operation calls have been specified for each possible provided interface, then the required interface can be determined dynamically as it depends upon the actual subset of the offered services (through the provided interface) used. This is done in [RBF04] using FSMs, which were discussed earlier.

Another state-based approach to the formal description of component behaviour is that of [dAHO1b, dAHO1a]. [dAHO1b] stresses the need for interface models in component-based design as a means of specifying what a component expects from its environment. The authors argue in favour of formal interface models with game-theoretic foundations that can support compatibility checks and refinement. The interface model proposed in [dAHO1a] is in this spirit and uses an automata-based language to model the behaviour of components at their interfaces. In particular, it is intended to capture assumptions about the order in which the methods or operations of a component are called and the order in which the component calls methods of other components.

The input/output behaviour of a component is described by an automaton, the so-called interface automaton, which is syntactically similar to the I/O automata of [LT87]. An interface automaton in [dAHO1a] consists of: (i) a finite set of states, which includes an initial state; (ii) input actions, which can be understood as events (for example operation calls and their return values) on the receiving end of communication channels; (iii) output actions, which can be understood as events (for instance, operation calls, message transmissions) on the sending end of communication channels; (iv) internal actions: to our understanding, an internal action is an action accepted at a state of the product of two automata when (at the projection of this state onto the state of each automaton) it is an input action of one, and an output action of the other; (v) a transition relation, which defines a step between states via some action. It might worth pointing out that the definition of an interface automaton, as given in [dAHO1a], does not include any deterministic condition, effectively allowing multiple target states for a single transition.

A component in [dAHO1a] is represented by a box whose ports correspond to the input and output actions, each port being associated with either an input or an output action. An interface automaton is used to capture guarantees about the specified component, in terms of sequences and
choices of actions via its ports. In doing so, an interface automaton also captures assumptions about the environment: each output step of the automaton incorporates the assumption that the corresponding output action is accepted by the environment as input and each input action that is not accepted at a state of the automaton incorporates the assumption that the environment does not provide that input.

In this way, when interface automata are combined, their composition includes not only the corresponding components’ guarantees but also the respective environment assumptions. The composition of two interface automata includes forming the product of the two automata and then restricting the product automaton to the set of compatible states. These are states from which the environment can prevent the product automaton from entering error states. In what follows, we will discuss composition of interface automata in more detail.

Each state of the product automaton, denoted by $P_1 \times P_2$ for interface automata $P_1$ and $P_2$, consists of a state of $P_1$ together with a state of $P_2$. Each step of the product automaton is either a joint step, which represents an output (resp. input) action of one automaton which is an input (resp. output) action of the other; or a simple step, which represents an input or output action from one automaton, providing it is not an output or input of the other.

We pause to make the observation that this does not cover the case where the two automata can engage in independent actions within a step. These could be: (i) an input (or output) action from each, or (ii) an input (resp. output) action of one which is not an output (resp. input) action of the other. To our understanding, such cases are not considered in defining the transition relation of the product interface automaton. Note that performing composition under the condition that the two interface automata have disjoint sets of actions, unless an input action of one is an output of the other, does not exclude the above cases.

The product automaton obtained following the construction given in [dAH01a] may contain states in which one automaton does an output or input action which exists in the set of actions of the other automaton, but is not yet ready in its current state. Notice that this is precisely the incompatibility issue that the approach of [SR02a], discussed earlier, is also concerned with.

Instead of attempting to coerce the automata into meeting the respective requirements, as done in [SR02a], [dAH01a] removes such incompatible states from the product and ends up with a set of compatible states only, which is considered to be the composite automaton of the two initial interface automata. Hence, the product automaton is seen as an intermediate step in constructing the composition of two interface automata $P_1$ and $P_2$, denoted by $P_1 \parallel P_2$. The compatibility checking which is performed at the level of the product automaton by computing compatible states can be viewed in a game-theoretic setting. It amounts to solving a game between the
product automaton, which tries to enter incompatible states, and its environment, which tries to prevent this.

The interface automata proposed in this approach [dAH01a] provide a useful way of specifying behaviour at the interfaces between components. They can be used to capture both guarantees about the specified component, in terms of legal component behaviours, and assumptions about the environment, in terms of permissible environment behaviours. The challenge in this approach, the so-called optimistic view, is to find some environment (rather than all) that satisfies the environment assumptions of all components in the composed system. This optimistic approach to specifying interfaces allows for an elegant treatment of refinement which comes down to choosing between the legal component behaviours without restricting the permissible environment behaviours.

A component in this approach has a dedicated port for each input and output action. This is somewhat restrictive and does not reflect the way components are understood in UML. Even in the Koala component model where the notion of a component is influenced by the fact that components are expected to sit directly on top and drive hardware devices, input and output ports are associated with more than one signal.

Further, there seems to be no way to express concurrency between input and/or output actions on distinct ports. The automata-based language used to capture the ordering of actions on ports allows for sequential execution only. This is manifested in the notion of composition given in [dAH01a] which is essentially synchronised on shared actions and interleaving of all other actions. Transitions of the composite automaton are curiously restricted to either shared actions from both constituent automata or individual actions from solely one automaton, and thus do not cater for the full range of independent actions, as discussed before.

From Arbab et al's paper [ABB00], they introduce another formal model for components. In this, every interface contains a FSM which is used to state the internal status of components, and which abstracts away from component internal details and the particular programming language of its implementation. The interface contains five elements: a name, a channel signature, and three predicates, namely a blocking invariant, a pre-condition, and a post-condition.

The name of a component uniquely identifies the component within a system, which also gives a type of the component.

The channel signature of a component is a list of channels representing its initial connections. With regard to simplicity of models, the channel in here is defined a one-to-one, unidirectional and first-in-first-out (FIFO) event channel. It represents a reliable and directed flow of information from its source to its sink. Every channel is an exclusively point-to-point communication medium between a single producer and a single consumer. The producer or the
consumer of a channel loses its exclusive control of its channel-end by writing its identifier end to another channel. Subsequently, a component may dynamically regain the exclusive control of a specific end of a channel, simply by reading its identifier as a value from another channel. This allows dynamic reconfiguration of channel connections among the components in a system. However, the number of initial components and channels is assumed to be static. A component may send a value to a channel only if it is connected to its source. Similarly, it may receive a value from a channel only if it is connected to its sink.

The blocking invariant is a predicate that specifies the possible deadlock-prone behaviour of the component. The pre-condition is a predicate that specifies the contents of the buffers of the initial external channels (i.e., the ones in the channel signature) of the component. The post-condition is a predicate that specifies the contents of the buffers of the external channels that exist upon termination.

Semantically, they define the behaviour of a component by a FSM. They define observable behaviours of a component in terms of sequence of values, one for each channel-end that the component has been connected to. The value sent or received by a component through a channel at a particular point in time will be independent of the time other values are sent or received through other channels. This implicitly means that channels work independently; each of them receives or sends information concurrently in the system. The observable behaviour of a CBS is given by the set of final global states of successfully terminating computations, provided that the system is deadlock-free. Notice that, a global state records for each channel the contents of its buffer.

As for the comments on the works from Arbab et al, the main contribution is that they gave a formal model for CBS. It contains the basic concepts of interfaces, which comprise a name, a set of channel ends, a set of blocking invariants, a pre-condition and a post-condition. And they introduced the architecture of CBS, which contains a set of components linked by a certain number of one-to-one and unidirectional event channels. Each component can exchange messages through the observable behaviours in their interfaces. A logical interface description language and the formal semantics of composing component interfaces are introduced in [ABB00].

The highlight of their work is that they proposed a dynamically reconfigurable component-based architecture, which is inspired by the works in a formal model of components [BRS'00]. This dynamically changeable architecture denotes that assuming the number of components and channels are static, the connections between each component in the architecture can change dynamically in the runtime environment and in an arbitrary manner.

However, a key shortcoming of their model is the definition of connections. That is, the channel in [ABB00] is not sufficiently powerful to handle some situations in the real world. For instance,
although a one-to-one and unidirectional kind of channel can be executed concurrently and independently, it cannot broadcast messages to many receivers. In addition, the unidirectional characteristic of this channel restricts the possibility of further composition of interfaces.

2.4 Conclusion and Goals of the Thesis

The component-based approach to software engineering offers a range of potential benefits, notably reuse and reduced product-development time. It has been maintained that the component-oriented paradigm inevitably places emphasis on the specification and analysis of components. In this chapter we have reviewed approaches to the specification and analysis of CBS.

Undoubtedly, it is common practice in modelling software systems to think in terms of drawing diagrams to provide a graphical representation of various aspects of software. However, there is an inherent difficulty with graphic modelling: the choice of what diagrams to use has a profound influence on how a solution is shaped. As if to make things worse, a diagram can be expressed at different levels of precision. In addition, it can be claimed that diagrammatic modelling is in a sense error-prone. However, it does not have to be error-prone, since the notation itself may inherently have precise semantics, but people naturally tend to use diagrams imprecisely.

A means of resolving ambiguity is to attach a formal interpretation to a diagram. Only then could diagrammatic notations be useful for analysis and verification (of the information they convey), in addition to their visualisation purposes. For instance, UML 2.0 [OMG04a] includes graphical representations for provided and required interfaces of components. Although the need for a formal notion of a contract between provided and required interfaces is acknowledged in UML 2.0 (see Chapter 8, 15 in [OMG04a]), such a formalisation is not laid out in its specification document.

It should be recognised that graphical descriptive approaches seem to lack an associated precise behavioural semantics for the elements being represented, in general. In an attempt to provide an easily comprehensible notation, formal rigour is sacrificed. On the other hand, the fact that diagrammatic-based descriptive techniques, including UML and Koala, do not commit to a specific formal semantics, allows for a number of formal interpretations to be attached.

In addition to resolving ambiguities of a certain class of diagrams, formal approaches can have an effect on the choice of diagrams used in graphical modelling. Formal methods have not been espoused by component designers, at present. We do not claim the experience to argue on this issue in depth, but two contributing factors seem to stand out. One has to do with the steep learning curve usually associated with formalisms and mathematics that makes component developers reluctant about their application in design. The second factor, which is in a sense
related to the first, is that formal approaches are often not blended with UML concepts and diagrams that underline current software design practices. The review of formal approaches in Section 2.3 suggests, for instance, that components are understood as having a single provided interface, unlike components in UML 2.0 which have multiple provided interfaces.

Nevertheless, a formal model for component-based design could be seen to add value if it is expressive enough to capture subtle issues such as concurrency, simultaneity, nondeterminism, and so forth, only with the sacrifice of visualisation.

The review of existing formal approaches to CBS provides a view of different approaches to behaviour modelling. Traditionally, the principal approaches are based on event and state.

Both state-based and event-based approaches draw upon the notion that behaviour can be captured by representing the lifecycles of objects as state-machines. The key difference between state-based and event-based approaches is whether the lifecycle analysis focuses on the internal status (or data conditions) of the object or on the event sequence (or event protocol) of the object. This has some interesting consequences for the form that state-transition models take.

In the state-based approaches, the focus of modelling an object’s lifecycle is the identification of the object states, where a state is characterised as “a situation or condition of the object in which certain physical laws, rules and policies apply” [SM92, page 5]. The modelling of an object lifecycle centres on identification of the relevant states based on an understanding of the domain and the processing requirements. Transitions, fired by events, are added to the state-machine to drive it from one state to another. An event is defined as “an abstraction of an incident or signal in the real world that tells us that something is moving to a new state” [SM92, page 42]. This approach is most widely used in the real-time/embedded system domain and the best known example is probably the Shlaer-Mellor Method, developed in the 1970s [SM92]. The Shlaer-Mellor approach has recently been repositioned as an MDA approach under the name ‘eXecutable UML’ [MB02, RFW’04].

The drawback of state-based approaches is that they represent system behaviour locally in each class or component; thus, there is lack of support for displaying the global behaviour of the system. In addition, they illustrate causality, concurrency, and conflict relations between events in terms of state sequences or state configurations (for example, state diamonds). For more succinct representation, it is very important to identify the set of causality relations, concurrent events, and conflict conditions separately from the state-based representation because they carry more useful information for the designers and design algorithms.

Furthermore, state-based approaches lack mechanisms for reuse of state-machine behaviour across different class definitions except in the case where two classes have identical state-machines. In addition, the lack of support for the concept of being able to refine state-machine
definitions as designers move down a class hierarchy entails that state-machines are always defined solely at the lowest level [MB02, page 227]. Thus, if two different types of bank account are being defined with slightly different behaviours, each would need to be given its own, entirely separate and complete, state-machine. It is not possible to define the common elements of behaviour in a common abstract class and refine this differently for the two specific types of account.

The event-based approaches, on the other hand, are to define the valid lifecycle of the objects, where a lifecycle is defined as a sequence of events. The approach therefore centres on identifying the complete vocabulary of events that can affect the object, whether they cause state change or not, and constructing a state-machine that describes the possible orderings of the events over the life of the object. The event-based approach has its root in the work of Jackson et al in the JSD method developed in the 1970s and 1980s [Jac83]. JSD used a diagrammatic tree diagram form of simple regular expressions to describe event sequences, which in JSD were called ‘entity life histories’. In event-based approaches, an event is defined as any incident in the real-world whose occurrence is allowed or constrained by the state of the object. This is a wider definition than that used in the state-based approach as it includes events that do not change the state. This wider definition is necessary to define protocols because the circumstances under which such an event can take place are part of the protocol. For instance a \textit{ChangeQuantity} event on an \textit{Order}, which does not change it state, cannot happen after the \textit{Deliver} event.

A by-product of the event-based approach to constructing the state-machine model is the resultant definition of the states. The states so defined will generally corresponding to the states that would be chosen in the state-based approach.

Compared to state-based approaches, event-based approaches allow a given state-machine to be re-used across the definition of different objects. For instance, if a number of different types of account were being defined, some of which could be frozen and some not, the two state-machines could be used selectively as building bricks in the definitions of the different types of account. This allows a pure Mixin-based approach [BC90] to behaviour definition.

This chapter has summarised the approaches of modelling CBS. As described in the preceding sections, we face several issues in modelling CBS.

\textbf{Formal Semantics}

Our, almost periodic, reference to the semantic issues of the existing diagrammatic/ADL-based software component models addresses the fact that, although useful for informal specification and structural design, these models cannot provide adequate support for rigorous analysis and formal verification. Therefore, the models need to be translated into other, more formal notation. This can determine a precise interpretation if the target notation has a well-defined semantics.
Chapter 2 Approaches to Modelling Component-based Systems

True Concurrency

The diversity of interleaving semantics and non-interleaving semantics (usually refer to as true concurrency) roots from the different standpoint of viewing concurrent events. Interleaving models believe the ordering of concurrent events is objective and irrelevant. The observations on systems exhibiting concurrency largely depend on the relative position of the observer or the actual timing of execution. Different observers may disagree on the ordering of concurrent events because they are in the different position\(^2\). Therefore, interleaving models assume observations are sequential in nature leading to the interpretation that concurrent events may occur in either order. On the other hand, non-interleaving models believe the ordering of concurrent events is subjective and thus is not distinguished. They represent relative concurrency under the assumption of the existence of a global clock in the universe, that is, under a global clock, concurrent events are synchronised in the time domain and their ordering is explicitly specified. Further, a notion of simultaneity is identified to specify the situation where events progress exactly at the same time.

Both UML 2.0’s sequence diagram and SOFA’s behaviour protocol express concurrency in interleaving semantics. Certainly, it has the advantage of an easier mathematical treatment of concurrency, but the interleaving approach is not originally suited for defining component behaviours, in the sense that its concurrency is reduced to sequentiality plus nondeterminism. For example, it is unable to faithfully manifest the parallel interactions among independent components. In non-interleaving models, parallel composition is more intuitive to capture concurrency and considered as a primitive operator. Non-interleaving semantics is more suitable to maintain the information that the system is composed of independently computing components, that is, some behavioural properties (typically deadlock and liveness properties) rest on the fact that each component is a separate entity independently making its own computation progress.

Concurrency Granularity

In the SOFA model, components are assumed to have a number of provided and required interfaces but events occur sequentially. Seeing that, the behaviour protocol in this approach approximates component behaviours in sequences of events occurring on all its interfaces, parallel behaviours only arise from the composition of separate components for exchanging services. There is no provision to show events taking place in parallel in the same interface of a component. This limit becomes evident in modelling interfaces with multi-threaded features, where an interface may concurrently serve different requests sent from connected interfaces of different components. This problem can also been seen in process algebra approaches (such as CSP and CCS), in which concurrency arises only through the composition across different components. In these approaches, we say that concurrency is considered at the CBS level. Moschoyiannis and Shields’s component vector language [MS03] is able to describe concurrency in a single

\(^2\) This thought-experiment was given by A. Einstein in [Bin21] to demonstrate the non-objectivity of contemporaneity in relativistic mechanics.
component, but with the assumption that services occur sequentially on a single interface. Concurrent service occurrences can only engage distinct interfaces of the component. Thus, we say that [MS03] considers concurrency at the component level. In order to provide adequate support for CBSE, it is desirable to have a well-established behaviour language which has ability to capture concurrency at the individual interface level, so that concurrency can be considered within a single interface of a component.

**Notation Readability**

In structure descriptions, CORBA’s IDL [OMG02] and its variants distinguish asynchronous services from synchronous services by a postfix ‘oneway’. The corresponding difference, however, does not appear justified in behaviour description approaches. A major instance in this chapter is again the behaviour protocol in SOFA. In this event-based approach, the event is regarded as atomic entity and the operation call is modelled as a pair of consecutive events representing the invocation and reaction. However, in a protocol expression, we cannot precisely determine the operation call from a sequence of events without the SOFA CDL’s reference. A clear structural entity becomes ambiguous in the behaviour description. This ambiguity reduces the readability of the model and increases the complexity of developers’ tasks.

In recent years, research about several major component models has been published, each model aimed to tackle part of the problem they are interested in. However, some software component models are diagrammatic/ADL-based and lack operational semantics. Up to now an intensive effort has been devoted to the formalisation of UML semantics (see a survey report [MB01]), but not yet on the Koala model. This limits reasoning Koala systems at a behaviour level at design time. The lack of consensus of the underlying behaviour semantics has resulted in serious difficulties with integrating Koala components that have been developed at different centres. Uchitel et al [UCK+04] employ a process algebra – Finite State Processes (FSP) [MK99], to specify the horizontal communication protocol in the Koala model. FSP’s operational semantics is given in terms of labelled transition systems (LTS), which is a purely interleaving approach and hence true concurrency cannot be addressed.

Our goal and the main motivation for the work developed in this thesis are to provide a formalisation and behavioural semantics for component-based ADLs, such as Koala. As this work is specifically targeted at the real-time and embedded software domain, addressing the above-mentioned issues of concurrency are central to our work. In addition, we target our work at the anticipated usage scenario of (for example) Koala components. That is, components are selected from a generic repository, and connectors are used to accommodate any behavioural mismatch between required and provided interfaces. In the next chapter, we are going to introduce the generic CBS framework we are working with.
Chapter 3

A Generic Component-based System Model

In this chapter, we describe the creation of a generic CBS model to set the context of this study. This CBS model is a generalised and simplified version of the Koala model. Being focused on fundamental principles, in Section 3.1, we discuss how we base our model on a primary component which exhibits functionalities and dependencies via services. Services can be either events (asynchronous) or functions (synchronous), according to the way of communication. An interface is an abstract definition of a coherent collection of services. It also can be considered as a contract of components. A coherent collection of interfaces normally resides on a service port, which is an interaction point between a component and its environment. In Section 3.2, based on Allen and Garlan’s architectural connection [AG97, AG98] that takes communication out of components, we proposed a connector that further takes control out of components. Our connectors, rather than components themselves, initiate and coordinate interactions amongst components, so that any control flow between components is encapsulated by connectors. With regard to the hierarchical design, we introduce the concept of composed component in Section 3.3. A composed component is a composite that is able to contain a group of sub-components together with connectors. Composed components provide proxy ports, which directly link to internal ports, in order to make the internal structure accessible from entities at higher levels. Finally, we consider the composed component without any proxy ports as a component-based system (CBS). Next, Section 3.4 introduces the CBS evolution lifecycle. We end this chapter in Section 3.5 with a discussion of our CBS model compared with other connector-centric approaches in the literature. We will use a ticket vending machine throughout this chapter as an example to illustrate this model.

3.1 Primary Components

A service is an atomic structure entity in our model. It characterises a basic functionality of component, and lists a dependency of the component. A component is an encapsulated computational unit that is only able to interact with its environment via exposed services. A component can interact with its environment in an asynchronous or synchronous manner. In our work, a service can be either an event or a function.
An event is used for asynchronous communications. It facilitates loosely-coupled message-oriented service interactions between components. An event is able to handle one action (or signal). Its definition comprises a *polarity*, denoting the way of handling the action; and an *action*, specifying an atomic computational unit. According to the different ways of dealing with an action, an event can be an: (i) *in event*, which can absorb an action from its environment; (ii) *out event*, which is able to emit an action to its environment; and (iii) *internal event*, which can internally process an action. The union of in and out events is called *observable events*, which contain all events exposed in the environment.

Function is introduced to specify procedure call-like synchronous communications. A function is actually an ordered pair of tightly-coupled events. The first event denotes *invocation*, and the second event denotes *reaction*. According to different methods of handling its invocation and reaction, a function can be *provided*, *required* and *internal*. For a provided function, its invocation is an in event and its reaction is an out event. That means the function is initially expecting a request issued from its partner, and reacts by emitting a result. On the other hand, a required function first sends a request via an out event, and absorbs a response through its subsequent in event. For an internal function, both its invocation and reaction are internal events. Similarly, the union of provided and required functions is called *observable functions*.

Notice that there is a characteristic in common for both in events and provided functions at runtime; that is, their threads only exist while being invoked by their partners. Therefore, we describe, the union of in events and provided functions, as *passive services*. In a similar vein, the union of out events and required functions can be called *active services*, as they are able to initialise autonomous threads of control at runtime. As well, we call the union of internal events and functions *internal services*. Note that we do not intend to model internal services, they only arise from a component hierarchical design. That is, when a component encapsulates several sub-components, some of whose services become internal, because external entities are no longer able to get in contact with them, they become isolated. Furthermore, the union of observable events and observable functions are often referred to as *observable services*.

The *interface* is a contract of the set of functionalities on components. In practice, the interface is an abstract definition of a collection of one or more observable services, and zero or more attributes, ideally one that defines a cohesive set of services. An interface may have multiple passive services through which it makes its services available and multiple active services through which it issues requests to other components in order to deliver its offered services. Note that a passive service may be related to more than one active service. We do not directly consider attributes in our component model because each of them can be safely replaced by a pair of retrieving and updating functions [OMG04b].
A port represents an interaction point between a component and its environment. Interfaces deliver its services to the environment and expect some services from the environment at the associated interaction point. A port that possesses a cohesive set of interfaces which often exhibits one module of services exposed to component’s environment, is often referred to as a service port, we refer readers to UML 2.0 Superstructure FTF Convenience Document [OMG04a] for details.

A primary component is a modular unit with one or more service ports that are replaceable within the environment.

**Example 3.1** In Figure 3.1, a primary component ComA comprises of two ports: PotA and PotB, where the interface IntA associated with PotA and the interface IntB associated with PotB. IntA exposes the provided function FunA, the required function FunB and the in event EvtC. Similarly, IntB exhibits the provided function FunD, the required function FunE and the out event EvtF.

### 3.2 Connectors

Several years ago, when Allen and Garlan [AG97, AG98] introduced the first-class connector concept in their Wright ADL, there was a significant disagreement among researchers as to whether connectors are really necessary or not. The problem of classical ADLs is that connections are very similar to compositions and therefore the question as to whether it is necessary to have two different abstractions naturally arise. Analysing both, the component is a piece of software design focusing on computational service delivery, while the connector is more like a channel accommodating component communications. One perceived advantage of this design should be the separation of concerns in reasoning about system behaviours. The connectors encapsulate the computational paths, while the components encapsulate computations. This separation of concerns should make it more tractable and hence practicable to reason about system behaviours by reasoning about communication and computation separately. Therefore, introducing the connectors to the CBS architecture is a milestone towards delivering complex, reusable and high-quality CBSs.

In Allen and Garlan’s model, components have to link to connectors in order to interact with other components in the CBS. Connectors are independent communication units at the same
abstraction level as components. Connectors deal with component collaborations, which are constituted by glue, specifying interactions, and a set of roles, restricting the components which the connector will link together. Namely, roles describe what kinds of components are expected in the interaction, whereas the glue describes how they interact with each other. Also, we can consider that a role provides a specification that determines the obligations of each component participating in the interaction, and the glue gives the interaction logics, i.e., the sequence constraints of inter-component communications. This arrangement distinguishes between computation relationships and communication relationships of software modules or components.

Allen and Garlan's glue plays an active role in coordinating roles, and their roles act as a specification for ports: provided that the ports satisfy the role specifications, they will stand in for those roles in the running system. The ports define the actual behaviour of the components and eventually interact with the glue while associating with the roles. It can be argued that Allen and Garlan's connectors encapsulate communications, but do not encapsulate control: they pass the control back to the components in their communications. That is, components have to invoke other's services and manage their returns via connectors. Consequently, in terms of control, components are not loosely-coupled, and control and computation are mixed up.

In order to minimise component coupling, and to maximise separation of communication from computation, our connectors are proposed to encapsulate control between components. The control flow of components is totally encapsulated within the connectors, i.e., they originate and coordinate all controls. This means that components need not to initiate interactions with other components via these connectors; rather, this is done by the connectors.

With respect to this intuition, in our connector, the glue structures the flow control in a sequence of interactions among expected components; we modularise the glue into separate roles and each role forms the kind of component that can participate in collaborations. This hints that, roles only contain localised interaction information, as a matter of fact, which is implicitly defined in the glue. Ports are used to specify the component behaviour in general. Components remain unaware of the connectors with which they are participating and the components with whom they are interacting. Once ports attached to the roles, they behave in the same way as the roles would behave.

From an architectural point of view, one major advantage of this design is that, as practical systems have quite sophisticated rules about component interactions, our connectors capture all design decisions of interaction logics. These logics are no longer spread over all communicating components and this therefore makes it easier to exploit them for analysis and maintenance. For CBS maintenance and evolution, such a complete encapsulation in connectors could also make it simpler to manage changes in the components and changes in the connectors separately.
Additionally, connectors explicitly prohibit behaviours outside the range of those defined by the glues to occur inside connectors, so that our connectors are conservative [AG98] in nature. In contrast, in order to support this notion, Allen and Garlan’s connectors do require extra effort.

Further, by means of regarding roles as a group of localised glue and restricting ports to roles, our model eliminates a layer of the interaction between glue and roles, as existed in Allen and Garlan’s model [AG98].

It has been shown the significant difference between Allen and Garlan’s model and our model in that: in Allen and Garlan’s model, components encapsulate computation and control, and connectors only encapsulate communication. Whereas in our model, components only encapsulate computation, and connectors encapsulate control and communication. Such differences result from the different emphasis on the reusability of CBS: Allen and Garlan’s approach presents a connector as a fixed set of mechanisms (for instance pipe/filter systems), and they are concentrating on promoting the mobility of connectors to adapt various components; while our approach represents a connector as a set of specific user-defined mechanisms, and this helps us to maximise the reusability of components in different contexts.

**Example 3.2** For illustration, in Figure 3.2, the component *ComA* linked to the connector *ConA*, by means of attaching the services *FunA*, *FunB* and *EvtC* of the interface *IntA* on the port *PotA* to the services *FunA*, *FunB* and *EvtC* of the role *RoleA* on *ConA*, respectively. In the glue, *FunA*, *FunB* and *EvtC* are binding with *FunG*, *FunH* and *Evtl* in order to exchange services. *RoleA* and *RoleB* specify the expected participants of *ConA*; they are thus attaching to corresponding components. Glue is the actual entity of describing interaction workflow, which comprises a collection of bindings between events and functions specified in roles. Although the connector in this example is binary, in general a connector can have more than two roles.

![Figure 3.2 A Component with a Connector](image)
As shown in the above example, we say that the binding is a primary manner of service communication. Services interacting in glue need binding methods to connect through. The binding method usually requires that two services must be bound by the match. By matched services we mean that, at a high enough level of abstraction, the service provided by one component is equivalent to the service required by the other. More especially in this model, we consider two services are matching if the polarities of services are pairwise opposite but corresponding actions of services are the same. Traditionally, the binding method only applies between a pair of events or a pair of functions, such as the approaches described in [LV95, OMG04a, PV02, MDE’95, YS97]. In this model, we further permit binding between an event and a function, called partial binding, and we refer to the traditional binding as full binding. For example, a partial binding could be an event invoking a function without expecting its result, or a function’s reaction notifying an event, and so on. In addition to traditional event-to-event and function-to-function bindings, partial binding supports event-to-function bindings, this pattern of cross-boundary interactions further increases the portability and interoperability of components. We will formally define this concept in Section 4.2.

Although all bindings in the example above are binary, in general they are able to have more than two parties. With the intention of naturally achieving the maximum concurrency in multi-party bindings, for dealing with the case where one active service requests multiple passive services, the interaction process will be finished when all responses (if any) have been received from the notified services. On the other hand, to deal with the case when one passive service serves multiple (i.e., separate but concurrent) active services, a mutual exclusion mechanism will be applied for each atomic service interaction. This mechanism is used especially for the case of one provided function binding with multiple required functions. In that context, once the provided function accepts a request, its invocation part will become frozen, so that later-coming requests will be blocked until the process of the current request has finished and subsequently, the invocation has been restored.

Finally, for the sake of brevity, our CBS model does not consider composed connectors, i.e., connectors are always the primary entities in our model.

3.3 Composed Components

In order to support the visibility and modularity restriction employed in CBS, components can be composed. A primitive component is a black-box entity which does not possess any other components and connectors, such as the ComA in Example 3.1. A composed component, on the other hand, is a composition structure that contains a coherent group of sub-components linked by connectors (if any), and a set of proxy ports.
Chapter 3 A Generic Component-based System Model

The services from a component’s internal structure can access external entities via proxy ports, which differ from service ports in that: (i) proxy ports are used to implement the component but are not part of the essential externally-observable functionality of the component and can, therefore, be altered or deleted along with the internal structure of the component in question; and (ii) proxy ports could act as interceptors to allow the (service/proxy) ports on sub-components to interact with external entities from higher hierarchies. The elaborate specification of communication over ports can be described as complex as being filtered, merged, modified; or as simple as to being routed, all depending on the contents specified in proxy ports. For simplicity, we only consider the proxy ports capable of routing communications in this study. This eases our later rigorous behaviour analysis of proxy ports.

In this model, internal ports can link to proxy ports by means of port mapping, which is another form of communication. In this, it is noteworthy that: (i) just as its name implies, only internal service/proxy ports can map to the proxy port; (ii) all services on the mapping port will be delegated to the mapped port, so incoming signals can be faithfully transmitted to the opposite side. A similar idea can be found in the hierarchical component of Fractal [BCL’04]. In this, the required and provided interfaces of a composite component are required to correspond to the interfaces of a sub-component.

Example 3.3 Consider the component ComC as sketched in Figure 3.3. In this context, ComA and ComB are primitive components and ComC is composed of them. ConA is an internal connector of ComC. Furthermore, PotD is a proxy port, which simply represents the internal service port PotB to the upper environment. At this proxy port, every request emitted from the sub-component ComA will be forwarded to the environment. As well, every signal absorbed by the proxy port PotD will be forwarded to the mapping service port PotB on the sub-component.

![Figure 3.3 A Composed Component](image-url)
As a result, if two sub-components are linked via a connector, such a connector also becomes internal in the composed component. From the composed component’s point of view, all services specified in its internal structure are internal services, but may become observable when focusing our perspective onto internal structures.

In order to achieve consistency in the behavioural specification of different layers of the hierarchical composed component, we impose an assumption that

*The ports on the same sub-components must be mapped to the same proxy ports.*

This assumption excludes the case in which the behaviour of a sub-component scatters across different proxy ports of the super component. This usually results in inconsistencies between the behaviours specified on the two levels. Detailed discussion on this point can be found in Section 6.2.

Based on this hierarchical design, every component and connector may be a part of a composed component. Notice that our model is recursive: a composed component can itself appear in the content of another. The roots of composed components are the ones without any proxy ports, which is normally called a *component-based system* (CBS).

### 3.4 The Lifecycle of Component-based Systems

To produce a CBS from the initial concept to the final product, we need to follow a certain development sequence. This sequence is called *CBS lifecycle*. Three stages of a CBS lifecycle are generally identified: the *design* phase, the *deployment* phase and the *runtime* phase.

- **Design Phase**: in this stage, components are designed and constructed in source code, and deposited into a repository. In the repository, components are types (like classes in the UML terminology) that cannot execute in isolation. Nevertheless, (unlike UML classes) their instances (like objects in the UML terminology) can be integrated into another component type. The constructed components, including composed ones, have to be catalogued and stored in a flat repository in such a way that they can be retrieved later, as and when needed.

- **Deployment Phase**: in this stage, component source codes are retrieved from the repository, and then compiled into executables and deployed to the target container (or platform).

- **Runtime Phase**: under the chosen runtime environment, component executables are instantiated with initial configuration settings, and they are ready to execute.
In the design phase, primary components are the encapsulated pieces of reuse that clearly state the services they offer and the requirements they need to fulfil their services. In order to achieve a greater degree of reusability, the specific binding knowledge is taken out of components, and components are designed independently of each other. New primary component types are constructed and then deposited in a repository for later reuse.

Connectors structure the patterns of sequencing of service exchanges. Connector types are always parameterised over all architecture-specific information (particularly expecting components) so as to keep them generic in principle and applicable in a variety of contexts. Connector types cannot directly involve composition, the reason for that is they and their parameters are abstract and generic, but in the reality, every interaction in the CBS is concrete and specialised for a specific context. This context is usually determined by a collection of instances of pre-defined component types. Therefore, any CBS interaction should be represented by a connector instance, which is simply created by the connector type with the specific component instances participating in this cooperation. Also, the actual component types of the components connected by the connector instance are determined at this time.

To construct a composed component, component designers need not start from scratch. Instead, they can retrieve appropriate components and connectors from the repository, compose them and then deposit the designed construct back to the repository. Any internal structure of the composed component type is described in terms of component and connector instances. It is noteworthy that any composition in the design stage is only between component and connector instances (rather than their types).

Our component repository is flat. This means that it contains both primary and composed constituents. For example, although component instances are encapsulated in a composed component, corresponding types are not. Therefore, it is possible to construct a second composed component with the same primary components in a different structure.

In the deployment phase, no new composition is possible. Composite types are retrieved from the repository and compiled into binaries, and released to the runtime environment. After that, final users are able to use the application in the runtime phase.

Besides being an explicit design-time entity, a connector should also remain as an explicit entity in an application’s runtime. Since every connector type has its concrete representative in the implementation, it is not difficult to localise the modifications that must be performed upon the code of the application every time the application’s deployment changes. No modifications should affect the application’s business logic concentrated in components.
3.5 Conclusion

Existing component models like [MDE95, AG97, BCL04, YS97, OMG02, OMG04a] and more generally developers of the component approach have proved their interest in designing, implementing and maintaining constantly evolving software systems. Component models allow an application to be designed as a group of interconnected components accessible through well-defined interfaces, the whole forming a potentially complex architecture.

Contracts [HHG90] are design formalisms used to express cooperation between objects, but they describe rather than enforce the constraints on the message exchange between participants. Also, many design patterns [GHJ94] can be expressed using connectors – in this way retaining and enforcing design decisions at the implementation level.

The problem of providing a language construct to express and also take charge of interaction relationships has been approached from a variety of angles. Pintado [Pin93] proposes gluons to mediate object collaborations. His approach emphasises collaborations between objects as client-server protocols, and does not allow for the specification of more general patterns of collaboration, in particular for ones where no server is required. Similarly, components in Darwin [MDE95] interact through required and provided services, so that the unit of connection is basically a binding of services of two components. In the Composition-Filters approach of SINA [AWB94], Abstract Communication Types are proposed for enforcing invariant behaviour among objects. However, object interfaces must be modified before an object can engage in a new kind of interaction – an impediment to the reuse of components in new contexts.

Yellin and Strom [YS97] described synchronous communication between two component interfaces using finite state machines and formally introduced the notion of an adaptor, as a software entity capable of bridging components with defined temporal interfaces that are semantically compatible but syntactically incompatible. When two components are functionally compatible, but their interface protocols are not compatible, adaptors are used to translate the interfaces. Adaptors are connector-like constructs but in the context of augmented interfaces they require a limited expressive power and can represent solely two-party relationships. Further, component hierarchical design was not considered there.

Sullivan and Notkin [SN92] separate connectors from the components at the implementation level by providing mediators, proposed to facilitate tool integration. Our connectors are close to mediators in the sense that they are based on an implicit message-passing mechanism which allows components to remain truly independent [SG96]. In contrast to mediators, however, our connectors not only can reply messages, but can forbid message delivery, redirect messages and so forth.
Frolund and Agha [FA93] propose synchronizers for multi-object coordination in a concurrent language. Synchronizers are similar to our connectors but there is a main difference between the two approaches. A synchronizer only updates its own state on receiving a message from a participant, but it cannot itself send messages to its participants and alter their states. A synchronizer only coordinates communication. Our connector is, in this sense, more active – it can enforce state changes in the participants, hence additionally coordinates control.

In the literature, the common problems of providing a construct for explicitly specifying interactions between components can be summarised as follows.

- **Inability to localise interaction information:** loss of design information. Some of the design of the application is lost during the implementation since we cannot localise information about interactions. This is most evident when we try to re-engineer an application. Program codes contain little of the interaction relationships identified at design time, making reverse engineering a much more difficult task.

- **Mixing of concerns:** impediment to reuse. Logically, components should have an identity independent of the different interactions in which they can engage. When no connector construct is available, component behaviour includes the connector behaviour, making for less reusable components. Providing a connector entity at implementation level allows abstraction and factorisation of all the information about a connection and also allows for the reuse of typical interaction relationships.

To address the first problem, interaction relationships should be represented by an explicit construct, as in [AG97], taking the binding knowledge out of the components. This is in contrast to approaches of enriching component interfaces with protocols which capture interaction information [YS94], and to the approach of Darwin [MDE95], where the connection of components is defined as the binding of services and is found inside the definition of a composite component. To address the second problem, components and connectors should be independent of each other – more specifically, although connectors must specify the kinds of components which they connect, components should not be aware of the relationship in which they may engage. This is in contrast to the component-filters approach of [AWB94], where object interfaces must be modified to allow them to engage in new kinds of interactions, and to the approach of gluons [Pin93], where objects must address the mediating gluon in order to collaborate with each other.

Our CBS model is proposed to overcome these common issues. As many practitioners find that it is more intuitive to describe inter-component behaviours outside components (for example [Car95]), and being enlightened by Allen and Garlan’s architectural connection [AG97], we define first-class connectors, as standalone design constructs, to specify a set of user-defined component communication mechanisms. Our CBS is modelled as a component-port-role-
connector-role-component sequence. This pattern makes explicit the interaction relationships between components using the abstraction of connector, clearly separating communication/interaction from components and making it much easier to design and build off-the-shelf components.

Completely treating our connectors as independent entities separate from components at the design time, deployment time and runtime helps us to solve the first problem, since design-time connectors are required to map to explicit runtime entities. The interaction information is adequately maintained at the implementation level and is easy to trace and locate. This design also facilitates the localisation of all the necessary modifications that must be made to the application’s code whenever the deployment of the application changes.

Based on Allen and Garlan’s approach, which takes communication out of components, our approach pushes forward to take control out of components. Our belief is that this significantly promotes the reusability and maintainability of components as well as connectors, allows us to reason about component cooperation in isolation, and makes our CBS clearer and simpler to implement, and also solves the second problem above. We can see that our connector instances play a dominant role in component communication. It not only explicitly captures the user-defined communication mechanisms, but actually coordinates and governs component interactions. A typical example could be a connector linking all components to perform the initialisation. Putting it another way, the attached components will be controlled (more specially, restricted) by the connector. They must follow the connector’s collaboration logics to communicate with others. We find it can be more elegant to analyse connector behaviour in a context-free manner, rather than having to provide a specific architectural instance. However, such a provision is necessary in Allen and Garlan’s approach. Because components will eventually stand in for those connectors in the running system, analysing connector behaviour needs to refer to the actual components participating in the connection. It is proving difficult to maintain design-time interaction logics in runtime, making the effort of designing communication mechanisms in connectors somewhat useless, and most importantly mixes up computation and control in components.

In our proposed approach, the separation of communication and control from computation is maintained. That is to say, our connectors not only describe component relationships, but also enforce them to change behavioural states. Our connectors initiate and coordinate services in components and handle their results. Thus, they encapsulate communication and determine control flow. At this point our approach is in line with Lau, Elizondo and Wang’s Exogenous Connector [LEW05], Ducasse and Richner’s executable connector [DR97] and Arbab’s exogenous coordination in the coordination language for concurrent computation [Arb96].
Kung-Kiu Lau et al’s Exogenous Connector [LEW05] is similar to our connector in terms that both approaches separate computation from control meaning that control flow does not originate from components, but from connectors. One main difference is that exogenous connectors are designed for deployment-time composition (i.e., connectors link component binaries), while our connectors are defined for design-time composition (i.e., connectors link component source codes). The Java [Sun]’s reflection technology is used to illustrate the feasibility of their model.

Ducasse and Richner’s executable connectors [DR97] are specified by connector templates, which describe all the information representing the connection between the participants by specifying in a set of rules how message exchanges influence the behaviour of the participants. The executable connectors observe and control the communication between participants and can also enforce state changes in the participants. Therefore, connectors can be seen as a kind of higher-level glue for synchronising and composing components. It can be seen that the main contribution of this work is to provide a descriptive and executable notation for connectors and thus enable the localisation of information about interaction of components at the level of implementation.

In addition to providing descriptive notations for the architecture of our CBS, we are also interested in how to formalise it and how to formally describe the behaviour of this model. With this concern, a formal specification of intra-component behaviour is required, and this mathematical framework will be explored in the next chapter.
Chapter 4

Formalisation of Component-based Systems

In this chapter, we introduce a formal model and a behaviour language for the CBS model introduced in Chapter 3. We adopt the initial algebra approach throughout this chapter, since it is well known as a natural way for defining algebraic languages. A brief introduction to the initial algebra approach is given in Section 4.1. In our approach, we base our behaviour language on component services, which are classified into events (asynchronous services) and functions (synchronous services) in Section 4.2. We further associate service names to each service and then define a component signature, which is the static specification of our formal model and provides a universe for the behaviour specification of the formal model. The component behaviour is captured in the time-slot model, as a set of service occurrences over a sequence of time slots. Following the initial algebra approach, the time-slot model introduced in Section 4.3 gives the semantic domain or interpretation of our behaviour language. The creation and manipulation of the time-slot model are facilitated by a set of operations defined in Section 4.4. We introduce an extended regular-like language in Section 4.5, component protocol, as the syntax of our behaviour language. Every expression or term in the component protocol may be interpreted in only one way in the time-slot model. Then, we give a formal model for our CBS in Section 4.6. An industrial case study will be demonstrated in Section 4.7. In Section 4.8, we end this chapter with a discussion of this approach compared with others in the literature.

4.1 Introduction

To formally describe a CBS, we need to formalise its behaviour in a specification language. The initial algebra approach [GT74, Hen88] from ADJ group is well-known as a natural way to give the syntax and semantics of process algebras and we will adopt this approach throughout this chapter. Now, we give a brief introduction to this approach.

We start off by introducing the notions of a signature. A signature is a set of formal functional symbols or combinators, normally denoted as $\Sigma$. Each function symbol has associated with it an arity which gives the number of arguments of the function it represents. Note that the function symbol whose arity is 0 is a constant. Formally the arity of a signature is a mapping, arity:
$\Sigma \rightarrow \mathbb{N}$. With each symbol $f$ in $\Sigma$, it associates its arity, $\text{arity}(f)$, a natural number. We use $\Sigma_n$ to denote the set of function symbols in $\Sigma$ of arity $n$.

**Definition 4.1.1** If $\Sigma$ is a signature, a $\Sigma$-algebra is a pair $(A, \Sigma_A)$ where

- $A$ is a set, called the carrier
- $\Sigma_A$ is a set of function symbols $\{f_A : f \in \Sigma\}$

such that if $\text{arity}(f) = n$ then $f_A$ is a function from $A^n \rightarrow A$.

A $\Sigma$-algebra consists of a set $A$ and an interpretation over $A$ of every function symbol in $\Sigma$. Naturally, a given signature can have many different interpretations, even different interpretations over the same carrier. One particular interpretation has, as a carrier, the set of terms or words which can be constructed using the function symbols; this particularly important $\Sigma$-algebra is called the *term algebra* for $\Sigma$. Term algebras will play a central role in what follows. Their carriers consist of sequences of symbols or strings, called *terms*, which are constructed using the function symbols in $\Sigma$.

**Definition 4.1.2** Let $T_\Sigma$ the set of terms over $\Sigma$, be the least set of strings which satisfies

a) if $f \in \Sigma$ has arity 0 then the string consisting of the symbol $f$ is in $T_\Sigma$

b) if $f \in \Sigma$ has arity $k > 0$ then the string of the form $f(t_1, ..., t_k)$ is in $T_\Sigma$, whenever $t_1, ..., t_k$ are strings in $T_\Sigma$

Thus the elements in $T_\Sigma$ are strings consisting of the symbols ‘(’, ‘)’ and ‘,’ together with the symbols from $\Sigma$, which can be constructed using the rules (a) and (b) above. Note that if $\Sigma$ contains no constants, then $T_\Sigma$ is empty. $T_\Sigma$ itself may be regarded as the carrier of a $\Sigma$-algebra in which every constant $c$ is interpreted as $c$ (i.e., $c_{T_\Sigma} = c$) and for every $f$ in $\Sigma$ of arity $k, k > 0$, then $f_{T_\Sigma}(t_1, ..., t_k) = f(t_1, ..., t_k)$.

The recursive nature of $T_\Sigma$ gives a very powerful proof method for deriving properties of terms. To show that the property $P$ holds of all the terms in $T_\Sigma$ it is sufficient –

- to prove $P$ holds of all constant symbols in $\Sigma$
- assuming $P$ holds of the terms $t_1, ..., t_k$, to prove $P$ holds of the term $f(t_1, ..., t_k)$ for every $f$ in $\Sigma$ of arity $k, k > 0$

This is called *structural induction* as the induction is actually on the syntactic structure of the terms. One uses structural induction to prove properties of elements of $T_\Sigma$. For example, we can
show that if a function $\sigma_A : T \rightarrow A$, where $A$ is a $\Sigma$-algebra, and $\sigma_A$ is a partial function such that,

- for every constant $c$ in $T$, $\sigma_A(c) = c_A$, and
- for every function symbol $f$ in $T$ of arity $k$: $\sigma_A(f(t_1, ..., t_k)) = f_A(\sigma_A(t_1), ..., \sigma_A(t_k))$

Then the structural induction shows that $\sigma_A$ is actually total and indeed a homomorphism, which is the most fundamental property of term algebras. This establishes part of the following result. Uniqueness is proved similarly.

**Theorem 4.1.1** For every $(A, \Sigma_A)$ there exists a unique $\Sigma$-homomorphism $\sigma_A : T \rightarrow A$.

If we view $T$ as the syntax of a language and a $\Sigma$-algebra $(A, \Sigma_A)$ as a semantic domain or interpretation, then Theorem 4.1.1 states that every expression or term in the language has a unique meaning as in $(A, \Sigma_A)$: thus, there is only one way to interpret any expression in the semantic domain. There is also a sense in which it can be interpreted as saying that $T$ is the ‘least’ $\Sigma$-algebra. In general, a $\Sigma$-algebra $(A, \Sigma_A)$ makes identifications between terms, i.e., $(A, \Sigma_A)$ identifies two terms $t_1, t_2$ if $\sigma_A(t_1) = \sigma_A(t_2)$. Then $T$ is the $\Sigma$-algebra which makes the least number of identifications. In practice it makes none. To see this, we apply Theorem 4.1.1 with $A$ equal to $T$; there exists a unique homomorphism $\sigma_T$ from $T$ to itself. Since the identity function is a $\Sigma$-homomorphism it follows that $\sigma_T$ must be the identity, i.e. $\sigma_T(t_1) = \sigma_T(t_2)$ if and only if $t_1$ is syntactically the same as $t_2$.

**Definition 4.1.3** Let $\mathcal{J}$ be a class of $\Sigma$-algebras. Then a $\Sigma$-algebra $I$ in $\mathcal{J}$ is initial in $\mathcal{J}$ if for every $\Sigma$-algebra $J$ in $\mathcal{J}$ there exists a unique $\Sigma$-homomorphism from $I$ to $J$.

Theorem 4.1.1 can now be rephrased to read: $T$ is initial in the class of all $\Sigma$-algebras. Hence, it is the key concept of the term initial algebra semantics.

**Definition 4.1.4** Given a $\Sigma$-algebra $A$, then there exists a congruence on $T$, $\equiv_A$, given by $t_1 \equiv_A t_2 \iff \sigma_A(t_1) = \sigma_A(t_2)$.

$\equiv_A$ is simply an equivalence relation between elements of $T$. We can actually construct a ‘quotient algebra’, $T_\equiv / \equiv_A$, denotes the set of equivalence classes induced by $\equiv_A$. There exists a natural injection mapping $\sigma_A : T \rightarrow T_\equiv / \equiv_A$, which is the initial algebra homomorphism.

---

1 The proof can be found in page 26 of Hennessy’s book [Hen88].
4.2 Component Signature

A component in our model is understood as an encapsulated computational entity that is capable of providing services to its environment and requiring services from its environment according to its contract (also called interfaces). At the specification level, a service is an atomic structural entity, which exhibits a functionality of its component, or lists a dependency of its component. A component without exposing any services is considered as a component-based system, which is a top-level component. From the ‘black-box’ view (or external view), CBS is a chunk of software that can complete its tasks independently without interacting with outside. From the ‘white-box’ view (or internal view), CBS could be regarded as a service interaction architecture in which sub-components are linking together for exchanging services.

The functionality of service is characterised by its actions. Adopting [OMG04a]’s event definition, “the specification of a significant occurrence that has a location in time and space and can cause the execution of an associated behaviour”, an action in this study is understood as an atomic computational unit. Services manipulate actions in the following ways: a service can emit an action to its environment; absorb an action from its environment and process action(s) internally.

Following most state-of-the-art component models (such as [OMG04a, JH04]) and also earlier work like [LV95], the services in our model can be specified as asynchronous services (called events) or synchronised services (called functions).

The event is a loosely-coupled service adopting an asynchronous message-based interaction mode, for example MOM as discussed by Steve [Ste95]. An event normally handles one action.

**Definition 4.2.1** Suppose we have a denumerable set $ACT$ of actions, events are actions equipped with polarity $\{?,?,!\}$, which denotes the way of handling actions: absorbing, emitting and internal. We define $E = \{?,?,!\} \times ACT$ to be the set of all events.

**Example 4.2.1** Considering a ticket vending machine, we can define a coin-inserting event $e_{ci} = \text{insertCoins}$, which means a customer inserts coins. In addition, we define a receipt-collecting event $e_{rc} = \text{printReceipt}$, means that the customer is expecting a printed receipt.

According to the polarity, events could be partitioned into subsets. An event can absorb an incoming action from the environment, called in event; emit an action to the environment, called out event; and process an internal action, called internal event. The in and out events are regarded as observable events.

**Definition 4.2.2** We define the partition of all events $E$ to be a collection of subsets,

\[ \downarrow E = \{(\rho,a) \in E : \rho = ?\}, \text{ is the set of all in events} \]
Chapter 4 Formalisation of Component-based Systems

- \( \uparrow E = \{(\rho, a) \in E : \rho = \uparrow\} \), is the set of all out events
- \( \tau E = \{(\rho, a) \in E : \rho = \tau\} \), is the set of all internal events
- \( E_{\text{obs}} = \downarrow E \cup \uparrow E \), is the set of all external event tokens

Following Example 4.2.1, we can see that \( e_{\text{ci}} \in \uparrow E \) and \( e_{\text{rc}} \in \downarrow E \). Also, \( \{e_{\text{ci}}\} \cup \{e_{\text{rc}}\} \subseteq E_{\text{obs}} \subseteq E \). The new symbols \( \{\downarrow, \uparrow\} \) are introduced solely for the sake of uniformity with the partition of functions defined later. By convention, an event \((\rho, a)\) can be written as \( \rho a \) for short. The polarity and action of an event give rise to the equality and match relations over events.

**Definition 4.2.3** Let \((\rho, a)\) and \((\sigma, b)\) be events, then

a) \( (\rho, a) = (\sigma, b) \Leftrightarrow \rho = \sigma \land a = b \), where \( = \) is the equality over events, it is reflexive, symmetric and transitive

b) \( (\rho, a) \equiv (\sigma, b) \Leftrightarrow \rho \neq \sigma \land a = b \), where \( \equiv \) is the match relation over events, it is irreflexive, symmetric and intransitive

**Remark 4.2.1** Let \((\rho, a), (\sigma, b), (\phi, c) \in E\), then \((\rho, a) \equiv (\sigma, b) \land (\sigma, b) \equiv (\phi, c) \Leftrightarrow (\rho, a) = (\phi, c)\).

The function is another form of service, which is a tightly-coupled service adopting a synchronous request-respond-based interaction mode, such as RPC [Rao95], procedure call-like. A function is in practice a pair of tightly-coupled events. It becomes clearer when defining an operation call in a message sequence chart, in which an operation call is normally decomposed into a send event followed by a receive event. Hence, a function can be defined as an ordered pair with first event invocation and second event reaction. By using events, we can ensure that the invocation and reaction of functions are atomic. We further make assumptions that, (i) the invocation happened strictly before the reaction; (ii) the underlying machine is infinitely fast and, hence, internal processes in-between the invocation and reaction are instantaneous; (iii) during reaction, the invocation is frozen. The functions are not entirely based upon the synchrony hypothesis as proposed by Benveniste and Berry [BB91]. Instead, it distinguishes the beginning of a function from the termination and assumes that the processes in-between are straightaway.

**Definition 4.2.4** Let \( \downarrow E, \uparrow E, \tau E \subseteq E \), we define a set \( F \) of all functions to be \( F = (\downarrow E \times \uparrow E) \cup (\uparrow E \times \downarrow E) \cup (\tau E \times \tau E) \).

It is intuitive that functions could be partitioned into subsets according to the polarities of invocation and reaction. That is, functions can be provided, required, or internal. For a provided function, its invocation must be an in event and its reaction must be an out event. In this way, the function is initially expecting a request (action), and reacts by emitting a result (action).
Conversely, for the required function, its invocation must be an out event and its reaction must be an in event. So the function always emits a request at first, followed by absorbing a result. For the internal functions, both the invocation and reaction must be internal events. Apart from these kinds of pairs, other cases are disqualified from being considered as a function. These cases are: (i) a pair of in/out events together with an internal event, and (ii) a pair of two events with the same polarity. For (i), from the environment viewpoint either its invocation or reaction is missing. It is more like a single event. For (ii), it cannot present a procedure call-like synchronous interaction. It is more like two consecutive but separate events. Therefore, it is more reasonable to exclude these cases in the definition of functions.

**Definition 4.2.5** We define the partition of all functions $F$ to be a collection of subsets,

- $\downarrow F = \downarrow E \times \uparrow E$, is the set of all provided functions
- $\uparrow F = \uparrow E \times \downarrow E$, is the set of all required functions
- $\tau F = \tau E \times \tau E$, is the set of all internal functions, and
- $F_{\text{obs}} = \downarrow F \cup \uparrow F$, is the set of all observable functions

**Example 4.2.2** In a ticket vending machine, we can define a ticket-selling function $f_{\text{ts}} = (\text{?insertCoins, !produceTicket})$, which is initially waiting for coins and will react by producing a ticket. Similarly, we define a receipt-printing function $f_{\text{rp}} = (\text{!produceTicket, !printReceipt})$, which is initially observing the $\text{produceTicket}$ signal. It will print out a receipt once the signal has been captured. We in addition define two pairs of functions: a transaction-printing function $f_{\text{tp}} = (\text{?recordTrans, !printTrans})$ and a sell-recording function $f_{\text{sr}} = (\text{!recordTrans, !printTrans})$. The pair exchanges actions to print out transaction data; a sell-monitoring function $f_{\text{sm}} = (\text{?startMon, !endMon})$ and a system-admin function $f_{\text{sa}} = (\text{!startMon, !endMon})$. This pair exchanges actions to monitor each transaction in the system. Furthermore, $f_{\text{ts}}, f_{\text{rp}}, f_{\text{tp}}, f_{\text{sm}}, f_{\text{sr}}, f_{\text{sa}} \in F$ and $f_{\text{ts}}, f_{\text{rp}}, f_{\text{tp}}, f_{\text{sm}}, f_{\text{sr}}, f_{\text{sa}} \in F$.

**Definition 4.2.6** Let a function $f$ be $(e, e')$, then

- $\text{INV}(f) = e$, where function $f$ is associated with its invocation $e$
- $\text{REC}(f) = e'$, where function $f$ is associated with its reaction $e'$

Then, we can define the relations over functions based on those relations over events. Consequently, their algebraic properties inherit those of the event relations.

**Definition 4.2.7** Let $f, f' \in F$, then
Chapter 4 Formalisation of Component-based Systems

\[ a) \quad f = f' \iff \text{INV}(f) = \text{INV}(f') \land\text{REC}(f) = \text{REC}(f') \]

\[ b) \quad f \equiv f' \iff \text{INV}(f) \equiv \text{INV}(f') \land\text{REC}(f) \equiv \text{REC}(f') \]

**Definition 4.2.8** Assume we have a set \( E \) of all events and a set \( F \) of all functions, we define a set \( S \) of all services to be \( S = E \cup F \).

Based on the partition of events and functions, we can define the partition of services. During service interaction, we view the services whose threads only exist while being invoked by their counterparts as *passive services*. They are always prefixed by the symbol \( \downarrow \). On the other hand, we call the services that are able to initialise autonomous threads of control as *active services*, which are normally equipped with the symbol \( \uparrow \). Finally, we call those services prefixed by the symbol \( \tau \), *internal services*. Notice that we do not intend to model internal services, which only arise from hierarchical encapsulations of components. That is to say, while composed components are constructed, some services may become their internal structures. Consequently, these services will be isolated from the environments of composed components and therefore become internal services. In contrast, we call the union of all active services and passive services, *observable services*, since they are usually exposed in component environments.

**Definition 4.2.9** We define the partition of all services \( S \) to be a collection of subsets,

- \( \downarrow S = \downarrow E \cup \downarrow F \), is called the set of all passive services
- \( \uparrow S = \uparrow E \cup \uparrow F \), is called the set of all active services
- \( \tau S = \tau E \cup \tau F \), is called the set of all internal services, and
- \( S_{\text{obs}} = E_{\text{obs}} \cup F_{\text{obs}} = \downarrow S \cup \uparrow S \), is called the set of all observable services

Based on the relations over events and functions, we further introduce a cross-boundary relation. We notice that for the previous match relations over events and functions, they require that all parts of both sides are pairwise matching. However, sometimes we can find some reasonable cases, such as two functions may only match one pair, and an event matches one part of a function. This is where the *partial match* relation comes into play and we call the preceding match relations *full match*.

**Definition 4.2.10** Let \( f, f' \in F \) and \( e \in E \), we define the partial match relation \( \succ \) over services by

\[ a) \quad e \succ f \iff e \equiv \text{INV}(f) \]

\[ b) \quad f \succ e \iff \text{REC}(f) \equiv e \]

\[ c) \quad f \succ f' \iff \text{REC}(f) \equiv \text{INV}(f') \]
The full and partial match relations denote all kinds of valid service interactions in this study of CBS, under the synchrony principle that services only respond to the same action as they are expecting. Otherwise they are waiting forever. In other words, the emitting action can be immediately omitted if no matched events are ready to absorb it (we will return to this point in the assumptions of the time-slot model in Section 4.3), but the absorbing action cannot be passed over if there are no incoming actions from the matched events. With this concern, we consider only three cases above are valid for service partial interactions. In the first case (a), an event can simply request a provided function without expecting the result. In Example 4.2.1 and 4.2.2, \( e_{cd} \succeq f_{ts} \), where the reaction of \( f_{ts} \) is omitted, we can simply consider that the customer can just depart after s/he inserts coins, regardless the system response. Another practical example can be found in most programming languages. It is always valid to call a method without capturing its returned value, which means the caller is not interested in the method result. With case (b), an event may absorb an action from the reaction of the provided function. For example, \( f_{tp} \succeq e_{rc} \) in above examples. For case (c), a function may respond to the reaction from another provided function, such as \( f_{ts} \succeq f_{tp} \) in Example 4.2.2. In this case, the reaction of \( f_{tp} \) will be omitted because \( f_{ts} \) has already finished. Note that partial match relation is not suitable for required functions, since once they emit an action, they will suspend forever if there are no responses to absorb.

The partial match relation is no longer symmetric. We call the first service in this relation, matching invoker and the second service, matching reactor, since the latter always responds to the former. Applying this to full match relations, we consider all active services as matching invokers, and consider all passive services as matching reactors.

**Definition 4.2.11** Let \( s, s' \in S \), then we define \( \text{IVR}(s, s') = \mathbb{B} = \{\text{true}, \text{false}\} \), such that

\[
\text{true, if } s \equiv s' \wedge s \in \uparrow S
\]

\[
\text{true, if } s \triangleright s'
\]

false, otherwise

Also, we define a function \( \text{MCH}(s, s') = \text{IVR}(s, s') \lor \text{IVR}(s', s) \).

**Definition 4.2.12** Let \( s, s' \in S \), we define the difference relation \( \neq \) over services to be \( s \neq s' \iff \neg s \equiv s' \land \neg \text{MCH}(s, s') \), which is irreflexive, symmetric and intransitive.

Notice that the function \( \text{MCH}(s, s') = s \equiv s' \lor s \triangleright s' \lor s' \triangleright s \).

Within our CBS model, every service associates at least one unique service name, but not vice versa. Expressed another way, a service may be named in different ways (as alias), but a name
cannot be reused. A name is nothing more than a handy symbol that serves to identify services. Hence, we can say all service names are in surjection with (onto) all services.

**Definition 4.2.13** We assume a denumerable set \( P \) of all service names, we define a surjective function \( \text{SEV}: P \rightarrow S \).

This function is total as a service cannot exist without a name. The surjective nature of the function is given mainly in consideration of the multi-party service interactions in a connector, where some service names may point onto the same service in certain circumstances.

**Example 4.2.3** Following Examples 4.2.1 and 4.2.2, we can define \( \text{SEV}(ts) = f_{ts} \), where \( ts \) is a service name associated with \( f_{ts} \). Likewise, we associate the following services with corresponding service names, respectively: \( \text{SEV}(ci) = e_{ci} \), \( \text{SEV}(rc) = e_{rc} \), \( \text{SEV}(rp) = f_{rp} \), \( \text{SEV}(tp) = f_{tp} \), \( \text{SEV}(sr) = f_{sr} \), \( \text{SEV}(sm) = f_{sm} \) and \( \text{SEV}(sa) = f_{sa} \).

**Definition 4.2.14** We define a component signature to be a triplet \( \mathcal{S} = (P, S, \text{SEV}) \), where

- \( P \) is a set of service names
- \( S \) is a set of services
- \( \text{SEV} \) is a surjective function from \( P \) to \( S \) that associates service names with services

A component signature \( \mathcal{S} \) comprises a set of service names, services, and a labelling function \( \text{SEV} \) associates each service name with a service. \( \mathcal{S} \) contains all static characteristics of a component, and serves to identify a component. Therefore, a component signature could be considered as the static specification of a component. It also hints that all services defined in \( \mathcal{S} \) are distinct, simply followed by set theory.

**Definition 4.2.15** Given a component signature \( \mathcal{S} = (P, S, \text{SEV}) \), we define

- function \( \text{NAM}: \mathcal{S} \rightarrow P \), which associates the component signature \( \mathcal{S} \) to its service name set \( P \)
- function \( \text{SEV}: \mathcal{S} \rightarrow S \), which associates the component signature \( \mathcal{S} \) to its service set \( S \)

**Example 4.2.4** Consider the services and their names defined in Example 4.2.3. Now we construct a component signature for the vendor component in the ticket vending machine. That is, \( \mathcal{S}_{\text{Vendor}} = (P_v, S_v, \text{SEV}_v) \), where

- \( P_v = \{ts, sr, sm\} \)

![Figure 4.1 The Vendor Component](image-url)
\( S_v = \{ (? \text{insertCoins}, \mid \text{produceTicket}), \)
\( (\mid \text{recordTrans}, ? \text{printTrans}), \)
\( (? \text{startMon}, \mid \text{endMon}) \} \)

\( \text{SEV}_v(ts) = (? \text{insertCoins}, \mid \text{produceTicket}) \)
\( \text{SEV}_v(sr) = (\mid \text{recordTrans}, ? \text{printTrans}) \)
\( \text{SEV}_v(sm) = (? \text{startMon}, \mid \text{endMon}) \)

Likewise, the component signature of the printer component is \( \mathcal{S}_{\text{printer}} = (P_p, S_p, \text{SEV}_p) \),

\( P_p = \{ rp, tp \} \)

\( S_p = \{ (? \text{produceTicket}, \mid \text{printReceipt}), \)
\( (\mid \text{recordTrans}, \mid \text{printTrans}) \} \)

\( \text{SEV}_p(rp) = (? \text{produceTicket}, \mid \text{printReceipt}) \)
\( \text{SEV}_p(tp) = (\mid \text{recordTrans}, \mid \text{printTrans}) \)

Further, the component signature of the customer component can be defined as \( \mathcal{S}_{\text{customer}} = (P_c, S_c, \text{SEV}_c) \), where

\( P_c = \{ ci, rc \} \)

\( S_c = \{ \mid \text{insertCoins}, ? \text{printReceipt} \} \)

\( \text{SEV}_c(ci) = \mid \text{insertCoins} \)
\( \text{SEV}_c(rc) = ? \text{printReceipt} \)

Finally, the component signature of the monitor component is \( \mathcal{S}_{\text{monitor}} = (P_m, S_m, \text{SEV}_m) \), where

\( P_m = \{ sa \} \)
\[ S_m = \{ (\text{startMon}, \text{endMon}) \} \]

\[ SEV_m(\text{sa}) = (\text{startMon}, \text{endMon}) \]

It can be seen from the example that a component signature conveys all static (or structural) specification of a component, which provides a universe for us to describe the behaviour of components. This dynamic specification can be semantically captured in our behaviour model – the time-slot model.

### 4.3 Time-slot Model

The time-slot model is introduced to give an operational semantics for our behaviour language. Just as its name implies, the time-slot model describes the occurrences of events over a sequence of time slots. Five assumptions are considered:

- **Instantaneous event**: to recap, an event is considered as handing an atomic action in a component. The occurrence of an event is treated as instantaneous, so we can consider that the occurrence of any event is at a single precise time, within a single time slot. It implies that the occurrence of one event does not blur into another, and can be judged for simultaneity. This assumption also leads us to a discrete time model [AD94], which simplifies reasoning about the temporal relations over events.

- **Time slot**: The discrete time model is modelled as a sequence of uninterrupted time slots. These time slots are regarded as atomic and equivalent slices of time. Assuming the unit length of a time slot is 1 and the maximal increment is 1, time slots can therefore be modelled as a set of positive integers \( N^+ = \{1, 2, 3, \ldots\} \). It implies that there is no minimum delay between events occurring in two continuous time slots.

- **Newtonian time**: The time is modelled as a single global conceptual clock and at the same rate for all components in CBS. The global time clock is used in the semantic framework for analysis and description of component behaviour.

- **Maximal parallelism**: Assuming every service is executed on its own dedicated processor, a component always has sufficient resources for infinite events to perform in one time slot, so that concurrent events are not in competition for processor time or memory and also no implicit scheduling is considered.

- **Maximal interaction**: Once an event has occurred, all possible matched events are willing to react. This assumption guarantees the maximal interactions over services in a CBS. However, it does not guarantee that the occurrence of an event needs all counterparts to be ready to accept. The occurrence will be immediately omitted if no services react.
To sum up, these assumptions encapsulate the time-slot model taken to the understanding of the component behaviour in the context of time. To compare with Schneider's [Sch00] timed computational model: in the time-slot assumption, Schneider employs real numbers to model time, while we chop time into a set of discrete but uninterrupted time slots and model them by positive integers. Schneider's [Sch00] maximal progress assumption requires that events must occur at the instant that all participants are ready. Our assumption says that the action will be immediately ignored or lost if a service emits an action at an instant when no other parties are ready to absorb it. With this assumption, our model belongs to the branch of synchronous machines in the literature. The rest of assumptions are in compliance with those of Schneider's model.

We restrict the time-slot model to describe observable services only; consequently it indicates that our time-slot model is concrete, in which no internal services are considered. Under the assumption of Newtonian time, the time-slot model is synchronous, with a global clock. We further assume that the time-slot model always has an initial state, and, as time proceeds, executes finitely many services within infinite time slots. This models the fact that time never ends. Moreover, the time-slot model only describes intended behaviour of components; thus the time-slot model is intensional. Bringing all factors together, the time-slot model could be classified as a synchronous intensional linear-time non-interleaving model [Gla90, WN95] in the concurrency community.

**Definition 4.3.1** Suppose that $\mathcal{S}$ is a component signature, we define a run over $\mathcal{S}$ to be a function $f : Q \times \mathbb{N}^+ \to E \cup \{\Omega\}$, where

- $Q \subseteq P$, is a set of service names
- $\mathbb{N}^+$ is a set of positive integers \{1, 2, 3, ...\}, denotes a sequence of time slots
- $E$ is a set of events, and the symbol $\Omega$ denotes nothing happened,

such that for all $n \in \mathbb{N}^+$ and $p \in Q$,

- if $SEV(p) \in E$, then $f(p,n) \neq \Omega \Rightarrow f(p,n) = SEV(p)$
- if $SEV(p) \in F$, then $\begin{cases} SEV(p) = (f(p,n), f(p,n + 1)), & \text{or} \\ SEV(p) = (f(p,n - 1), f(p,n)) \end{cases}$

The component signature $\mathcal{S}$ is the 'universe' of run $f$. Mathematically, run $f$ is a mapping from each service name $p$ together with a time slot $n$ to an event; this association can be understood as the event(s) of service $p$ that occurred in the time slot $n$. It delivers information about the occurrences of events in a time frame. If $p$ denotes a function, its occurrence will be
interpreted as two events (invocation and reaction) that occurred in two continuous time slots. If nothing happened in the time slot \( n \), the symbol \( \Omega \) will be associated.

**Example 4.3.1** Suppose we have a component signature \( \mathcal{S} = (P, S, SEV) \), where \( P = \{s, t\} \), \( S = \{\langle a, b \rangle, \langle c, d \rangle \} \) and \( SEV(s) = \langle a, b \rangle \), \( SEV(t) = \langle c, ?d \rangle \). We define a run \( f_s : \{s, t\} \times \mathbb{N}^+ \rightarrow \{\langle a, b \rangle, \langle c, ?d \rangle \} \cup \{\Omega\} \) over \( \mathcal{S} \). Run \( f_s \) can be visualised tabularly as below.

<table>
<thead>
<tr>
<th>ID</th>
<th>Component Signature</th>
<th>Time Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( s ) \langle a, b \rangle</td>
<td>( a ) ( b )</td>
</tr>
<tr>
<td>2</td>
<td>( t ) \langle c, ?d \rangle</td>
<td>( c ) ?d</td>
</tr>
</tbody>
</table>

The rest of elements in domain \( f_s \) associates \( \Omega \).

In general, every run can be illustrated by a time-slot table, in which each row represents a service and each column denotes a time slot. In the above case, the time-slot table of \( f_s \) graphically represents an execution of service \( s \) and \( t \), meaning that function \( s \)'s \( \langle a, b \rangle \) occurred first, and then \( s \)'s \( !b \) and function \( t \)'s \( !c \) occurred simultaneously, followed by \( t \)'s \( ?d \).

**Definition 4.3.2** Suppose \( f \) is a run over a component signature \( \mathcal{S} \); then we define a function \( NAM(f) = Q \), where each run \( f \) associates its set \( Q \) of service names.

**Definition 4.3.3** We define \( \text{Runs}_{\mathcal{S}} \) to be a set of all possible runs over \( \mathcal{S} \); we define \( \text{Runs}_{\mathcal{S}}(Q) = \{f \in \text{Runs}_{\mathcal{S}} : NAM(f) = Q\} \) to be a set of all possible runs over a set \( Q \) of service names.

If a run represents a component execution, \( \text{Runs}_{\mathcal{S}}(Q) \) comprises the set of all possible executions for a component, given its set \( Q \) of service names.

**Definition 4.3.4** We define \( IDL(f, p) = \text{true} \) if \( \forall n \in \mathbb{N}^+ : f(p, n) = \Omega \). Further, we define \( IDL(f, n) = \text{true} \) if \( \forall p \in NAM(f) : f(p, n) = \Omega \). In this case \( n \) is an idle time slot of run \( f \).

By convention, \( IDL(f, 0) = \text{true} \) for all runs. Within a global clock framework, we allow runs to have idle time slots during execution.

**Definition 4.3.5** Suppose a run \( f \) is finite if \( \exists N \in \mathbb{N}^+ \exists f \in \text{Runs}_{\mathcal{S}} : IDL(f, n) \) for all \( n > N \), we define the length of run to be \( \ell(f) = \min\{N : IDL(f, n) \text{ for all } n > N\} \).

Notice that as a consequence, we say \( \forall f \in \text{Runs}_{\mathcal{S}} : !IDL(f, \ell(f)) \). In words, the length of a run is the last nonempty time slot of a run, for example, \( \ell(f_1) = 3 \) in Example 4.3.1.

**Definition 4.3.6** Let \( f, g \in \text{Runs}_{\mathcal{S}} \), then we define \( f \leq g \) if
a) \( \text{NAM}(f) = \text{NAM}(g) \wedge \)

b) \( \forall p \in \text{NAM}(f), \forall n \in \mathbb{N}^+: f(p,n) = g(p,n) \wedge \)

c) \( \ell(f) \leq \ell(g) \)

It is easy to show that \( \leq \) is a partial order on the set of all runs in \( \text{Runs}_\sigma \).

Informally, the partial order \( f \leq g \) requires that (a) the sets of service names over runs \( f \) and \( g \) are equal; (b) any occurrence in run \( f \) should also appear in run \( g \); (c) the length of run \( f \) is equal and less than the length of run \( g \).

**Example 4.3.2** Following Example 4.3.1, we define a run \( f' \), which is tabularly defined below.

It is obvious that the run \( f' \) doubled the occurrence of the events in service \( s \) and \( t \).

\[
\begin{align*}
  f'(s,1) &= ?a \\
  f'(s,2) &= !b \\
  f'(s,4) &= ?a \\
  f'(s,5) &= !b \\
  f'(t,2) &= !c \\
  f'(t,3) &= ?d \\
  f'(t,5) &= !c \\
  f'(t,6) &= ?d
\end{align*}
\]

The rest of elements in domain \( f' \) associates symbol \( \Omega \). Further, we can say that \( f \leq f' \).

**Proposition 4.3.1** Let \( f, g, h \in \text{Runs}_\sigma \). If \( f, g \leq h \), then either \( f \leq g \) or \( g \leq f \).

*Proof.* Suppose \( f, g \leq h \) and without loss of generality, that \( \ell(f) \leq \ell(g) \), we show that \( f \leq g \). Axiom (c) holds by the assumption. If Axiom (a) holds and if \( p \in \text{NAM}(f) \) and \( n \in \mathbb{N}^+ \), then \( f(p,n) = h(p,n) = g(p,n) \), so Axiom (b) holds, which completes the proof. \( \square \)

**Definition 4.3.7** We define \( \perp_Q \in \text{Runs}_\sigma(Q) \) to be the least element of \( \text{Runs}_\sigma(Q) \), where \( \perp_Q \) satisfies that \( \forall n \in \mathbb{N}^+ : \text{IDL}(_Q, n) \).

In general, the \( \perp_Q \) represents a run over a set \( Q \) of service names where nothing happened. By convention, \( \ell(\perp_Q) = 0 \). We remark that \( \perp_Q \leq f \) for all \( f \in \text{Runs}_\sigma(Q) \). Notice that if \( Q = \emptyset \), then \( \perp_Q \) becomes an empty function, as we can see that \( \perp_{\emptyset} : \emptyset \times \mathbb{N}^+ \rightarrow \{\Omega\} = \perp_{\emptyset} : \emptyset \rightarrow \{\Omega\} \), we denote \( \perp_{\emptyset} \) as \( \perp \) for short.

In order to build a complete behaviour of the component, we need to construct runs recursively. The simplest case is to combine two runs sequentially via the concatenation operation on runs.
Definition 4.3.8 Let \( f, g \in \text{Runs}_\sigma \), then we define the concatenation on runs by \( f \cdot g \in \text{Runs}_\sigma (\text{NAM}(f) \cup \text{NAM}(g)) \), such that
\[
(f \cdot g)(p, n) = \begin{cases} 
  f(p, n), & \text{if } n \leq \ell(f) \\
  g(p, n - \ell(f)), & \text{otherwise}
\end{cases}
\]

Lemma 4.3.1 Let \( f, g \in \text{Runs}_\sigma \), then we say that \( \ell(f \cdot g) = \ell(f) + \ell(g) \).

Proof. It suffices to show that,
1. \(-\text{IDL}(f \cdot g, \ell(f) + \ell(g))
2. \text{IDL}(f \cdot g, n), \text{if } n > \ell(f) + \ell(g)

Firstly, if \((f \cdot g)(p, \ell(f) + \ell(g)) = g(p, \ell(g))\) and \(-\text{IDL}(g, \ell(g))\), then we can deduce that \(-\text{IDL}(f \cdot g, \ell(f) + \ell(g))\).

Secondly, if \(\exists n \in \mathbb{N}^+: n > \ell(f) + \ell(g)\), then \((f \cdot g)(p, n) = g(p, n - \ell(f))\), since \(n - \ell(f) > \ell(g)\) and \(\text{IDL}(g, n - \ell(f))\). As a result of this, we conclude that \(\text{IDL}(f \cdot g, n)\).

Remark 4.3.1 Let \( f, g, h \in \text{Runs}_\sigma \), then
\[a) \quad (f \cdot g) \cdot h = f \cdot (g \cdot h)\]
\[b) \quad \bot \cdot f = f \cdot \bot\]
\[c) \quad \bot_{\text{NAM}(f)} \cdot f = f \cdot \bot_{\text{NAM}(f)}\]
\[d) \quad \bot_{\text{NAM}(g)} \cdot f = f \cdot \bot_{\text{NAM}(g)}\]

Note that if \(\text{NAM}(f) \neq \text{NAM}(g)\), then \(\bot_{\text{NAM}(g)} \cdot f = f \cdot \bot_{\text{NAM}(g)}\), but the result is not equal to the run \(f\) itself, because the result of \(\bot_{\text{NAM}(g)} \cdot f\) is run \(f\) with additional service names \((\text{NAM}(g) - \text{NAM}(f))\), but they are all blank.

Example 4.3.3 Following Example 4.3.1, we define run \( g_r : \{u, v\} \times \mathbb{N}^+ \to \{?g, !h, !e\} \cup \{\Omega\} \), where
\[
g_r(u, 1) = ?g \quad g_r(v, 1) = \Omega \\
g_r(u, 2) = !h \quad g_r(v, 2) = \Omega \\
g_r(u, 3) = \Omega \quad g_r(v, 3) = !e
\]

Then, \((f_s \cdot g_r) : \{s, t, u, v\} \times \mathbb{N}^+ \to \{?a, !b, !c, ?g, !h, !e\} \cup \{\Omega\}\), where
\[(f \cdot g_2)(s,1) = ?a \quad (f \cdot g_2)(t,3) = ?d\]
\[(f \cdot g_2)(s,2) = !b \quad (f \cdot g_2)(u,4) = ?g\]
\[(f \cdot g_2)(t,2) = !c \quad (f \cdot g_2)(u,5) = !h\]
\[(f \cdot g_2)(v,6) = !e\]

It is noteworthy that the same services may appear in two operands of concatenation operation, such as \(f \cdot f_1\). In such contexts, we view this phenomenon as service periodic recurrences. In the time-slot table, service recurrences are displayed in the same row.

**Definition 4.3.9** Let \( f \in \text{Runs}_{\tau} \) and \( m \in \mathbb{N}_0 \), we define the time-slot restrictions on runs by

\[ f|_m (p,n) = \begin{cases} f(p,n), & \text{if } n \leq m \\ \Omega, & \text{otherwise} \end{cases} \]

\[ f|_m (p,n) = f(p,n + m) \]

**Remark 4.3.2** Let \( f \in \text{Runs}_{\tau} \) and \( m, n \in \mathbb{N}_0 \), then

\[ a) \quad (f|_m)|_n = f|_{\min(m,n)} \]
\[ b) \quad (f|_m)|_n = f|_{\max(m,n)} \]
\[ c) \quad \bot|_m = \bot = \bot|_m \]
\[ d) \quad \bot|_{\text{NAM}(/)} = \bot|_{\text{NAM}(/)} = \bot|_{\text{NAM}(/)} \]
\[ e) \quad f|_0 = \bot|_{\text{NAM}(/)} \]
\[ f) \quad f|_0 = f \]
\[ g) \quad f|_m = \bot|_{\text{NAM}(/)}, \text{ if } m \geq \ell(f) \]
\[ h) \quad (f|_m) \cdot (f|_m) = f \]

**Lemma 4.3.2** Let \( f \in \text{Runs}_{\tau} \) and \( m \in \mathbb{N}_0 \), so \( \ell(f|m) = \begin{cases} 0, & \text{if } m \geq \ell(f) \\ \ell(f) - m, & \text{otherwise} \end{cases} \)

**Proof.** For \((a)\), \( m \geq \ell(f) \Rightarrow \ell(f|m) = \ell(\bot|_{\text{NAM}(/)}) = 0 \) by \((g)\) of Remark 4.3.2. For \((b)\), it suffices to remark that,
Chapter 4 Formalisation of Component-based Systems

1. \( \neg \text{IDL}(f \upharpoonright m, \ell(f) - m) \)

2. \( \text{IDL}(f \upharpoonright m, n) \), if \( n > \ell(f) - m \)

Firstly, it is apparent that \( (f \upharpoonright m)(p, \ell(f) - m) = f(p, \ell(f)) \) from Definition 4.3.9 and of course \( \neg \text{IDL}(f, \ell(f)) \), so we can deduce that \( \neg \text{IDL}(f \upharpoonright m, \ell(f) - m) \).

Secondly, if \( n + m > \ell(f) \), then \( \text{IDL}(f, n + m) \), so we can conclude that \( \text{IDL}(f \upharpoonright m, n) \).

\[ \square \]

**Example 4.3.4** Consider run \( f_s \cdot g_x \) in Example 4.3.3, \((f_s \cdot g_x)\mid_5\) is represented as

\[
\begin{align*}
((f_s \cdot g_x)\mid_3)(s, 1) &= ?a \\
((f_s \cdot g_x)\mid_3)(s, 2) &= !b \\
((f_s \cdot g_x)\mid_3)(t, 2) &= !c \\
((f_s \cdot g_x)\mid_3)(t, 3) &= ?d \\
((f_s \cdot g_x)\mid_3)(u, 4) &= ?g \\
((f_s \cdot g_x)\mid_3)(u, 5) &= !h
\end{align*}
\]

<table>
<thead>
<tr>
<th>ID</th>
<th>Component Signature</th>
<th>Time Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s ((?a, !b))</td>
<td>1 2</td>
</tr>
<tr>
<td>2</td>
<td>t ((!c, ?d))</td>
<td>3 4</td>
</tr>
<tr>
<td>3</td>
<td>u ((?g, !h))</td>
<td>5 6</td>
</tr>
<tr>
<td>4</td>
<td>v (!e)</td>
<td>3</td>
</tr>
</tbody>
</table>

The time-slot table above gives us an intuitive view of how the time-slot restriction works. In this example, the operation makes all subsequent time slots of the time slot 5 to be the idle time slots. Consequently, \((f_s \cdot g_x)\mid_5)(v, 6) = \Omega \). Compared to the classical restriction function in mathematics, time-slot restrictions on runs only change the occurrences of events, they do not trim runs’ graphs\(^2\) to smaller domains.

**Lemma 4.3.3** Let \( f, g, h \in \text{Runs}_s \), then

\( a) \ f \upharpoonright m \leq f \)

\( b) \ f \leq g \Rightarrow (f \upharpoonright m \leq g \upharpoonright m) \)

\( c) \ f \leq g \Rightarrow g = f \cdot (g \mid^{(f)}_m) \)

\( d) \ f \leq g \leq h \Rightarrow (g \mid^{(f)}_m) \leq (h \mid^{(f)}_m) \)

\(^2\) The graph of a function \( f \) is the set of all ordered pairs \((x, f(x))\), for all \( x \) in the domain \( X \).
Proof. For (c), suppose \( f \leq g \), then \( f = g \upharpoonright_{a(f)} \), so \( f \cdot (g \mid_{a(f)}) = (g \mid_{a(f)}) \cdot (g \mid_{a(f)}) = g \) by (h) of Remark 4.3.2. For (d), suppose \( f \leq g \leq h \), then \( g = f \cdot (g \mid_{a(f)}) \) and \( h = f \cdot (h \mid_{a(f)}) \). Since \( g \leq h \), then \( f \cdot (g \mid_{a(f)}) \leq f \cdot (h \mid_{a(f)}) \Rightarrow (g \mid_{a(f)}) \leq (h \mid_{a(f)}) \). □

**Lemma 4.3.4** If \( Q \subseteq P \), then \( \text{Runs}_Q(Q) \) is a monoid\(^3\) with respect to \( \cdot \) and with the identity \( \bot_Q \). \( \text{Runs}_Q \) is a monoid with respect to \( \cdot \) and with the identity \( \bot \).

**Proof.** For \( \text{Runs}_Q(Q) \), we have shown the associativity of binary operation \( \cdot \) in (a) of Remark 4.3.1, and the identity element \( \bot_Q \) in Definition 4.3.6, such that for all \( f \in \text{Runs}_Q(Q) \), \( f \cdot \bot_Q = \bot_Q \cdot f = f \) by (b) of Remark 4.3.1. Therefore, we can conclude that \( \text{Runs}_Q(Q) \) is a monoid \( (\text{Runs}_Q(Q), \cdot) \). The proof of \( \text{Runs}_Q \) is analogous. □

Mathematically, these monoids \( \text{Runs}_Q(Q) \) and \( \text{Runs}_Q \) are free and non-commutative, with respect to the least element \( \bot_Q \) and the binary operation of concatenation \( \cdot \).

The \( \text{Runs}_Q(Q) \) describes all possible behaviours of a component with its service set \( Q \). However, when describing component behaviour we are mostly interested in what the component is intended to do. Component-based design is concerned with interconnecting pre-fabricated components to provide some specific overall CBS functionality, and for this purpose it is crucial to have a description of the expected behaviour of each component before the CBS is developed, executed and tested as a whole.

Within our CBS model, this amounts to restricting to an appropriate subset of \( \text{Runs}_Q(Q) \) comprising runs that describe intended or permitted behaviour only.

**Definition 4.3.10** Let \( Q \subseteq P \), then a time-slot language over \( Q \) is a set \( L \subseteq \text{Runs}_Q(Q) \), such that
\[
\forall f \in L, \forall g \in \text{Runs}_Q(Q) : g \leq f \Rightarrow g \in L.
\]

In plain words, we require that, if run \( f \) is in a time-slot language and run \( g \leq f \), then \( g \) must also be in that time-slot language. It is called the downward closure of time-slot languages. As a result of this, \( \bot_Q \in L \) for all \( L \) over \( Q \). And if \( Q = \emptyset \), then \( L = \text{Runs}_\emptyset(\emptyset) = \{\bot\} \).

**Definition 4.3.11** Suppose \( L \subseteq \text{Runs}_Q(Q) \), then we define a function as
\[
\text{NAM}(L) = Q.
\]

By convention, \( \text{NAM}(\{\bot\}) = \emptyset = \text{NAM}(\emptyset) \).

**Definition 4.3.12** Let \( L_1, L_2 \) be time-slot languages, then we define concatenation on time-slot language as \( L_1 \cdot L_2 = \{f \cdot g : f \in L_1, g \in L_2\} \).

\(^3\) A monoid is an algebraic structure with a single, associative binary operation and an identity element.
In general, operations on time-slot languages induce operations on set of runs by element-wise application. For example, the concatenation on time-slot languages generates a set of runs formed by the Cartesian product concatenating runs in $L_1$ and $L_2$.

**Proposition 4.3.2** If $L_1, L_2$ are time-slot languages, then so is $L_1 \cdot L_2$. Further, $\text{NAM}(L_1 \cdot L_2) = \text{NAM}(L_1) \cup \text{NAM}(L_2)$.

**Proof.** We first show downward closure, let $f \in L_1, g \in L_2$ and $h \in \text{Runs}_\emptyset(\text{NAM}(L_1 \cdot L_2))$, suppose $h \leq f \cdot g$, by Proposition 4.3.1, since $f \leq f \cdot g$, either $f \leq h$ or $h \leq f$. In the first case, suppose $f \leq h \leq f \cdot g$, then $h = f \cdot (h|^{t(f)})$ by (c) of Lemma 4.3.3, and by (d) of Lemma 4.3.3 $(h|^{t(f)}) \leq ((f \cdot g)|^{t(f)})$, which implies $(h|^{t(f)}) \leq g$, then $(h|^{t(f)}) \in L_2$ by downward closure. As a result $h \in L_1 \cdot L_2$. In the second case, suppose $h \leq f \leq f \cdot g$, then $h \in L_1$ and $h = h \cdot \perp_{\text{NAM}(g)} \in L_1 \cdot L_2$.

**Remark 4.3.3** Let $L_1, L_2$ and $L_3$ be time-slot languages, then

1. $L_1 \cdot (L_2 \cdot L_3) = (L_1 \cdot L_2) \cdot L_3$
2. $\emptyset \cdot L_1 = \emptyset = L_1 \cdot \emptyset$
3. $L_2 \subseteq L_3 \Rightarrow (L_1 \cdot L_2) \subseteq (L_1 \cdot L_3)$ and $(L_2 \cdot L_1) \subseteq (L_3 \cdot L_1)$

**Definition 4.3.13** Let $L$ be a time-slot language and $m \in \mathbb{N}_0$, we define the time-slot restrictions on time-slot languages by

- $L|_m = \{f|^{t(f)} : f \in L\}$
- $L|^{m} = \{f|^{m} : f \in L\}$

**Remark 4.3.4** Let $L$ be a time-slot language and $m, n \in \mathbb{N}_0$, then

1. $(L|_m)|_n = L|_{\min(m,n)}$
2. $(L|^{m})|^{n} = L|^{\max(m,n)}$
3. $\emptyset|_m = \emptyset|^{m} = \emptyset$

**Definition 4.3.14** Let time-slot model $M_{\mathcal{S}}$ be a set of all possible time-slot languages over $\mathcal{S}$, we define the time-slot model $M_{\mathcal{S}}(Q) = \{L \subseteq \text{Runs}_\emptyset : \text{NAM}(L) = Q\}$ to be a set of all possible time-slot languages over a set $Q$ of service names.
Adopting the initial algebra approach, $M_\mathcal{S}$ is the carrier of the semantic domain or interpretation of our behaviour language. Note that the carrier $M_\mathcal{S}$ depends on a specific component signature $\mathcal{S}$ associated with it. It means the behaviour of a component is completely based upon the static structure of the component.

To sum up, the time-slot model gives us an intentional view of the component behaviour in the context of time. In our approach, the time-slot model provides operational semantics for our syntax language – component protocol which is introduced later. The time-slot model approximates time slots in positive integers to ensure that a run of component proceeds in an orderly way – in discrete time slots. In addition, we require that a finite number of services may occur within infinite time slots, which gives evidence for a collection of runs to be realisable in an intuitive sense, and further gives rise to the notions of regularity for component protocol. Moreover, the time-slot assumption requires that time slots are uninterrupted. It ensures that that there is no ‘gap’ in the time continuum; this fine time granularity holds the repletion property as in Shields [Shi97]. Further, we defined the downward closure for every time-slot language; it guarantees the left-closure property introduced by Shields [Shi97]. That is, all event occurrences in a component are faithfully recorded in the corresponding time-slot language, which is able to present the behaviour of a component. Bringing all concepts together, we believe a time-slot model can be naturally transformed into a discrete behavioural presentation as proposed by Shields [Shi88, Shi97].

### 4.4 Operations of Time-slot Model

In the previous section, the concatenation gives us the first flavour of how to construct new runs and time-slot languages from given ones. Now we are going to provide more such operations in order to facilitate creation and manipulation in the time-slot model. In the initial algebra approach, the signature of $M_\mathcal{S}$ is defined by these operations introduced below.

**Definition 4.4.1** Let $Q, Q' \subseteq P$, then we define the service restriction on runs by

$$\big|_Q : \text{Runs}_\mathcal{S}(Q \cup Q') \to \text{Runs}_\mathcal{S}(Q).$$

For readability, we write $f\big|_Q$ instead of $\big|_Q(f)$. Unlike time-slot restrictions, the service restriction trims the graphs (domain and codomain) of runs. In mathematics, the service restriction is indeed a typical projection function $\text{PROJ}_Q$.

**Remark 4.4.1** Let $f \in \text{Runs}_\mathcal{S}$ and $Q, Q' \subseteq P$, then

a) $$(f\big|_Q)\big|_{Q'} = f\big|_{Q \cap Q'} = (f\big|_{Q'})\big|_Q.$$
Chapter 4 Formalisation of Component-based Systems

Example 4.4.1 Following Example 4.3.3, \((f \cdot g)\mid_{(u,x)}\) is tabularly presented as

\[
\begin{align*}
((f \cdot g)\mid_{(u,x)})(s,1) &= \?a \\
((f \cdot g)\mid_{(u,x)})(s,2) &= !b \\
((f \cdot g)\mid_{(u,x)})(u,4) &= ?g \\
((f \cdot g)\mid_{(u,x)})(u,5) &= !h
\end{align*}
\]

<table>
<thead>
<tr>
<th>ID</th>
<th>Component Signature</th>
<th>Time Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name</td>
<td>Service</td>
</tr>
<tr>
<td>1</td>
<td>s</td>
<td>(?a, !b)</td>
</tr>
<tr>
<td>2</td>
<td>u</td>
<td>(?g, !h)</td>
</tr>
</tbody>
</table>

Definition 4.4.2 Let \(L \in M_{\mathcal{S}}\) and \(Q \subseteq \mathcal{P}\), then we define the service restriction on time-slot languages by \(L \mid Q = \{ f \mid Q : f \in L \}\):

Remark 4.4.2 Let \(L \in M_{\mathcal{S}}\) and \(Q, Q' \subseteq \mathcal{P}\), then

\[
\begin{align*}
a) & \quad (L \mid Q) \mid Q' = L \mid Q \cap Q' = (L \mid Q) \mid Q \\[1.2ex]
b) & \quad \emptyset \mid Q = \emptyset \\[1.2ex]
c) & \quad L \mid Q = L, \text{ if } \text{NAM}(L) \subseteq Q \\[1.2ex]
d) & \quad L \mid Q = \{ \bot \}, \text{ if } \text{NAM}(L) \cap Q = \emptyset
\end{align*}
\]

Definition 4.4.3 Let \(f, g \in \text{Runs}_\mathcal{S}\), then we define the inclusion on runs by \(f[g] \in \text{Runs}_\mathcal{S}(\text{NAM}(f) \cup \text{NAM}(g))\), such that

\[
\ell(f[g]) = \begin{cases} f\mid_{(\ell(f)/2)} \cdot g \cdot f\mid_{(\ell(f)/2)}, & \text{if } \ell(f) \in 2\mathbb{Z} \\ \text{undefined, otherwise} & \end{cases}
\]

Lemma 4.4.1 Let \(f, g \in \text{Runs}_\mathcal{S}\), assuming \(\ell(f) \in 2\mathbb{Z}\), then we say \(\ell(f[g]) = \ell(f) + \ell(g)\).

Proof. It is sufficient to show that

1. \(-\text{IDL}(f[g], \ell(f) + \ell(g))\)
2. \(\text{IDL}(f[g], \ell(f) + \ell(g) + n), \text{ if } n > 0\)
According to Definition 4.4.3, it is immediate that

\[ f(g)(p, n) = \begin{cases} 
  f(p, n), & \text{if } n \leq \ell(f)/2 \\
  g(p, n - \ell(f)/2), & \text{if } n - \ell(f)/2 \leq \ell(g) \\
  f(p, n - \ell(g) + \ell(f)/2), & \text{if } n - \ell(f)/2 > \ell(g)
\end{cases} \]

Firstly, it is obvious that \((f[g])(p, \ell(f) + \ell(g)) = f(p, \ell(f))\) and \(\neg \text{IDL}(f, \ell(f))\), so we can deduce that \(\neg \text{IDL}(f[g], \ell(f) + \ell(g))\).

Secondly, \((f[g])(p, \ell(f) + \ell(g) + n) = f(p, \ell(f) + n)\), so \(\text{IDL}(f[g], \ell(f) + \ell(g) + n)\). □

**Proposition 4.4.1** Let \(f, g \in \text{Runs}_\sigma\), then

\[
\begin{align*}
  a) & \quad (f[g])[h] = f[g[h]] \\
  b) & \quad f[\bot] = \bot[f] \\
  c) & \quad f[\bot_{\text{NAM}(f)}] = \bot_{\text{NAM}(f)}[f] \\
  d) & \quad f[\bot_{\text{NAM}(g)}] = \bot_{\text{NAM}(g)}[f]
\end{align*}
\]

**Proof.** To prove associativity (a), first of all, \(f[g[h]]\) is defined if and only if \(\ell(f) \in 2\mathbb{Z}\) and \(g[h]\) is defined, which holds if and only if \(\ell(g) \in 2\mathbb{Z}\). By Lemma 4.4.1, we can deduce that \(\ell(f) \in 2\mathbb{Z} \land \ell(g) \in 2\mathbb{Z} \Rightarrow \ell(f[g]) \in 2\mathbb{Z}\). Consequently, \(f[g[h]]\) is defined if and only if \((f[g])[h]\) is defined. Secondly, if \(f[g]\) is defined, we can deduce that

\[
\begin{align*}
  a) & \quad f[g]_{\ell(f) + \ell(g)/2} = f_{\ell(f)/2} \cdot g_{\ell(g)/2} \\
  b) & \quad f[g]_{\ell(f) + \ell(g)/2} = g_{\ell(g)/2} \cdot f_{\ell(f)/2}
\end{align*}
\]

by Definition 4.4.3 and Lemma 4.4.1. So we can conclude that

\[
\begin{align*}
  f[g[h]] = f_{\ell(f)/2} \cdot g[h] \cdot f_{\ell(f)/2} \\
  = f_{\ell(f)/2} \cdot g_{\ell(g)/2} \cdot h \cdot g_{\ell(g)/2} \cdot f_{\ell(f)/2} \\
  = f[g]_{\ell(f) + \ell(g)/2} \cdot h \cdot f[g]_{\ell(f) + \ell(g)/2} \\
  = (f[g])[h]
\end{align*}
\]

**Example 4.4.2** Following Example 4.3.3, we define a run

\(h_1 : \{x, y\} \times \mathbb{N}^+ \rightarrow \{!k, ?j\} \cup \{\Omega\}\) as

\[
\begin{align*}
  h_1(x, 1) = !k & \quad h_1(y, 1) = \Omega \\
  h_1(x, 2) = \Omega & \quad h_1(y, 2) = ?j
\end{align*}
\]

<table>
<thead>
<tr>
<th>ID</th>
<th>Component ID</th>
<th>Signature</th>
<th>Time Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>![k]</td>
<td>![k]</td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td>![j]</td>
<td>![j]</td>
</tr>
</tbody>
</table>

- 90 -
Then, \( (h_1[f_1 \cdot g_1]) \colon \{s,t,u,v,x,y\} \times \mathbb{N}^* \rightarrow \{?a,!b,!c>?d,!?e,!?g,!?h,!?j\} \cup \{\Omega\} \), where

\[
\begin{align*}
h_1[f_1 \cdot g_1](x,1) &= ?k \\
h_1[f_1 \cdot g_1](s,2) &= ?a \\
h_1[f_1 \cdot g_1](s,3) &= !b \\
h_1[f_1 \cdot g_1](t,3) &= !c \\
h_1[f_1 \cdot g_1](t,4) &= ?d \\
h_1[f_1 \cdot g_1](u,5) &= !g \\
h_1[f_1 \cdot g_1](u,6) &= !h \\
h_1[f_1 \cdot g_1](v,7) &= !e
\end{align*}
\]

Note that duplicated services are allowed in inclusion, such as \( f_1[f_1] \). We can regard this as the service self-recursive calls.

**Definition 4.4.4** Let \( L_1, L_2 \in \mathcal{M}_\sigma \), then we define the inclusion on time-slot languages

\[
L_1[L_2] = \{f[g] : f \in L_1 \land g \in L_2 \land f[g] \neq \text{undefined}\}.
\]

It should be noteworthy that, \( L_1[L_2] \) excludes ‘undefined’ results; the benefit is that we can always guarantee that the operation produces another time-slot language.

**Remark 4.4.3** If \( L_1 \) and \( L_2 \) are time-slot languages, then so is \( L_1[L_2] \). Furthermore, if \( L_1[L_2] \neq \emptyset \), then \( \text{NAM}(L_1[L_2]) = \text{NAM}(L_1) \cup \text{NAM}(L_2) \).

**Remark 4.4.4** Let \( L_1, L_2, L_3 \in \mathcal{M}_\sigma \), then

a) \( (L_1[L_2])[L_3] = L_1[L_2[L_3]] \)

b) \( L_1[\emptyset] = \emptyset = \emptyset[L_1] \)

**Definition 4.4.5** Let \( f, g, h \in \text{Runs}_\sigma \), then we define the parallel on runs \( f \parallel g \) by

\[
\begin{align*}
f \parallel g &= \begin{cases} h, & \text{if } h\big|_{\text{NAM}(f)} = f \land h\big|_{\text{NAM}(g)} = g \\
\text{undefined, otherwise} & \end{cases}, & \text{where } h \in \text{Runs}_\sigma(\text{NAM}(f) \cup \text{NAM}(g))
\end{align*}
\]

The parallel on runs is used to describe two runs that proceed concurrently and independently. As a matter of fact, the operation is a parallel merge of runs within a shared time slot sequence, in which the ordering imposed by each operand is preserved. No synchronisation on event occurrences is imposed. To some extent, the operation solely synchronises the whole time frames.
of two runs. This parallel operation is similar to the \( \parallel \) composition in CSP as the projection function \( h|_{\text{NAM}(f)} \). This constructor occurs also in various guises in 'classical' net theory related to place decomposition and in COSY [LTS79] in the composition of path expressions.

**Lemma 4.4.2** Let \( f, g \in \text{Runs}_{\neq} \) and assume \( f \parallel g \) is defined, then \( \ell(f \parallel g) = \max\{\ell(f), \ell(g)\} \).

**Remark 4.4.5** Let \( f, g, h \in \text{Runs}_{\neq} \), then

\[
\begin{align*}
  a) & \quad f \parallel f = f \\
  b) & \quad f \parallel g = g \parallel f \\
  c) & \quad (f \parallel g) \parallel h = f \parallel (g \parallel h) \\
  d) & \quad \bot \parallel f = f \\
  e) & \quad \bot|_{\text{NAM}(f)} \parallel f = f
\end{align*}
\]

**Example 4.4.3** Following Example 4.4.2, \( (h \parallel (f \cdot g)) : \{s, t, u, v, x, y\} \times \mathbb{N}^+ \rightarrow \{?a, !b, !c, ?d, !e, ?g, !h, !k, ?j\} \cup \{\Omega\} \), where

\[
\begin{align*}
  (h \parallel (f \cdot g))(s, 1) &= ?a \\
  (h \parallel (f \cdot g))(x, 1) &= !k \\
  (h \parallel (f \cdot g))(s, 2) &= !b \\
  (h \parallel (f \cdot g))(t, 2) &= !c \\
  (h \parallel (f \cdot g))(y, 2) &= ? j \\
  (h \parallel (f \cdot g))(t, 3) &= ? d \\
  (h \parallel (f \cdot g))(u, 4) &= ? g \\
  (h \parallel (f \cdot g))(u, 5) &= ! h \\
  (h \parallel (f \cdot g))(v, 6) &= ! e
\end{align*}
\]

In some contexts, we may want the runs’ time frames to be synchronised with certain time slots. For example, we require the 3rd time slot of run \( f \) to be synchronised with the 5th time slot of run \( g \). This is where the time-slot parallel operator comes into play. It is defined by the assistant function below, which inserts \( i \) time slots to the time slot \( m \) of the run in question.

**Definition 4.4.6** Let \( f \in \text{Runs}_{\neq}, m \in \mathbb{N}^+ \) and \( i \in \mathbb{Z} \), then we define a function \( \text{IS}(f, m, i) = f' \), such that if \( i \leq 0 \), then \( \text{IS}(f, m, i) = f \), otherwise,
\[ f'(p,n) = \begin{cases} f(p,n), & \text{if } n < m \\ \Omega, & \text{if } m \leq n < m + i \\ f(p,n-i), & \text{otherwise} \end{cases} \]

By convention, \( \text{IS}(\bot, m, i) = \bot \) and so is \( \text{IS}(\bot, m, i) = \bot \), for all \( m \in \mathbb{N}^+ \) and \( i \in \mathbb{Z} \).

**Remark 4.4.6** Let \( f \in \text{Runs}_\sigma, m \in \mathbb{N}^+, i \in \mathbb{N}_0, \) \( \ell(\text{IS}(f,m,i)) = \begin{cases} \ell(f) + i, & \text{if } m \leq \ell(f) \\ \ell(f), & \text{otherwise} \end{cases} \).

**Lemma 4.4.3** Let \( f, g \in \text{Runs}_\sigma, m \in \mathbb{N}^+ \) and \( i \in \mathbb{Z}, \) then \( \text{IS}(f \mid g, m, i) = \text{IS}(f,m,i) \mid \text{IS}(g,m,i) \).

**Lemma 4.4.4** Let \( f \in \text{Runs}_\sigma, m, n \in \mathbb{N}^+ \) and \( i, j \in \mathbb{N}_0, \) if \( n \leq m \leq n + i \), then

\[ \text{IS}(\text{IS}(f,n,i),m,j) = \text{IS}(f,n,i + j) \]

**Lemma 4.4.5** Let \( f, g \in \text{Runs}_\sigma, m, n \in \mathbb{N}^+ \) and \( i, j \in \mathbb{N}_0, \) if \( m = n \lor (m > n \land j > i), \) then

\[ \text{IS}(f,m,i) \mid \text{IS}(g,n,j) = \text{IS}(f \mid \text{IS}(g,n,j-i),m,i) \]

**Example 4.4.4** Regarding the run \( h_x \) in Example 4.4.2, \( \text{IS}(h_x,1,2) \) generates a new run \( h'_x \),

\[
\begin{align*}
h'_x(x,1) &= \Omega \\
h'_x(y,1) &= \Omega \\
h'_x(x,2) &= \Omega \\
h'_x(y,2) &= \Omega \\
h'_x(x,3) &= k \\
h'_x(y,3) &= \Omega \\
h'_x(x,4) &= \Omega \\
h'_x(y,4) &= ?j
\end{align*}
\]

**Definition 4.4.7** Let \( f, g \in \text{Runs}_\sigma, m, n \in \mathbb{N}^+ \) and \( k = \max(m,n) \), we define the *time-slot parallel on runs* by

\[ f \mid_m g = \text{IS}(f,1,k-m) \mid \text{IS}(g,1,k-n) \]

\[ f \mid_{m\leftarrow n} g = \text{IS}(f,m,k-m) \mid \text{IS}(g,n,k-n) \]

The time-slot parallel is used to synchronise the \( m \)-th time slot of run \( f \) with the \( n \)-th time slot of run \( g \). The operation \( m \leftarrow n \) is a derivative version of \( m \mid n \), instead of inserting the gap time slots to the beginning of run, \( m \leftarrow_n \) inserts them at the specific time slot \( m \) (or \( n \)). In the sequel, this operation will be used to synchronise runs' last nonempty time slots.

**Lemma 4.4.6** Let \( f, g \in \text{Runs}_\sigma, m, n \in \mathbb{N}^+ \) and \( k = \max(m,n) \), then

\[ \ell(f \mid_m g) = k + \max(\ell(f) - m, \ell(g) - n) \]
Remark 4.4.7 Let \( f, g \in \text{Runs}_{\text{c}} \) and \( m, n \in \mathbb{N}^+ \), then

\[ f \upharpoonright_m \downharpoonleft_n g = g \upharpoonright_m f \]

\[ \downharpoonleft_m \downharpoonleft_n f = f \]

\[ \text{NAM}(f) \downharpoonleft_m \downharpoonleft_n f = f \]

\[ f \upharpoonright_m \downharpoonleft_n g = f \upharpoonright g \iff m = n \]

\[ f \upharpoonright_m \downharpoonleft_n g = g \upharpoonright_n f \iff (m = n) \vee (m > \ell(f) \wedge n > \ell(g)) \]

Lemma 4.4.7 Let \( f, g, h \in \text{Runs}_{\text{c}} \) and \( l, m, n \in \mathbb{N}^+ \), then

\[ (f \upharpoonright_m \downharpoonleft_n g) \max(l, m) \downharpoonleft_n h = f \upharpoonright \max(l, m) \downharpoonleft_n \text{IS}(g, m, l - m) \downharpoonleft_n h \]

Proof. To prove this result, we need to examine each case in turn.

**Case 1:** let \( l = m = n \), then by definition 4.4.7,

\[ \text{LHS}^4 := (f \upharpoonright_m \downharpoonleft_n g) \max(l, m) \downharpoonleft_n h = (f \upharpoonright g) \downharpoonleft_n h \]

\[ \text{RHS}^5 := f \upharpoonright \max(l, m) \downharpoonleft_n \text{IS}(g, m, l - m) \downharpoonleft_n h = f \upharpoonright g \downharpoonleft_n h = \text{LHS} \]

**Case 2:** let \( n > l = m \), then by definition 4.4.7,

\[ \text{RHS} := f \upharpoonright \max(l, m) \downharpoonleft_n \text{IS}(g, m, l - m) \downharpoonleft_n h = \text{IS}(f, l, n - l) \downharpoonleft_n \text{IS}(g, l, n - l) \downharpoonleft_n h \]

\[ \text{LHS} := (f \upharpoonright_m \downharpoonleft_n g) \max(l, m) \downharpoonleft_n h = \text{IS}(f, l, n - l) \downharpoonleft_n \text{IS}(g, l, n - l) \downharpoonleft_n h \]

\[ = \text{IS}(f, l, n - l) \downharpoonleft_n \text{IS}(g, l, n - l) \downharpoonleft_n h \]

\[ = \text{RHS} \]

**Case 3:** let \( l > m > n \), then by definition 4.4.7,

\[ \text{LHS} := (f \upharpoonright_m \downharpoonleft_n g) \max(l, m) \downharpoonleft_n h = (f \upharpoonright \text{IS}(g, m, l - m) \downharpoonleft_n \text{IS}(h, l, n - l) \downharpoonleft_n h \]

\[ \text{RHS} := f \upharpoonright \max(l, m) \downharpoonleft_n \text{IS}(g, m, l - m) \downharpoonleft_n h = f \upharpoonright \text{IS}(g \downharpoonleft \text{IS}(h, n, m - n) \downharpoonleft_n m, l - m) \downharpoonleft_n h \]

\[ = f \downharpoonright \text{IS}(g, m, l - m) \downharpoonleft_n \text{IS}(\text{IS}(h, n, m - n), m, l - m) \]

\[ \text{Remark 4.4.7 (c)} \]
\[ = f | IS(g, m, l - m) | IS(h, n, l - n) \quad \text{Lemma 4.4.4} \]
\[ = \text{LHS} \quad \text{Remark 4.4.5 (c)} \]

Case 4: let \( n > m > l \), then by definition 4.4.7,
\[ \text{RHS} := f \big|_\text{max(m,n)} (g \big|_m \big|_n h) = IS(f, l, n - l) | (IS(g, m, n - m) | h) \]
\[ \text{LHS} := (f \big|_l \big|_m g) \big|_{\text{max(l,m)|n}} h = IS(IS(f, l, m - l) | g, m, n - m) | h \]
\[ = IS(f, l, n - l) | IS(g, m, n - m) | h \]
\[ = \text{RHS} \quad \text{Lemma 4.4.5} \]
\[ = \text{RHS} \quad \text{Remark 4.4.5 (c)} \]

More cases are proven in Appendix A. We suggest that the proofs of all other cases among \( l, m \) and \( n \) are analogous. □

The results from Lemma 4.4.7 also holds for \( \cdot |_n \), i.e., \((f \cdot |_m g) \cdot |_{\text{max(l,m)|n}} h = f \cdot |_{\text{max(m,n)}} (g \cdot |_n h) \) and the proof is analogous.

**Example 4.4.5** Following Example 4.4.2, we define \( h_{x \cdot y} (f \cdot g) : \{s, t, u, v, x, y\} \times \mathbb{N}^+ \rightarrow \{?a, !b, !c, ?d, !e, ?g, h, !k, ?j\} \cup \{\Omega\} \), where

\[
\begin{align*}
(h_{x \cdot y} (f \cdot g))(s, 1) &= ?a \\
(h_{x \cdot y} (f \cdot g))(s, 2) &= !b \\
(h_{x \cdot y} (f \cdot g))(t, 2) &= !c \\
(h_{x \cdot y} (f \cdot g))(x, 2) &= !k \\
(h_{x \cdot y} (f \cdot g))(t, 3) &= ?d \\
(h_{x \cdot y} (f \cdot g))(y, 3) &= ?j \\
(h_{x \cdot y} (f \cdot g))(u, 4) &= ?g \\
(h_{x \cdot y} (f \cdot g))(u, 5) &= !h \\
(h_{x \cdot y} (f \cdot g))(v, 6) &= !e
\end{align*}
\]

<table>
<thead>
<tr>
<th>ID</th>
<th>Component Signature</th>
<th>Time Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name</td>
<td>Service</td>
</tr>
<tr>
<td>1</td>
<td>s</td>
<td>(a, !b)</td>
</tr>
<tr>
<td>2</td>
<td>t</td>
<td>(c, ?d)</td>
</tr>
<tr>
<td>3</td>
<td>u</td>
<td>(g, !h)</td>
</tr>
<tr>
<td>4</td>
<td>v</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

**Definition 4.4.8** Let \( L_1, L_2 \in M_\sigma \) and \( m, n \in \mathbb{N}^+ \), we define the parallel operations on time-slot languages by

- \( L_1 | L_2 = \{ f | g : f \in L_1 \land g \in L_2 \land f | g \neq \text{undefined} \} \)
\begin{itemize}
  \item L_{1 \ln l} L_2 = \{ f_{m \ln g} : f \in L_1 \land g \in L_2 \land f_{m \ln g} \neq \text{undefined} \}
  \item L_{1 \ln l} L_2 = \{ f_{m \ln g} : f \in L_1 \land g \in L_2 \land f_{m \ln g} \neq \text{undefined} \}
\end{itemize}

**Remark 4.4.8** If \( L_1 \) and \( L_2 \) are time-slot languages, then so is \( L_1 \mid L_2 \). Further, if \( L_1 \mid L_2 \neq \emptyset \), then \( \text{NAM}(L_1 \mid L_2) = \text{NAM}(L_1) \cup \text{NAM}(L_2) \). It also applies to \( L_{1 \ln l} L_2 \) and \( L_{1 \ln l} L_2 \).

**Remark 4.4.9** Let \( L_1, L_2, L_3 \in M \) and \( m, n \in \mathbb{N}^+ \), then
\begin{itemize}
  \item[1)] \( L_1 \mid L_1 = L_1 \)
  \item[2)] \( L_1 \mid L_2 = L_2 \mid L_1 \)
  \item[3)] \( L_{m \ln l} L_2 = L_2 \mid n \ln L_1 \)
  \item[4)] \( L_{m \ln l} L_2 = L_2 \mid n \ln L_1 \)
  \item[5)] \( (L_1 \mid L_2) \mid L_3 = L_1 \mid (L_2 \mid L_3) \)
  \item[6)] \( \emptyset \mid L_1 = \emptyset \)
  \item[7)] \( \emptyset \mid m \ln L_1 = \emptyset \)
  \item[8)] \( \emptyset \mid m \ln L_1 = \emptyset \)
  \item[9)] \( f \in L_1 \iff f \in f \mid L_1 \)
  \item[10)] \( m = n \iff L_{m \ln l} L_2 = L_{1 \ln m} L_1 \)
\end{itemize}

In addition to the parallel operation, we introduce a more sophisticated operation, called simultaneity. Based on parallel, it further requires that two runs start and finish at the same time slots (which are called synchronisation points), and leaves the processes in-between running in parallel. Namely, it makes two processes perform in the same period of time. Simultaneity will use the following assistant functions.

**Definition 4.4.9** Let \( f \in \text{Runs}_\sigma \), we define a function \( \text{SAT}(f) = \min \{ n : -\text{IDL}(f, n) \} \).

The function \( \text{SAT}(f) \) returns the first nonempty time slot of run \( f \). For any \( f \) over \( Q \subseteq P \), if \( f \) is not \( \perp_Q \), at least one time slot will be nonempty. Otherwise, by convention, \( \text{SAT}(\perp_Q) = 0 \). For instance, \( \text{SAT}(\text{IS}(h, 1, 2)) = 3 \) in Example 4.4.4.

**Remark 4.4.10** Let \( f \in \text{Runs}_\sigma \) and \( n \in \mathbb{N}_0 \), then \( \text{SAT}(\text{IS}(f, 1, n)) = \text{SAT}(f) + n \).

**Definition 4.4.10** Let \( f \in \text{Runs}_\sigma \), we define a function \( \text{LST}(f) = \ell(f) - \text{SAT}(f) \).

**Definition 4.4.11** Let \( f, g \in \text{Runs}_\sigma \) and \( k = \max(\text{SAT}(f), \text{SAT}(g)) \), then we define the simultaneity on runs by \( f \parallel g = \text{IS}(f, 1, k - \text{SAT}(f)) \mid_{\text{LST}(f)+k} \mid_{\text{LST}(g)+k} \text{IS}(g, 1, k - \text{SAT}(g)) \).

**Lemma 4.4.8** Let \( f, g \in \text{Runs}_\sigma \), \( k = \max(\text{SAT}(f), \text{SAT}(g)) \), then
\begin{itemize}
  \item \( \ell(f \parallel g) = k + \max(\text{LST}(f), \text{LST}(g)) \)
\end{itemize}
Chapter 4 Formalisation of Component-based Systems

Proof. According to Definition 4.4.9, it is immediate that

\[ \ell(f \parallel g) = \ell(IS(f,1,k-SAT(f)) \downarrow_{LST(f,k)} IS(g,1,k-SAT(g))) \]  

(a)

Let \( f' = IS(f,1,k-SAT(f)) \), \( g' = IS(g,1,k-SAT(g)) \), \( l = k + \text{max}(LST(f),LST(g)) \), then

\[ (a) = \text{max}(IS(f',LST(f)+k,l-LST(f)-k),IS(g',LST(g)+k,l-LST(g)-k)) \]  

(b)

where \( \ell(IS(f',LST(f)+k,l-LST(f)-k)) = \ell(f') + l - LST(f) - k \)

Remark 4.4.6

\[ = \text{LST}(f) + k + l - LST(f) - k = l \]

In analogy, \( \ell(IS(g',n,l-n)) = l \), so in fact \((b) = \text{max}(l,l) = l\). We therefore conclude that \(\ell(f \parallel g) = l = k + \text{max}(LST(f),LST(g))\). \(\square\)

Lemma 4.4.9 Let \( f, g \in \text{Runs}_\sigma \), \( m \in \mathbb{N}^+ \) and \( i \in \mathbb{Z} \), \( IS(f \parallel g, m, i) = IS(f, m, i) \parallel IS(g, m, i) \).

Lemma 4.4.10 Let \( f, g, h \in \text{Runs}_\sigma \), \( SAT(f) = SAT(g) = SAT(h) \Rightarrow (f \parallel g) \parallel h = f \parallel (g \parallel h) \).

Proof. Let \( m = \text{max}\{SAT(f),SAT(g),SAT(h)\} \), and \( f' = IS(f,1,m-SAT(f)) \)

4.4.10 , \( SAT(f') = SAT(g') = SAT(h') = m \), then we claim that

\[ (f \parallel g) \parallel h = (f' \parallel g') \parallel h' = (f' \ell(f'),g') \max(\ell(f'),\ell(g')) \ell(h') \]

\[ = f' \ell(f') \max(\ell(f'),\ell(h')) (g' \ell(g')) \ell(h') = f'' \parallel (g' \parallel h') \]  

Lemma 4.4.7

\[ = f \parallel (g \parallel h) \]  

\(\square\)

Remark 4.4.11 Let \( f, g \in \text{Runs}_\sigma \), then

\( a) \quad f \parallel f = f \)

\( b) \quad f \parallel g = g \parallel f \)

\( c) \quad \perp f = f \)

\( d) \quad \perp_{\text{Nam}(f)} f = f \)

Proposition 4.4.2 Let \( f, g, h \in \text{Runs}_\sigma \), then

\( a) \quad (f \parallel g) \parallel h = f \parallel (g \parallel h) \)

\( b) \quad SAT(f) = SAT(g) \land \ell(f) = \ell(g) \iff f \parallel g = f \mid g \)
Proof. For (a), let \( m = \max\{\text{SAT}(f), \text{SAT}(g), \text{SAT}(h)\} \), and \( f' = \text{IS}(f,1,m - \text{SAT}(f)) \), \( g' = \text{IS}(g,1,m - \text{SAT}(g)) \) by \( h' = \text{IS}(h,1,m - \text{SAT}(h)) \).

Remark 4.4.10, we have \( \text{SAT}(f') = \text{SAT}(g') = \text{SAT}(h') = m \), so Lemma 4.4.9 applies. We need to show that

1. \((f \| g) \| h = (f' \| g') \| h')\)
2. \(f \| (g \| h) = f' \| (g' \| h')\)

By commutativity, we can easily deduce (2) from (1). Therefore, we only need to consider (1) in a number of cases.

Case 1: Let \( \text{SAT}(f) > \text{SAT}(g) > \text{SAT}(h) \), and \( k = \text{SAT}(f) - \text{SAT}(g) \), \( k' = \text{SAT}(f) - \text{SAT}(h) \),

\[
(f \| g) \| h = (f' \| (g' \| h'))^{\max(\ell(f'), \ell(g'), \ell(h'))} IS(h,1,k')
\]

\[
= (f' \| (g' \| h'))^{\max(\ell(f'), \ell(g'), \ell(h'))} h'
\]

By Lemma 4.4.7

Case 2: Let \( \text{SAT}(f) = \text{SAT}(g) < \text{SAT}(h) \), and \( k = 0 \), \( k' = \text{SAT}(h) - \text{SAT}(f) \)

Since \( \text{SAT}(f \| g) = \text{SAT}(f) < \text{SAT}(h) \), then

\[
(f \| g) \| h = \text{IS}(f \| g,1,k')^{\max(\ell(f), \ell(g)) + k' \ell(h')} h'
\]

\[
= \text{IS}(f,1,k') \| \text{IS}(g,1,k')^{\max(\ell(f), \ell(g)) + k' \ell(h')} h'
\]

By Lemma 4.4.9

Remark 4.4.6

We suggest that the proofs of others are analogous. Therefore, an application of Lemma 4.4.10 completes the proof.

For (b), if \( \text{SAT}(f) = \text{SAT}(g) = k \), then

- \( \text{IS}(f,1,k - \text{SAT}(f)) = \text{IS}(f,1,0) = f \)
- \( \text{IS}(g,1,k - \text{SAT}(g)) = \text{IS}(g,1,0) = g \)
so that $f \parallel g = f \upharpoonright \{t(f) \mid t(g) \} g = f \mid g$ by (d) of Remark 4.4.7.

**Example 4.4.6** Following Example 4.4.2, so we define $h_\parallel (f_s \cdot g_s) : \{s,t,u,v,x,y\} \times \mathbb{N}^* \rightarrow \{?a,!b,c,d,e,g,h,k,?f\} \cup \{\Omega\}$, where

\[(h_\parallel (f_s \cdot g_s))(s,1) = ?a\]
\[(h_\parallel (f_s \cdot g_s))(x,1) = !h\]
\[(h_\parallel (f_s \cdot g_s))(s,2) = !b\]
\[(h_\parallel (f_s \cdot g_s))(t,2) = !c\]
\[(h_\parallel (f_s \cdot g_s))(t,3) = ?d\]
\[(h_\parallel (f_s \cdot g_s))(u,4) = ?g\]
\[(h_\parallel (f_s \cdot g_s))(u,5) = !h\]
\[(h_\parallel (f_s \cdot g_s))(v,6) = !e\]
\[(h_\parallel (f_s \cdot g_s))(y,6) = ?j\]

For some technical purposes, we may require runs to synchronise on a common service. For example, each run describes a partial view of component behaviour and we need to combine them together to generate a complete view. For this reason, we introduce the service simultaneity, which is facilitated by the following assistant functions.

**Definition 4.4.12** Let $f \in \text{Runs}_s$, we define a function $\text{OCT}(f,p,e) = \{n : f(p,n) = e\}$. Suppose $\text{OCT}(f,p,e) \neq \emptyset$, then $\text{OCT}(f,p,e) = \{n_1, \ldots, n_k\}$ such that $n_1 < \ldots < n_k$, so we define

\[
\text{OCT}(f,p,e,i) = \begin{cases} 
  n_i, & \text{if } i \leq k \\
  \text{undefined, otherwise}
\end{cases}
\]

This function returns a time slot of run $f$ in which the event token $t$ occurred for the $i$-th time. For instance, $\text{OCT}(h_\parallel (f_s \cdot g_s), y, ? j, 1) = 6$ in Example 4.4.6.

**Definition 4.4.13** Let $f \in \text{Runs}_s$, we define a function $\text{OCE}(f,p,e)$ by

\[
\text{OCE}(f,p,e) = \begin{cases} 
  \text{OCT}(f,p,e,1), & \text{if } |\text{OCT}(f,p,e)| = 1 \\
  \text{undefined, otherwise}
\end{cases}
\]

For example, $\text{OCE}(h_\parallel (f_s \cdot g_s), y, ? j) = 6$ in Example 4.4.6, but $\text{OCE}(f', s, ?a)$ is undefined in Example 4.4.2.
Chapter 4 Formalisation of Component-based Systems

Definition 4.4.14 Let \( f, g \in \text{Runs}_E, \ p \in P \) and \( s = \text{SEV}(p) \), we define the service simultaneity on runs by

\[
\begin{align*}
\text{♦ } s \in E &\Rightarrow f \parallel g = f \circ \text{OCE}(f,p,s) \circ \text{OCE}(g,p,s) \parallel g \\
\text{♦ } s \in F &\Rightarrow f \parallel g = \text{IS}(f,1,k-m) \circ \text{OCE}(g,p,s) \circ \text{IS}(g,1,k-n), \text{ where} \\
&\quad a) \ m = \text{OCE}(f,p,\text{INV}(s)) \\
&\quad b) \ n = \text{OCE}(g,p,\text{INV}(s)) \\
&\quad c) \ i = \text{OCE}(f,p,\text{REC}(s)) \\
&\quad d) \ j = \text{OCE}(g,p,\text{REC}(s)) \\
&\quad e) \ k = \max(m,n) \\
\end{align*}
\]

The service simultaneity will be undefined if no \( p \) or multiple \( p \) is found on either side. It behaves differently according to the kind of service \( p \) is denoting. For events, it simply synchronises the time slots in which their events occurred. For functions, it first synchronises their invocations, and then synchronises their reactions and the region in-between is called simultaneous region, which can be considered as an interpretation of coregions in LSCs [DH01]. The rest of executions in operands retain their order. A close interpretation could be found in the interface parallel in CSP [Soh00].

Proposition 4.4.3 Let \( f, g, h \in \text{Runs}_E, p \in P \), assuming \( f \parallel g \) and \( g \parallel h \) are defined, then

\[
\begin{align*}
&\quad a) \ f \parallel g = g \parallel f \\
&\quad b) \ (f \parallel g) \parallel h = f \parallel (g \parallel h) \\
\end{align*}
\]

Proof: For (a), we can simply deduce the commutativity of \( \parallel \) from (a) of Remark 4.4.6.

For (b), let \( s = \text{SEV}(p) \), if \( s \in E \), we can show that \((f \parallel g) \mid_{\text{max}(l,m)} h = f \mid_{\text{max}(l,m)} (g \mid_{\text{max}(l,m)} h)\), where \( l = \text{OCE}(f,p,s), m = \text{OCE}(g,p,s), n = \text{OCE}(h,p,s) \) by Lemma 4.4.7.

If \( s \in F \), then the proof is analogous to the method applied in (a) of Proposition 4.4.2, where

\[
\begin{align*}
&\quad m = \max\{\text{OCE}(f,p,\text{INV}(s)),\text{OCE}(g,p,\text{INV}(s)),\text{OCE}(h,p,\text{INV}(s))\} \\
&\quad k = \text{OCE}(f,p,\text{INV}(s)) - \text{OCE}(g,p,\text{INV}(s)) \\
&\quad k' = \text{OCE}(f,p,\text{INV}(s)) - \text{OCE}(h,p,\text{INV}(s)) \\
\end{align*}
\]

Therefore, we can claim the associativity of service simultaneity. □
**Example 4.4.7** Following Example 4.4.2, we define a run

\[ i : \{u, z\} \times \mathbb{N}^+ \to \{g, h, l, ?m\} \cup \{\Omega\} \]

where

\[ i(u, 1) = ?g \]
\[ i(z, 2) = !l \]
\[ i(u, 4) = !h \]
\[ i(z, 3) = ?m \]

So, \( i \in (h_s [f_k \cdot g_r]) : \{s, t, u, v, x, y, z\} \times \mathbb{N}^+ \to \{a, b, c, d, e, f, g, h, k, ?j, ?l, ?m\} \cup \{\Omega\} \), where

<table>
<thead>
<tr>
<th>ID</th>
<th>Component Signature</th>
<th>Time Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>u ( (?g, !h) )</td>
<td>?g !h</td>
</tr>
<tr>
<td>2</td>
<td>z ( (l, ?m) )</td>
<td>!l ?m</td>
</tr>
</tbody>
</table>

**Definition 4.4.15** Let \( L_1, L_2 \in \mathcal{M}_p \) and \( p \in \mathcal{P} \), we define the simultaneity operations on time-slot languages by

\[ L_1 \parallel L_2 = \{ f \parallel g : f \in L_1 \land g \in L_2 \land f \parallel g \neq \text{undefined} \} \]
\[ L_1 \parallel L_2 = \{ f \parallel g : f \in L_1 \land g \in L_2 \land f \parallel g \neq \text{undefined} \} \]

**Remark 4.4.12** If \( L_1 \) and \( L_2 \) are time-slot languages, then so is \( L_1 \parallel L_2 \). Furthermore, if \( L_1 \parallel L_2 \neq \emptyset \), then \( \text{NAM}(L_1 \parallel L_2) = \text{NAM}(L_1) \cup \text{NAM}(L_2) \). It also applies to \( L_1 \parallel L_2 \).

**Remark 4.4.13** Let \( L_1, L_2, L_3 \in \mathcal{M}_p \) and \( p \in \mathcal{P} \), then

\( a) \ L_1 \parallel L_1 = L_1 \)

\( b) \ L_1 \parallel L_2 = L_2 \parallel L_1 \)

\( L_1 \parallel L_2 = L_2 \parallel L_1 \)
Chapter 4 Formalisation of Component-based Systems

c) \((L_1 \parallel L_2) \parallel L_3 = L_1 \parallel (L_2 \parallel L_3)\)  
\((L_1 \parallel L_2) \parallel L_3 \neq L_1 \parallel (L_2 \parallel L_3)\)

d) \(\emptyset \parallel L_1 = \emptyset\)  
\(\emptyset \parallel L_1 = \emptyset\)

**Definition 4.4.16** Let \(f, g \in \text{Runs}_\mathcal{S}\), \(p, q \in P\) where \(s_p = \text{SEV}(p) \in \uparrow S\), \(s_q = \text{SEV}(q) \in \downarrow S\) and \(k = i = 0\), we define the binding on runs by

\[ f \overset{p}{\rightarrow} g = f \mid g, \text{ iff} \]
\[ a) \ f = \perp \wedge \text{IDL}(g, q), \text{ or} \]
\[ b) \ \text{OCE}(f, p, s_p) \in \mathbb{N}^+ \wedge \text{IDL}(g, q), \text{ otherwise} \]

\[ f \overset{p}{\rightarrow} g = \begin{cases} \text{IS}(f, k, i) \mid_{m+1} g, & \text{if IVR}(s_p, s_q) \\ \text{undefined}, & \text{otherwise} \end{cases} \]

1. if \(s_p, s_q \in F \wedge s_p \equiv s_q\) (cf. Definition 4.2.7), then

\[ a) \ k = \text{OCE}(f, p, \text{REC}(s_p)) \]
\[ b) \ i = \text{LST}(f \mid_{\{p\}}) - \text{LST}(g \mid_{\{q\}}) + 2 \]
\[ c) \ m = \text{OCE}(f, p, \text{INV}(s_p)) \]
\[ d) \ n = \text{OCE}(g, q, \text{INV}(s_q)) \]

2. if \(s_p, s_q \in E \wedge s_p \equiv s_q\) (cf. Definition 4.2.3), then

\[ a) \ m = \text{OCE}(f, p, s_p) \]
\[ b) \ n = \text{OCE}(g, q, s_q) \]

3. if \(s_p \in E \wedge s_q \in F \wedge s_p \succ s_q\) (cf. Definition 4.2.10 (a)), then

\[ a) \ m = \text{OCE}(f, p, s_p) \]
\[ b) \ n = \text{OCE}(g, q, \text{INV}(s_q)) \]

4. if \(s_p \in F \wedge s_q \in E \wedge s_p \succ s_q\) (cf. Definition 4.2.10 (b)), then

\[ a) \ m = \text{OCE}(f, p, \text{REC}(s_p)) \]
\[ b) \ n = \text{OCE}(g, q, s_q) \]

5. if \(s_p, s_q \in F \wedge s_p \succ s_q\) (cf. Definition 4.2.10 (c)), then

\[ a) \ m = \text{OCE}(f, p, \text{REC}(s_p)) \]
\[ b) \ n = \text{OCE}(g, q, \text{INV}(s_q)) \]

The binding is a kind of composition with regard to service interactions. Each binding operation is based on an atomic service interaction; so, in general, the operator is defined if and only if, (i) service \(p\) and \(q\) occurred once in runs \(f\) and \(g\) respectively and, (ii) \(p\) is a matching invoker. It hints that service \(p\) fully or partially matches service \(q\). Based on the match relations defined in Section 4.2, five binding methods are considered, as all kinds of valid service interactions. They are, case
(1) two fully matched functions; case (2) two fully matched events; cases (3, 4) a pair of partially matched event and functions; and case (5) two partially matched functions. The first two cases are called full binding, and the latter three are called partial binding.

Two particular cases are in addition defined if the operands fail to meet above cases. Assuming service \( q \) does not occur in run \( g \) (note IDL(\( g, q \)) always holds for \( g = \perp, \perp_0 \)), case (a) run \( f = \perp \), and case (b) service \( p \) is an out event that occurred once in run \( f \). This case follows the synchrony principle of service interactions discussed in Section 4.2. That is to say, emitting actions from service \( p \) can be omitted if service \( q \) does not (is not ready to) absorb. Notice that \( p \) cannot be a function, because if so, once \( p \) emits a request, it would be stuck there if no response returned. Technically, the term OCE(\( f, p, s_p \)) is undefined if \( s_p \in F \), because \( s_p \) is an ordered pair. Since no interactions happened, we treat \( f \mapsto_q g \) under these two cases as \( f \circ g \) composition. This setting will be employed to check the glue's anti-deadlock property in Section 5.3. Apart from these cases, all other cases are left undefined. Because of the nature of binding operator, it is neither associative nor commutative.

**Example 4.4.8** Following Example 4.3.1, we define a run \( j_x : \{p, w\} \times \{a, b\} \rightarrow \{!a, b, !n\} \cup \{\Omega\} \), where

\[
\begin{align*}
  j_x(p, 1) &= !a \\
  j_x(w, 2) &= !n \\
  j_x(p, 2) &= ?b
\end{align*}
\]

then \( j_x \mapsto_x f_x : \{s, t, p, w\} \times \{a, b\} \rightarrow \{?a, !b, !c, ?d, !a, b, !n\} \), where

\[
\begin{align*}
  (j_x \mapsto_x f_x)(p, 1) &= !a \\
  (j_x \mapsto_x f_x)(s, 2) &= ?a \\
  (j_x \mapsto_x f_x)(w, 2) &= !n \\
  (j_x \mapsto_x f_x)(s, 3) &= !b \\
  (j_x \mapsto_x f_x)(t, 3) &= !c \\
  (j_x \mapsto_x f_x)(p, 4) &= ?b \\
  (j_x \mapsto_x f_x)(t, 4) &= ?d
\end{align*}
\]

This example shows the composition over run \( f \) and \( g \) with regard to the interaction between a pair of functions \( p \) and \( s \), as case (1).
**Definition 4.4.17** Let \( L_1, L_2 \in M_\sigma \) and \( p, q \in P \), then we define the binding on time-slot languages by \( L_1 \mapsto_q L_2 = \{ f \mapsto q g : f \in L_1 \land g \in L_2 \land f \mapsto_q g \neq \text{undefined} \} \).

**Remark 4.4.14** If \( L_1 \) and \( L_2 \) are time-slot languages, then so is \( L_1 \mapsto_q L_2 \). Furthermore, if \( L_1 \mapsto_q L_2 \neq \emptyset \), then \( \text{NAM}(L_1 \mapsto_q L_2) = \text{NAM}(L_1) \cup \text{NAM}(L_2) \).

**Remark 4.4.15** Let \( L_1 \in M_\sigma \) and \( p, q \in P \), then \( L_1 \mapsto_q \emptyset = \emptyset = \emptyset \mapsto_q L_1 \).

The interleaving semantics in our approach is interpreted as performing two operands in any order, included in exactly the same period of time (simultaneity), so long as the ordering imposed by each operand as such is preserved. We consider the interleaving to be a default way of thinking about the behaviours on different components.

**Definition 4.4.18** Let \( L_1, L_2 \in M_\sigma \), then we define the interleaving on time-slot languages by \( L_1 || L_2 = L' \cup L'' \), where

\[
\begin{align*}
a) \quad & L' = L_1[L_2] \cup L_2[L_1] \cup L_1 || L_2, \\
b) \quad & L'' = \bigcup_{i=1}^{\sigma(f)} \{ f \mapsto_i g \} \cup \bigcup_{i=1}^{\sigma(g)} \{ g \mapsto_i f \}, \text{for all } f \in L_1, g \in L_2.
\end{align*}
\]

\( L' \) produces a set of runs in such a way that, it time-slot synchronises two operands in any possible order as long as no additional idle time slots (gap) introduced in-between. Note that \( f \cdot \tau (f), g = f \cdot g \) for any \( f, g \in \text{Runs}_\sigma \). We view \( L_1 || L_2 \) as the most liberal composition over language \( L_1 \) and \( L_2 \).

**Remark 4.4.16** If \( L_1 \) and \( L_2 \) are time-slot languages, then so is \( L_1 || L_2 \). Furthermore, if \( L_1 || L_2 \neq \emptyset \), then \( \text{NAM}(L_1 || L_2) = \text{NAM}(L_1) \cup \text{NAM}(L_2) \).

**Remark 4.4.17** Let \( L_1, L_2, L_3 \in M_\sigma \), then

\[
\begin{align*}
a) \quad & L_1 || L_2 = L_2 || L_1, \\
b) \quad & (L_1 || L_2) || L_3 = L_1 || (L_2 || L_3), \\
c) \quad & \emptyset || L_1 = \emptyset, \\
d) \quad & L_2 \subseteq L_3 \Rightarrow (L_1 || L_2) \subseteq (L_1 || L_3) \text{ and } (L_2 || L_1) \subseteq (L_3 || L_1).
\end{align*}
\]

**Example 4.4.9** Consider \( f_\beta \) in Example 4.3.1 and \( g_\beta \) in Example 4.3.3, we define \( \{f_\beta\} || \{g_\beta\} \), which is tabularly represented in Appendix B.
**Definition 4.4.19** Let \( L_1, L_2 \in M_S \), we define the union on time-slot languages \( L_1 \lor L_2 = (\bot \cdot L_1) \cup (\bot \cdot L_2) \).

We just recall that \( \bot \cdot L_1 \) appends new blank rows to all runs of \( L_1 \).

**Remark 4.4.18** If \( L_1 \) and \( L_2 \) are time-slot languages, then so is \( L_1 \lor L_2 \). Furthermore, \( \text{NAM}(L_1 \lor L_2) = \text{NAM}(L_1) \cup \text{NAM}(L_2) \).

**Remark 4.4.19** Let \( L_1, L_2, L_3 \in M_S \), then

\[
\begin{align*}
a) & \quad L_1 \lor L_1 = L_1 \\
b) & \quad L_1 \lor L_2 = L_2 \lor L_1 \\
c) & \quad (L_1 \lor L_2) \lor L_3 = L_1 \lor (L_2 \lor L_3) \\
d) & \quad L_1 \cdot (L_2 \lor L_3) = L_1 \cdot L_2 \lor L_1 \cdot L_3 \\
e) & \quad \emptyset \lor L_1 = L_1 \\
f) & \quad \text{NAM}(L_1) = \text{NAM}(L_2) \iff L_1 \lor L_2 = L_1 \cup L_2
\end{align*}
\]

Regarding \( \emptyset \lor L_1 \) in (e), \( \emptyset \lor L_1 = (\bot \cdot L_1) \cup (\bot \cdot L_1) \), since \( \bot \cdot \emptyset = \bot \), then \( (\bot \cdot L_1) \cup (\bot \cdot L_1) = \emptyset \cup (\bot \cdot L_1) = \emptyset \cup L_1 = L_1 \). In addition, similar to (d), all other operations are distributive over \( \lor \).

### 4.5 Component Protocol

The behaviour of a component is not as simple as handling a single event. Normally, it describes all possible maximal execution sequences throughout all services exhibited on components. Theoretically, these sequences can be infinite and even unrestricted [HU79] in general. A challenge is to find a finite notation to approximate the behaviour of a component by corresponding time-slot language. The notation should be simple enough to be easily applied in any component ADL specifications, manipulated by automated tools, and facilitated specification verification. The approach we proposed is mainly motivated by the Behaviour Protocols [PV02], which specifies behaviours by an extended regular language over event tokens. By adopting the initial algebra approach, we give a simple syntax for our behaviour language, the *component protocol* \( P \). It formulates the behaviour of the component at a higher level of abstraction – services. The language \( P \) is in effect a term algebra \( T_{\Sigma} \) for a particular signature \( \Sigma \). Each term \( t \) in \( T_{\Sigma} \) will represent a process. A semantics or interpretation for the language \( P \) is given by
defining a particular $\Sigma$-algebra $(M_\varphi, \Sigma_{M_\varphi})$, $M_\varphi$ for short. The meaning of a process $t$ is thus uniquely defined: it is $\sigma_{M_\varphi}(t)$, where $\sigma_{M_\varphi}$ is the unique $\Sigma$-homomorphism from the term algebra $T_\varphi$ to the $\Sigma$-algebra $M_\varphi$. The language $\mathcal{P}$ is in fact initial in the class of all $\Sigma$-algebras $[M_\varphi]$.

A simple view of components is that they are software entities which perform services. A very simple component is the one that can perform solely one service. We introduce a constant, or a function symbol of arity 0, to denote this process, $p$, which is the service name of that service. In general, every service in a component gives rise to (at least) one constant $p$. Therefore, we can consider the set $P$ of service names as a constant set of that $\Sigma$. Put in another way, the set $P$ is the alphabet of $\mathcal{P}$. Further, $\Sigma$ has an empty string denoted as $\lambda$.

Further, if $t_1, t_2$ are terms and $p, q \in P$, we introduce the following function symbols in our $\Sigma$.

$;(t_1, t_2)$ sequencing; represents the process which can perform $t_1$ and then perform $t_2$.

$+(t_1, t_2)$ alternative; represents the process that can act either like $t_1$ or like $t_2$.

$f(t_i)$ repetition; $^n(t_i)$ represents the process which can perform $t_i$ zero or more times; $^1(t_i)$ represents the process which can perform $t_i$ once or more times; $^? (t_i)$ represents the process which can perform $t_i$ zero or once times; and $^m (t_i)$ represents the process which can perform $t_i$ exactly $n$ times.

$\square (t_1, t_2)$ including; represents the process that can perform $t_2$ inside $t_1$.

$\mid (t_1, t_2)$ parallel; represents the process which can perform $t_1$ and $t_2$ in parallel.

$\parallel (t_1, t_2)$ synchronising; represents the process that can perform $t_1$ and $t_2$ simultaneously.

$\parallel (t_1, t_2)$ service synchronising; represents the process which can synchronise $t_1$ and $t_2$ on the common service $p$.

$\parallel (t_1, t_2)$ interleaving; represents the process which can perform $t_1$ and $t_2$ in any order.

$p \mapsto q (t_i)$ binding; represents a process of service interaction, where service $p$ requests the service $q$ in $t_i$. Note that $p$ could be the empty string $\lambda$.

This completes the description of the signature. So $\Sigma_p$, the signature in question, is completely determined by,
Chapter 4 Formalisation of Component-based Systems

- $(\Sigma_P)_0 = P \cup \{\lambda\}$, where $P$ is a set of service names
- $(\Sigma_P)_1 = \{^{(f)}\} \cup \{p \mapsto_q : p \in P\}$
- $(\Sigma_P)_2 = \{;+,[],||,|||\} \cup \{p : p \in P\}$
- $(\Sigma_P)_n = \emptyset$, for $n > 2$

The language $P$ is simply an element of the term algebra $T_{\Sigma_P}$. For readability, we use $P$ in place of $T_{\Sigma_P}$. It should be emphasised that elements of $P$ are open terms; they contain a set $P$ of service names from the component in question. We introduce the following conventions, which will make these function symbols more readable:

- for all $f \in (\Sigma_P)_2$, the function symbol $f$ is treated as an infix symbol so that $f(t_1,t_2)$ is written as $t_1 f t_2$
- repetition function symbols $f(t_1)$ are often rendered as $t_1 f$
- binding function symbol $p \mapsto_q (t_1)$ is rendered as $p \mapsto_q t_1$
- brackets are omitted whenever possible, following is the precedence of function symbols
  1. (highest) repetition ($^*,+,?,^*$)
  2. including ($[]$)
  3. binding ($p \mapsto_q$)
  4. synchronising ($||$)
  5. parallel ($|$)
  6. (lowest) service synchronising ($|||_p$), interleaving ($|||_p$), sequencing (;), alternative (+)

**Example 4.5.1** Let $a,b,c,d \in P$, then

- $a^*;b[c+d]$, represents a component which can perform service $a$ any number of times, and then perform service $b$, which further includes performing service $c$ or $d$ inside.
- $a[b][c;d]$, represents a component which can concurrently perform the sequences of services $c$ and service $a$ where service $b$ executes inside, followed by service $d$.
- $a[b][c;d]$, represents a component which can synchronously perform service $c$ and service $a$ in which service $b$ proceeds inside, followed by the service $d$. 

---

- 107 -
Chapter 4 Formalisation of Component-based Systems

- $a \rightarrow_{b} b[c \rightarrow_{d} d]$, suppose service $a$ and $d$ are in component $Comp_1$, service $b$ and $c$ are in component $Comp_2$, this expression represents a process of service interaction between $Comp_1$ and $Comp_2$, such that service $a$ in $Comp_1$ requests service $b$ in $Comp_2$, in which service $c$ in $Comp_2$ successively makes a call-back to service $d$ in $Comp_1$.

A semantic domain or interpretation for $P$ is given by the $\Sigma$-algebra $(M\varphi, \Sigma_{M\varphi})$, in which the time-slot model $M\varphi$ is the carrier, and for the signature $\Sigma_{M\varphi}$, each function symbol (and constant) $f_{M\varphi}$ interprets $f$ in $\Sigma_{P}$ over $M\varphi$. We also apply the above conventions to all $f_{M\varphi}$.

**Definition 4.5.1** Let $L_1, L_2 \in M\varphi$, $p, q \in P$ and $Q \subseteq P$, then we define

- $\lambda_{M\varphi} := \{L\}$
- $p_{M\varphi} := \{f_p\}$ for all $p_{M\varphi} \in \Pi_{M\varphi}$, where $f_p$ is a run, such that
  
  a) if $SEV(p) \in E$, then $f_p : \{p\} \times N^* \rightarrow E \cup \{\Omega\}$ by

  \[
  f_p(p, 1) = SEV(p) \\
  f_p(p, n + 1) = \Omega, \ n > 0
  \]

  b) if $SEV(p) \in F$, then $f_p : \{p\} \times N^* \rightarrow E \cup \{\Omega\}$ by

  \[
  (f_p(p, 1), f_p(p, 2)) = SEV(p) \\
  f_p(p, n + 1) = \Omega, \ n > 1
  \]

- $L_1 \cdot_{M\varphi} L_2 := L_1 \cdot L_2$
- $L_1 \lor_{M\varphi} L_2 := L_1 \lor L_2$
- $L_1^{\Pi_{M\varphi}} := \{\Pi_{NAM(L_1)}\}$, then $L_1^{(n+1)_{M\varphi}} := L_1^{n_{M\varphi}} \cdot L_1$, where $n = 0$, further
  
  a) $L_1^{\Pi_{M\varphi}} := \bigcup_{n=0}^{\infty} L_1^{n_{M\varphi}}$ b) $L_1^{\Pi_{M\varphi}} := \bigcup_{n=1}^{m} L_1^{n_{M\varphi}}$ c) $L_1^{\Pi_{M\varphi}} := L_1^{0_{M\varphi}} \cup L_1$
- $L_1[L_2]_{M\varphi} := L_1[L_2]$
- $L_1|_{M\varphi} L_2 := L_1 \mid L_2$
- $L_1 ||_{M\varphi} L_2 := L_1 \parallel L_2$
- $L_1(\bot)_{M\varphi} L_2 := L_1 \bot L_2$
- $L_1 \parallel_{M\varphi} L_2 := L_1 \parallel L_2$
- $(p \rightarrow_{q})_{M\varphi} L_1 := p_{M\varphi} p \rightarrow_{q} L_1$
For every $p \in \mathcal{P}$, it associates a unique singleton time-slot language $p_{\mathcal{M}}$, which denotes a single occurrence of a service named $p$. Therefore, we consider $p_{\mathcal{M}}$ as the dynamic specification of service $p$. According to the static definition of services, an occurrence of event lasts one time slot where it handles an action; and an occurrence of function lasts two time slots in which it sequentially handles actions in its invocation and reaction part. With regard to the assumptions of functions discussed earlier, the invocation must happen strictly before the reaction in the context of time slot sequence.

The alternative function symbol '+' semantically denotes the union of two time-slot languages. Notice that the choice is nondeterministic, in the sense that the choice has no control by the function symbol. Therefore, + can be called internal choice. However, taking the polarity of services into account, the choice becomes not purely nondeterministic. That is, for passive services, the choice made depends on whether their counterparts, active services, invoked them or not.

The repetition function symbols are used to specify how often the service is allowed to occur. From the definition, we can deduce that $L_{\mathcal{M}}^{(n+1)} \subseteq \cdots \subseteq L_{\mathcal{M}}^{n+1} \subseteq L_{\mathcal{M}}$. Note that $\text{NAM}(L_{\mathcal{M}}^{n+1}) = \text{NAM}(L_{\mathcal{M}})$ for all $n$ (by an easy induction), so $L_{\mathcal{M}}^{n+1}$, $L_{\mathcal{M}}^{n+2}$, and $L_{\mathcal{M}}^{n+3}$ have the same set of service names as $L_{\mathcal{M}}$ by ($f$) of Remark 4.4.19. Further, it is apparent that $L_{\mathcal{M}}^{n+1} = L_{\mathcal{M}}^{n+2} \cup L_{\mathcal{M}}^{n+3}$. Therefore, syntactically, $p^* = p^* + p^*$, for all $p \in \mathcal{P}$.

For the including function symbol '[]', the simplest $L_1$ is usually an occurrence of function. In this context, the including function symbol is very useful in modelling service nesting calls, which are mainly discussed in [M04, PV02]. For example, suppose a component has a provided function $p$ and a required function $q$, the behaviour of the component could be $p[q]$, which means service $p$ needs service $q$ to provide support for its service. Also, it makes easier to express nesting calls on the same service (function only).

For the parallel function symbol '|', the simplest application is to denote the occurrences of two services are in parallel. It is important to remark that it is not intended to synchronise the progress of two processes. It only means two processes execute in shared time slot sequences. For example, for a concurrent occurrence of an event with a function, the event may finish earlier than the function.

For the synchronising function symbol '||', the simplest usage is to synchronise the occurrences of a pair of services. If the pair is an event and a function, it implicitly means that the occurrence of an event is synchronising with that of the reaction of function.
Example 4.5.2 In order to demonstrate the difference between the parallel and synchronising function symbol, given function \(s, t\) and \(u\), \(s[t]\|u\) means synchronising both \(s\) and \(u\)'s invocations and reactions, while \(s[t]|u\) just leaves \(s\) and \(u\)'s reactions ordered free.

\[
s[t]\|u
\]

The service synchronising function symbol \(\|\) is mainly used for multi-party service interactions.

Example 4.5.3 We define a run \(k_s : \{r,z\} \times \mathbb{N}^* \rightarrow \{?a,?b,?l,?m\} \cup \{\Omega\}\) where

\[
\begin{align*}
k_s(r,1) &= ?a \\
k_s(z,2) &= ?l \\
k_s(r,4) &= ?b \\
k_s(z,3) &= ?m
\end{align*}
\]

The singleton time-slot language with run \(k_s\) can be denoted by the expression \(r[z]\). Then, the term \(p \leftrightarrow_s s \| p \leftrightarrow_r r[z]\) describes service \(p\) which broadcasts its request to both service \(s\) and \(r\), where service \(r\) further includes service \(z\). So, service \(p\) will complete as soon as all its requested sequences are completed, as illustrated above.

The interleaving function symbol \(\|\) denotes \(L_1\) and \(L_2\) execute independently of the other. This function symbol denotes the most liberal ordering for service execution. Therefore, we consider interleaving is the default execution order for all services. Any other function symbols introduced are just for restricting this order.

All function symbols above are expressive enough to specify any service execution sequences in a component. However, to be able to specify the service interaction logics over components, we introduce the binding function symbol \(\leftrightarrow_q\), which represents an atomic service interaction: for any constant \(p_M\), if its service matches the requested service \(q\) in \(L_1\), then they can interact in some way.
Example 4.5.4 Following Example 4.2.4, now we can use component protocol to specify the behaviour of the connector \textit{Conn} in the ticket vending machine.

- \( t_{\text{Glue}} = ci \rightarrow ts (sa \rightarrow sm sm \parallel ts \rightarrow rp rp[sp \rightarrow rp tp]) \parallel rp \rightarrow rc rc \), where

- \( t_{\text{RSel}} = ts \parallel sm; sr \)

- \( t_{\text{RPnt}} = rp[tp] \)

- \( t_{\text{RBuy}} = ci; rc \)

- \( t_{\text{RMon}} = sa \)

As introduced in Chapter 3, a connector is comprised of a glue and a set of roles. In this case, the connector has 4 roles: \textit{RSel}, \textit{RPnt}, \textit{RBuy} and \textit{RMon}. Each role describes a certain service set the potential component(s) should have. Let us assume that, \textit{RSel} acts as a vendor, \textit{RPnt} acts as a printer, \textit{RBuy} acts as a customer and \textit{RMon} acts as a monitor. Regarding the glue's interaction logics, it is noteworthy to point out that, (i) \( ci \rightarrow ts (sa \rightarrow sm sm \parallel ts) \), here, we synchronised the occurrence of \( sm \) and \( ts \), which means in order to monitor ticket selling, the self-monitoring service (\( sm \)) has to start and finish at the same time as the ticket-selling service (\( ts \)) does; (ii) \( rp[sp \rightarrow rp tp] \), means while issuing a receipt to the customer, the printer has to export the transaction data to the vendor. Furthermore, the printer needs to make sure that the latter process completes successfully before the former process completes; (iii) \( ci \rightarrow ts \rightarrow rp rp \), this is a kind of multi-party interaction, a so-called 'binding chain', that simply specifies the behaviour, such as the coin-inserting service (\( ci \)) calls the ticket-selling service (\( ts \)), which successively invokes the receipt-printing service (\( rp \)); (iv) \( ts \rightarrow rp rp[sp \rightarrow rp tp] \parallel rp \rightarrow rc rc \), in brief describes the procedure of the vendor asking the printer to issue a receipt to the customer, and to issue the transaction data to itself as well. Technically, without considering \( rp[sp \rightarrow rp tp] \), the term can simply be \( ts \rightarrow rp rp \rightarrow rc rc \), but taking into account this sub-expression, we cannot directly write \( ts \rightarrow rp rp[sp \rightarrow rp tp] \rightarrow rc rc \), because \( rp[sp \rightarrow rp tp] \) is not a service name to link \( \rightarrow rc rc \). An alternative solution is writing the
two parts (i.e. \( ts \rightarrow_{\rho} rp[\sigma_{sr} \rightarrow_{\rho} tp] \) and \( rp \rightarrow_{rc} rc \)) separately and using service synchronising to combine them together.

Following the initial algebra approach, \( (M_{S}, \Sigma_{M_{S}}) \) is a \( \Sigma \)-algebra and with every term in the language \( \mathcal{P} \) we can associate a unique meaning in \( M_{S} \), namely \( \sigma_{M_{S}} : \mathcal{P} \rightarrow M_{S} \), which states that there is only one way to interpret the syntactic language \( \mathcal{P} \) in the semantic domain \( M_{S} \).

We observe that the \( M_{S} \) determines an equality (technically, a congruence) over \( \mathcal{P} \). In this, \( t_{1}, t_{2} \in \mathcal{P} \), then \( t_{1} =_{M_{S}} t_{2} \iff \sigma_{M_{S}}(t_{1}) = \sigma_{M_{S}}(t_{2}) \). Thus, all \( =_{M_{S}} \) equations between terms of \( \mathcal{P} \) are based on algebraic properties of the operations on time-slot languages.

It is important to remark that for the equation \( t_{1};(t_{2} + t_{3}) =_{M_{S}} t_{1};t_{2} + t_{1};t_{3} \), it is usually frowned on by process algebraists, following from the fact that our semantics is entirely language based; effectively, it avoids internal choice. A more elaborate semantics involving failure set as in CSP is possible, but that is beyond the scope of this thesis, as is any discussions of equational reasoning as in, for example, Hennessy [Hen88].

### 4.6 Formalisation of Component-based Systems

So far, we have introduced the component signature \( \mathcal{S} \) (Definition 4.2.14), time-slot model \( M_{S} \) (Definition 4.3.14) and component protocol \( \mathcal{P} \). Now, we are able to use them to formalise a component.

**Definition 4.6.1** A component is a pair \((\mathcal{S}, L)\), where

- \( \mathcal{S} \), is the component signature
- \( L \), is the time-slot language over \( \mathcal{S} \)

The definition above indicates that (i) all services listed in the component signature \( \mathcal{S} \) are distinct; (ii) all services defined in \( \mathcal{S} \) will be manifested in the time-slot language \( L \), followed by Definition 4.3.10.

This definition gives a complete description of a component. It consists of the static structure described by a signature \( \mathcal{S} \), together with the dynamic description in a time-slot language \( L \) formed over \( \mathcal{S} \). The time-slot language can be built from a regular-like expression language, component protocol \( \mathcal{P} \). Therefore, the definition can be rephrased to read:

\[
\text{A component is a pair } (\mathcal{S}, \sigma_{M_{S}}(t)) \text{ where } t \in \mathcal{P}.
\]
Intuitively, the idea is that the component protocol indicates possible constraints on the order in which several services of the component can be or should be called.

It might be noteworthy that, there are a number of ways to restrict the component behaviour to allow sequences of services. In [BRF03, YS97] a finite state machine is attached to each component interface, in which case the allowed sequences are essential given by the language accepted by the machine. [Mos04] advocates the use of sequence diagrams, LSCs [DH01] in particular, for obtaining the component language based on the partial order induced by a sequence diagram, effectively building on earlier work in [Kf04b, Kf04a] on formalising the interactions that appear on sequence diagrams. Jin and Han [JH04] presented a language, PEIDL, for the interaction protocol specification of software components. Their language uses the labelled transition systems (LTS) as its formal basis. Our approach relies on the use of protocols. We use component protocols to model sequences of service occurrences at time slots. An alternative option might simply be a textual description (use cases) of intended behaviour.

In our CBS model, any kind of structural entity (such as services, interfaces, ports, components, connectors, and even roles and glues) can be formed as the above pair. The simplest (or atomic) one is a service, in which \( S \) solely contains one service name, and \( L \) denotes a single occurrence of the service (cf. Definition 4.5.1).

**Definition 4.6.2** Given a component \( C = (S, L) \), we define

- **function** \( NAM(C) = NAM(S) \), which associates the ordered pair \((S, L)\) to its service name set \( P \) of its \( S \)
- **function** \( SEV(C) = SEV(S) \), which associates the ordered pair \((S, L)\) to its service set \( S \) of its \( S \)

Generally speaking, our framework provides a formal basis for both Lau, Elizondo and Wang's exogenous connectors [LEW05] and Ducasse and Richner's executable connectors [DR97]. In these approaches, components are static entities that provide a set of method signatures as in CORBA's IDL [OMG02] and have not any dynamic specifications. In our model, both components and connectors are defined in ordered pairs, as \((S, L)\) (cf. Definition 4.6.1), which consists of a static signature \( S \), together with a dynamic language \( L \) over \( S \). This definition provides a more complete specification for a CBS model.

**Example 4.6.1** Following Example 4.2.4, now we can formally define those components and the connector in the ticket vending machine as follows.

- The vendor is \((S_{vendor}, L_{vendor})\), where \( L_{vendor} \) is formed by the expression \( ts || sm;sr \) on the port \( PSel \).
♦ The printer is \((S_{\text{Printer}}, L_{\text{Printer}})\), where \(L_{\text{Printer}}\) is formed by the expression \(rp || tp\) on the port \(PPnt\).

♦ The customer is \((S_{\text{Customer}}, L_{\text{Customer}})\), where \(L_{\text{Customer}}\) is formed by the expression \(ci;rc\) on the port \(PBuy\).

♦ The monitor is \((S_{\text{Monitor}}, L_{\text{Monitor}})\), where \(L_{\text{Monitor}}\) is formed by the expression \(sa\) on the port \(PMon\).

♦ The connector is \((S_{\text{Conn}}, L_{\text{Conn}})\), where \(L_{\text{Conn}}\) is actually formed by its glue \(ci \mapsto ts \ (sa \mapsto sm \ || \ ts \mapsto rp \ (sr \mapsto tp)) \ || \ rp \mapsto rc \ rc\), and \(S_{\text{Conn}} = (P_n, S_n, SEV_n)\)

\[P_n = \{ts, sr, sm, rp, tp, ci, rc, sa\}\]

\[S_n = \{(? insertCoins, ! produceTicket), (? recordTrans, ! printTrans), (? startMon, ! endMon),
(? produceTicket, ! printReceipt), (? recordTrans, ! printTrans), ! insertCoins,
? printReceipt, (? startMon, ! endMon)\}\]

\[SEV_n(ts) = (? insertCoins, ! produceTicket)\]
\[SEV_n(sr) = (? recordTrans, ! printTrans)\]
\[SEV_n(sm) = (? startMon, ! endMon)\]
\[SEV_n(rp) = (? produceTicket, ! printReceipt)\]
\[SEV_n(tp) = (? recordTrans, ! printTrans)\]
\[SEV_n(ci) = ! insertCoins\]
\[SEV_n(rc) = (? printReceipt\]
\[SEV_n(sa) = (? startMon, ! endMon)\]

4.7 Industrial Case Study

Starting with the scenario-based notation within the Koala component model \[vO03, vO04\], we give a more concrete semantics in terms of the time-slot model. In this section, we illustrate our approach by means of obtaining the formal definition of a configuration from the Koala model in the CE industry. This configuration includes four components and a fork; it is a real-life example extended from van Ommering’s thesis \[vO04\]. \[vO04\] makes a very interesting benchmark - his is much more focused on the industrial practice of components with quite limited behavioural
modelling. We solely focus on the behavioural modelling of Koala model, providing a complementary experiment to his work. We shall use this configuration as a running example to illustrate our formal approach throughout the rest of the thesis.

A typical use of a fork is to route the communication between a tuner and two (or more) video output components. Figure 4.6 shows such a configuration, where we use a shorthand notation (• and o) for a pair of Koala interfaces as introduced before. As a further simplification, we have omitted the driver components; this will not change the essence of our explanation.

We start again from the top-level control software calling the function \textit{Tune(f)} in \textit{A}. Component \textit{A} issues a drop request to fork \textit{F}, which in turn forwards this request to output component \textit{B}. Suppose that \textit{B} answers \textit{true}, \textit{F} subsequently forwards the drop request to the second output component \textit{C}. If \textit{C} answers positively too, then \textit{F} can return \textit{true} to \textit{A}, that can in turn change the frequency in its driver (not shown in Figure 4.6). \textit{A} then calls the restore command in \textit{F}, which forwards this to \textit{B} and \textit{C}, respectively.

We have shown the synchronous case only in Figure 4.6. If one of the output components returns \textit{false}, delaying the approval of the request, then fork \textit{F} must keep track of this and return \textit{false} as well (after having the drop request of both components). Component \textit{A} cannot proceed with the tune operation then. Fork \textit{F} must now wait for the component that has returned \textit{false} to call a drop acknowledgement in \textit{F}. On receipt of that, \textit{F} can call a drop acknowledgement in \textit{A}, which can in turn change the frequency and call the restore operation. Fork \textit{F} forwards the restore to \textit{B} and \textit{C} just as sketched in Figure 4.6. Note that the restore can be called synchronously or asynchronously, but this adds no extra complexity here.

If both output components delay the approval of the drop request, then fork \textit{F} must remember this, and keep count of the drop acknowledgements that \textit{B} and \textit{C} send later. Only on the second acknowledgement, \textit{F} may forward this acknowledgement to \textit{A}. The protocol then proceeds as described above. We illustrate this case in Figure 4.7. Naturally, this protocol can easily be extended to forks with more than two outputs.
Chapter 4 Formalisation of Component-based Systems

Figure 4.7 A Koala Fork with Asynchronous Drop Requests and Synchronous Restores

The above protocol gives an informal description of the normative behaviours of fork \( F \). We are interested in obtaining the formal description of fork \( F \) as following.

The signature of \( F \) is given by \( \mathcal{S}_F = (P_F, S_F, SEV_F) \) where


- \( S_F = \{!dropReq, ?dropReq,!dropAck, ?dropAck, 
\ (!dropReq, ?dropAck), (?dropReq,!dropAck), 
\ (?restore,!restoreAck), (!restore, ?restoreAck)\} \)

- \( SEV_F(EdrA) = !dropReq \)

\[ SEV_F(EdrB) = SEV_F(EdrC) = ?dropReq \]

\[ SEV_F(EdrAckA) = ?dropAck \]

\[ SEV_F(EdrAckB) = SEV_F(EdrAckC) = !dropAck \]

\[ SEV_F(FdrA) = (!dropReq, ?dropAck) \]

\[ SEV_F(FdrB) = SEV_F(FdrC) = (?dropReq,!dropAck) \]

\[ SEV_F(FrstA) = (!restore, ?restoreAck) \]

\[ SEV_F(FrstB) = SEV_F(FrstC) = (?restore,!restoreAck) \]
Chapter 4 Formalisation of Component-based Systems

From the picture above, the behaviour of fork $F$ is defined in the time-slot language $L_F$ which corresponds to a component protocol $t_F$, i.e., $\sigma_{M_F}(t_F) = L_F$.

- $t_F = (((EdrA \rightarrow EdrB \parallel EdrA \rightarrow EdrC EdrC);(EdrAckB \rightarrow EdrAckA EdrAckA \parallel 

\text{EdrAckC} \rightarrow EdrAckA EdrAckA)) + (FdrA \rightarrow FdrB FdrB \parallel FdrA \rightarrow FdrC FdrC));

(FrstA \rightarrow FrstB \parallel FrstA \rightarrow FrstC FrstC)$

The formal description of fork $F$ is thus given by $(S_F, L_F)$ where $S_F$ describes the structure of the fork in terms of its services (in/out events and provided/required functions) together with the set of service names associated with each. The fork's language $L_F$ comprises a comprehensive set of runs describing the observable behaviours of the fork in terms of its services occurring in discrete time intervals. Intuitively, a more detailed graphic presentation of fork $F$ is displayed in Figure 4.8.

![Figure 4.8 The 3-ary Koala Fork Connector F](image)

As depicted in Figure 4.8, fork $F$ is a connector comprising three roles ($RoleA$, $RoleB$ and $RoleC$) and a glue. The definition of fork $F$ is in effect from its glue that integrates all information held by roles, and coordinates and constrains the interactions between roles. On the other hand, each role stores the information concerning one participant alone. The definition of a role should be a partition of connector descriptions.
In the glue of fork $F$, the drop signal command is modelled in synchronous and asynchronous fashions. In the synchronous mode, the drop signal command is modelled as a function (for example, $F\text{dr}A$), that embodies a tightly-coupled causality relationship between the drop request and drop acknowledgement, i.e., regardless internal processes the reaction is normally initiated immediately after receiving the request. While in the asynchronous mode, the drop signal command is modelled by two independent events (for instance, $E\text{dr}A$ and $E\text{dr}\text{Ack}A$), in the sense that the causal relationship between the drop request and drop acknowledgement is loosely-coupled; that is, the reaction may be returned anytime after the request.

Unlike the behaviour protocol [PV02] plainly using a pair of events in modelling an operation call-like synchronous service, the component protocol $\mathcal{P}$ simplifies this thereby introducing the concept of function (comprising an invocation event and a reaction event), and associating functions with primitive protocol alphabet -- service names. This improvement helps reduce the complexity and increase the readability of modelling synchronous communications in protocol. For instance, the restore signal command of fork $F$ needs to be modelled as $E\text{rst}A; E\text{rst}\text{Ack}A$ in behaviour protocols, whereas simply $F\text{rst}A$ in component protocols.

The interacting logic $t_F$ is in fact a process of four multi-party bindings between services from $\text{Role}A$, $\text{Role}B$ and $\text{Role}C$. In each multi-party binding, the interacting services are two pairs of either matched events (for instance $E\text{dr}\text{Ack}B \leftrightarrow_{F\text{dr}A} E\text{dr}\text{Ack}A$ ) or matched functions (for example $F\text{rst}A \leftrightarrow_{F\text{rst}C} F\text{rst}C$). At this point, we should remark that: firstly, in addition to these ‘classic’ full binding modes, the component protocol further includes partial bindings, such as event-to-function binding, function-to-event binding, and etc. For instance, suppose $\text{Role}A$ replaces its drop acknowledgement event ($\text{dropAck}A$) with a provided function ($\text{dropAck}, !\text{AckConfirm}$) called $F\text{dr}\text{Ack}A$, which expands $\text{Role}A$ to send a confirmation message after receiving the drop acknowledgement in a more secure protocol. After that, $\text{Role}B$ (and $\text{Role}C$) cannot directly interact with the new $\text{Role}A$ anymore, because the traditional binding methods do not support this kind of event-to-function, cross-boundary bindings. One possible solution is to extend $\text{Role}B$ and $\text{Role}C$ correspondingly, in order to accommodate $\text{Role}A$’s change, but it may require further changes to connecting components ($B$ and $C$) so as to avoid incompatibility with the connector roles (see Section 5.4 for details). Fortunately, we can bypass this difficulty by using component protocol’s partial bindings. We can simply change $E\text{dr}\text{Ack}A$ to $F\text{dr}\text{Ack}A$ in $t_F$, i.e., $(E\text{dr}\text{Ack}B \leftrightarrow_{F\text{dr}A} F\text{dr}\text{Ack}A || E\text{dr}\text{Ack}C \leftrightarrow_{F\text{dr}A} F\text{dr}\text{Ack}C)$, where $!\text{dropAck} > (\text{dropAck}, !\text{AckConfirm})$ with the sense that $\text{Role}B$ (also $\text{Role}C$) need not necessarily wait for $\text{Role}A$’s confirmation. As we can see that the new kinds of binding would not disrupt the overall interactions and most importantly, they extend the collaborativity between services, thus they become one of the unique advantages of the component protocols.
Secondly, according to the horizontal communication protocol [vO03, vO04], RoleA interacts with RoleB and RoleC independently and there is no explicit order between the activities of RoleB and RoleC, so we use $\mathcal{P}$’s most liberal composition operator (||) to model the order between them. Multi-party bindings can be conveniently expressed in the component protocol $\mathcal{P}$. A more constraint multi-party binding example could be $FrstA \leftrightarrow FrstB FrstB[FrstA \leftrightarrow FrstC FrstC]$ , indicating that the interaction of RoleA to RoleC must happen inside the interaction between RoleA with RoleB.

The features of concurrency and nondeterminism are introduced into the protocol $\tau_r$ by use of the interleaving function symbol || and the Cartesian product-based language operations in the semantic domain. To understand $\tau_r$ in detail, we break $\tau_r$ down into the following three ports.

a) $(EdrA \rightarrow_{EdrB} EdrB \parallel EdrA \rightarrow_{EdrC} EdrC); (EdrAckB \rightarrow_{EdrAckA} EdrAckA) \parallel EdrAckC \rightarrow_{EdrAckA} EdrAckA)$

b) $(FdrA \rightarrow_{FdrB} FdrB \parallel FdrA \rightarrow_{FdrC} FdrC)$

c) $(FrstA \rightarrow_{FrstB} FrstB \parallel FrstA \rightarrow_{FrstC} FrstC)$

Part (a) is a sequential process of two multi-party bindings, describing the drop signal command in asynchronous mode. The first sub-protocol $(EdrA \rightarrow_{EdrB} EdrB \parallel EdrA \rightarrow_{EdrC} EdrC)$ denotes 7 runs.

<table>
<thead>
<tr>
<th>ID</th>
<th>Component Signature</th>
<th>Time Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EdrA</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>2</td>
<td>EdrB</td>
<td>1 !dr !dr</td>
</tr>
<tr>
<td>3</td>
<td>EdrC</td>
<td>1 ?dr</td>
</tr>
</tbody>
</table>

6 is a shorthand for the event !dropReq. Similarly, ?dr is a shorthand for the event ?dropReq.
The second sub-protocol \((EdrAckB \rightarrow EdrAckA \parallel EdrAckA \leftrightarrow EdrAckA, EdrAckA)\) has the similar structure, thus represents another 7 runs. By Cartesian concatenating two set of runs together, the protocol of part (a) generates 49 \((7 \times 7)\) runs which cover all possible orders of a drop signal command in an asynchronous fashion. The run on the right is one of the scenarios that shows RoleA sends a drop signal request to RoleB and RoleC simultaneously, thereafter RoleA receives an acknowledgement from RoleC followed by another one from RoleB.

Part (b) specifies the drop signal command in the synchronous mode. The protocol \((FdrA \leftrightarrow FdrB \parallel FdrA \leftrightarrow FdrC, FdrC)\) interprets a time-slot language including 9 runs.

\(^7\) is a shorthand for the event \(!\text{dropAck}\). Similarly, \(!\text{da}\) is a shorthand for the event \(!\text{dropAck}\).
Comparing part (a) that exhibits 49 runs, part (b) only denotes 9 runs. Technically, let us rewrite the protocol \((FdrA \rightarrow_{FdrB} FdrB \parallel FdrA \rightarrow_{FdrC} FdrC)\) in events and compare it with part (a).

- Part (b): \((EdrA \rightarrow_{EdrB} EdrB; EdrAckB \rightarrow_{EdrAckA} EdrAckA) \parallel (EdrA \rightarrow_{EdrC} EdrC; EdrAckC \rightarrow_{EdrAckA} EdrAckA)\)

- Part (a): \((EdrA \rightarrow_{EdrB} EdrB \parallel EdrA \rightarrow_{EdrC} EdrC); (EdrAckB \rightarrow_{EdrAckA} EdrAckA \parallel EdrAckC \rightarrow_{EdrAckA} EdrAckA)\)

It is easy to see that part (b) concatenates \(EdrA \rightarrow_{EdrB} EdrB\) with \(EdrAckB \rightarrow_{EdrAckA} EdrAckA\), and \(EdrA \rightarrow_{EdrC} EdrC\) with \(EdrAckC \rightarrow_{EdrAckA} EdrAckA\) individually at first, and then combines the two processes in interleaving. On the other hand, part (a) interleaves \(EdrA \rightarrow_{EdrB} EdrB\) with \(EdrA \rightarrow_{EdrC} EdrC\), and \(EdrAckB \rightarrow_{EdrAckA} EdrAckA\) with \(EdrAckC \rightarrow_{EdrAckA} EdrAckA\) individually first, and then concatenates the two processes together. Part (b) applies two concatenation and one interleaving operators, whereas part (a) exploits two interleaving and one concatenation operators, hence it makes part (a) more productive. This reinforces the fact that asynchronous communication is more flexible than synchronous communication.

With the same pattern of part (b), the restore signal part (c) presents 9 runs too. Now, using the alternative operator to link part (a) and part (b), and then concatenating the resulting protocol with part (c), the final protocol \(\tau_F\) denotes \((49 + 9) \times 9 = 522\) possible orders of the interaction amongst RoleA, RoleB and RoleC. The run below is one of the scenarios that chooses asynchronous drop signal command and synchronous restore signal command.
Chapter 4 Formalisation of Component-based Systems

<table>
<thead>
<tr>
<th>ID</th>
<th>Component Signature</th>
<th>Time Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name Service</td>
<td>1 2 3 4 5 6 7 8 9 10 11 12 13</td>
</tr>
<tr>
<td>1</td>
<td>EdrA ?dr Idr Idr</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>EdrB ?dr ?dr</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>EdrC ?dr ?dr</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>EdrAckA ?da ?da</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>EdrAckB ?da ?da</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>EdrAckC ?da ?da</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>FrstA (?r, ?ra)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>FrstB (?)r, !ra</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>FrstC (?)r, !ra</td>
<td></td>
</tr>
</tbody>
</table>

One such an alternative technique is used by a formal verification of the protocol, using the Labelled Transition System Analyser (LTSA) [MK99]. In LTSA, models are described textually by an algebraic language, Finite State Processes (FSP). In Uchitel et al’s paper [UCK’04], the authors use FSP to describe the behaviour of a Koala fork.

\[
T_{Tuner} = (t.f.dropReq \rightarrow f.t.dropReq\_ack \rightarrow t.f.restore \rightarrow f.t.restore\_ack \rightarrow T_{Tuner}).
\]

\[
V1\_Video = (t.f.dropReq \rightarrow f.v1.dropReq \rightarrow v1.f.dropReq\_ack \rightarrow f.t.dropReq\_ack \rightarrow t.f.restore \rightarrow f.v1.restore \rightarrow v1.f.restore\_ack \rightarrow f.t.restore\_ack \rightarrow V1\_Video).
\]

\[
V2\_Video = (t.f.dropReq \rightarrow f.v2.dropReq \rightarrow v2.f.dropReq\_ack \rightarrow f.t.dropReq\_ack \rightarrow t.f.restore \rightarrow f.v2.restore \rightarrow v2.f.restore\_ack \rightarrow f.t.restore\_ack \rightarrow V2\_Video).
\]

\[
|| F\_Fork = (T_{Tuner} || V1\_Video || V2\_Video).
\]

Figure 4.9 A Koala Fork with Asynchronous Drop Signal and Restore Commands in FSP

As shown above, the behaviour of a Koala fork, \( F\_Fork \), is a parallel composition of primitive processes \( T_{Tuner}, V1\_Video \) and \( V2\_Video \). Three processes synchronise with the common events in \( T_{Tuner} \) and interleave the \( f.v1.dropReq \) and \( f.v2.dropReq \), \( v1.f.dropReq\_ack \) and \( v2.f.dropReq\_ack \), \( f.v1.restore \) and \( f.v2.restore \), \( v1.f.restore\_ack \) and \( v2.f.restore\_ack \) in \( V1\_Video \) and \( V2\_Video \). To extend this fork to serve more video outputs, we can simply parallel compose \( F\_Fork \) with more drop video signal processes.

\(^8\) is a shorthand for the function \((\text{restore}, \text{restoreAck})\). Similarly, \((?r, !ra)\) is a shorthand for the function \((?\text{restore}, !\text{restoreAck})\).
FSP models concurrency by synchronising shared events and interleaving disjoint events from different processes. Shared events are the way that process interaction is modelled. Disjoint events are the way that independent process execution is modelled, representing in the form of composing arbitrary relative order of events from different processes but preserving of each process order. This parallel composition is similar to the interface parallel $||$ in CSP in such a way that the events of the processes must agree on occurrences of the events in the common interface $A$, but are otherwise independent. The operational semantics of this operator in FSP is given by Labelled Transition Systems (LTS). We note that while the use of LTS is extremely helpful to increase our insight into the protocol, LTS is a typical interleaving model since it implicitly describes concurrency, meaning that parallel executions are encoded by means of alternative computations obtained by interleaving transitions corresponding to actions executed by independent components. As a result of this, FSP is unable to address the simultaneity semantic in an adequate way.

In Figure 4.9, we use atomic events and parallel composition to describe asynchronous execution of the drop signal and restore commands. In modelling synchronous execution, since no synchronous service is introduced in FSP, we have to treat the drop signal and restore commands as single events $\text{dropReq}$ and $\text{restore}$, respectively. When combined in parallel, the resulting traces for the drop signal command are $f.v1.\text{dropReq} \rightarrow f.v2.\text{dropReq}$ and $f.v2.\text{dropReq} \rightarrow f.v1.\text{dropReq}$. Despite these two possibilities, by component protocols, we can additionally account for the case in which the reaction of $f.v1.\text{dropReq}$ is happening simultaneously with the invocation of $f.v2.\text{dropReq}$.

$$T\_\text{Tuner} = (t.f.\text{dropReq} \rightarrow t.f.\text{restore} \rightarrow T\_\text{Tuner}).$$

$$V1\_\text{Video} = (t.f.\text{dropReq} \rightarrow f.v1.\text{dropReq} \rightarrow t.f.\text{restore} \rightarrow f.v1.\text{restore} \rightarrow V1\_\text{Video}).$$

$$V2\_\text{Video} = (t.f.\text{dropReq} \rightarrow f.v2.\text{dropReq} \rightarrow t.f.\text{restore} \rightarrow f.v2.\text{restore} \rightarrow V2\_\text{Video}).$$

$$|| F\_\text{Fork} = (T\_\text{Tuner} || V1\_\text{Video} || V2\_\text{Video}).$$

Figure 4.10 A Koala Fork with Synchronous Drop Signal and Restore Commands in FSP.

In modelling concurrency and synchronous execution, this example demonstrates the component protocols to providing better expressive power than FSP. On the other hand, FSP has its own advanced features, such as guarded actions, action relabelling, and etc.
The Koala fork connector has three roles: RoleA, RoleB and RoleC. The formal definition of RoleA is an ordered pair \((S_{RA}, L_{RA})\), in which

- \(S_{RA}\), the signature of RoleA, is defined as \((P_{RA}, S_{RA}, SEV_{RA})\), where
  
a) \(P_{RA} = \{EdrA, EdrAckA, FdrA, FrstA\}\)
  
b) \(S_{RA} = \{!dropReq, ?dropAck, (?dropReq, ?dropAck), (!restore, ?restoreAck)\}\)
  
c) \(SEV_{RA}(EdrA) = !dropReq\)

\[SEV_{RA}(EdrAckA) = ?dropAck\]
\[SEV_{RA}(FdrA) = (?dropReq, ?dropAck)\]
\[SEV_{RA}(FrstA) = (!restore, ?restoreAck)\]

- \(L_{RA}\), the behaviour language of RoleA, is interpreted by the protocol \(t_{RA}\), in which

\[t_{RA} = (((EdrA||EdrA);(EdrAckA||EdrAckA)) + (FdrA||FdrA);(FrstA||FrstA))\]

Notice RoleA runs every service twice in the order of interleaving executions, because it needs to interact with both RoleB and RoleC. For example, the orderings of the interleaving over event \(EdrA\) could be \(EdrA; EdrA\) and \(EdrA||EdrA\) shown as below.

**ID** | **Component Signature** | **Time Slots**
---|---|---
1 | EdrA | !dr !dr !dr

**EdrA; EdrA**

**ID** | **Component Signature** | **Time Slots**
---|---|---
1 | EdrA | !dr !dr

**EdrA || EdrA**

Furthermore, the possible orderings of the interleaving on function Frst could be \(FrstA; FrstA\), \(FrstA || FrstA\) and \(FrstA[FrstA]\).

**ID** | **Component Signature** | **Time Slots**
---|---|---
1 | FrstA | (!r, ?ra) !r ?ra !r ?ra

**FrstA; FrstA**

**ID** | **Component Signature** | **Time Slots**
---|---|---
1 | FrstA | (!r, ?ra) !r ?ra

**FrstA || FrstA**

**ID** | **Component Signature** | **Time Slots**
---|---|---
1 | FrstA | (!r, ?ra) !r ?ra ?ra
Thus, $t_{RA}$ reflects $(2 \times 2 + 3) \times 3 = 21$ distinct runs in its operational semantics.

Similarly, we can define RoleB as a pair $(s_{RB}, l_{RB})$, where

- the signature of RoleB $s_{RB}$ is described in terms of $(p_{RB}, s_{RB}, sev_{RB})$, in which
  
  a) $p_{RB} = \{EdrB, EdrAckB, FdrB, FrstB\}$
  
  b) $s_{RB} = \{?dropReq, !dropAck, (?dropReq, !dropAck), (?restore, !restoreAck)\}$

  c) $sev_{RB}(EdrB) = ?dropReq$

  $sev_{RB}(EdrAckB) = !dropAck$

  $sev_{RB}(FdrB) = (?dropReq, !dropAck)$

  $sev_{RB}(FrstB) = (?restore, !restoreAck)$

- the behaviour of RoleB $l_{RB}$ is captured by the component protocol $t_{RB}$, where

  $t_{RB} = (EdrB; EdrAckB + FdrB); FrstB$

Equally, the description of RoleC can be defined in the same way.

Figure 4.11 The Component A Attached with Fork Connector F

Component $A$ is illustrated in Figure 4.11 above. Its formal description is $A = (s_A, l_A)$ where
the signature $\mathcal{S}_A$ of $A$ is defined by $(P_A, S_A, SEV_A)$, in which

a) $P_A = \{Ftune, EdrA, EdrAckA, FdrA, FrstA\}$

b) $S_A = \{(\text{?tuneReq, !tuneAck}), !\text{dropReq}, ?\text{dropAck},$  
\hspace{2cm} ($!\text{dropReq}, ?\text{dropAck}, !\text{restore}, ?\text{restoreAck})\}

c) $SEV_A(Ftune) = (\text{?tuneReq, !tuneAck})$

$SEV_A(EdrA) = !\text{dropReq}$

$SEV_A(EdrAckA) = ?\text{dropAck}$

$SEV_A(FdrA) = (!\text{dropReq}, !\text{dropAck})$

$SEV_A(FrstA) = (!\text{restore}, ?\text{restoreAck})$

the behaviour $L_A$ of $A$ is generated through the protocol $t_A$, where

$$t_A = \text{Ftune}(((\text{EdrA} \parallel \text{EdrA}); (\text{EdrAckA} \parallel \text{EdrAckA})) + (\text{FdrA} \parallel \text{FdrA})) \parallel (\text{FrstA} \parallel \text{FrstA})$$

As shown in Figure 4.11, component $A$ has two ports $PVA$ and $PHA$. Each port holds respectively interfaces $IVA$ and $IHA$. Interface $IVA$ can be defined as $(SIVA, LIVA)$, in which

the signature $\mathcal{S}_{IVA} = (P_{IVA}, S_{IVA}, SEV_{IVA})$, where

a) $P_{IVA} = \{Ftune\}$

b) $S_{IVA} = \{(\text{?tuneReq, !tuneAck})\}$

c) $SEV_{IVA}(Ftune) = (\text{?tuneReq, !tuneAck})$

the behaviour $L_{IVA} = \sigma_{M_A}(t_{IVA})$, in which $t_{IVA} = \text{Ftune}$

Clearly, $IVA$ is an interface having a single service $Ftune$. This definition can be also used to describe the service $Ftune$. Recall that any structural entity in this CBS model can be modelled in terms of $(\mathcal{S}, L)$, ranging from a single service to the whole CBS. In this definition, $(\text{?tuneReq, !tuneAck})$ is the actual service, $Ftune$ is the name, and the singleton time-slot language $L_{IVA}$ is the behaviour of the service being specified by the protocol $t_{IVA}$, denotes a single occurrence of the service illustrated as the table above.

Likewise, the definition of interface $IHA$ is a pair $(\mathcal{S}_{IHA}, L_{IHA})$, where

is a shorthand for the function $(\text{?tuneReq, !tuneAck})$. 

- 126 -
}\text{in which}
\begin{itemize}
\item[a)] \( P_{\text{IHA}} = \{ \text{EdrA}, \text{EdrAckA}, \text{FdrA}, \text{FrstA} \} \)
\item[b)] \( S_{\text{IHA}} = \{ \text{!dropReq}, \text{?dropAck}, (\text{!dropReq}, \text{?dropAck}), (\text{!restore}, \text{?restoreAck}) \} \)
\item[c)] \( \text{SEV}_{\text{IHA}}(\text{EdrA}) = \text{!dropReq} \)
\[ \text{SEV}_{\text{IHA}}(\text{EdrAckA}) = \text{?dropAck} \]
\[ \text{SEV}_{\text{IHA}}(\text{FdrA}) = (\text{!dropReq}, \text{?dropAck}) \]
\[ \text{SEV}_{\text{IHA}}(\text{FrstA}) = (\text{!restore}, \text{?restoreAck}) \]
\end{itemize}
\[ L_{\text{IHA}} = \sigma_{M_{\text{IHA}}} (t_{\text{IHA}}) , \text{ where} \]
\[ t_{\text{IHA}} = (((\text{EdrA} \mid \text{EdrA}); (\text{EdrAckA} \mid \text{EdrAckA})); (\text{FdrA} \mid \text{FdrA})); (\text{FrstA} \mid \text{FrstA})) \]

In this CBS model, concurrency starts from individual interfaces. For instance, the behaviour of interface \( \text{IHA} \) specifies every service occurring twice in the interleaving order. More concurrency will be produced when composing interface behaviours to ports, port behaviours to components, and so on. Note that we do not intend to further pursue concurrency at services, where a service occurrence is either a single event or a pair of events whose first event must happen strictly before its second event (denotes a function). Hence, there is no concurrency at this level of granularity.

The definitions of ports \( \text{PHA} \) and \( \text{PVA} \) could be directly obtained from their interfaces. Due to the fact that they merely have a single interface, no additional compositions are needed. However, the definition of component \( A (\mathcal{S}_A, L_A) \) is really generated from its ports \( \text{PHA} \) and \( \text{PVA} \), in such a way that the behaviour of \( \text{PHA} \) is included in that of \( \text{PVA} \), i.e., \( t_{\text{PVA}}(t_{\text{PHA}}) \).

We can define component \( B \) (also \( C \)) directly from their ports \( \text{PHB} \) and \( \text{PHC} \), so \( B = (\mathcal{S}_B, L_B) \),
\begin{itemize}
\item the signature of \( B \) is given by \( \mathcal{S}_B = (P_B, S_B, \text{SEV}_B) \), where
\item[a)] \( P_B = \{ \text{EdrB}, \text{EdrAckB}, \text{FdrB}, \text{FrstB} \} \)
\item[b)] \( S_B = \{ \text{?dropReq}, \text{!dropAck}, (\text{?dropReq}, \text{!dropAck}), (\text{?restore}, \text{!restoreAck}) \} \)
\item[c)] \( \text{SEV}_B(\text{EdrB}) = \text{?dropReq} \)
\[ \text{SEV}_B(\text{EdrAckB}) = \text{!dropAck} \]
\[ \text{SEV}_B(\text{FdrB}) = (\text{?dropReq}, \text{!dropAck}) \]
\end{itemize}
the behaviour of $B$ is specified as $L_B = \sigma_{\mathcal{M}_B}(t_B)$, in which $t_B = (EdrB; EdrAckB + FdrB) || FrstB$.

Component $D$ is designed as a configuration having a tuner controlling two video output components. As depicted in Figure 4.12, component $D$ is a composed component which wraps components $A$, $B$, $C$ and connector $F$, further maps component $A$'s port $PVA$ to its proxy port $PVD$. If the top-level control software wants to change the frequency of the tuner, it can simply calls the $Tune(f)$ in $D$. $D$ hides all communications between the tuner and two video outputs, and only replies a tune acknowledgement. From the perspective of top-level control software, the $Tune(f)$ function is the only service visible in $D$, all activities among $A$, $B$, $C$ through $F$ become internal.

As component $D$ just has a proxy port $PVD$, whose definition $(\mathcal{S}_{pvd}, L_{pvd})$ is identical to the definition $(\mathcal{S}_D, L_D)$ of $D$, where

- $\mathcal{S}_D$ denotes the component signature, formed as $(P_D, S_D, SEV_D)$ in which
  
  a) $P_D = \{Ftune\}$
4.8 Conclusion

Software components are increasingly expected to operate in a distributed and concurrent setting [KFP+04]. This makes strong demands in terms of component interactions and parallel behaviours. Therefore, the study of a suitable formal model for components points towards models introduced for describing concurrent computations such as Petri nets [Pet79b, Pet79a], CCS [Mil80], CSP [Hoa85], LOTOS [BB87], Mazurkiewicz traces [Maz88], event structures [NPW81, Win88], asynchronous transition systems [Shi85, Bed88], \(\pi\)-calculus [Mil99] among others.

The review of existing formal approaches to CBS design in Chapter 2 indicates that most models are concerned with concurrency arising through composition of components. The approaches seem to converge on treating this notion of concurrency in terms of the notion of parallel composition found in process algebras such as CCS [Mil80] and CSP [Hoa85]. In this context, parallel activity is modelled by imposing synchronisation on events in common between components (on bound matched services from each) which allow potentially concurrent execution of all other events.

Further difficulties arise when considering concurrency within a single interface. This is the case of multi-threaded interfaces where events can co-occur. It became apparent that existing formal approaches to the specification and analysis of components could not adequately incorporate this notion of concurrency as a property expressed with the formalism.

The interleaving interpretation of parallel behaviour, as well as corresponding parallel constructs in UML 2.0 sequence diagrams, Message Sequence Charts (MSC) [ITU00] and Live Sequence Charts (LSC) [DH01, HM03], in existing formalisms for components, with the exception of [KF04b, KF02] who consider true concurrency, re-enforce the concept that only a single event may occur at a time. Such an interpretation cannot faithfully distinguish between concurrency and nondeterminism and does not seem to be powerful enough for component-based design where the communication activities run in parallel and can change arbitrarily many variables at the same time.
In this sense, a non-interleaving model which incorporates at least the notion of true concurrency found in partial order models such as the event structures [NPW81, Win88], is needed for expressing concurrency at the level of individual interfaces. Further, in a concurrent setting it would be appropriate to model explicitly the case where events occur at exactly the same time, which amounts to our understanding of simultaneity. This case is not excluded in true concurrency but is not modelled explicitly either.

Considering that even if a component is to be implemented on a single processor machine there are options such as multi-threading to allow for servicing multiple requests in parallel, we would argue that simultaneity is a useful abstraction and it would be desirable to model it explicitly within a formal framework for components.

In this chapter, we have presented a formal behaviour specification language. It is purely a language-based model, where the time-slot model $M_\sigma$, its non-interleaving semantics introduced to encompass the above. Also, it gives an operational semantic domain for our syntactic language, the component protocol $P$. By the initial-algebra approach, we have demonstrated that every expression in $P$ has a unique meaning in $M_\sigma$.

A formal CBS model has also been introduced, in which a component is associated with (i) a signature, which describes its functionalities and (ii) a language defined over this signature, which describes its intended behaviour. The component signature defines the functionalities the component provides and requires, as well as their associated operations. The language part of a component comprises runs which contain a finite sequence of event occurrences that may be experienced on that component within an infinite sequence of time. Naturally, in component-based development we are interested in the intended or allowed sequences of services occurring on components. As a result, our goal is to restrict our works to an appropriate subset of runs that describe the intended behaviour only. This subset comprises the corresponding time-slot language, which can be approximated by the syntax language $P$.

Our syntax language, the component protocol $P$, is mainly inspired by Plasil and Visnovsky [PV02]. The authors propose a simple regular-like notation, the behaviour protocols, which originate in the path expressions earlier defined by Campbell and Habermann [CH74], aiming to provide a finite notation for expressing infinite and even unrestricted behaviours. Being a regular-like expression, a behaviour protocol is constructed by a set of events (alphabet) combining with classic operators used in regular expressions together with enhanced operators (such as interleaving/shuffling and restriction) and composed operators (such as composition and adjustment), to describe the ordering of events on component interfaces. More details can be found in Plasil and Visnovsky [PV02] and Section 2.3.1.
In our model, the component behaviour is approximated by the component protocol $P$, which also takes a form similar to regular expressions (see for example [Coh97]), constructed by component services as its alphabet. Like other state-of-the-art models for example, [OMG04a, JH04, LV95], two kinds of services are specialised: asynchronous events and synchronous functions. A function is in fact a pair of events. It distinguishes the beginning of an operation invocation from its termination. According to Jin and Han [JH04], compared with existing approaches such as that of, Plasil and Visnovsky [PV02] who consider operation invocations to be atomic units of control, this distinction leads to more accurate specifications of the relative sequencing between operation invocations.

In addition, we introduced the including operator to facilitate describing service nesting calls, which are expressed via $\{\}$ in [Men04, PV02]. We provided a formal definition for this notion.

In terms of the concurrency semantics, we proposed the use of an additional operator for capturing simultaneous service occurrences. Simultaneity was considered, as one possible case, in the semantics of the parallel operator. The semantics of the simultaneity is different in that it says the associated services have to occur simultaneously. The difference between the parallel and the simultaneity can be understood as the distinction between services may and must happen concurrently. Hence, the simultaneity can be understood as a refinement of parallelism. In addition, considering the parallel operator synchronises the whole time sequences of associated service occurrences, the time-slot parallel is introduced to synchronise partial time sequences of them. Also, the service simultaneity is used to synchronise two processes on common service. The interleaving semantics is proposed to capture the idea in terms of services that are running in any possible order. On the other hand, the behaviour protocol simply provides two interleaving-like operators (and/or-parallel).

Finally, we introduced a binding operator in modelling service interactions. It not only supports typical event-to-event and function-to-function kinds of classic interactions, but also event-to-function, function-to-event and function-to-function (chained) kinds of partial interactions. When compared with conventional binding operators, our binding operator offers a wider range possibility of service interactions, it maximises the notion of component interoperability. We believe it is also more practical and closer to the implementation level. Further, multi-party interactions are deliberately taken into account (cf. Example 4.5.3).

In addition to traditional event-to-event and function-to-function bindings, partial binding supports event-to-function bindings, this pattern of cross-boundary interactions further increases the portability and interoperability of components. We will formally define this concept in Section 4.2.
Like related formal models [Bro03, BRS'00, KRB96], our semantic domain, the time-slot model, works with discrete time and regards time as a chain of equidistant time intervals. We use \( \mathbb{N}^+ \) as an abstract time axis and assume a time synchronous model because of the resulting simplicity and generality. This means that there is a global time scale that is valid for all part of the modelled system.

To recap, [Bro03, BRS'00] use the notion of timed streams (a data stream with discrete time), i.e. finite or infinite sequences of elements from a given domain, to represent histories of conceptual entities that change over time. Given a message set \( M \), a timed stream is defined by a function

\[
s : \mathbb{N} \rightarrow M^*
\]

where \( M^* \) is the set of sequences over \( M \). For each time \( t \) the sequence \( s(t) \) denotes the sequence of messages communicated at time \( t \) in the stream \( s \). The expression \( s \downarrow t \) denotes the stream containing only the elements of \( s \) till time \( t \). The behaviour of a component is comprised of a set of timed streams. Each of them models the communication history of the associated input and output channel of the component.

Klein et al [KRB96] produced a different representation of timed streams. Assuming that time proceeds in a sequence of equivalent time intervals, the authors model the proceeding of time by one time interval using a time signal \( \sqrt{\cdot} \), called tick. \( M^t \) is denoted as a set \( M \cup \{ \sqrt{\cdot} \} \), and the authors define

\[
\begin{align*}
\bullet M^\infty &= \{ s \in (M^t)^\infty : \#(\text{Filter}(\{\sqrt{\cdot}, s\})) = \infty \} \\
\bullet M^* &= (M^t)^*
\end{align*}
\]

where \( \#: M^\infty \rightarrow \mathbb{N} \cup \{\infty\} \) delivers the length of the stream and \( \text{Filter} : \mathcal{P}(M) \times M^\infty \rightarrow M^\infty \) deletes all elements in \( s \in M^\infty \) which are not contained in set \( N \in M^\infty \).

The set \( M^\infty \) is the set of all infinite sequences of elements from \( M^t \), which contains infinitely many copies of \( \sqrt{\cdot} \). The requirement of infinitely many copies of \( \sqrt{\cdot} \) embodies the reality that time never ends and that the authors consider only infinite communication histories. Streams over \( M^t \) contain only finitely many messages from \( M \) between two ticks. The set \( M^* \) will be used in the sequel to speak about finite prefixes of infinite streams.

Comparing timed streams and the time-slot model, the most notable difference is that the timed stream is a fictitious clock model [AD94], in which the interpretation of a timed execution trace is that events occur in the specified order at real-valued times, but only the (integer) readings of the actual times with respect to a digital clock are recorded in the trace; while the time-slot model is a
discrete-time model [AD94], which requires that continuous time be approximated by choosing some fixed quantum \textit{a priori}. The \textit{fictitious clock model} is similar to the \textit{discrete-time model}, in that both models are conceptually simple to manipulate behaviours using finite automata, but the compensating disadvantage is that they represent time only in an approximate sense. In addition, the timed stream is designed to model data flows between input and output channels of components, while the time-slot model is devised to model control flows of the occurrences of services during component executions. Furthermore, the time-slot model provides a richer set of operations for managing process compositions especially in a concurrent setting.

In the earlier work of timed streams Klein et al [KRB96], the authors do not impose any requirement on the time difference between the sending and the receiving of a message. This is so-called instantaneous reaction: the output at time \( t \) may depend on the input at time \( t \). This would result in an oracle, which is normally impractical for modelling functions (where two events run in strict sequence), in the sense that there does not exist a machine which ultimately runs fast so that the local computation time can be easily ignored. Instantaneous reactions may also lead into problems with causality if we consider additional delay free feedback loops. Therefore, upon receiving a message, we believe it is more realisable to respond it in the next time slot, and leave a time unit for the machine to process this request. In this way, the causality between input and output is guaranteed and explicitly visible according to the fine time granularity. A similar thought appeared in later works on time streams by [BRS'00, Bro03, Bro04].

The explicit notion of time, i.e., global clocks, makes the time-slot model highly synchronous. That is to say, synchronisation is determined by time slots, rather than the identity of the events occurring in them. Synchronous CCS (SCCS) algebra as developed by Miller [Mil83] is another well-known synchronous model in the literature. We can observe that the parallel composition in SCCS is synchronous. Considering

\[
q_1 \rightarrow^a q_1'q_2 \rightarrow^b q_2'
\]

In this, event \( a \) and \( b \) are some elements of the set of actions. After parallel composition, event \( a \) and \( b \) happen simultaneously, but their identities or the processes which involve them are lost. For example, \( a \cdot b = b \cdot a \) (presumably) and maybe \( a \cdot b = c \cdot d \). Therefore \( q_1 | q_2 \rightarrow^{a \cdot b} q_1' | q_2' \) does not give any information about the individual component actions. This is a ‘good thing’ from the algebraic point of view, but we need to preserve more detail. Technically, our \( \text{Runs}_\varphi \) is a free and non-commutative monoid (so events \( a \) and \( b \) can be recaptured from \( d = a \cdot b \)). Certainly, the approach of hiding unnecessary detail has all the benefits of abstraction, though the question of what constructed ‘unnecessary detail’ depends on specific requirements. In our case, we feel that
it is necessary to keep more detail, since any discussion of behaviour needs to refer to the events participating in it. The time-slot model is much more concrete, and makes implicit reference to global clocks.

Comparing with other parallel compositions, CSP and LOTOS processes can synchronise on identically named actions, CCS agents synchronise only on actions with complementary names, for example, \( a \) and \( \overline{a} \), while the time-slot model synchronises runs on time slots. The main difference to others is again the explicit use of time slots and the concrete description of interactions.

The fundamental difference in expressing concurrency is as follows. By departing from classic CSP concurrency, we are able to consider concurrency within a single interface of a component. In CSP, and related process algebras, concurrency arises through composition. Here we have not yet been concerned with composing sequences from different interfaces, though this may also produce concurrency. Moschoyiannis and Shields’s component vector language [MS03] considers concurrency within a single component, on the assumption that services occur sequentially on a single interface (one at a time). Concurrent and simultaneous service occurrences can only engage distinct interfaces of the component. Our language-based description of component behaviour further relaxes this assumption to allow services happen concurrently and even simultaneously within a single interface. By considering concurrency at the interface level, our time-slot model shows a finer granularity of concurrency than that of the component vector language [MS03]. This improvement enables component engineers to define multi-thread component interfaces.

Adopting the vector based approach, the time-slot model is, in part, based on vectors of sequences in the sense that if \( f \) is a run \( f: Q \times N^+ \rightarrow E \cup \{Q\} \), then we can regard \( f \) as a vector in which \( f(p) = f(p,1)...f(p,\ell(f)) \).

Other vector based models can be found in [Shi97, Arn02, MS03]. In each case, we have a language of vectors, where a vector is a function from some fixed indexing set \( I \) into a set of sequences. These models are distinguished by constraints on the kind of vector allowed.

In each case, we have a function \( \alpha: I \rightarrow \wp(A) \) (with \( A = \bigcup_{i \in I} \alpha(i) \)); if the elements of \( I \) are, for example, process identifiers, then \( \alpha(i) \) is the set of actions associated with process \( i \). In all three, we have a concatenation operation where \( (a \cdot b)(i) = a(i) \cdot b(i) \).

In Shields’ model [Shi97], vectors are constrained to be concatenations of ‘event vectors’, \( a_a \), \( a \in A \), where \( a_a(i) = a \) if \( a \in \alpha(i) \), otherwise \( a_a(i) = Q \).

Nivat and Arnold [Arn02] take a more general approach in which vectors are constrained to be concatenations of a fixed set of ‘synchronisation vectors’, that is vectors \( x \) such that \( |x(i)| \leq 1 \).
Moschyiannis and Shields [MS03] make no constraints on individual vectors. Instead, the constraints are on possible languages. The component vector language is required to be normal (also known as ‘well formed’), a concept arising from theoretical considerations of partial order semantics, but [MS03] have shown to have practical value in detecting emergent behaviour.

The main difference between Shields’ vector language [Shi97] and Moschyiannis and Shields’ component vector language [MS03] is that the former describes the purely asynchronous behaviour of the system of processes synchronising on shared events. The latter expresses simultaneity and concurrency, and temporal relations are defined globally with respect to the language. For example, \((a, -), (-, b)\) are

- **mutually exclusive** in \{\((-,-), (a,-), (-, b)\}\)
- **concurrent** in \{\((-,-), (a,-), (-, b), (a,b)\}\) (a)

Therefore, concurrency of component vectors [MS03] has to depend on the whole language. In other vector models [Shi97, Arn02], the concurrency is intrinsic to the column vectors (a specific kind of vectors in that each of their coordinates is either the empty sequence or a single event). For example, in the case (a) above, \((-, b, b)\) is a column vector where the event in question is 'b'.

Another difference among three vector models is vector decomposition. Shields’ vectors [Shi97] can decompose uniquely - up to the order of independent column vectors. Thus,

\[(ab, c, ac) = (a, -, a), (b, -, c), (-, c, c), (-, c, b)\]

This is because of the way the vector model is defined - vectors are products of column vectors. And these generators satisfy relations but only of the form \(a.b = b.a\). For instance, it is possible in Shields’ vector language [Shi97] for one to have two distinct sets of column vectors \(s_1, \ldots, s_m\) and \(t_1, \ldots, t_n\), such that \(s_1, \ldots, s_m = t_1, \ldots, t_n\) if and only if \(s_1, \ldots, s_m\) is some permutation of \(t_1, \ldots, t_n\), as the example above.

Similarly, Nivat and Arnold’s vectors [Arn02] are formed from coordinatewise products of column vectors, as in the path expressions model [LTS79]. However, they do not have unique decomposition (up to partial commutativity) into synchronisation vectors, because these generators may satisfy non-trivial relations other than those of the form \(a.b = b.a\). For example, it is possible for a Nivat and Arnold’s vector language [Arn02] to have two distinct sets of synchronisation vectors \(s_1, \ldots, s_m\) and \(t_1, \ldots, t_n\), such that \(s_1, \ldots, s_m = t_1, \ldots, t_n\). There are not any specific relations they all satisfy. Nivat and Arnold did not interpret any contemporaneous relation (for example concurrency and simultaneity) in his model. Shields did this in his book [Shi97] (in Chapter 25).
Conversely, Moschoyiannis and Shields' component vectors [MS03] can decompose into column vectors uniquely up to commutativity, but only within the context of a particular normal component language. In general, component vectors are not generated by column vectors. It is a property of normal languages that this happens, it is not part of their definition. For example, in the language

\[ \{(-,-),(a,-),(-,b),(a,b)\} \]

\((a,b)\) can decompose as \((a,-),(-,b) = (-,b),(a,-)\), whereas in the language

\[ \{(-,-),(a,b)\} \]

\((a,b)\) cannot decompose any further. There is unique decomposition here too, but for a different reason.

The runs in our time-slot model do not satisfy any relation because essentially we are working within free monoids. Additionally, our time-slot model differs from Nivat and Arnold's vector model [Arn02] in that we preserve empty time slots in concatenation but Nivat and Arnold's model [Arn02] does not.

Overall, these vector models allow explicit representation of concurrency and/or simultaneity. We feel, however, given the intended application of the model, that it is safe to assume the existence of global clocks, which can unambiguously assign service occurrences to specific time slots. Finally, all above models can be transformed into discrete behavioural presentations as initially developed by Shields [Shi88, Shi97]. The interested reader should refer to [Shi97, MS03] for further details.
Chapter 5

Properties of Component-based Systems

In this chapter, we discuss the architectural properties of our CBS model, by means of the properties of well-definedness, well-behavedness and refinement on component protocols introduced in Section 5.1. The well-definedness means the expression can be completely interpreted into a time-slot language; no illegal phrases are allowed. The well-behavedness can be considered as a weaker version of well-definedness, where illegal phrases are permitted as long as the resulting time-slot language is not an empty set. The refinement is to evaluate the semantical inclusion between two expressions. The deadlock-freedom on the glue is covered in Section 5.2, aiming to assess the safety in the workflow of inter-component cooperation. The loyalty of a role with a glue is presented in Section 5.3, in order to ensure each role is faithfully representing a certain portion of the glue. A discussion of the compatibility of a role with a port is introduced in Section 5.4. Within our generic CBS model, the compatibility is checked by determining if a role is able to delegate a port to interact with its partners. The substitutability on ports is discussed in Section 5.5. It is used to assure the safety in component replacement. The demonstration of our approach is taken from a case study on a Koala configuration in the CE industry in Section 5.6. We end this chapter with a discussion about where our architectural properties reside in the literature of component interoperability.

5.1 Properties of the Component Protocol

By using the component protocol to specify the behaviour of components, we are also interested in the properties of the component protocol, such as whether a given expression is meaningful. In what follows, we are going to discuss several properties of component protocol.

In mathematics, the term well-defined is used to specify that a certain concept (for example a function, a property or a relation) is defined in a mathematical or logical way using a set of base axioms in an entirely unambiguous way. Usually definitions are stated unambiguously, and there is no question about their well-definition.

In our model, a well-defined $P$ expression basically requires all its sub-expressions to be meaningful in the semantic domain. Put in another way, given an expression, by interpreting it into a time-slot language, none of its sub-sequences will encounter an empty set $\emptyset$. That means
any sub-sequences in the given expression are able to represent a certain portion of time slot sequence. If so, we shall say that the given expression is well-defined. Defining the well-definedness of component protocol will use the following assistant function.

**Definition 5.1.1** We define a function \( \text{NAM} : \mathcal{P} \rightarrow \wp(\mathcal{P}) \), such that given an expression \( t \), \( \text{NAM}(t) \) is the set of all service names (alphabet) in \( t \).

Considering the expression \( t \) as a word of \( \mathcal{P} \), its constant set of service names is the alphabet of \( \mathcal{P} \), which is indeed the set of atomic elements in \( \mathcal{P} \). By convention, \( \text{NAM}(\lambda) = \emptyset \).

**Example 5.1.1** Following Example 4.5.1, \( \text{NAM}(a^*; b[c + d]) = \text{NAM}(a[b]; c; d) = \text{NAM}(a[b]; c; d) = \text{NAM}(a \mapsto b; c \mapsto d; d) = \{a, b, c, d\} \).

**Definition 5.1.2** Suppose that \( t \in \mathcal{P} \), then we shall say that \( t \) is well-defined if \( \text{NAM}(\sigma_{M_p}(t)) = \text{NAM}(t) \).

In effect, the well-definedness property estimates the syntactical correctness of the given expression. We know that, using the semantic function \( \sigma_{M_p} \), every expression in \( \mathcal{P} \) can be interpreted into a series of operations on time-slot languages. By the definitions of these operations in Chapter 4, we can notice that while combining individual runs, the operations always exclude 'undefined' results. It leads to the fact that the service names in the meaningless phrase will not be included in the result time-slot language. Therefore, it turns out that, given an expression \( t \) in \( \mathcal{P} \), if its atomic elements set \( \text{NAM}(t) \) equals to the set \( \text{NAM}(\sigma_{M_p}(t)) \) of service names in the corresponding time-slot language, then the expression \( t \) is well-defined; it also implies that all operations over services are semantically well-defined. By convention, the expression \( \lambda \) is well-defined, since \( \text{NAM}(\sigma_{M_p}(\lambda)) = \text{NAM}(\{1\}) = \emptyset = \text{NAM}(\lambda) \).

**Example 5.1.2** Case 1: consider the term \( t_1 = a; b[c + d] \), suppose \( \text{SEV}(b) \in E \), then

\[
\text{NAM}(\sigma_{M_p}(t_1)) = \text{NAM}(\sigma_{M_p}(a) \cdot (\sigma_{M_p}(b)[\sigma_{M_p}(c)] \vee \sigma_{M_p}(b)[\sigma_{M_p}(d)]))
\]

\[
= \text{NAM}((f_a \cdot (\emptyset \vee \emptyset)) \quad \text{Remark 4.4.19 (e)}
\]

\[
= \text{NAM}((f_a \cdot \emptyset) \quad \text{Remark 4.3.3 (b)}
\]

\[
= \text{NAM}(\emptyset)
\]

\[
= \emptyset
\]

while \( \text{NAM}(t_1) = \{a, b, c, d\} \), so \( \text{NAM}(\sigma_{M_p}(t_1)) \neq \text{NAM}(t_1) \). As a result of this, the expression \( t_1 \) is not well-defined.
Case 2: consider the term \( t_2 = a;b[c + d] + e[f] \), where \( \text{SEV}(b) \in F \) and \( \text{SEV}(e) \in E \), then

\[
\text{NAM}(\sigma_{\mathcal{M}_2}(t_2)) = \text{NAM}(\sigma_{\mathcal{M}_2}(a) \bullet (\sigma_{\mathcal{M}_2}(b)[\sigma_{\mathcal{M}_2}(c)] \lor \sigma_{\mathcal{M}_2}(b)[\sigma_{\mathcal{M}_2}(d)]) \lor \sigma_{\mathcal{M}_2}(e)[\sigma_{\mathcal{M}_2}(f)])
\]

\[
= \text{NAM}(\{f_a\} \bullet (\{f_b\} \lor \{f_b\} \lor \{f_d\})) \lor \emptyset
\]

\[
= \text{NAM}(\{f_a\} \bullet (\{f_b\} \lor \{f_b\} \lor \{f_d\}))
\]

Remark 4.4.19 (e)

\[
= \{a, b, c, d\}
\]

which is a subset of \( \text{NAM}(t_2) = \{a, b, c, d, e, f\} \). So the expression \( t_2 \) is not well-defined.

Additionally, if \( \text{SEV}(e) \in F \), then

\[
\text{NAM}(\sigma_{\mathcal{M}_2}(t_2)) = \text{NAM}(\sigma_{\mathcal{M}_2}(a) \bullet (\sigma_{\mathcal{M}_2}(b)[\sigma_{\mathcal{M}_2}(c)] \lor \sigma_{\mathcal{M}_2}(b)[\sigma_{\mathcal{M}_2}(d)]) \lor \sigma_{\mathcal{M}_2}(e)[\sigma_{\mathcal{M}_2}(f)])
\]

\[
= \text{NAM}(\{f_a\} \bullet (\{f_b\} \lor \{f_b\} \lor \{f_d\})) \lor \emptyset
\]

\[
= \text{NAM}(\{f_a\} \bullet (\{f_b\} \lor \{f_b\} \lor \{f_d\}))
\]

\[
= \{a, b, c, d, e, f\}
\]

\[
= \text{NAM}(t_2)
\]

Hence, we can say that the expression \( t_2 \) is well-defined.

Case 3: if \( t_3 = a; b[c + \lambda] + e; \lambda \), where \( \text{SEV}(b) \in F \), then

\[
\text{NAM}(\sigma_{\mathcal{M}_3}(t_3)) = \text{NAM}(\sigma_{\mathcal{M}_3}(a) \bullet (\sigma_{\mathcal{M}_3}(b)[\sigma_{\mathcal{M}_3}(c)] \lor \sigma_{\mathcal{M}_3}(b)[\sigma_{\mathcal{M}_3}(\lambda)]) \lor \sigma_{\mathcal{M}_3}(e) \bullet \sigma_{\mathcal{M}_3}(\lambda))
\]

\[
= \text{NAM}(\{f_a\} \bullet (\{f_b\} \lor \{f_b\} \lor \{f_d\})) \lor \{f_a\} \bullet \{\downarrow\})
\]

\[
= \text{NAM}(\{f_a\} \bullet (\{f_b\} \lor \{f_b\} \lor \{f_d\}) \lor \{f_a\} \bullet \{\downarrow\})
\]

Prop. 4.4.1 (b), Rem. 4.3.1 (b)

\[
= \{a, b, c, e\}
\]

which equals to \( \text{NAM}(t_3) = \{a, b, c, e\} \). Therefore, \( t_3 \) is well-defined.

Comparing the first two cases above, we can see that in case (1), the whole process collapsed under the sub-expression \( b[c + d] \), so the final result is \( \emptyset \). However, in case (2), part of the interpretation (of \( a; b[c + d] \)) remains in the resulting language, so the result will not be an empty set \( \emptyset \) (as case (1)). In order to identify this discrepancy, we further introduce the well-behavedness property to component protocols.
**Definition 5.1.3** Suppose that \( t \in \mathcal{P} \), then we shall say that \( t \) is well-behaved if \( \sigma_{M,s}(t) \neq \emptyset \).

In plain words, this definition says that, given an expression \( t \), by interpreting it to a time-slot language using the semantic function \( \sigma_{M,s} \), if the resulting time-slot language is not empty (i.e., there exists at least one run), then we shall say that the expression \( t \) is well-behaved. Also, it implicitly means that the expression \( \lambda \) is always well-behaved, as \( \sigma_{M,s}(\lambda) = \{1\} \neq \emptyset \).

**Example 5.1.3** Following Example 5.1.2, we say that the terms \( t_2, t'_2 \) and \( t_3 \) are well-behaved, because their corresponding time-slot languages are not empty. However, \( t_1 \) is not well-behaved, because \( \sigma_{M,s}(a;b[c + d]) = \emptyset \).

To investigate the relation between two properties further, the definition of well-behavedness can be rephrased to read: the expression \( t \) is well-behaved, if

- \( \text{NAM}(\sigma_{M,s}(t)) = \text{NAM}(t) = \emptyset \), or
- \( \text{NAM}(\sigma_{M,s}(t)) \neq \emptyset \land \text{NAM}(\sigma_{M,s}(t)) \subseteq \text{NAM}(t) \)

Despite the empty string \( \lambda \), the well-behavedness property is in fact a weaker version of the well-definedness property. It weakens the condition from equivalent set (\( \text{NAM}(\sigma_{M,s}(t)) = \text{NAM}(t) \)) to nonempty subset (\( \text{NAM}(\sigma_{M,s}(t)) \neq \emptyset \land \text{NAM}(\sigma_{M,s}(t)) \subseteq \text{NAM}(t) \)).

**Remark 5.1.1** Let \( t \in \mathcal{P} \), if \( t \) is well-defined, then \( t \) is also well-behaved.

Using the bottom-up approach to specifying component behaviours, the behaviour of a component must come from those of its ports, which in turn comes from the behaviours of their interfaces. Moreover, while attaching ports with roles, we should be able to guarantee that the port respects all of the role's obligations to interact with its partner. To perform these kinds of behaviour checking, we introduce a refinement relation to component protocols.

**Definition 5.1.4** Let \( t_1, t_2 \in \mathcal{P} \), then we define \( t_1 \sqsubseteq t_2 \iff \sigma_{M,s}(t_2) \subseteq \sigma_{M,s}(t_1) \).

The \( \sqsubseteq \) relation is related to the congruence \( =_{M,s} \) on \( \mathcal{P} \), just as \( t_1 =_{M,s} t_2 \iff \sigma_{M,s}(t_1) = \sigma_{M,s}(t_2) \), we can define \( t_1 =_{M,s} t_2 \iff t_1 \sqsubseteq t_2 \land t_2 \sqsubseteq t_1 \).

**Remark 5.1.2** Let \( t_1, t_2, t_3 \in \mathcal{P} \), then

\[ a) \quad t_1 \sqsubseteq t_1 \]
\[ b) \quad t_1 \sqsubseteq t_2 \land t_2 \sqsubseteq t_3 \Rightarrow t_1 \sqsubseteq t_3 \]
\[ c) \quad t_1 \sqsubseteq t_2 \land t_2 \sqsubseteq t_1 \Rightarrow t_1 = t_2 \]

- 140 -
Chapter 5 Properties of Component-based Systems

d) \( t_2 \) is well-behaved \( \land t_1 \sqsubseteq t_2 \Rightarrow t_1 \) is well-behaved

The relation may also be captured algebraically as follows:

\[ t_1 \sqsubseteq t_2 \Leftrightarrow \sigma_{M_x}(t_2) = \sigma_{M_x}(t_1) \cap \sigma_{M_x}(t_2) \]

A similar idea appears in CSP's refinement relationship. Formally, the refinement is based on [Hoa85, Chapter 3], the characterisation of a process as the triple \((A, F, D)\) of alphabet, failures and divergences. A process \( P \) is refined by process \( Q \) written \( P \sqsubseteq Q \), if (i) their alphabets are the same, (ii) the failures of \( P \) are a superset of the failures of \( Q \), and (iii) the divergences of \( P \) are a superset of the divergences of \( Q \). Another close approach is found in Plasil and Visnovsky’s [PV02]. The authors defined the protocol conformance as, given two interface protocols \( PI_1 \) and \( PI_2 \), we say that \( PI_1 \) conforms to \( PI_2 \) if \( L(PI_1) \subseteq L(PI_2) \). In other words, \( PI_1 \) conforms \( PI_2 \) only if it can generate a trace language which is a subset of the trace language generated by \( PI_2 \).

Example 5.1.5 Case 1: following Example 5.1.2, if \( t_4 = a; b[c + d] + g[h] \) where \( \text{SEV}(g) \in E \),

\[
\sigma_{M_x}(t_4) = \sigma_{M_x}(a) \bullet (\sigma_{M_x}(b)[\sigma_{M_x}(c)] \lor \sigma_{M_x}(b)[\sigma_{M_x}(d)]) \lor \sigma_{M_x}(g)[\sigma_{M_x}(h)]
\]

\[
\sigma_{M_x}(t_4) = \{f_a\} \bullet (\{f_b\}{\{f_c\}} \lor \{f_b\}{\{f_d\}}) \lor \emptyset
\]

\[
\sigma_{M_x}(t_4) = \{f_a\} \bullet (\{f_b\}{\{f_c\}} \lor \{f_b\}{\{f_d\}})
\]

which is equivalent to \( \sigma_{M_x}(t_2) \), so that \( \sigma_{M_x}(t_4) \sqsubseteq \sigma_{M_x}(t_2) \Rightarrow t_2 \sqsubseteq t_4 \), and also \( t_4 \sqsubseteq t_2 \), eventually \( t_4 = t_2 \) by (c) of Remark 5.1.2.

Case 2: suppose \( \text{SEV}(g) \in F \) for \( t_4 \), and \( t_5 = a; b[c + d] + g[h^2] \), then

\[
\sigma_{M_x}(t'_4) = \{f_a\} \bullet (\{f_b\}{\{f_c\}} \lor \{f_b\}{\{f_d\}}) \lor \{f_g\}{\{f_h\}}
\]

\[
\sigma_{M_x}(t_5) = \{f_a\} \bullet (\{f_b\}{\{f_c\}} \lor \{f_b\}{\{f_d\}}) \lor (\{f_g\}{\{f_h\}} \lor \{f_g\}{\{f_h\}})
\]

in which \( \sigma_{M_x}(t'_4) \sqsubseteq \sigma_{M_x}(t_5) \), then \( t_5 \sqsubseteq t'_4 \).

In this simpler and more liberal definition of a refinement relation (\( \sqsubseteq \)), we would like to highlight that, (i) only meaningful parts of compared expressions will affect the final result, such as in case (1) where the refinement relation holds between \( t_2 \) and \( t_4 \), regardless of the illegal phrases \( e[f] \) and \( g[h] \); (ii) the expression \( h \) always refines \( h^7 \), \( h^* \) and \( h^+ \) according to their interpretations, for example in case (2) where \( \sigma_{M_x}(h^7) = \{f_h\} \lor \{f_h\} \), and \( \sigma_{M_x}(h) = \{f_h\} \) is
obviously a refinement of \( h^7 \), where \( h \) does not exercise the internal choice of engaging \( h^0 \), but is otherwise consistent with the behaviour of \( h^7 \). However, \( t'_4 \) is not a refinement of \( t'_2 \), and vice versa, because \( t'_1 \) refuses \( g[h] \) when \( t'_4 \) does not, and \( t'_4 \) refuses \( e[f] \) when \( t'_2 \) does not; (iii) the empty string (\( \lambda \)) is not a refinement of any other expressions, due to \( \sigma_{M,\sigma}(\lambda) = \{ \bot \} \), not \( \emptyset \).

While intuitively motivated, three properties on the component protocol above might at first glance appear obscure, and the sceptical reader may well ask, “What good is it anyway?” Like type correctness for programming languages, introducing these properties is intended to provide certain guarantees that the CBS is well-formed. We will demonstrate the utility of these properties in various parts of CBS behaviour checking in the following sections.

5.2 Deadlock-freedom (of a glue)

An important goal of architectural design is to guarantee that components can safely communicate under a particular form of interaction. In this study, this goal is separated into three questions “Can roles safely communicate with each other under a glue?”; “Can a role faithfully localise a glue?” and, “Can a port participate in a role to communicate with others?” In this section, we are focusing on the first question.

Scale down to our model; in order to make certain guarantee that every role communicates safely with each other, we have to ensure the glue works properly. A key aspect of determining this property is detecting whether the glue will get stuck during execution. By ‘get stuck’ we normally mean that the glue is waiting for the rest of the system to emit some action that will never be forthcoming. The causes of this situation will be: (i) the operands of the binding operator are not matching; (ii) the services are not being bound; or (iii) any kinds of illegal descriptions.

In order to satisfy this property, we need to prevent all these three cases appearing in the glue in question. To exclude the first and third case, we can simply check the glue with well-definedness. For the second case, we believe that not all unbound services will lead to deadlock.

Example 5.2.1 Following Example 4.5.4, suppose we simplify the glue’s behaviour to \( ts \rightarrow_{\tau} rp[tp] \), so the process will get stuck at service \( tp \) through execution, since the service \( tp \) is expecting to be invoked but nothing is forthcoming. However, if we change the glue’s behaviour to \( ts \rightarrow_{\tau} rp[tp'] \), in which \( tp' \) means service \( tp \) may or not happen, depending on whether any matched active service (such as service \( sr \)) invokes it or not. From a design point of view, we can consider the glue does not enforce the printing of the transaction data to the vendor while printing receipt to the customer anymore if not requested. The service \( tp \) becomes optional, it is up to the
vendor to decide whether requesting this service or not. Therefore, the glue $ts \mapsto rp[tp^?]$ works properly with/without the invocation of service $tp$.

To handle this situation, firstly the glue can be understood as a set of bindings being ordered in certain way, in the sense that the glue in effect is a legal sequence of service pairs that specifies the service exchanges between components and their mates. For unbound services, we can consider they are binding with the empty string $\lambda$, so that semantically, they are binding to a singleton with an empty function $\bot$. Note that services with different polarities will be bound in different ways. To modify glue in this way, we introduce a function $\text{FIL}$.

**Definition 5.2.1** Given an expression $t$, we define a function $\text{FIL}(t) \in \mathcal{P}$, it goes through every constant $p$ of $t$, if $p$ is unbound, then a function $\text{BID}(p)$ will be applied in place of $p$, such that

$$
\text{BID}(p) =
\begin{cases}
\lambda \mapsto p, \text{ if } \text{SEV}(p) \in \downarrow S \\
\lambda, \text{ otherwise}
\end{cases}
$$

Equivalently, the function $\text{BID}(p)$ is defined as

$$
\text{BID}(p) = \begin{cases} 
\lambda \mapsto p, \text{ if } \text{SEV}(p) \in \downarrow S \\
\lambda, \text{ otherwise}
\end{cases}
$$

State that the empty string $\lambda$ is always thought to be unbound. Since $\text{BID}(\lambda) = \lambda$, it is always true that $\text{FIL}(\lambda) = \lambda$.

**Example 5.2.2** Suppose the behaviour of a glue $t_{\text{Glue}} = a \mapsto b \mapsto c \mapsto d \mapsto e \mapsto f \mapsto g \mapsto h$ where $\text{SEV}(e) \in \downarrow S$ and $\text{SEV}(h) \in \uparrow S$, so $\text{FIL}(t_{\text{Glue}}) = a \mapsto b \mapsto c \mapsto d \mapsto e \mapsto f \mapsto g \mapsto h \mapsto \lambda$.

By employing the function $\text{FIL}$, we can simplify deadlock detecting to a series of binding checkings, to check whether each binding is well-defined via the binding operator defined in Definition 4.4.16. Finally, if the ‘filled’ glue is well-behaved, i.e., there exists (at least) one run which describes the occurrences of services in the semantic domain, then we can say that the glue is deadlock-free.

**Definition 5.2.2** Let $t_0 \in \mathcal{P}$, and a glue $G = (\mathcal{S}_G, \sigma_{M,\sigma}(t_0))$, we shall say that the glue $G$ is deadlock-free, if

1. $t_0 \neq \lambda$, and
2. $t_0$ is well-defined, and
3. $\text{FIL}(t_0)$ is well-behaved

First of all, the empty string expression $\lambda$ is never deadlock-free, in the sense that, the process of $\lambda$, never performs any services, is indeed the simplest deadlock-prone process. Secondly, $t_0$ must be completely well-defined; there are no illegal descriptions. Thirdly, $\text{FIL}(t_0)$ must be well-
behaved (i.e. $\sigma_{M,S}(\text{FIL}(t_G)) \neq \emptyset$); it is used to evaluate the well-definedness of each service binding (including unbound services with a subsequent added $\lambda$). Below, let us illustrate this definition by some examples.

**Example 5.2.3 Case 1**: following Example 4.5.4, assume the behaviour of the glue is denoted as $t_{\text{glue}} = ts \mapsto_{rp} rp[tp]$, apparently it is not an empty string expression and also is well-defined, so Axiom (a) and Axiom (b) hold. Then, $\text{FIL}(t_{\text{glue}}) = ts \mapsto_{rp} rp[\lambda \mapsto_{rp} tp]$ where

$$\begin{align*}
\sigma_{M,S}(ts \mapsto_{rp} rp[\lambda \mapsto_{rp} tp]) &= \sigma_{M,S}(ts) \mapsto_{rp} \sigma_{M,S}(rp)[\sigma_{M,S}(\lambda) \mapsto_{rp} \sigma_{M,S}(tp)] \\
&= \{f_{\lambda} \} \mapsto_{rp} \{f_{\lambda}\} \{\bot\} \mapsto_{rp} \{f_{\lambda}\} \\
&= \{f_{\lambda}\} \mapsto_{rp} \emptyset \\
&= \emptyset
\end{align*}$$

Therefore $\text{FIL}(t_{\text{glue}})$ is not well-behaved, Axiom (c) does not hold. As a result, the glue is not deadlock-free.

**Case 2**: suppose now the behaviour of the glue $t_{\text{glue}} = ts \mapsto_{rp} rp[tp^2]$, of course Axiom (a) and Axiom (b) still hold. Regarding Axiom (c), $\text{FIL}(t_{\text{glue}}) = ts \mapsto_{rp} rp[\lambda \mapsto_{rp} tp^2]$, where

$$\begin{align*}
\sigma_{M,S}(ts \mapsto_{rp} rp[\lambda \mapsto_{rp} tp^2]) &= \sigma_{M,S}(ts) \mapsto_{rp} \sigma_{M,S}(rp)[\sigma_{M,S}(\lambda) \mapsto_{rp} \sigma_{M,S}(tp^2)] \\
&= \{f_{\lambda}\} \mapsto_{rp} \{f_{\lambda}\} \{\bot\} \mapsto_{rp} \{\bot\} \lor \{\bot\} \mapsto_{rp} \{f_{\lambda}\} \\
&= \{f_{\lambda}\} \mapsto_{rp} \{f_{\lambda}\} \{\bot\} \lor \emptyset \\
&= \{f_{\lambda}\} \mapsto_{rp} \emptyset
\end{align*}$$

the result will not be $\emptyset$. Therefore, $\text{FIL}(t_{\text{glue}})$ is well-behaved, Axiom (c) also holds. Hence, the glue is deadlock-free. It is noteworthy that following Definition 4.4.16, $\{\bot\} \mapsto_{rp} \{\bot\} = \{\bot\} \lor \{\bot\}$ by the case (a): $f \mapsto_{g} g = f \lor g$, if $f = \bot \land \text{IDL}(g, q)$. However, $\{\bot\} \mapsto_{rp} \{f_{\lambda}\} = \emptyset$ since $\text{IDL}(f_{\lambda}, tp)$ is not true, it results in $\bot \mapsto_{rp} f_{\lambda}$ is undefined.

\[1\] Here, $p$ could be any service name.
Case 3: the case above also holds when the glue’s behaviour is described by \( t_{\text{Glue}} = ts \rightarrow rp[tp^*] \), as shown in follows. \( \text{FIL}(t_{\text{Glue}}) = ts \rightarrow rp[\lambda \rightarrow yp \cdot tp^*] \), where

\[
\sigma_{M_{p}}(ts \rightarrow rp[\lambda \rightarrow yp \cdot tp^*]) = \sigma_{M_{p}}(ts) \rightarrow rp[\sigma_{M_{p}}(\lambda) \rightarrow yp \cdot \sigma_{M_{p}}(tp^*)]
\]

\[
= \{f_{ts}\} \rightarrow rp(\{\perp\} \rightarrow yp \cdot \perp_{yp}) \vee \{\perp\} \rightarrow yp \cdot \{f_{tp}\} \vee \ldots \vee \{\perp\} \rightarrow yp \cdot \{f_{tp}\}
\]

\[
= \{f_{ts}\} \rightarrow rp(\{\perp_{yp}\} \vee \varnothing \vee \ldots \vee \varnothing)
\]

\[
= \{f_{ts}\} \rightarrow rp(\{\perp_{yp}\})
\]

Def. 4.4.16, Remark 4.4.5 (d)

In all examples above, \( tp \) denotes a provided function. Likewise, these examples also hold when \( tp \) denotes an in event.

To sum up, for any well-defined glue, if there exists an unbound passive service, then the glue is not deadlock-free. However, if the unbound passive service is optional or postfixed \( * \), then the glue may be deadlock-free. At this point, another factor needs to be concerned: whether the glue has any unbound active services.

Example 5.2.4 Case 1: consider the glue’s behaviour \( t_{\text{Glue}} \) defined in Example 4.5.4, now we modify it to \( ci; sa \rightarrow sm \cdot sm \), where the service \( ci \) is an unbound out event. Now let us assess the anti-deadlock property of the glue. Firstly, \( t_{\text{Glue}} \) is not an empty string \( \lambda \) and is well-defined. Secondly, \( \text{FIL}(t_{\text{Glue}}) = ci \rightarrow cl \cdot \lambda; sa \rightarrow sm \cdot sm \), where

\[
\sigma_{M_{p}}(ci \rightarrow cl \cdot \lambda; sa \rightarrow sm \cdot sm) = \sigma_{M_{p}}(ci) \rightarrow cl \cdot \sigma_{M_{p}}(\lambda) \cdot \sigma_{M_{p}}(sa) \rightarrow sm \cdot \sigma_{M_{p}}(sm)
\]

\[
= \{f_{ci}\} \rightarrow cl(\perp) \cdot \{f_{sa}\} \rightarrow sm \cdot \{f_{sm}\}
\]

\[
= \{f_{ci}\} \cdot \{f_{sa}\} \rightarrow sm \cdot \{f_{sm}\}
\]

Def. 4.4.16, Rem. 4.4.5 (d)

because the result is not \( \varnothing \), it is immediate that \( \text{FIL}(t_{\text{Glue}}) \) is well-behaved. Hence, the glue is free from deadlock. Notice that \( \{f_{ci}\} \rightarrow cl(\perp) = \{f_{ci}\} \cdot \{\perp\} = \{f_{ci}\} \) by the case (b) of Definition 4.4.16: \( \text{OCE}(f, p, s_{p}) \in \mathbb{N}^+ \land \text{IDL}(g, q) = f \rightarrow p \rightarrow g \neq f \mid g \).

Case 2: however, if the service \( ci \) denotes a function, then

\[
\sigma_{M_{p}}(ci \rightarrow cl \cdot \lambda; sa \rightarrow sm \cdot sm) = \sigma_{M_{p}}(ci) \rightarrow cl \cdot \sigma_{M_{p}}(\lambda) \cdot \sigma_{M_{p}}(sa) \rightarrow sm \cdot \sigma_{M_{p}}(sm)
\]

\[
= \{f_{ci}\} \rightarrow cl(\perp) \cdot \{f_{sa}\} \rightarrow sm \cdot \{f_{sm}\}
\]

\[
= \varnothing \cdot \{f_{sa}\} \rightarrow sm \cdot \{f_{sm}\}
\]

Def. 4.4.16, Rem. 4.4.5 (d)
where \( \{ f_{ci} \} s \rightarrow s_m \{ f_{sm} \} \) is not defined. As a result of this, \( f_{ci} p \rightarrow c_i  \) is undefined. Hence, if \( ci \) denotes a function, then the glue is not deadlock-free. We can explain this phenomenon in terms of our principle of interaction: if \( ci \) is an out event, once it emits an action, even if this action will be absorbed by other services, the process will continue its own processing. Note that this is a common mechanism for asynchronous communication. However, if \( ci \) is a required function, once it has emitted an action, it will immediately enter a state of waiting for reply. If nothing is forthcoming, the process will get stuck at that point because of the synchronous nature of this process.

The situations are the same if the unbound active services are optional or postfixed with *.

**Example 5.2.5** Case 1: following Example 4.5.4, we modify the glue’s behaviour to \( \tau_{Glue} = ci^7; sa \rightarrow s_m \). Still, \( \tau_{Glue} \) is not an empty string and is well-defined. Regarding the Axiom (c) of Definition 5.2.2, \( FILG /! (e) = c_i p \rightarrow c_i \text{ sa } \rightarrow s_m \text{ sm } \), then

\[
\sigma_{M_p}(ci^7 \rightarrow c_i \lambda; sa \rightarrow s_m sm) = \sigma_{M_p}(ci^7) p \rightarrow c_i \sigma_{M_p}(\lambda) \ast \sigma_{M_p}(sa) sa \rightarrow s_m \sigma_{M_p}(sm)
\]

\[
= ((\bot c_i) p \rightarrow c_i \bot) \ast \{ f_{ci} \} s \rightarrow s_m \{ f_{sm} \}
\]

\[
= (\emptyset \ast \{ f_{ci} \} s \rightarrow s_m \{ f_{sm} \}) \ast \{ f_{sa} \} sa \rightarrow s_m \{ f_{sm} \}
\]

The nonempty result means that \( FILG/! \) is well-behaved, so the glue is deadlock-free.

**Case 2:** consider the service \( ci \) denotes a function, then

\[
\sigma_{M_p}(ci^7 \rightarrow c_i \lambda; sa \rightarrow s_m sm) = \sigma_{M_p}(ci^7) p \rightarrow c_i \sigma_{M_p}(\lambda) \ast \sigma_{M_p}(sa) sa \rightarrow s_m \sigma_{M_p}(sm)
\]

\[
= ((\bot c_i) p \rightarrow c_i \bot) \ast \{ f_{ci} \} s \rightarrow s_m \{ f_{sm} \}
\]

\[
= (\emptyset \ast \emptyset) \ast \{ f_{sa} \} sa \rightarrow s_m \{ f_{sm} \}
\]

\[
= \emptyset \ast \{ f_{sa} \} sa \rightarrow s_m \{ f_{sm} \}
\]

\[
= \emptyset \ast \{ f_{sa} \} sa \rightarrow s_m \{ f_{sm} \}
\]

Remark 4.4.16

Remark 4.4.19 (e)

Remark 4.3.3 (b)
Therefore, \( \text{FIL}(t_{\text{Glue}}) \) is not well-behaved, and the glue is not deadlock-free.

As shown in above examples, we can generalise that for any well-defined glues, if the unbound active service is an event, then the glue is deadlock-free, subject to unbound passive services.

To sum up, this section discussed the question in terms of, "Can roles safely communicate with each other under a glue?" From a formal point of view this question can be cast as one of detecting glue deadlock, such as the works being done in [AG98, AP04b, AP04c, BCD01]. In short, we say that a glue is free from deadlock if it does not get 'stuck' midway through its execution; if it stops, that means the overall process is finished and every constant has been passed through. To evaluate this property, firstly we ensure that the given glue is not an empty string \( \lambda \) and is well-defined. Secondly, since we believe that optional or asterisk passive services and active events will not lead the glue to deadlock, we employ a FIL function to make the glue in question to be a full set of service bindings (let unbound services bind with the empty string \( \lambda \), if any), and check the resulting glue with well-behavedness. All deadlock bindings will be semantically spotted by the definition of binding operator.

### 5.3 Loyalty (of a role with a glue)

In the context of our connector, the glue conveys the 'global' information about how components are cooperating together; when decomposing the glue into a set of roles, each role delivers the localised information about what certain tasks the component should be able to fulfil during the interaction. When viewed as individual processes, each role should, on the other hand, faithfully represent a certain part of glue's process. In other words, the behaviour of each role should be loyal to that of the glue.

To formally define this property, we start off by defining assistant functions DEL and SRK.

**Definition 5.3.1** Let \( f, f' \in \text{Runs}_\varphi, m \in \mathbb{N}^+ \), we define a function \( \text{DEL}(f, m) = f' \), such that

\[
\begin{cases}
  f'(p,n) = \begin{cases}
    f(p,n), & \text{if } n < m \\
    f(p,n + 1), & \text{otherwise}
  \end{cases}
  \\
  \text{Informally, the DEL function removes all event occurrences at the given time slot in the run. By convention, } \text{DEL}(\bot, n) = \bot \text{ for all } n \in \mathbb{N}^+, \text{ and so as } \bot_\varphi .
\end{cases}
\]

**Remark 5.3.1** Let \( f \in \text{Runs}_\varphi, m \in \mathbb{N}^+ \), if \( m > \ell(f) \Rightarrow \text{DEL}(f, m) = f .

**Example 5.3.1** Consider Example 4.4.1, \( \text{DEL}((f_s \cdot g_e)|_{(x,1)}, 3) = l_x \), it is tabularly illustrated as
Chapter 5 Properties of Component-based Systems

\[ l(s, 1) = \gamma \alpha \]
\[ l(s, 2) = \gamma \beta \]
\[ l(u, 3) = \gamma \gamma \]
\[ l(u, 4) = \gamma \beta \]

As shown in the example, the third column (time slot) in the preceding table is removed.

**Definition 5.3.2** Let \( f \in \text{Runs}_\mathcal{E} \), we define a function \( \text{SRK}(f) = f \), such that for all \( n < \ell(f) \in \mathbb{N}^+ \), \( \text{IDL}(f, n) \Rightarrow f = \text{DEL}(f, n) \).

As a matter of fact, the resulting run is a shrunken version of run \( f \), in which all idle time slots before the last nonempty time slot are removed (if any). Observe that \( \text{SRK}(f) = f \) if no idle time slot can be found before the length of run \( f \), and it is also the case of \( f = \lambda, \lambda_G \) by convention.

**Definition 5.3.3** Let \( \mathcal{L} \in \mathbb{M}_\mathcal{E} \), then we define a function \( \text{SRK}(\mathcal{L}) = \{\text{SRK}(f) : f \in \mathcal{L}\} \).

To achieve the loyalty property, the role should behave in the same way as the corresponding localised glue docs. Technically, the localised glue behaviour is achieved by restricting the whole glue behaviour to the one only containing role’s services, thereafter restoring the restricted behaviour to the original pattern (as being initially constructed), by means of the SRK function. In addition, we require that both glue and role are well-defined.

**Definition 5.3.4** Let \( t_G, t_R \in \mathcal{P} \), a glue \( G = (\mathcal{S}_G, \sigma_{M_G}(t_G)) \) and a role \( R = (\mathcal{S}_R, \sigma_{M_R}(t_R)) \), we shall say that the role \( R \) is loyal to the glue \( G \), if

- \( t_G \) and \( t_R \) are well-defined, and
- \( \sigma_{M_R}(t_R) = \text{SRK}(\sigma_{M_G}(t_G)|_{\text{NAM}(G)}) \)

State that the empty string role is always in loyalty with any glues (including the empty string one), because technically \( \text{SRK}(\sigma_{M_G}(t_G)|_{\text{NAM}(G)}) = \text{SRK}(\sigma_{M_G}(t_G)|_{\emptyset}) = \text{SRK}(\{\emptyset\}) = \{\emptyset\} = \sigma_{M_G}(\lambda) \) according to the \((d)\) of Remark 4.4.1, and also \( \lambda \) is well-defined. The intuition behind this is that we leave the empty string role as a special role in connector development. For example, we can add (or leave) a role with solely an empty string beforehand and assign its responsibility afterwards; when we merge and separate roles, we can move one’s functionality to the other and leave it blank with an empty string, and check the loyalty property of the other roles with the glue, as well as the connector well-formedness property discussed later. It facilitates rolling back changes as soon as the problem has been estimated.
Example 5.3.2 Case 1: given the glue and roles defined in Example 4.5.4, we shall say that the role $\text{RSel}$ is loyal to the glue $\text{Glue}$. Since first of all, $t_{\text{RSel}} = ts || sm; sr$ and $t_{\text{Glue}} = ci \rightarrow rc (sa \rightarrow sm; sm || ts \rightarrow rp [sr \rightarrow rp, tp]) || rp \rightarrow rc$ are well-defined; and secondly, $\sigma_{M_{G}}(t_{\text{RSel}}) = \text{SRK}(\sigma_{M_{G}}(t_{\text{Glue}})|_{ts, sm, sr})$. However, if we change $t_{\text{RSel}}$ to $ts || sm[r, sr]$, the loyalty property between the $\text{RSel}$ and the $\text{Glue}$ does not hold, due to $\sigma_{M_{G}}(t_{\text{RSel}}) \neq \text{SRK}(\sigma_{M_{G}}(t_{\text{Glue}})|_{ts, sm, sr})$.

Case 2: also, we shall say that the role $\text{RPnt}$ is in loyalty with the glue $\text{Glue}$. We can observe that $t_{\text{RPnt}} = rp [tp]$ is well-defined because the service $rp$ denotes a function; and also $\sigma_{M_{G}}(t_{\text{RPnt}}) = \text{SRK}(\sigma_{M_{G}}(t_{\text{Glue}})|_{tp, rp})$.

Case 3: similarly, we claim that the role $\text{RBuy}$ is loyal to the glue $\text{Glue}$. Firstly, $t_{\text{RBuy}} = ci; rc$ is always well-defined. Secondly, $\sigma_{M_{G}}(t_{\text{RBuy}}) = \text{SRK}(\sigma_{M_{G}}(t_{\text{Glue}})|_{ci, rc})$.

Case 4: finally, we can conclude that the role $\text{RMon}$ is loyal to the glue. The role only has a service $sa$, so its behaviour expression $t_{\text{RMon}} = sa$ is well-defined. In addition, $\sigma_{M_{G}}(t_{\text{RMon}}) = \text{SRK}(\sigma_{M_{G}}(t_{\text{Glue}})|_{sa})$.

In closing, this section has concentrated on the second question "Can a role faithfully localise a glue?" We answered this question by introducing a loyalty property between roles and glues. It requires that the role represents a part of glue's behaviour and does not introduce any new behaviour to the glue. In short, this approach is to localise the glue behaviour and check it with the corresponding role behaviour; the loyalty property holds when two behaviours are both well-defined and semantically equivalent to each other. Finally, the loyalty property plays an important role in connector development.

5.4 Compatibility (of a role with a port)

From our architectural perspective, components and connectors are independent of each other. Components are computational entities that offer a number of services but they should not be aware of the relationships in which they may engage. On the other hand, connectors are communication entities that control component interactions. More precisely, connectors can
observe and manipulate cooperation between components and can even enforce state changes in components.

Following this thought we investigate the relationship between a role and a port. Each role represents the localised service interaction logics in a connector. Each port specifies (a part of) the behavioural information in a component. From the methodological concerns, in order to increase the reusability of individual components, the behaviour of a port should be more liberal, because a component should fill as broad set of connectors as possible. Conversely, the behaviour of a role should be more restricted and deterministic, since it represents the actual activities of a component in composition.

Furthermore, if a given role is acting in a way that a given port is able to act, we shall say that the given role is compatible with the given port. One benefit of this property is that we can safely constrain a port behaviour down to a role behaviour. Thus, certain guarantee can be made for components to safely communicate with each other through the connector, as long as the underlying glue is free from deadlock and the attached roles are in loyalty with that glue.

The notion in terms of compatibility can be formally captured by means of the refinement relationship between component protocols (cf. Definition 5.1.4).

**Definition 5.4.1** Let $t_R, t_P \in \mathcal{P}$, a role $R = (\mathcal{S}_R, \sigma_{M_R}(t_R))$ and a port $P = (\mathcal{S}_P, \sigma_{M_P}(t_P))$, we shall say that the role $R$ is compatible with the port $P$, if

1. $\text{SEV}(R) \neq \emptyset \wedge \text{SEV}(R) \subseteq \text{SEV}(P)$, and
2. $t_R; t^0_P \subseteq t^0_R; t^0_P$

Generally, the compatibility between roles and ports requires that (i) statically, the role’s service set is nonempty and is a subset of the port’s one (so that the port’s service set is also nonempty); and (ii) dynamically, the role represents fewer possible behaviours (less nondeterministic) than the port. While the port permits any internal choices, the role may further constrain those.

There is one reason why it is not possible to use $\mathcal{P}$’s definition of refinement directly to define role-port compatibility. That is, the port may normally have additional service names while internal choices existed. That makes the alphabets of two compared expressions different from each other. Unfortunately, the $\subseteq$ relation only assumes that the alphabets of the compared expressions are the same. However, we can overcome this technical limit by concatenating one party’s expression to a counterpart’s 0-postfixed expression, such as $t_P; t^0_R$ and $t^0_R; t^0_P$. In semantic domain, $\sigma_{M_P}(t_P; t^0_R) = \sigma_{M_P}(t_P) \circ_0 \{ \text{\_NAM}(\sigma_{M_R}(t_R)) \}$, recall that this operation appends new blank rows to the language of $t_P$ (if any). This operation helps us to unify the alphabets of compared expressions before the refinement checking.
Furthermore, the static constraint \( \text{SEV}(R) \neq \emptyset \) on Axiom (a) excludes the possibility of using empty string to describe the behaviour of roles and ports. Any roles and ports with an empty string behaviour description denote that they have no observable services to their environments, and also they have no observable behaviours. Consequently, a port without observable services (and their behaviours) will not be able to communicate with any roles, and vice versa. Hence, we claim that the empty string party is never compatible with its counterparts.

Also, the static constraint \( \text{SEV}(R) \subseteq \text{SEV}(P) \) on Axiom (a) helps us to handle the case of 0-postfixed expressions on compared parties. For example, given a port and a role whose behaviour are \((a;b)^0\) and \((a[d])^0\), respectively. In dynamic, \((a;b)^0; (a[d])^0 \subseteq (a[d])^0; (a;b)^0\) and vice versa, because in effect \(\downarrow_{(a,d)} \bullet \downarrow_{(a,b)} = \downarrow_{(a,b)} \bullet \downarrow_{(a,d)}\). However, the two entities are in fact functionally incompatible, although neither of them performs anything. This kind of incompatibility cannot be perceived in refinement checking, but can be easily prevented in the static world by setting the condition \(\text{SEV}(R) \subseteq \text{SEV}(P)\). Note that inconsistency between service names is acceptable here. It is possible for roles and ports to have the same service with different names.

Again, let us illustrate the definition of compatibility through following examples.

Example 5.4.1 Case 1: consider the role \(\text{RPnt}\) whose behaviour \(t_{\text{RPnt}} = rp[tp]\) is defined in Example 4.5.4, and the port \(\text{PPnt}\) with its behaviour \(t_{\text{PPnt}} = rp || tp\) defined in Example 4.6.1, we say that the role \(\text{RPnt}\) is compatible with the port \(\text{PPnt}\), as \(\text{SEV}(\text{RPnt}) \subseteq \text{SEV}(\text{PPnt})\) and \(rp || tp \subseteq rp[tp]\). From a design viewpoint, this case can be understood as, in order to maximise the reusability of a printer component, its receipt-printing service \((rp)\) and the transaction-printing \((tp)\) are designed to be able to run in any order. On the other hand, the role \(\text{RPnt}\) describes the responsibility of a printer in a particular context; that of a ticket vending machine, in this a constraint has been placed: whilst the printer is issuing a receipt to the customer, the printer also has to issue one to the vendor. Hence, when attaching the \(\text{PPnt}\) to \(\text{RPnt}\), we can believe that it is safe to cast the behaviour of \(\text{PPnt}\) to that of \(\text{RPnt}\), as \(rp[tp]\) is in fact the one possible execution of \(rp || tp\).

Case 2: if we introduce another provided function to the printer component, called receipt-preview \((rv)\), and change the behaviour of the port \(\text{PPnt}\) to \(rv; rp || tp\), which means the printer always displays nominal receipt preview before printing out the actual one. In this case, the compatibility between \(\text{RPnt}\) and \(\text{PPnt}\) does not hold, since obviously we cannot safely cast \(\text{PPnt}\)'s
behaviour $rv;rp || tp$ to role $RPnt$’s $rp(tp)$. However, if the port $PPnt$’s behaviour is defined as $rv';rp || tp$, then the role $RPnt$ will be compatible with it, because

$$
\sigma_M \{ rp\} \cdot \{ \bot_{rv;rp,(tp,p)} \} \subseteq \{ \bot_{rv} \} \cdot \{ \bot_{rp} \} \subseteq \sigma_M \{ \{ f_{rp} \} \} \cdot \{ f_{tp} \} \cdot \{ \bot_{rp,tp} \}
$$

where $rv'$ is translated into $\{ \bot_{rv} \} \cup \{ f_{rv} \}$. From a design point of view, the port $PPnt$ in this case allows the receipt-preview service ($rv$) to run optionally. That is, if the role does not request $rv$, then the port can get around this service and continue to proceed.

**Case 3**: assuming we have bought a printer–fax machine component, its port $PPntFax$’s behaviour is $t_{PPntFax} = rp || tp + dp || df$. In this, $dp$ denotes a provided function, document-printing, which prints out incoming documents; and $df$ denotes a required function, document-faxing, which faxes out documents. We further introduce a role called $RFax$ whose behaviour $t_{RFax} = dp || df$. Then, we attach the port $PPntFax$ to both role $RPnt$ and $RFax$ together, as shown in Figure 5.1 above. Assuming that all three parts are well-defined, now let us check their compatibility. Firstly, $SEV(RFax) \subseteq SEV(PPntFax)$ and $SEV(RPnt) \subseteq SEV(PPntFax)$. Secondly, $t_{PPntFax} \equiv t_{RFax}$, since

$$
\sigma_M \{ dp \} \cdot \{ \bot_{rp;dp,df} \} \subseteq \sigma_M \{ rp \} \cdot \{ tp \} \cdot \{ \bot_{dp,df} \}
$$

Similarly, $t_{PPntFax} \equiv t_{RPnt}$, since

$$
\sigma_M \{ rp \} \cdot \{ \bot_{rp;dp,df} \} \subseteq \sigma_M \{ rp \} \cdot \{ tp \} \cdot \{ \bot_{rp,dp} \}
$$

As we can see, both roles are compatible with the port. The $RPnt$ and $RFax$ do not exercise the choice of engaging in $dp || df$ and $rp || tp$, respectively, but are otherwise consistent with $PPntFax$’s behaviour. We can understand that, the printer–fax machine is able to act either as a printer or as a fax machine. However, if we change the port $PPntFax$’s behaviour to $rp || tp; dp || df$, informally, the machine has to act as a printer first, and then as a fax machine. In this case, neither $RPnt$ nor $RFax$ is compatible with $PPntFax$. Formally,

$$
\sigma_M \{ rp \} \cdot \{ \bot_{rp;dp,df} \} \nsubseteq \sigma_M \{ rp \} \cdot \{ tp \} \cdot \{ \bot_{rp,dp} \}
$$

$\Rightarrow$ $t_{PPntFax} \nsubseteq t_{RPnt}$, and
From the design viewpoint, \( RPnt \) is not compatible with \( PPntFax \) because we cannot safely restrict \( PPntFax \)'s behaviour to \( RPnt \)'s one, in the sense that \( RPnt \) does not offer the functionality of fax. In a similar vein to \( RFax \), it does not support the functionality of print.

Case 4: on the contrary, now imagine that we have a role \( RPntFax \) whose behaviour is defined as \( t_{RPntFax} = rp [ tp ] + dp \parallel df \), and two ports: \( PFax \) with its behaviour \( t_{PFax} = dp \parallel df \) and \( PPnt \) with its behaviour \( t_{PPnt} = rp \parallel tp \), then the role \( RPntFax \) is not compatible with either \( PFax \) or \( PPnt \), because \( SEV(RPntFax) \nsubseteq SEV(PPnt) \) as well as \( SEV(RPntFax) \nsubseteq SEV(PFax) \). Both ports failed in the static checking.

With a similar reason as provided in the previous case, it is unsafe to restrict \( PFax \) (or \( PPnt \))'s behaviour to \( RPntFax \)'s one, because both \( PFax \) and \( PPnt \) are incomplete descriptions of \( RPntFax \).

In general, due to the assessment of compatibility which unifies the alphabets of candidate roles and ports, this property holds if only one port is used in a role. As shown in case (3) above, roles \( RPnt \) and \( RFax \) may be compatible with the port \( PPntFax \), because for both roles, \( PPntFax \) is the only port working with them. In the following case, the role \( RPntFax \) will be never compatible with the ports \( PFax \) and \( PPnt \), because there are two ports used in this role. However, we can simply get around this limit through component hierarchical design, i.e., merging port \( PFax \) and \( PPnt \) into a super port, thereafter attaching it to the role \( RPntFax \).

Probably the closest to our work is the port/role compatibility checking in the Wright language developed by Allen and Garlan [AG98]. The language uses a variant of Hoare’s CSP [Hoa85] to define ports and roles. The compatibility of a port \( P \) and a role \( R \) is captured by means of CSP’s refinement relationship. Firstly, to unify the alphabets of \( P \) and \( R \), the \( P \) is replaced by \( P_{\mathit{STOP}} = P \parallel \mathit{STOP}_{R} \) (where \( \mathit{STOP}_{R} \) is the \( \mathit{STOP} \) process over alphabet \( R \)), and so is \( R_{\mathit{STOP}} = R \parallel \mathit{STOP}_{P} \). Secondly, with the concern that compatibility evaluation only cares about the behaviour of the port over traces described by the role, the authors restrict the process \( P \) to a process \( R \) by placing \( P \) in parallel with \( R' \) (a deterministic version of \( R \), obtained by replacing all of the internal choices from \( R \) by external choices.) Formally, the compatibility is defined as,
\[ P \text{ compat } R ('P \text{ is compatible with } R') \text{ if } R_{*p} \subseteq P_{*R} \parallel R'_{*p} \]

It appears that this compatibility definition is a reversed version of ours. It results from the different methodology of using connectors. Generally speaking, both approaches are based on the recognition that relations between components are as important as the components themselves and the first-class connector is a construct introduced for explicitly specifying collaborations between components. A connector is normally defined by a group of roles and a glue.

In Allen and Garlan’s approach [AG97, AG98], the authors adopt semi-independent connectors to standardise workflow for component cooperation. Connectors are regarded as prefabricated templates for a collection of components to collaborate on certain task. In following this principle, the glue defines a common interaction protocol for a set of parties and each role specify the requirement of a party catering for a broad range of components. The port is a de facto participant in the connector that will take the place of the role and interact with the glue directly in the runtime. Thus, the compatibility is to measure if the port can be considered a refinement when being restricted to the behaviour of the role.

In our approach, we use standalone connectors to explicitly capture communication mechanisms among components. Connectors play a more proactive role than that in [AG98]. Connectors are considered as explicit independent constructs that provide a necessary design abstraction to describe, also in effect initialise and take charge of inter-component communications. In respect to this approach, the glue describes the user-defined behaviour specific to the interactions of expected components and this designed behaviour is explicitly executed by the connector in the running system. Each connector’s role is actually a portion of the glue, characterises a potential component which is able to fulfil the desired role in the interaction. The port outlines the (part of) design-time behaviour of component regarding to the order of services to be served to deliver the component functionality. Once plugging in the connector, components should work exactly as the role guided. Therefore, the compatibility in this context becomes to evaluate if the role can be considered a refinement when being attached by the port.

One benefit of our approach is that, by completely treating connectors as standalone design entities separate from components, roles’ behaviour will not be replaced by (but actually restricts) the ports’ behaviour when attaching ports to roles. Therefore, it makes the collaboration logics ‘encapsulated’, instead of spreading across participating components during communication. Further, the result of glue deadlock checking can be always guaranteed. Also, it makes the component behaviour separate from the connector behaviour during execution. This significantly increases the reusability of both components and connectors, and makes the whole model cleaner and easier.
In summary, this section discussed the third question, “Can a port participate in a role to communicate with others?” Our approach draws upon the concept of a component being able to work with a connector so long as it provides a port compatible to a role required by the connector. This notion can be formally captured by means of the refinement relationship between component protocols, and further takes into account the static specifications. In this model, the connectors are fully independent entities that play a proactive role for component collaborations and they can control component behaviours. This setting minimises coupling between components. We suggest that roles should behave more deterministically than ports (or conversely ports should behave more liberally than roles), which provides the means for the assessment of compatibility between roles and ports.

5.5 Substitutability (of a port with a port)

The aim of software component technology is to provide means for easy creation and modification of software systems. A frequent kind of such a modification is upgrade, which is the replacement of an outdated version of a component by a more current one. An upgrade is therefore a special case of component substitutability.

The basic scenario is the standard one: we have a CBS consisting of a group of components connected with a connector. For whatever reason, we want to substitute one of these components by another one from a different provider. It is natural to require that the substitution to be side-effect free, i.e., that after such a substitution, the whole system must function correctly and its behaviour must be consistent with that before the change.

In general, the principle of substitutability was coined by Wegner and Zdonic [WZ88]: the replacement component should be usable whenever the current one was expected, without the client noticing it.

Many approaches exist that attempt to ensure this, ranging from the area of behavioural subtyping [ZW97, VHT00, HL99, Bra03] to intercepting incorrect functioning at runtime, for example, in fault-tolerant systems [Kop87]. The aim is usually to try all the options at hand before concluding, in the worst case, that the substitution is undesirable.

[ZW97, VHT00, HL99] employ the rationale of so-called ‘strict substitutability’. That is, the replacement component provides at least the same, and requires, at most the same. Based on the standard contravariant subtyping between component types, this kind ensures substitutability ‘a-priori’, without any information about their environments. In practice, this commonly used notion may be overly restrictive when taking into account environments. This observation is supported by other works that attempt to provide a more flexible notion of subtyping, such as [Bra03].

---

2 In this work we are not concerned whether the substitution occurs at design time or at runtime (usually called ‘update’).
work, a 'context-aware' form of substitutability is proposed, with the sense that (i) the replacement component only cares about those of current component's provided features which actually have bindings to the particular required features of other components in the given architecture; (ii) whether the environment provides features which the replacement component declares as required, not necessarily considering the requirements of the current component.

Indeed, many definitions of substitutability can be found using different notations in various frameworks. Some like Szyperski, emphasises the need for a holistic view, arguing that substitutability involves global integrity checking [Szy96]. While this is certainly right, the complexity of such checks may be prohibitive. We therefore prefer the approach that uses local solutions, which are based on the comparison of solely the component ports directly involved in the substitution.

Getting benefits from our CBS architecture, the substitutability in this model is much simpler than others. Since the port behaviour is to some degree controlled by connectors, the replacement of port does not actually affect the collaboration in the connector, and will not be noticed by other parties, as long as the replacement port is able to continue taking all jobs of the current one. The principle of substitutability can be therefore achieved in this architecture in a straightforward manner.

**Definition 5.5.1** Let $t_p, t'_p \in P$, a port $P = (s_p, \sigma_{M_p}(t_p))$ and a port $P' = (s_{p'}, \sigma_{M_{p'}}(t_{p'}))$, we shall say that the port $P'$ is substitutable for the port $P$, if

\[ a) \ SEV(P) \neq \emptyset \land SEV(P) \subseteq SEV(P'), \ and \]
\[ b) \ t_p; t'_p \subseteq t_p; t_p' \]

This definition reflects the natural understanding of substitutability in our model: the changes in the provided and required features of the port do not affect substitutability in a uniform way. The replacement port in most cases should have at least the same features as those of the current one and should not impose any new ordering constraints. Namely, the replacement port should behave more freely or at least the same as the current port does, and may occasionally exercise choices. Otherwise, the attached role will not be able to work properly with the new port.

Intuitively, one would expect that substitutability implies compatibility. This is proven in the following remark.

**Remark 5.5.1** If a port $P'$ is substitutable for a port $P$ and a role $R$ is compatible with the port $P$, then the role $R$ is also compatible with the port $P'$.

**Proof.** Suppose $R = (s_r, \sigma_{M_r}(t_r))$, $P = (s_p, \sigma_{M_p}(t_p))$ and $P' = (s_{p'}, \sigma_{M_{p'}}(t_{p'}))$. We have to show that
Chapter 5 Properties of Component-based Systems

1. SEV(R) ⊆ SEV(P')

2. \( t_p' \subseteq t_R \)

First of all, given \( \text{SEV}(P) \subseteq \text{SEV}(P') \) and \( \text{SEV}(R) \subseteq \text{SEV}(P) \), we can deduce that \( \text{SEV}(R) \subseteq \text{SEV}(P') \) by the transitivity of subset in set theory.

Secondly, given \( t_p' \subseteq t_p \) and \( t_p \subseteq t_R \), then the transitivity of \( \subseteq \) holds by \((b)\) of Remark 5.1.2 as \( t_p \subseteq t_p' \wedge t_p \subseteq t_R \Rightarrow t_p' \subseteq t_R \).

The result of 'substitutability implies compatibility' can be useful in certain common cases. For instance, given the replacement port \( P' \), the current port \( P \) and the role \( R \), suppose \( P \) is compatible with \( R \), when comparing two ports \( P \) and \( P' \) alone, we can always derive that the underlying compatibility holds between \( P' \) and \( R \) as long as \( P' \) is substitutable for \( P \). This remark guarantees that we can safely upgrade the port without spewing compatibility errors to the current system.

However, a failed substitutability does not immediately lead to compatibility failure. For example, provided that port \( P \)'s behaviour is \( t_p = a \parallel b + c[d] \), role \( R \)'s behaviour is \( t_R = a; b \) and \( R \) is compatible with \( P \), i.e. \( \text{SEV}(R) \subseteq \text{SEV}(P) \) and \( t_p \subseteq t_R \), now we want to replace port \( P \) with another port \( P' \) whose behaviour is \( t_p' = a \parallel b + e \parallel f \). \( P' \) is not substitutable for \( P \) since the Axiom \((a)\) fails, i.e., \( \text{SEV}(P) \not\subseteq \text{SEV}(P') \), which results from the different services they are engaged while exercising the internal choice. However, \( P' \) is compatible with \( R \), as we can see that \( \text{SEV}(R) \subseteq \text{SEV}(P') \) and \( t_p' \subseteq t_R \). Thus, it is still safe to transform \( P' \)'s behaviour to \( R \)'s one.

Example 5.5.1 Following Example 5.4.1, given a role \( RPnt \) with its behaviour \( t_{RPnt} = rp[p] \) and a port \( PPnt \) with its behaviour \( t_{PPnt} = rp \parallel tp \). It is straightforward to prove the compatibility between the role \( RPnt \) and the port \( PPnt \) by the fact that the axioms \( \text{SEV}(RPnt) \subseteq \text{SEV}(PPnt) \) and \( t_{PPnt} \subseteq t_{RPnt} \) remains true. Suppose now we want to replace \( PPnt \) with a new port \( PPntFax \) whose \( t_{PPntFax} = rp \parallel tp + dp \parallel df \). Apparently, the port \( PPntFax \) is substitutable for the port \( PPnt \):

\[ \text{SEV}(PPnt) \subseteq \text{SEV}(PPntFax) \]

\[ \sigma_{M_p}(rp \parallel tp) \bullet (\downarrow_{(r_{ppnt}, dp, df)}) \subseteq \sigma_{M_p}(rp \parallel tp \parallel dp \parallel df) \bullet (\downarrow_{(r_{ppnt})}) \Rightarrow t_{PPntFax} \subseteq t_{PPnt} \]

From the design point of view, it perfectly makes sense to replace the older printer with a printer-fax machine and keeps this procedure from being noticeable to other parts of the ticket vending machine.

- 157 -
In conclusion, substitutability is the ability of components to be safely substituted for current one. In general, substitutability requires the procedure to be side-effect free and unnoticeable to other parts of the system. The mainstream works on substitutability exploit the behavioural subtyping technique, such as the works of [ZW97, VHT00, HL99, Bra03]. Within our model, the requirements of substitutability can be easily achieved thanks to the independent connectors. We present a local approach (on port) to evaluate the substitutability by means of the subset relationship on service sets of two ports, and the refinement relationship on the behaviours of two ports. The substitutable port hints that it is also compatible with the attached role. Finally, we believe the definition of substitutability on ports can be easily rephrased for components.

5.6 Industrial Case Study

As we discussed earlier that the Koala model [vOvdL’00, vO02b, vO03, vO04] suffers from a lack of a solid semantic foundation, most of the current analysis on the Koala model is based on its informal semantics written in natural language. Such an informal description for Koala is often accompanied by Message Sequence Charts (MSC) [ITU00] that describe the series of interactions the Koala component should perform for its correct participation in fulfilling the certain task of the system.

By unfolding the MSCs into time-slot languages, following the formal construction described in Section 4.6, we may obtain the formal description of the Koala model. In this case study, we apply the formalism to a Koala fork configuration elaborated in [vO03, vO04], as detailed in Section 4.7.

Given the formalism of a Koala’s fork configuration, we now turn our attention to the architectural properties of the configuration. This entails considering properties of the anti-deadlock of the glue, loyalty of the roles with the glue, compatibility of the port and role, and substitutability between ports. In what follows we will examine these properties and illustrate why these properties are important in CBS design.

Example 5.6.1 Now based on the definition of fork $F$, we first examine the deadlock-freedom of the fork. Again, the interacting logic (glue) of fork $F$ is programmed through the protocol,

\[
t_F = ((E_{drA} \rightarrow E_{drB} \ || \ E_{drA} \rightarrow E_{drC} \ || \ E_{drC} \rightarrow E_{drA}) \ ||
\]
\[
(E_{drA} \rightarrow E_{drA} \ || \ E_{drB} \ || \ E_{drC} \rightarrow E_{drC})
\]
\[
(F_{rstA} \rightarrow F_{rstB} \ || \ F_{rstA} \rightarrow F_{rstC} \ || \ F_{rstC} \rightarrow F_{rstA})
\]

The protocol $t_F$ immediately satisfies the Axiom $(a)$ of deadlock-freedom as $t_F \not= \lambda$. Secondly, in order to demonstrate $t_F$ is well-defined, we have to show that every one of its sub-expressions

- 158 -
can be translated to a time-slot language and no empty sets arise in this interpretation. We can prove this property by evaluating whether the interpretation of $t_F$ in terms of a time-slot language displays the same set of service names as the alphabet of $t_F$, as the alphabet of the illegal sub-expression will not result in presence of the service name set of the time-slot language. The alphabet of $t_F$ can be obtained by $\text{NAM}(t_F) = \{EdrA, EdrB, EdrC, EdrAckA, EdrAckB, EdrAckC, FdrA, FdrB, FdrC, FrstA, FrstB, FrstC\}$. In the semantic domain, the set of service names of the corresponding time-slot language $L_F$ is retrieved from $\text{NAM}(L_F)$ where $L_F = \sigma_{M_s}(t_F)$. Briefly, $\text{NAM}(L_F) = \{EdrA, EdrB, EdrC, EdrAckA, EdrAckB, EdrAckC, FrstA, FrstB, FrstC\} \cup \{FdrA, FdrB, FdrC, FrstA, FrstB, FrstC\}$, is indeed equivalent to the resulting set of $\text{NAM}(t_F)$. From this outcome we can conclude that $t_F$ is well-defined and hence meets the Axiom $(b)$ of dead-lock freedom.

Thirdly, in considering the well-behavedness of $\text{FIL}(t_F)$, we need to prove that the interpretation of $\text{FIL}(t_F)$ does not result in an empty set. That is to say, we can tolerate a certain amount of illegal sub-expressions, but we have to guarantee that the whole expression properly constructs at least a run in the time-slot language. The rationale of doing this is that, regarding the interactions of connector purely as a sequence of bindings between services of roles, it is possible that certain unbound services are exposed in interaction design. Some unbound services are permitted, such as passive services $s^?_7$ and $s^?$, since they provide an option for their counterparts to opt for exercising them; and any out events, since it is up to their counterparts to trace their output actions. These unbound services do not cause deadlock in the overall interaction. However, other unbound services do, such as a required function will not proceed further until it receives a reply, and etc. We can detect these deadlock-prone services thereby examining the well-behavedness property of the resulting expression of $\text{FIL}(t_F)$. The FIL function always seeks unbound services in protocol and establishes each virtual binding for them with the empty string $\lambda$. Then, by interpreting the result protocol of $\text{FIL}(t_F)$ into the time-slot language, based on the binding operator, the virtual binding over deadlock-prone services will yield an undefined result. That will eventually reduce the whole time-slot language to an empty set, and therefore make the protocol not well-behaved. By contrast, the deadlock-free protocols will never end up with an empty set, thus are always well-behaved. When meeting unbound service $s^?$ in the protocol where $s \in S$, $\text{FIL}(s^?) = \lambda \mapsto s^?$ and then $\sigma_{M_s} = (\lambda \mapsto s^?) = \{\bot\} \cup \{\bot\} \cup \{\bot\} = \{\bot\} \cup \{\bot\}$. Remember that unbound $s^?$ is deadlock-free as explained earlier, but it will produce an empty set in the processing. That is the main reason why we do not apply well-definedness here. In terms of $t_F$, $\text{FIL}(t_F) = t_F$ because there are no unbound services. As
discussed in Section 4.7, \( \sigma_{\mathcal{M}}(t_F) \) delegates a time-slot language holding 522 runs. Based on this, we can derive that \( FIL(t_F) \) is well-behaved, and consequently, qualifies the Axiom (c) of deadlock-freedom.

We have seen that, to prevent deadlock on the glue, Axiom (a) excludes the inherently deadlock-prone process \( \lambda \); Axiom (b) blocks any problematic expressions (including unmatched bindings); and Axiom (c) isolates all unbound deadlock-prone service expressions. In the following, we present some counterexamples based on \( t_F \) to exhibit the ability of this property to catch deadlocks in the component cooperation.

**Example 5.6.2** In the current example, RoleA’s drop signal protocol could be an asynchronous process: once sent out a request to component RoleB for permission to drop the signal, RoleA need not wait for an acknowledgement to advance further. On the other hand, once it has received a drop signal request from RoleA, RoleB need not reply immediately. This loose-coupled relationship between drop request and drop acknowledgement can be modelled in two separate events: \( EdrA \) and \( EdrAckA \). Alternatively, RoleA’s drop signal command could be a synchronous process: RoleA requests RoleB for permission to drop the signal, while in this call, RoleB calls its driver to blank the output and then returns an approval. Assuming the drop acknowledgement is immediate, RoleA then informs its tuner driver to change frequency. In such a case, the time interval between the drop request and drop acknowledgement is regarded as instantaneous. This reflects a tight-coupled causal relationship between the request and acknowledgement. Instead of modelled via two independent events \( EdrA \) and \( EdrAckA \), we formalise this relationship in a function \( FdrA = (\text{dropReq}, \text{dropAck}) \), emphasising that after sending a dropReq, RoleA is blocked until receiving a dropAck. Conversely, we create \( FdrB = FdrC = (\text{dropReq}, \text{dropAck}) \).

In this example, let us switch around the drop signal event \( EdrA \) and function \( FdrA \) in \( t_F \).

\[ t'_F = (((FdrA \mapsto_{EdrB} EdrB \parallel EdrA \mapsto_{EdrC} EdrC);(EdrAckB \parallel EdrAckC) + \\
(EdrA \mapsto_{FdrB} FdrB \parallel EdrA \mapsto_{FdrC} FdrC));(FrstA \mapsto_{FrstB} FrstB \parallel FrstA \mapsto_{FrstC} FrstC) 
\]

As we can see that in \( t'_F \), the positions of \( EdrA \) and \( FdrA \) appear to swap, \( EdrAckB \) and \( EdrAckC \) are left unbound in that there are no longer acknowledgement messages for them.

To evaluate the deadlock-freedom property for \( t'_F \), Axiom (a) is passed by \( t'_F \neq \lambda \); Axiom (b) is failed in the sub-expressions \( (FdrA \mapsto_{EdrB} EdrB \parallel FdrA \mapsto_{EdrC} EdrC) \). In this, both bindings semantically return an empty set, because the binding operations of \( f_{FdrA \mapsto_{EdrB} FdrB} \) and \( f_{FdrA \mapsto_{EdrC} FdrC} \) give undefined results due to \( FdrA \neq EdrB \) and \( FdrA \neq EdrC \). Apparently, interacting between \( FdrA \) and \( EdrB \) (or \( EdrC \)) causes deadlock, that is, after \( FdrA \) emits a request
to $EdrB$ (or $EdrC$), the latter only absorbs it but cannot produce an acknowledgement, then $FdrA$ suffers infinitely waiting for that acknowledgement. It might be worth noting that all service mismatches would be detected at this stage; $t'_{f}$ fails Axiom (c) due to the preceding error. The resulting protocol of $FIL(t'_{f})$ is listed as follows:

$\text{FIL}(t'_{f}) = (((FdrA \rightarrow EdrB \ || \ FdrA \rightarrow EdrC) ; (EdrAckB \rightarrow EdrAckB \ \lambda) + (EdrA \rightarrow FdrB \ || \ EdrA \rightarrow FdrC) ; (FrstA \rightarrow FrstB \ || \ FrstA \rightarrow FrstC)) \ |

Since $(FdrA \rightarrow EdrB \ || \ FdrA \rightarrow EdrC)$ generates an empty set, the overall interpreting process on $FIL(t'_{f})$ will reach an empty set, that violates the well-behavedness property. We need to point out here that $(EdrAckB \rightarrow EdrAckC \ \lambda) + (EdrA \rightarrow FdrB \ || \ EdrA \rightarrow FdrC)$ does not introduce an empty set in the binding operation. It can be understood that the emitting message from $EdrAckB$ and $EdrAckC$ is not directed to anyone, and is lost through neglect. In the similar vein, $(EdrA \rightarrow FdrB \ || \ EdrA \rightarrow FdrC)$ does not result in an empty set, because it is not compulsory for $EdrA$ to listen $FrstB$ and $FrstC$’s acknowledgements.

In general, roles are decentralised design of glue. To insure consistency on the specifications of roles and glue, we introduce the loyalty property for them. Assuming that a role $R$ and a glue $G$ is defined by $(SG, <jm a (tG))$ and $(SR, gMs (tR))$, respectively, by examination of the loyalty with $R$ and $G$, we need to check that $tG$ and $tR$ are not illegal expressions, and prove that $gM_{s}(tR)$ is semantically equivalent to $SRK(\sigma_{M_{s}}(tG)\big|_{NAM(g)})$. The principle behind this is that considering glue as the interaction logics over all parties, dividing up the logics into separate portions, each role should be a portion representing one party involved in the collaboration. The requirement comes down to the behaviour of each role should be undertaken in a way that is consistent with that of the glue as a whole. To implement this checking, the operation $\sigma_{M_{s}}(tG)\big|_{NAM(g)}$ helps us to obtain the time-slot language from $G$ that covers the services only belonging to $R$. For the moment, the service restriction retains all idle time slots which are previously occupied by the services from other roles. It is now the time for the function $SRK$ to come into play in wiping out these idle time slots. And the resulting time-slot language should be exactly the one interpreted from $R$. In short, the procedure is to use service restriction and the $SRK$ function to derive a time-slot language over a restricted set of services from the glue, and to measure if such a time-slot language is identical to the one of the role in question that is created from scratch. Let us revisit the example of the Koala fork configuration.
Example 5.6.3 As set forth in Section 4.7, the fork connector $F$ has RoleA, RoleB and RoleC. Due to a large number of runs produced from the glue protocol $t_F$, for the sake of simplicity, we only present the process involved with the tabular run shown on page 122. In terms of RoleA,

- $t_{RA} = (((EdrA \| EdrA); (EdrAckA \| EdrAckA)) + (FdrA \| FdrA)); (FrstA \| FrstA)$ is well-defined;

- as the run example exhibited below, $\text{SRK}(\sigma_{M,}\left|_{\text{NAMU}_{\text{SI}}} \right|_{\text{t}_F})$ is corresponding to the time-slot language constructed through $t_{RA}$;

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
\text{ID} & \text{Component} & \text{Signature} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
\hline
1 & EdrA & !dr & 1dr & & & & & & & & & & & & \\
\hline
\end{array}
\]

In view of this outcome, we can conclude that RoleA is loyal to the glue of fork $F$.

Analogously, the loyalty property between RoleB and the glue of fork $F$ can be demonstrated as follows:

- $t_{RB} = (EdrB; EdrAckB + FdrB); FrstB$, is clearly well-defined; and furthermore

- $\text{SRK}(\sigma_{M,}\left|_{\text{NAMU}_{\text{SI}}} \right|_{\text{t}_F}) = \sigma_{M,}\left|_{\text{t}_{RB}} \right|$, from the process on the run example outlined below

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
\text{ID} & \text{Component} & \text{Signature} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
\hline
\hline
\end{array}
\]
In the same way, we can conclude that RoleC is in loyalty with the glue of fork F.

When integrating a component into a CBS context, the component will be wired to (at least) a connector in the CBS, thereby attaching the component’s port(s) to the connector’s role(s). To recap, connectors dominate components’ collaboration in this model. A role is the explicitly defined interface for connectors to identify the responsibilities of a service module in the attaching component, whereas a port is an interaction point for the attaching component to provide concrete operational competence to deliver the promised service. In this concern, roles are extensively considered to be the interfaces of ports. The compatibility with bound port and role is applied here to reason about whether the port satisfies the requirements of the role, so that the port can perform the designed behaviour of the role in the resulting system. To investigate this property, it shall suffice to show that the port can do whatever the role wants. This criterion can be formulated as follows. Axiom (a): in the static domain, the port should include all services of the role; and Axiom (b): in the dynamic domain, the port should be able to perform all possible behaviours of the role. Using the formal terms in this thesis, Axiom (a) can be written as $\text{SEV}(R) \subseteq \text{SEV}(P)$. We impose an extra condition $\text{SEV}(R) \neq \emptyset$ to prevent empty roles which have no ‘sockets’ for ports to plug in; Axiom (b) could be indicated via $t^* \cap t^* P$. The technique involves first unifying the signatures of time-slot languages of role $R$ and port $P$ by appending blank rows for the differentiated services to each other, and second verifying if the runs of role $R$ is a subset of which of port $P$. Let us consider the Koala fork configuration again.

Example 5.6.4 Based on the definitions of fork F’s RoleA, and component A’s port PHA (as IHA) formalised in Section 4.7, we can reason about their compatibility as follows.

$\text{SEV(ROLE)} = S_{R_{A}} = \{!\text{dropReq}, ?\text{dropAck}, (!\text{dropReq}, ?\text{dropAck}), (!\text{restore}, ?\text{restoreAck})\}$

\[ \neq \emptyset \] , and

$\text{SEV(IHA)} = S_{I_{HA}} = \{!\text{dropReq}, ?\text{dropAck}, (!\text{dropReq}, ?\text{dropAck}), (!\text{restore}, ?\text{restoreAck})\}$

Axiom (a) is attained by $\text{SEV(ROLE)} = \text{SEV(IHA)} \neq \emptyset$;
✿ given \( t_{RA} = (((EdrA || EdrA);(EdrAckA || EdrAckA)) + (FdrA || FdrA));(FrstA || FrstA) \),

\[
\begin{align*}
& t_{RA} = (((EdrA || EdrA);(EdrAckA || EdrAckA)) + (FdrA || FdrA)) || (FrstA || FrstA),
\end{align*}
\]

and equation (a), we can derive \( t_{RA};t_{HA}^0 =_{M_x} t_{RA} \) and \( t_{HA};t_{RA}^0 =_{M_x} t_{HA} \) from Remark 4.3.1 (a). We need to show that \( t_{HA} \sqsubseteq t_{RA} \). By Definition 4.4.18, interleaving typically includes concatenation. We can deduce the following:

\[
\sigma_{M_x}(t_{RA}) \subseteq \sigma_{M_x}(t_{HA})
\]

Definition 4.4.18

\[
\Rightarrow t_{HA} \sqsubseteq t_{RA}
\]

Definition 5.1.4

So we can claim that Axiom (b) holds.

Since both Axioms (a) and (b) hold true, we can conclude that fork F’s Role A is compatible with component A’s port PHA.

Dually, the compatibility between fork F’s Role B and component B’s port PHB (as IHB) can be analysed as below.

✿ SEV(RoleB) = \( S_{RB} = \{\?\text{dropReq},!\text{dropAck},(?\text{dropReq},!\text{dropAck}),(?\text{restore},!\text{restoreAck})\} \neq \emptyset \), and

\[
\begin{align*}
& SEV(IHB) = S_{IHB} = \{\?\text{dropReq},!\text{dropAck},(?\text{dropReq},!\text{dropAck}),(?\text{restore},!\text{restoreAck})\},
\end{align*}
\]

Axiom (a) is satisfied by \( SEV(RoleB) = SEV(IHB) \neq \emptyset \); (a)

✿ given that \( t_{RB} = (EdrB;EdrAckB + FdrB);FrstB \), \( t_{IHB} = (EdrB;EdrAckB + FdrB) || FrstB \),

and equation (a), we can deduce that \( t_{RB};t_{IHB}^0 =_{M_x} t_{RB} \) and \( t_{IHB};t_{RB}^0 =_{M_x} t_{IHB} \) by Remark 4.3.1 (c). It suffices to show that \( t_{IHB} \sqsubseteq t_{RB} \). In the same vein as above, we can prove this relation as follows.

\[
\sigma_{M_x}(t_{RB}) \subseteq \sigma_{M_x}(t_{IHB})
\]

Definition 4.4.18

\[
\Rightarrow t_{IHB} \sqsubseteq t_{RB}
\]

Definition 5.1.4

Based on the results obtained, we can claim that fork F’s Role B is compatible with component B’s port PHB. This result also applies to the compatibility assessment for fork F’s Role C with component C’s port PHC.

Component substitution and in particular update of components is a vital mechanism for maintaining installed applications up-to-date. The key requirement is that the upgrade must not introduce new problems, but rather fix the old one or enhance the application. In configuration management terms, it must preserve (or improve) the configuration consistency of the application.
The main effect of substitutability is that, given the compatibility between the role \( R \) and the port \( P \), the substitutability of the new port \( P' \) with the current one \( P \) ensures the compatibility between \( P' \) and \( R \). Let us further assume \( P' \) has the same service kit as \( P \). Standing on the role \( R \)'s point of view, substitutability is a much-relaxed version of compatibility, and compatibility could be considered as a special case of substitutability, i.e., \( t_P \subseteq t_P \subseteq t_R \). With these concerns, the method used to validate the compatibility of \( P \) and \( R \) is also applicable to assess the substitutability of \( P' \) and \( P \). In practice, the new component should be designed to be as flexible as possible to increase the chances on substitution. On the other hand, the role should be specified as rigidly as possible to increase the chances of achieving successful compositions. Let us return back to the Koala example.

**Example 5.6.5** In addition to the current Koala fork configuration, we arrange to upgrade the component \( A \) to \( A' \) which provides an auto tune service to align a signal. The formal description is \( A' = (S_{A'}, L_{A'}) \), where

\> the signature \( S_{A'} = (P_{A'}, S_{A'}, SEV_{A'}) \), in which

\> a) \( P_{A'} = \{FautoTune, EdrA', EdrAckA', FdrA', FrstA'\} \)

\> b) \( S_{A'} = \{(!autoTuneReq, !autoTuneAck), !dropReq, ?dropAck, \)

\> (?!dropReq, ?dropAck), (?!restore, ?restoreAck)\)

\> c) \( SEV_{A'}(FautoTune) = (!autoTuneReq, !autoTuneAck) \)

\> \( SEV_{A'}(EdrA') = !dropReq \)

\> \( SEV_{A'}(EdrAckA') = ?dropAck \)

\> \( SEV_{A'}(FdrA') = (!dropReq, ?dropAck) \)

\> \( SEV_{A'}(FrstA') = (?!restore, ?restoreAck) \)

\> the behaviour \( L_{A'} = \sigma_{M_{A'}}(t_{A'}) \), where

\> \( t_{A'} = FautoTune([(EdrA' || EdrA') || (EdrAckA' || EdrAckA')] + (FdrA' || FdrA')) || \)

\> (FrstA' || FrstA')\)

The definition of component \( A' \) is generated from the composition of two of its ports \( PVA' \) and \( PHA' \). The definition of port \( PVA' \) is declared as \( (S_{PVA'}, L_{PVA'}) \), in which

\> its signature \( S_{PVA'} = (P_{PVA'}, S_{PVA'}, SEV_{PVA'}) \), where
a) \( P_{PV'} = \{\text{autoTune} \} \)

b) \( S_{PV'} = \{ (? \text{autoTuneReq}, !\text{autoTuneAck}) \} \)

c) \( \text{SEV}_{PV'}(\text{autoTune}) = (? \text{autoTuneReq}, !\text{autoTuneAck}) \)

- its behaviour \( L_{PV'} = \sigma_{M_\phi}(t_{PV'}) = \text{autoTune} \)

As well, the definition of port \( PHA' \) is treated as \( (S_{PHA}, L_{PHA}) \), where

- the signature is \( S_{PHA} = (P_{PHA}, S_{PHA}, \text{SEV}_{PHA}) \), in which

  a) \( P_{PHA} = \{ \text{EdrA}', \text{EdrAckA}', \text{FdrA}', \text{FrstA}' \} \)

  b) \( S_{PHA} = \{ !\text{dropReq}, ?\text{dropAck}, (? \text{dropReq}, ?\text{dropAck}), (? \text{restore}, ?\text{restoreAck}) \} \)

  c) \( \text{SEV}_{PHA}(\text{EdrA'}) = !\text{dropReq} \)

      \( \text{SEV}_{PHA}(\text{EdrAckA'}) = ?\text{dropAck} \)

      \( \text{SEV}_{PHA}(\text{FdrA'}) = (? \text{dropReq}, ?\text{dropAck}) \)

      \( \text{SEV}_{PHA}(\text{FrstA'}) = (? \text{restore}, ?\text{restoreAck}) \)

- the behaviour is \( L_{PHA} = \sigma_{M_\phi}(t_{PHA}) \), where

  \( t_{PHA} = (((\text{EdrA'} || \text{EdrA'}) || (\text{EdrAckA'} || \text{EdrAckA'}) + (\text{FdrA'} || \text{FdrA'}) || (\text{FrstA'} || \text{FrstA'}) \)

To validate the substitutability between the new port \( PHA' \) and the current port \( PHA \), we can carry out the following diagnoses.

- \( \text{SEV}(PHA') = \{ !\text{dropReq}, ?\text{dropAck}, (? \text{dropReq}, ?\text{dropAck}), (? \text{restore}, ?\text{restoreAck}) \} = \text{SEV}(PHA) \neq \emptyset \) \hs (a)

  Hence, Axiom (a) is met.

- from equation (a), we can have \( t_{PHA}; t_{PHA} =_{M_\phi} t_{PHA} \) and \( t_{PHA}; t_{PHA} =_{M_\phi} t_{PHA} \) by Remark 4.3.1 (c). It is suffice to show \( t_{PHA} \sqsubseteq t_{PHA} \). We demonstrate this relation in the following.

  \( \sigma_{M_\phi}(t_{PHA}) \subseteq \sigma_{M_\phi}(t_{PHA}) \) \hs Definition 4.4.18

  \( \Rightarrow t_{PHA} \sqsubseteq t_{PHA} \) \hs Definition 5.1.4

  Therefore, Axiom (b) is met.

Since Axioms (a) and (b) are both satisfied, we assert that the new port \( PHA' \) is substitutable for the current port \( PHA \).
To show the underlying compatibility between $PHA'$ and RoleA,

- $SEV(PHA') = \{\{dropReq, ?dropAck\}, \{dropReq, ?dropAck\}, \{restore, ?restoreAck\}\} = SEV(RoleA) \neq \emptyset$

- Based on equation (a), we can have $t_{PHA}^0 \cdot t_{RA}^0 = M_{SE}^t$ and $t_{RA} \cdot t_{PHA}^0 = M_{SE}^t$ through Remark 4.3.1 (c). We need to show $t_{PHA} \sqsubseteq t_{RA}$. We can prove this axiom by,

\[
\sigma_{M_{SE}} (((EdrA \parallel EdrA); (EdrAckA \parallel EdrAckA)) + (FdrA \parallel FdrA); (FrstA \parallel FrstA)) \sqsubseteq
\sigma_{M_{SE}} (((EdrA' \parallel EdrA') \parallel (EdrAckA' \parallel EdrAckA')) + (FdrA' \parallel FdrA') \parallel (FrstA' \parallel FrstA'))
\]

Definition 4.4.18

\[\Rightarrow t_{PHA} \sqsubseteq t_{PHA} \quad \text{Definition 5.1.4}\]

All axioms (a) and (b) are validated. We can say that $PHA'$ is compatible with $RoleA$.

From the maintainer's viewpoint, the new component $A'$ preserves all functionalities of component $A$. In addition, it enhances the tune function of its port $PVA'$, and relaxes the behaviour constraint from sequencing $((EdrA \parallel EdrA); (EdrAckA \parallel EdrAckA))$ to interleaving $((EdrA' \parallel EdrA') \parallel (EdrAckA' \parallel EdrAckA'))$ in its port $PHA'$. At this point, of course it is safe to replace the current port $PHA$ of $A$ with the new port $PHA'$ of $A'$. As from the role $RoleA$'s point of view, $PHA'$ just creates new execution orders, but it does not hinder $RoleA$ to extract some of the old orders designed to participate in the interaction. Finally, the substitutability of components $A'$ and $A$ is subject to the substitutability of all their ports.

5.7 Conclusion

Nowadays, the use of the term 'compatibility' and 'conformance' in the literature is rather ambiguous and has various meanings in different contexts. Therefore, we need to clarify these terms in what follows.

We first distinguish between correctness and conformance. A component implementation is correct in relation to the component interface when it is both consistent and complete. Theoretically, we would say that an implementation is consistent with a specification if all its behaviours conform to the specification. It is then complete with respect to the specification if it implements all the behaviours that conform to a specification. A simple form of completeness implies that all features listed in the interface are actually implemented; more complex forms of specification requires all possible orders of calls permitted according to the specification to be served by the implementation.
Correctness is thus a relation between implementations and specifications. Quite distinct from correctness, we need conformance, as a relation between specifications. In our context, the relation is between behaviour descriptions on different components, such that either components can cooperate properly or one can replace the other. The conformance regarding compatibility is defined for a component with its environment. A component is compatible to its environment if, (i) statically, the structure of the component conforms to a given architecture context; (ii) dynamically, the component is always available to provide services to its environment, and vice versa. The conformance regarding substitutability is defined to relate the same kind of components, for example, for component updating. It is based on the classical principle of behaviour subtyping.

Interoperability finally, extends the above relations. A component is interoperable to its environment if it is compatible to its environment and its implementation must be correct. This is a ‘formal’ view of interoperability. It is in line with the work by Vallecillo et al [VHT00], who understand it as a mutual correspondence of interface (to be) bound, so that their owners can interoperate. Likewise, a component is interoperable to another component if it is substitutable to that component and has correct implementation. This is the ‘practical’ view of interoperability; one understands the term as the ability of a new version (of a component) to safely substitute for a previous one.

Interoperability is one of key issues of building applications from reusable components. In principle, interoperability is the absence of any interoperability error. Traditionally, two main levels of interoperability have been distinguished: the signature level (names and signatures of services), and the semantic level (the ‘meaning’ of services). The first one deals with the ‘plumbing’ issues, while the second one covers the ‘behavioural’ aspects of component interoperation. Konstantas also refers to them as the ‘static’ and ‘dynamic’ levels [Kon95], and Bastide and Sy talk about components that ‘plug’ and ‘play’ when referring to those two levels [BS00].

In the works of Vallecillo et al [VHT00], these authors further contemplate a different interoperability level – the protocol level, which deals just with the relative order in which a
component expects its services to be called, the order in which it invokes other components' services, and the rules that initialise and govern the object interactions.

Within this big picture, we placed our interest down to the conformance relationship between components and introduced our checks across all three interoperability levels. Scaling down to our model, we introduced the compatibility between port and role, to capture the relationship between components and their environments (connectors); that provides a support for a 'formal' view of interoperability. Further, we proposed the substitutability between ports, to formalise the relationship between components; that offers an approach for 'practical' view of interoperability.

In our model, composition is given at the instance level: component instances interacting in a specific instance of a system architecture (connector), the compatibility and substitutability properties are normally considered between component/connector instances.

Being focused on connectors, we introduced the deadlock-freedom property on the glue and the loyalty property between roles and glue. Since our connectors are discrete entities, these two properties are applicable for both connector types and instances. Also, these properties will play a key role in defining the well-formedness of connectors in the next chapter.
Chapter 6

Well-formedness of Component-based Systems

In this chapter, we propose the property of 'well-formedness' in our CBS model based on the CBS architecture design introduced in Chapter 3, the behaviour description and formal CBS definition proposed in Chapter 4, and the architectural properties discussed in Chapter 5. First of all, the well-formedness on interfaces is given in Section 6.1; it requires that a well-formed interface must have a set of unmatched services and its behaviour be well-defined. Secondly, the well-formedness of ports is provided in Section 6.2, in which two kinds of port, service port and proxy port, are treated differently. The well-formedness of a service port examines the service unmatchability over interfaces, and also the refinement relationship of the user-defined behaviour with the inherent behaviour (the theoretically most liberal behaviour). For proxy ports, we present a way to keep the specification consistency in CBS hierarchical design. Thirdly, the well-formedness of components is introduced in Section 6.3, where the main principle is similar to the well-formedness of service ports. Finally, we discuss the well-formedness property of connectors in Section 6.4. This section evaluates the unmatchability of services over a connector's roles, the loyalty of all roles with its glue and the deadlock-freedom of the glue. All properties above consider both the static and dynamic aspects of the model in question. We end each section by providing an illustrative example. Finally, a CE industry case study is used to demonstrate our approach in Section 6.5.

6.1 Interfaces

An interface is a cohesive group of one or more observable services. We can formally define an interface as an ordered pair \((S, L)\) according to the Definition 4.6.1. In this, \(S\) records the static service signatures exhibiting on the interface, and \(L\) specifies the occurrence order of these services within a time frame. This dynamic description can be captured by our component protocol expression and underlying time-slot language.

One common demand for the CBS modeller is to be able to check whether the user-defined interface description is well-formed. That is, whether the interface description is valid regarding
its structure and behaviour. In order to meet this demand, we introduce the well-formedness of component interfaces. A well-formed interface description is required to respect certain rules defined as below.

**Definition 6.1.1** Let \( t_i \in \mathcal{P} \), and an interface \( I = (\mathcal{E}_i, \sigma_{MCH}(t_i)) \), we should say that the interface \( I \) is well-formed, if

\[
\begin{align*}
& a) \quad \forall s \in \text{SEV}(I), \forall t \in \text{SEV}(I) : \text{MCH}(s, t), \text{ and} \\
& b) \quad t_i \text{ is well-defined}
\end{align*}
\]

Based on the preceding discussion on the component description in Section 4.6, we already know that there are no duplicated services in the signature and the time-slot language describes the behaviours of all services listed in the signature. To be able to make the whole description well-formed at the interface level, we further require that, (a) from the static aspect, there are no (full or partial) matched service pairs found in \( \mathcal{E}_i \); (b) from the dynamic aspect, the behaviour expression \( t_i \) must be well-defined. Implicitly, the corresponding time-slot language \( L_i \) is valid and nonempty.

Under Axiom (a), (full or partial) matched services will not coexist in any interfaces. Otherwise, it breaks the basic concept of component design (discussed in Chapter 2): *maximal encapsulation*. If a component requires a service which is also provided by the same component, it is unnecessary to expose this dependency. That entails redundant service interactions and unnecessarily exposes the system to potential hacker attack. By using the matching invoker function MCH (cf. Definition 4.2.11), we can easily identify any full and partial matched service pair in the interface.

Axiom (b) states that for any well-formed descriptions of interfaces, their behaviour expressions have to be well-defined. Semantically, their time-slot languages are never empty. Normally, the empty time-slot language is yielded from improper behaviour expressions, which encounter undefined operations over time-slot languages, and in turn generate empty set. Therefore, we need to exclude these possibilities on \( L_i \). Besides, the simplest well-formed description of the interface is \( I = (\mathcal{E}_i, L_i) \) where \( \text{SEV}(I) = \emptyset \) and \( L_i = \{1\} \). Diagrammatically, this description specifies an interface without any observable services.

**Example 6.1.1 Case 1:** following Example 4.2.4, we define the interface \( ISel \) under its port \( PSel \). The definition of \( ISel \) is exactly the same as that which is specified in \( PSel \): we would say that \( ISel \) is well-formed, since its services \( sm, ts \) and \( sr \) are apparently distinct, and its behaviour language \( L_{ISel} \) is not empty.

![Figure 6.1 The Vendor Component](image)
is generated by a well-defined behaviour expression \( ts \parallel sm; sr \).

**Case 2:** now, let us change the behaviour of the interface to \( ts^0 \parallel sm^0; sr^0 \). In the semantic domain,

\[
L'_{jSel} = \sigma_{M,p}(ts^0 \parallel sm^0; sr^0) = \sigma_{M,p}(ts^0) \parallel \sigma_{M,p}(sm^0) \cdot \sigma_{M,p}(sr^0)
\]

\[
= \{ \perp_x \} \parallel \{ \perp_y \} \cdot \{ \perp_z \}
\]

As we can see that, although there is no service occurring in time slots, the expression is still well-defined. Therefore, the interface is well-formed. In this case, we can consider the interface ISel has been switched off.

**Case 3:** suppose the behaviour language \( L''_{jSel} \) is formed by the expression \( ts \mapsto sm; sm; sr \). Obviously the required function \( ts \) does not match the provided function \( sm \). Semantically,

\[
L''_{jSel} = \sigma_{M,p}(ts \mapsto sm; sm; sr) = \sigma_{M,p}(ts) \mapsto sm \sigma_{M,p}(sm) \cdot \sigma_{M,p}(sr)
\]

\[
= \{ f_{ts} \} \mapsto sm \{ f_{sm} \} \cdot \{ f_{sr} \}
\]

\[
= \emptyset \cdot \{ f_{sr} \}
\]

\[
= \emptyset
\]

Axiom (b) does not hold since the expression \( ts \mapsto sm; sm; sr \) is not well-defined, and also we can see that \( L''_{jSel} = \emptyset \). Therefore, we conclude that the interface is not well-formed.

**Case 4:** similar to case (1), we define an interface IPnt under its port PPnt and IPnt’s description is the same as that on PPnt. The interface IPnt is well-formed, because its services \( rp \) and \( tp \) are two distinct functions, and its behaviour \( rp[tp] \) is well-defined.

**Case 5:** again, we define an interface IBuy under its port PBuy, and IBuy’s description is equivalent to that of PBuy. We would say that
the interface $IBuy$ is well-formed, as its services $cI$ and $rc$ are different with each other, and its behaviour description $ci;rc$ is well-defined.

Case 6: finally, we define an interface $IMon$ under its port $PMon$ and the description of $IMon$ is exactly the same as $PMon$’s description. Since the interface has only the service $sa$, and of course its behaviour description $sa$ is well-defined, we can assert that the interface $IMon$ is well-formed.

In summary, this section discussed the well-formedness of the interface. Based on the formal description given in Definition 4.6.1, to be able to define a well-formed interface, we require that (i) statically, no matched services are listed on the interface; and (ii) dynamically, the behaviour specification is well-defined, implying that the corresponding behaviour language is always nonempty. Practical examples of a ticket vending machine are given to demonstrate this property.

6.2 Ports

In general, a port represents an interaction point between a component and its environment. There are two kinds of port introduced in Chapter 3: (i) service port, possesses a cohesive set of interfaces which often exhibits one module of functionalities exposed to a component’s environment; (ii) proxy port, mainly acts as an interception point allowing sub-components to interact with the external entities from higher hierarchies. In this section, we firstly focus on the well-formedness property of service ports, thereafter turn our attention to the proxy ports.

Definition 6.2.1 Suppose a service port $SP = (\mathcal{E}_{SP}, \sigma_{M_{SP}}(t_{SP}))$ is made up of a set of interfaces $I^* = \{I_1, \ldots, I_n\}$ in form of $I_i = (\mathcal{E}_{I_i}, \sigma_{M_{I_i}}(t_{I_i}))$, where $t_{SP}, t_{I_i} \in P$, we shall say that the service port $SP$ is well-formed, if

- $\forall I_i \in I^*: I_i$ is well-formed, and
- $\forall s \in SEV(I_i), \exists t \in SEV(I_j): MCH(s, t)$, for any $I_i \neq I_j \in I^*$, and
- $\{SEV(I_i): I_i \in I^* \land SEV(I_i) \neq \emptyset\}$ is a partition of $SEV(SP)$, and
- $t_{I_i} \sqcup \cdots \sqcup t_{I_n} \subseteq t_{SP}$
Chapter 6 Well-formedness of Component-based Systems

Axiom (a) reflects the recursive nature of well-formedness. Axiom (b) and (c) express the requirements for the static model of service port, while Axiom (d) stresses on its dynamic model.

First of all, Axiom (a) requires that all interfaces of the service port are locally well-formed, i.e., there are no matched services on any interface and the behaviour specification of each interface is well-defined.

Secondly, Axiom (b) is to validate service unmatchability across the scope of all interfaces on the service port in question. In plain words, for any specific services in a certain interface, there does not exist a service in another interface such that the two services are matching with each other. By imposing this axiom, we are able to apply the principle of component maximal encapsulation at the port level. Note that, instead of directly checking the service unmatchability on $S_{sp}$, this approach benefits from the efforts done on interface well-formedness validation.

Thirdly, by using a set partition in Axiom (c), it imposes a restriction that the port signature $S_{sp}$ must contain all services under its nonempty interfaces. On the other hand, there are no duplicated services can be found between interfaces, with respect to the concept of partition in set theory.

Fourthly, regarding the behaviour aspect of a service port, Axiom (d) states that any user-defined behaviour of the service port has to be a refinement of the inherent behaviour of the service port, which is theoretically the most liberal execution order amongst interfaces in a service port, formed by placing all interface behaviours in an interleaving pattern. Therefore, the maximal degree of parallelism in a service port can be achieved. That is, all its interfaces execute independently with each other. In other words, the service port can run among its interfaces in any order. Therefore, we consider any user-defined behaviour of a service port is to place certain order constraints between interfaces. Moreover, the benefit from this axiom is that, by constructing components following a bottom-up approach, the upper-level (for example port) behaviour description is always able to keep respect to the lower-level (for example interface) behaviour description so as to avoid the duplication of design efforts.

For any given service port, if all axioms above are held, we can say that the port in question is well-formed. Let us demonstrate this property via the following example.

Example 6.2.1 Case 1: consider the service port $PSel$ in Example 6.1.1, it has an interface $ISel$, and which is well-formed as discussed previously, so Axiom (a) holds. Since $ISel$ is the only interface in $PSel$, Axiom (b) always holds because there is no $I_i$ found in $I^*$. In addition, Axiom (c) holds by a remark of partition: Every singleton set \{x\} has exactly one partition. And Axiom (d) holds by (a) of Remark 5.1.2, $t_{ISel} \subseteq t_{PSel}$. Therefore, the service port $PSel$ is well-formed. For exactly the same reason, the service ports $PPnt$, $PBuy$ and $PMon$ are well-formed too.
Case 2: assuming we add a goods-delivery module to the service port $P_{Sel}$ as depicted in Figure 6.5. The $I_{Dev}$ interface has an address-requesting function ($ar$), a ticket-selling function ($ts$) and a goods-delivering function ($gd$), where the $ts$ function is exactly equivalent to the $ts$ function in $I_{Sel}$. We shall say that the new service port $P_{Sel}$ is not well-formed: as $I_{Sel}$ and $I_{Dev}$ have the duplicated function $ts$, Axiom (c) is no longer holding.

Now, let us turn our attention to proxy ports. The nature of a proxy port is actually an interception point on a composed component such that sub-components can communicate with their parent architecture through proxy ports. Further, as mentioned in Chapter 3, only service and proxy ports are permitted to map to proxy ports.

Since the services on proxy ports normally originate from the lower level, the well-formed specifications of proxy ports should be somehow consistent with those of their lower-level structures. We are especially interested in whether the behaviour of the internal structure is preserved while mapping it to the higher levels in the hierarchy. To evaluate this property, an assumption has to be made:

*The ports on the same sub-components have to be mapped to the same proxy ports.*

Under this assumption, we can prevent the situation where the behaviour of a sub-component spreads over distinct proxy ports. This would most likely alter the sub-component behaviour.
Example 6.2.2 In Figure 6.6 above, there are two components \( \text{Com1} \) and \( \text{Com2} \) that are defined inside the composed component \( \text{ComA} \), which has two proxy ports \( \text{PotF} \) and \( \text{PotG} \). The \( \text{PotF} \) is mapped by the service ports \( \text{PotA} \) and \( \text{PotD} \), on which the observable services \( \text{FunA} \) and \( \text{EvtB} \) are listed. The \( \text{PotG} \) is mapped by the service port \( \text{PotB} \), on which the observable service \( \text{FunC} \) is exhibited. Further, \( \text{ComA} \) contains a connector whose roles \( \text{RoleA} \) and \( \text{RoleB} \) are linked with the ports \( \text{PotC} \) of \( \text{Com1} \) and \( \text{PotE} \) of \( \text{Com2} \), respectively. Assuming that the behaviour of \( \text{Com1} \) is \( t_{\text{Com1}} = t_{\text{PotA}}; t_{\text{PotB}}; t_{\text{PotC}} \) and that of \( \text{Com2} \) is \( t_{\text{Com2}} = t_{\text{PotD}}; t_{\text{PotE}} \), we specify the behaviour of \( \text{PotF} \) to be \( t_{\text{PotF}} = t_{\text{PotA}}; t_{\text{PotD}} \), and specify that of \( \text{PotG} \) to be \( t_{\text{PotG}} = t_{\text{PotB}} \). To define \( \text{ComA} \)'s behaviour, its proxy ports \( \text{PotF} \) and \( \text{PotG} \) will be taken into consideration in this context, such as \( t_{\text{ComA}} = t_{\text{PotF}} \parallel t_{\text{PotG}} = t_{\text{PotA}}; t_{\text{PotD}} \parallel t_{\text{PotB}} \); that means \( \text{PotF} \) and \( \text{PotG} \) run in interleaving mode. Obviously, this behaviour does not conform to the behaviour \( t_{\text{Com1}} \) specified in the sub-component \( \text{Com1} \), because \( \text{PotA} \) and \( \text{PotB} \) in \( \text{Com1} \) run sequentially with no gap in-between, but \( \text{PotD} \) starts immediately after \( \text{PotA} \) completed in \( \text{ComA} \). On the other hand, \( \text{PotB} \) runs independently of \( \text{PotA} \) followed by \( \text{PotC} \). Note that the behaviours of \( \text{PotC} \) and \( \text{PotE} \) become internal from \( \text{ComA} \). When reasoning about the behaviour consistency in different layers of hierarchy, internal behaviours are normally omitted according to the assumptions of time-slot model (as stated on page 75).

![Figure 6.7 The ComA Component without PotG](image-url)

The assumption introduced previously helps us to exclude this kind of design. According to that assumption, \( \text{PotA} \) and \( \text{PotB} \) must map to the same proxy port (\( \text{PotF} \), as sketched in Figure 6.7) because they reside in the same component \( \text{Com1} \). Then, the behaviour of \( \text{PotF} \) will take \( \text{PotB} \) into account: it could be \( t_{\text{PotF}} = t_{\text{PotA}}; t_{\text{PotB}}; t_{\text{PotD}} \), now which preserves \( \text{Com1} \)'s \( t_{\text{Com1}} = \)...
Chapter 6 Well-formedness of Component-based Systems

Let $t_{PotA}, t_{PotB}, t_{PotC}$ and $Com2$'s $t_{Com2} = t_{PotD}, t_{PotE}$ be internal behaviour. Note that the internal behaviour $t_{PotC}, t_{PotE}$ will be neglected while considering the behaviour consistency between $t_{Com1}, t_{Com2}$ and $t_{PotF}$.

Then, the measure on the behaviour consistency in the hierarchy can be achieved by checking a proxy port with a high degree of confidence. In each proxy port, we can check whether the refinement relationship has held between the user-defined behaviour and the inherent behaviour. The latter can be generated by a function introduced below.

**Definition 6.2.2** Given a proxy port $PP$ which is mapped by a set of ports $P^\star = \{P_1, \ldots, P_n\}$, a partition of $P^\star$ is made such that $\{\{P_1, P_2, P_3\}, \{P_4, P_5, P_6\}\}$ where $P_1, P_3, P_5$ belong to the component $C_1 = (S_{C_1}, L_{C_1})$ and $\{P_2, P_4, P_6\}$ belong to the component $C_2 = (S_{C_2}, L_{C_2})$, we define a function

$$\text{IPB}(PP) = L \in M_{\mathcal{G}}$$

in such a case

- $L = \text{SRK}(L_{C_1} | \text{NAM}(P_1) \cup \text{NAM}(P_3) \cup \text{NAM}(P_5)) \cup \text{SRK}(L_{C_2} | \text{NAM}(P_2) \cup \text{NAM}(P_4) \cup \text{NAM}(P_6))$

The IPB function helps us to produce the inherent behaviour of a proxy port from the given internal structure, in which two possibilities need to be taken into account: (i) the ports with the same sub-components; in such cases the behaviours in terms of those ports can be retrieved by using a service restriction on the behaviour of sub-components, and then applying the SRK function to remove all unnecessary idle time slots; (ii) the ports with different sub-components, in this case the resulting behaviour can be generated by placing behaviours of these ports in the interleaving scheme.

**Example 6.2.3** Continuing from Example 6.2.2, the inherent behaviour of port $PotF$ is $\text{IPB}(PotF) = \sigma_{M_{\mathcal{G}}} \left(t_{PotA}, t_{PotB} \parallel t_{PotD}\right)$, and as we can see that, the mapping ports $PotA$ and $PotB$ are in the same component but $PotD$ is not.

Taking all of the above concepts together, now we can define the well-formedness of proxy ports.

**Definition 6.2.3** Suppose a proxy port $PP = (S_{PP}, \sigma_{M_{\mathcal{G}}} (t_{PP}))$ is mapped by a set of ports $P^\star = \{P_1, \ldots, P_n\}$, where $t_{PP} \in \mathcal{P}$, we shall say that the proxy port $PP$ is well-formed, if

- $\forall P_i \in P^\star : P_i$ is well-formed, and
- $\forall s \in \text{SEV}(P_i), \exists t \in \text{SEV}(P_1) : \text{MCH}(s, t)$ for any $P_i \neq P_j \in P^\star$, and
- $\{\text{SEV}(P_i) : P_i \in P^\star\}$ is a partition of $\text{SEV}(PP)$,
- $\sigma_{M_{\mathcal{G}}} (t_{PP}) \subseteq \text{IPB}(PP)$
The proxy port is an intermediary for sub-components to interact with the external entities from parent environments. Therefore, there is no interface defined in proxy ports and they are solely mapped by the ports from sub-components. Further, Axiom (c) requires that every mapping port should at least contain one service. In other words, the proxy port does not accept a port without any observable services to connect through. Axiom (d) states that the user-defined behaviour of the proxy port should be a subset of the inherent behaviour of the proxy port in the semantic domain, which is generated by the function IPB.

**Example 6.2.4** Following on from Examples 6.2.2 and 6.2.3, we should say that the proxy port $PotF$ is well-formed. First of all, its mapping ports $PotA$ and $PotB$ are well-formed. Secondly, we consider the relevant services $FunA \neq EvtB \neq FunC$. Thirdly, the definition of $PotF$ covers all services from mapping ports $PotA$, $PotB$ and $PotD$. Fourthly,

\[ \lnot \left( (IPotA \neq PotB \neq PotD) \Rightarrow \text{IPB}(PotF) \right) \]

In conclusion, in this section we discussed about the well-formedness regarding service ports and proxy ports. A well-formed service port requires that (i) all interfaces of the port are well-formed; (ii) there are no matched services throughout all interfaces; (iii) the collection of service sets on interfaces is a partition of the service set on the port; and (iv) any user-defined port behaviour must be a refinement of the inherent port behaviour. For proxy ports, we further proposed an approach to keep behaviour consistent in CBS hierarchical encapsulation.

### 6.3 Components

A component is a modular unit with one or more ports that are replaceable within the environment. In order to support the visibility and modularity restriction employed in CBS, a component can be either primitive or composed. A primitive component is a black-box entity possessing no other components and all of its ports are service ports. A composed component is a composition structure defined as a coherent group of components linked by connectors (if any), and all its ports are proxy ports. In this section, we define a generic property of well-formedness for both kinds of components.

**Definition 6.3.1** Suppose a component $C = (\mathcal{S}_C, \sigma_{M,p}(t_C))$ is made up of a set of ports $P' = \{P_1, \ldots, P_n\}$ in form of $P_i = (\mathcal{S}_R, \sigma_{M,p}(t_R))$, where $t_C, t_R \in \mathcal{P}$, we shall say that the component $C$ is well-formed, if

1. $\forall P_i \in P' : P_i$ is well-formed, and
2. $\forall s \in \text{SEV}(P_i), \forall t \in \text{SEV}(P_j) : \text{MCH}(s, t)$, for any $P_i \neq P_j \in P'$, and
c) \( \{ \text{SEV}(P_j) : P_j \in P^* \land \text{SEV}(P_j) \neq \emptyset \} \) is a partition of \( \text{SEV}(C) \), and

d) \( t_{P_i} \parallel \ldots \parallel t_{P_n} \subseteq t_c \)

The basic principle of well-formedness of components is similar to that of service ports (as explained in Section 6.2). It is noteworthy that \( P^* \) mentioned above is a generalised term for ports. For primary components, \( P^* \) denotes a set of service ports. For composed components, \( P^* \) denotes a set of proxy ports.

**Example 6.3.1 Case 1:** following Example 6.1.1, we should say that primary component \( \text{Vendor} \) is well-formed, since it has only one port \( P_{Sel} \), and which is well-formed by Example 6.2.1. In the similar vein, the primary components \( \text{Printer} \), \( \text{Customer} \) and \( \text{Monitor} \) are well-formed too.

**Case 2:** according to Example 6.2.2, we should say that composed component \( \text{ComA} \) is well-formed, as its only proxy port \( P_{Of} \) is well-formed.

To sum up, in this section we introduced a generic property of well-formedness for both primary and composed components. For primary components, we require that they only have a set of service ports; while for composed components, we require that they only contain a set of proxy ports. The general principle of well-formedness for components is similar to the well-formedness of service ports.

### 6.4 Connectors

As discussed, the connectors in our model are separate entities, that are defined explicitly and able to coordinate component collaborations. In general, a connector comprises a certain number of roles and a glue. A connector can be formally specified as an ordered pair \( C = (S_C, L_C) \), which is in effect equivalent to the description of its glue \( G = (S_G, L_G) \). The signature \( S_C \) of the connector is made up of a union of all services on its roles. The behaviour language \( L_C \) of the connector is actually from the interaction workflow specified on its glue. To be able to evaluate the well-formedness of a connector, this property is formally defined as follows.

**Definition 6.4.1** Suppose a connector \( C \) is made up of a set of roles \( R^* = \{R_1, \ldots, R_n\} \) and a glue \( G \), in form of \( R_i = (S_{R_i}, \sigma_{M_{R_i}}(t_{R_i})) \) and \( G = (S_G, \sigma_{M_G}(t_G)) \), where \( t_{R_i}, t_G \in \mathcal{P} \), we shall say that the connector \( C \) is well-formed, if

a) \( \forall s \in \text{SEV}(R_i), \exists t \in \text{SEV}(R_i) : \text{MCH}(s, t) \), for all \( R_i \in R^* \), and
Chapter 6 Well-formedness of Component-based Systems

b) \( \text{SEV}(G) = \bigcup_{R_j \in R^*} \text{SEV}(R_j) \), and

c) \( \forall R_j \in R^*: R_j \text{ is loyal to the glue } G \), and

d) the glue \( G \) is deadlock-free

In consideration of the static aspect of a connector, we require that: (a) there are no (full or partial) matched service pairs defined in each role; (b) regardless of empty string roles, the sets of services on other roles is a division of the glue services into non-overlapping role services that cover all services of the glue. From the dynamic aspect of a connector, we state that (c) all roles should be in loyalty with the glue; and (d) of course the glue must be deadlock-free.

Axiom (a) requires that there are no (full or partial) matched services defined in every role of the connector, with the same consideration on the well-formedness for interfaces and ports. To recap, a role describes the desired kind of component (port) to be involved in the service interactions on the connector.

Axiom (b) says that the union of the service sets on all roles must be equal to the service set on the glue. This constraint excludes the circumstance of ‘orphan’ services. More precisely, that denotes the services which are mentioned in the glue but not appeared in any roles. Note that the situation of overlapping services across different roles is permitted. This is often happened in the connectors with multi-party interactions. Once all axioms on \( S \) are satisfied, we turn our attention to the behaviour language \( L \).

Axiom (c) further ensures that all roles are loyal to their glue. That eliminates the possibility of inconsistent behaviour descriptions between roles and port. Notice that this axiom complemented with Axiom (b) completely define the relationship between roles and glue in connectors.

Finally, Axiom (d) states that the process on the glue is free from deadlock. It ensures that the inter-component cooperation is specified in well-formed logics.

Overall, once all four axioms above are satisfied, we are able to conclude that the connector in question is well-formed. Now, let us consider the following example.

Example 6.4.1 Following Examples 4.5.4 and 4.6.1 (stated on page 111, 113), the connector \( \text{Conn} \) has four roles: \( R_{Sel} \), \( R_{Pnt} \), \( R_{Buy} \), and \( R_{Mon} \). Note that there is no empty string role defined in this connector. Axiom (a) holds, as

- \( \text{SEV}(R_{Sel}) = \{ (\text{?insertCoins},!\text{produceTicket}), (!\text{recordTrans},?\text{printTrans}),
\quad (?\text{startMon},!\text{endMon}) \} \)

- \( \text{SEV}(R_{Pnt}) = \{ (?\text{produceTicket},!\text{printReceipt}), (!\text{recordTrans},!\text{printTrans}) \} \)
♦ \( \text{SEV} (\text{RBuy}) = \{ \text{insertCoins}, ?\text{printReceipt} \} \)

♦ \( \text{SEV} (\text{RMon}) = \{ \text{(startMon, ?endMon)} \} \)

It appears that there is no matched service pair in each role. Axiom (b) also holds because \( \text{SEV} (\text{RSel}) \cup \text{SEV} (\text{RPnt}) \cup \text{SEV} (\text{RBuy}) \cup \text{SEV} (\text{RMon}) = \text{SEV} (\text{Conn}) \) and even they are pairwise disjoint. As shown earlier in Example 5.3.2 (on page 149), all roles are in loyalty with the glue \( \text{Glue} \), thus Axiom (c) holds. Finally, \( t_{\text{Glue}} = ci \rightarrow u \ (sa \rightarrow sm \ sm || ts \rightarrow rp [sr \rightarrow ip \ tp]) || rp \rightarrow rc \)

is well-defined, and \( \text{FIL} (t_{\text{Glue}}) = t_{\text{Glue}} \). By interpreting it in the semantic domain

\[
\sigma_{M_{gl}} (t_{\text{Glue}}) = \sigma_{M_{gl}} (ci \rightarrow u \ (sa \rightarrow sm \ sm || ts \rightarrow rp [sr \rightarrow ip \ tp]) || rp \rightarrow rc)
\]

\[
= \sigma_{M_{gl}} (ci \rightarrow u \ (\sigma_{M_{gl}} (sa) \rightarrow sm \ \sigma_{M_{gl}} (sm) || \sigma_{M_{gl}} (ts) \rightarrow rp \ \sigma_{M_{gl}} (rp))
\]

\[
= \{ f_{ci} \} \rightarrow t \ ((f_{sa}) \rightarrow sm \ (f_{sm}) \rightarrow t \ (f_{ts}) \rightarrow rp \ (f_{rp}) \rightarrow ip \ (f_{tp})) || (f_{rp}) \rightarrow rc \ (f_{rc})
\]

it turns out that \( \text{FIL} (t_{\text{Glue}}) \) is well-behaved. Therefore, the glue \( \text{Glue} \) is deadlock-free, and Axiom (d) is satisfied. Finally, we can conclude that the connector \( \text{Conn} \) is well-formed.

In closing this section, we discussed what it means for a connector to be well-formed in this section. A qualified well-formed connector requires that, (i) no matched services are defined in each role; (ii) the service set on the glue is a union of the collection of service sets on all roles; (iii) all roles are in loyalty with the glue; and (iv) the glue is free from deadlock. The first two axioms concern the static information about the connector, while the last two axioms focus on its dynamic aspect. We end this section by illustrating this property using the example regarding the connector of the ticket vending machine introduced earlier.

6.5 Industrial Case Study

In this section, we apply the formal constructions introduced in this chapter to examine the well-formedness property of the case study discussed in Section 4.7 and 5.6. The idea is that, from an initial set of system descriptions provided by the component designer(s), our proposed formal formedness properties are checked prior to integration. This will then confirm that the individual units are compliant with the general principles of CBS design. We list here for reference the properties to be checked:
service unmatchability on the scope of individual components;
• hierarchical consistency in the specifications of composed components;
• deadlock-freedom and glue-roles loyalty within the connectors, and etc.

The advantage in doing so is that, we can catch unit-level errors before integration testing (for example compatibility testing), making life easier for component integrators to maintain focus on high-level system integration issues. In the following examples, we will evaluate each property in turn.

We start by considering the well-formedness of interfaces. In general, a component is an encapsulated software element offering a predefined service kit. Interface refers to an abstraction that a component provides of itself to the outside world. This separates the methods of external communication from internal operation, and allows it to be internally modified without affecting the way outside entities interact with it. An interface publishes sets of provided services that are implemented by components, and also declares sets of required services which normally tie to provided ones in supporting the functionalities in absence of the components. The main well-definedness concern of interfaces here is to keep all services unmatched, so as to prevent the redundant service dependencies from being exposed to the outside world. In addition, a well-defined interface model requires its behavioural description to be well-defined.

Example 6.5.1 Referring back to the current definition of component \( A, B \) and \( C \) given in Section 4.7, we now evaluate interfaces \( IVA, IHA \) and \( IHB \) (or \( IHC \)) for the well-formedness property.

First of all, the well-formedness of interface \( IVA \) \((S_{IVA}, L_{IVA})\) can be simply proved from

• \( SEV(S_{IVA}) = \{(?\text{tuneReq},!\text{tuneAck})\} \), there does not exist a service \( s \), such that \( MCH(?\text{tuneReq},!\text{tuneAck}), s) = true \), so Axiom (a) of interface well-formedness holds;
• \( L_{IVA} = \sigma_{M,\text{r}} (t_{IVA}) \), in which \( t_{IVA} = \text{Ftune} \), \( \text{NAM}(t_{IVA}) = \{\text{Ftune}\} = \text{NAM}(L_{IVA}) \Rightarrow t_{IVA} \) is well-defined, and hence respects Axiom (b) of interface well-formedness.

With respect to the results above, we can assert that interface \( IVA \) is well-formed. Note that the interface containing a single service with single occurrence of the service is always well-formed.

Secondly, as we can see from the definition of interface \( IHA \) \((S_{IHA}, L_{IHA})\)

• \( SEV(S_{IHA}) = \{(!\text{dropReq},?\text{dropAck}), (!\text{dropReq},?\text{dropAck}), (!\text{restore},?\text{restoreAck})\} \)

let \( s_{EdrA} = !\text{dropReq}, s_{EdrAckA} = ?\text{dropAck}, s_{FdrA} = (!\text{dropReq},?\text{dropAck}) \) and
\( s_{PrstA} = (!\text{restore},?\text{restoreAck}) \), then

\[ MCH(s_{EdrA}, s_{EdrAckA}) = MCH(s_{EdrA}, s_{FdrA}) = MCH(s_{EdrA}, s_{PrstA}) = MCH(s_{EdrAckA}, s_{FdrA}) = \]
Chapter 6 Well-formedness of Component-based Systems

MCH($s_{EdrAckA}, s_{FrstA}$) = MCH($s_{FdrA}, s_{FrstA}$) = false, therefore Axiom (a) of the well-formedness of interfaces holds true;

♦ $L_{IHA} = \sigma_{M,A}(t_{IHA})$, where

$$t_{IHA} = (((EdrA || EdrA); (EdrAckA || EdrAckA)) + (FdrA || FdrA)) || (FrstA || FrstA)$$

NAM($t_{IHA}$) = {EdrA, EdrAckA, FdrA, FrstA} = NAM($L_{IHA}$) ⇒ $t_{IHA}$ is well-defined, and therefore satisfies Axiom (b) of the well-formedness of interfaces.

As a result, we claim that interface IHA is well-formed.

Thirdly, by examining the well-formedness of interface IHB ($S_{IHB}, L_{IHB}$) or IHC ($S_{IHC}, L_{IHC}$),

♦ SEV($S_{IHB}$) = {?dropReq, !dropAck, (?dropReq, !dropAck), (?restore, !restoreAck)},

let $s_{EdrB} = ?dropReq$, $s_{EdrAckB} = !dropAck$, $s_{FdrB} = (?dropReq, !dropAck)$ and

$s_{FrstB} = (?restore, !restoreAck)$, then

MCH($s_{EdrB}, s_{EdrAckB}$) = MCH($s_{EdrB}, s_{FdrB}$) = MCH($s_{EdrB}, s_{FrstB}$) = MCH($s_{EdrAckB}, s_{FdrB}$) =

MCH($s_{EdrAckB}, s_{FrstB}$) = MCH($s_{FdrB}, s_{FrstB}$) = false, it follows Axiom (a) of the well-formedness of interfaces;

♦ $L_{IHB} = \sigma_{M,A}(t_{IHB})$, in which $t_{IHB} = (EdrB; EdrAckB + FdrB) || FrstB$,

NAM($t_{IHB}$) = {EdrB, EdrAckB, FdrB, FrstB} = NAM($L_{IHB}$) ⇒ $t_{IHB}$ is well-defined, it meets Axiom (b) of the well-formedness of interfaces.

It can be seen in Figure 4.12 that the definition of interface IHC is analogous to that of interface IHB. Therefore, we conclude that interfaces IHB and IHC are well-formed.

A port is a feature of a component which specifies a distinct connection point between that component and its environment, or between the (behaviour of the) component and its internal parts. The first case is called a service port and the second case is called a proxy port. Service ports support a collection of interface descriptions specifying the nature of the interactions that may occur over a port. The interactions from the required services of a port characterise the requests that may be made from the component to its environment through this port. The interactions from the provided services of a port characterise requests to the component that its environment may make through this port. From the perspective of service ports, well-formedness should check whether (a) all owning interfaces are well-formed; (b) all owning services across different interfaces are mutually unmatched; (c) all owning interface signatures are a partition of the port signature into non-overlapping signatures that cover all of the elements defined in the port
signature; and, (d) the behaviour of port is built on the basis of the behaviours of owing interfaces. In brief, Axiom (a) shows the bottom-up assembly of well-formedness. For example, a well-formed port implies that all associating interfaces are well-formed, and so forth. Axiom (b) prohibits port-wide service matching. Usually, the required services of the matched pair need not appear in the port, in the sense that the port can enquire its provided services internally. There is no need to increase the degree of external dependency of the port. Axiom (c) ensures all nonempty interfaces’ signatures are pairwise disjoint and the union of them is equal to the signature of port. Finally, Axiom (d) is used to protect the design of interfaces. It is required that the port behaviour is obtained by combining the interface behaviours. In this way, the original design of the lower-level entities (for instance, interfaces) can be held at the higher level (for example, port). Concerning behaviour combination, as interleaving produces most order combinations, any customer-designed behaviour must be a subset of behaviour combination by interleaving, so that behaviour preservation checking can be reduced to behaviour refinement checking.

**Example 6.5.2** Returning to the Koala fork configuration, let us examine the well-formedness property for ports $PVA$, $PHA$, and $PHB$ (or $PHC$).

We first validate the well-formedness for port $PVA$. Since its only interface $IVA$ is well-formed as discussed in Example 6.5.1, we can immediately prove that Axioms (a), (b), and (c) are true. Further, from Remark 5.1.2 (a), $Ftune_C \subseteq Ftune$, that denotes Axiom (d) holds. Therefore, we can conclude that $PVA$ is well-formed.

In the same vein, based on the results of Example 6.5.1 and the nature of single interface structures, ports $PHA$ and $PHB$ (or $PHC$) satisfy Axiom (a), (b), and (c) of well-formedness. Also, by reflexivity of the refinement relation $\subseteq$, we can justify their Axiom (d) of well-formedness, and subsequently assert the well-formedness of $PHA$ and $PHB$ (or $PHC$).

A proxy port is an interception point on which the component’s internal ports are able to communicate with other ports at levels in the hierarchy. No services originally belong to proxy ports. The services exposed on the proxy port must be eventually pinned to an internal service port which is a mapping with the proxy port or one of its interconnected proxy ports. The main criteria (Axioms (a), (b) and (c)) for evaluating well-formedness of proxy ports are similar to those for service ports. However, because of the diverse nature of two kinds of ports, in the behaviour aspect (Axiom (d)), the emphasis of proxy ports is mainly placed on verifying model consistency over interconnecting ports at different hierarchy levels. An architectural assumption needs to be made before proceeding with the analysis. That is, the ports on the same sub-components must map to the same proxy ports. As we can see from Figures 6.6 and 6.7, this assumption saves us from all the general cases that cause behavioural inconsistency: by
decomposing internal port behaviours into different higher-level proxy ports, the behaviours on
the proxy ports would clearly represent inconsistency with the behaviours on the internal ports.
Except for these cases, all other causes of behaviour inconsistency need to be identified, thereby
assessing the subset relationship between the user-defined behaviour and inherent behaviour of
the proxy port in the semantic domain. It is noteworthy that the inherent behaviour of proxy ports
is built upon extracting mapped ports’ behaviours from each sub-component and then placing
them in an interleaving fashion.

**Example 6.5.3** $PVD$ is the only proxy port in this example. Let us examine its well-formedness.

- the mapping port of $PVD$, $PVA$, is well-formed from Example 6.5.2, Axiom (a) holds;
- by the first result of Example 6.5.1, Axioms (b) and (c) are also true;
- $\sigma_{M_{A}}(t_{PVD}) = \text{IPB}(PVD) = \text{SRK}(L_{PVD} \mid \{\text{Flame}\}) = \{f_{\text{Flame}}\}$, satisfies Axiom (d).

Hence, we can claim that proxy port $PVD$ is well-formed.

A component is a system element offering predefined services and able to communicate with
other components (via connectors). A component in this model takes the form of collections of
ports, and adheres to its formal specification $(S, L)$ so that the component can exist
autonomously from other components in the system and gain features like reusability and so forth.
Generally speaking, the so-called well-formed component in this context is equivalent to saying
that its formal specification (in all hierarchies) is well-formed. The method of identifying this
property is closely analogous to that of service ports. The only difference comes from different
hierarchy levels between the primary/composed components and the service ports.

**Example 6.5.4** Let us go back to the Koala fork configuration to examine the well-formedness
property for components $A$, $B$, $C$ and $D$.

In consideration of component $A$ with its formal description $(S_{A}, L_{A})$,

- from Example 6.5.2, all its ports $PVA$ and $PHA$ are well-formed, which meets Axiom (a);
- $\text{SEV}(S_{A}) = \{(\text{tune Req}, \text{tune Ack}), (\text{drop Req}, \text{drop Ack})\}$,
  
  let $s_{\text{Flame}} = (\text{tune Req}, \text{tune Ack})$, $s_{\text{Edra}} = (\text{drop Req})$, $s_{\text{Edra Ack}} = (\text{drop Ack})$,

  $s_{\text{Fdra}} = (\text{drop Req}, \text{drop Ack})$ and $s_{\text{Frad}} = (\text{restore Req}, \text{restore Ack})$, then

  \[
  \text{MCH}(s_{\text{Flame}}, s_{\text{Edra}}) = \text{MCH}(s_{\text{Flame}}, s_{\text{Edra Ack}}) = \text{MCH}(s_{\text{Flame}}, s_{\text{Fdra}}) = \text{MCH}(s_{\text{Edra}}, s_{\text{Edra Ack}}) = \text{MCH}(s_{\text{Edra}}, s_{\text{Fdra}}) = \text{MCH}(s_{\text{Edra Ack}}, s_{\text{Fdra}}) = \text{MCH}(s_{\text{Fdra}}, s_{\text{Frad}})
  \]
Chapter 6 Well-formedness of Component-based Systems

MCH(s_{EdrAck}, s_{FrstA}) = MCH(s_{FdrA}, s_{FrstA}) = false, this result qualifies Axiom (b);

- SEV(\mathcal{S}_{PV_A}) \cup SEV(\mathcal{S}_{PH_A}) = \{(?tuneReq, !tuneAck)\} \cup \{!dropReq, ?dropAck, (?dropReq, !dropAck), (?restore, !restoreAck)\} = SEV(\mathcal{S}_{A}) above, that satisfies Axiom (c); finally

- t_{PV_A} \parallel t_{PH_A} = Ftune \parallel (((EdrA \parallel EdrA); (EdrAckA \parallel EdrAckA)) + (FdrA \parallel FdrA)) \parallel (FrstA \parallel FrstA), and

\begin{align*}
t_A &= Ftune[(((EdrA \parallel EdrA); (EdrAckA \parallel EdrAckA)) + (FdrA \parallel FdrA)) \parallel (FrstA \parallel FrstA)] ,
\end{align*}

we can deduce \( t_{PV_A} \parallel t_{PH_A} \subseteq t_A \) by Definition 4.4.18.

The result of this experiment is that component \( A \) is well-formed.

In terms of component \( B \) with its formal description \((\mathcal{S}_B, L_B)\),

- we have seen that \( B \)'s only port \( PH_B \) is well-formed in Example 6.5.2, Axiom (a) is therefore satisfied;

- \( SEV(\mathcal{S}_B) = \{ ?dropReq, !dropAck, (?dropReq, !dropAck), (?restore, !restoreAck) \} \),

let \( s_{EdrB} = ?dropReq \), \( s_{EdrAckB} = !dropAck \), \( s_{FdrB} = (?dropReq, !dropAck) \) and \( s_{FrstB} = (?restore, !restoreAck) \), then

\begin{align*}
MCH(s_{EdrB}, s_{EdrAckB}) &= MCH(s_{EdrB}, s_{FdrB}) = MCH(s_{EdrB}, s_{FrstB}) = MCH(s_{EdrAckB}, s_{FdrB}) ,
\end{align*}

\begin{align*}
MCH(s_{EdrAckB}, s_{FrstB}) &= MCH(s_{FdrB}, s_{FrstB}) = false , \text{ Axiom (b) is passed;}
\end{align*}

- \( SEV(\mathcal{S}_{PH_B}) = SEV(\mathcal{S}_B) \), Axiom (c) is qualified; and finally

- \( t_B = t_{PH_B} = (EdrB; EdrAckB + FdrB) \parallel FrstB \Rightarrow t_B \subseteq t_{PH_B} \subseteq t_B \), so Axiom (d) is attained.

This proof shows that the definition of component \( B \) satisfies all axioms of the well-formedness property for components, hence component \( B \) is well-formed. This result also applies to testify for the well-formedness of component \( C \).

With regard to component \( D \) with its formal description \((\mathcal{S}_D, L_D)\),

- we have shown that port \( PV_D \) is well-formed in Example 6.5.3, so Axiom (a) is qualified;

- \( SEV(\mathcal{S}_D) \) is a singleton \{(?tuneReq, !tuneAck)\}, there does not exist a service \( s \) in its signature, such that \( MCH(\{?tuneReq, !tuneAck\}, s) = true \), Axiom (b) is valid;

- \( SEV(\mathcal{S}_{PV_D}) = SEV(\mathcal{S}_D) \), it turns out that Axiom (c) is satisfied; and finally
\[ t_D = t_{PVD} = Ftune \Rightarrow t_D \subseteq t_{PVD} \wedge t_{PVD} \subseteq t_D, \text{ Axiom (d) is passed.} \]

As all axioms are true, we can conclude that component \( D \) is well-formed.

Connectors are standalone design-time to runtime units that mediate interactions among components. A connector is made up of one glue and a collection of roles. The glue specifies the entire interaction flow over all participating components. Each role specifies the behaviour encountered in one component during its interaction with others. Recall that all parts in this model can be defined into an ordered pair \((S,L)\), including role and glue. The definition of a connector is consequently formed by the definition of its glue and roles. Considering the nature of glue, unlike components, the definition of a connector is certainly identical to that of its glue. To measure the design quality of a connector, we need to look at: (a) the service unmatchability on each role; (b) the service union of roles to the glue; (c) the loyalty of all roles with the glue; and (d) the deadlock-freedom of the glue.

**Example 6.5.5** Regarding the fork \( F \) of the Koala fork configuration, it has three roles \( RoleA, RoleB \) and \( RoleC \), where

\[
\begin{align*}
\text{SEV}(RoleA) &= \{!\text{dropReq}, ?\text{dropAck}, (!\text{dropReq}, ?\text{dropAck}), (!\text{restore}, ?\text{restoreAck})\}, \\
\text{SEV}(RoleB) &= \text{SEV}(RoleC) = \{?\text{dropReq}, !\text{dropAck}, (!\text{dropReq}, !\text{dropAck}), (!\text{restore}, !\text{restoreAck})\}, \\
(?!\text{restore}, !\text{restoreAck})
\end{align*}
\]

We can validate the well-formedness of fork \( F \) as below:

\[ \begin{align*}
\text{suppose } s_{EdrA} &= !\text{dropReq}, s_{EdrAckA} = ?\text{dropAck}, s_{FdrA} = (!\text{dropReq}, ?\text{dropAck}), \\
& s_{FrstA} = (!\text{restore}, !\text{restoreAck}), \text{ and } s_{EdrB} = ?\text{dropReq}, s_{EdrAckB} = !\text{dropAck}, \\
& s_{FdrB} = (?\text{dropReq}, !\text{dropAck}), s_{FrstB} = (?\text{restore}, !\text{restoreAck}), \text{ then} \\
\text{MCH}(s_{EdrA}, s_{EdrAckA}) &= \text{MCH}(s_{EdrA}, s_{FdrA}) = \text{MCH}(s_{EdrA}, s_{FrstA}) = \text{MCH}(s_{EdrAckA}, s_{FdrA}) = \\
& \text{MCH}(s_{EdrAckA}, s_{FrstA}) = \text{MCH}(s_{FdrA}, s_{FrstA}) = false, \text{ and} \\
\text{MCH}(s_{EdrB}, s_{EdrAckB}) &= \text{MCH}(s_{EdrB}, s_{FdrB}) = \text{MCH}(s_{EdrB}, s_{FrstB}) = \text{MCH}(s_{EdrAckB}, s_{FdrB}) = \\
& \text{MCH}(s_{EdrAckB}, s_{FrstB}) = \text{MCH}(s_{FdrB}, s_{FrstB}) = false, \text{ also applies to } RoleC.
\end{align*} \]

This result implies that all services in each role are pairwise unmatched;

\[ \begin{align*}
\text{SEV}(RoleA) \cup \text{SEV}(RoleB) \cup \text{SEV}(RoleC) &= \{?\text{dropReq}, !\text{dropReq}, ?\text{dropAck}, !\text{dropAck}, \\
& (?\text{dropReq}, !\text{dropAck}), (!\text{dropReq}, ?\text{dropAck}), (?\text{restore}, !\text{restoreAck}), \\
& (!\text{restore}, !\text{restoreAck})\} = \text{SEV}(F), \text{ that satisfies Axiom (b)};
\end{align*} \]
all roles RoleA, RoleB and RoleC are loyal to the glue, as shown in Example 5.6.3;

as manifested in Example 5.6.1 that the glue of fork F is deadlock free.

Finally, we claim that the fork connector F is well-formed.

6.6 Conclusion

Perhaps the term 'well-formed' should be supplanted by the term 'rigorously defined'. That means we can unrestrictedly design any kind of CBS, the rigorously defined CBSs (also called well-formed CBSs) are a particular class of CBSs that are developed following certain rules to assure the quality of design. In this chapter, we use our mathematical notations to identify the rules in designing interfaces, service/proxy ports, primary/composed components and connectors. The underlying idea is that in reasoning about the well-formedness property we can identify design defects in individual parts before combining them together. For example, we should be able to detect the deadlock-prone interaction mechanism in a connector before attaching components to it. Also, when defining a component, we should be able to check whether any matched service pair appears in the component; if so the maximal encapsulation principle for components is breached, because the component itself has already provided this service and it is unnecessary to reveal the dependency to the component’s environment. Furthermore, we should also able to maintain the description consistency in the hierarchical component design. In particular, we need to be able to guarantee that the behaviour of a composed component always originates from that of its internal structures. Following the bottom-up software testing approach, the well-formedness property of individual components and connectors will be unit-tested before wiring them together to construct a CBS.
Chapter 7

Conclusion

The work in this thesis presents a formal framework for the specification of connector-oriented CBS development which supports rigorous analysis and reasoning about the component interaction and composition. The additional information on the observable behaviours of components can be exploited within pragmatic approaches to software engineering in uncovering inconsistencies of CBS specifications.

In this chapter, we include some concluding remarks along with a more detailed summary of the results of the thesis. We also discuss possible directions for further work.

7.1 Summary

It should be recognised that describing software architectures in terms of interaction relationships between components brings us closer to a compositional view, and hence a more flexible or open view of an application [ND95]. First-class connectors allow us to view an application’s architecture as a composition of independent components. We gain in flexibility, since each component could engage in a number of different agreements, increasing the reuse potential of individual components. Separating connectors from the components also promotes reuse and refinement of typical interaction relationships. It opens the possibility of the refinement of connectors and the construction of complex connectors out of simpler ones. Providing our connector capability to dominate component collaboration completes the encapsulation that we wish to achieve. That is, we separate a connector’s communication and control logic from a component’s computation logics, thereby maximising loose encapsulation in terms of computation. Being inspired by this notion, we provided a generic CBS model based on the Koala model in Chapter 3. This model significantly promotes the reusability and maintainability of components as well as connectors, and makes our CBSs clearer and simpler to implement.

Also, it should be realised that the specification of a CBS requires additional behavioural information about components. This information is necessary to analyse and reason about the behaviour of the CBS and also to know what to expect when individual components are placed in a different context.
Chapter 7 Conclusion

This research work has been motivated by the observation that current approaches to CBSE typically lack a formal behavioural semantics. We use the Koala component model as an illustration of this. Koala’s graphical descriptive techniques have been extended to include useful notation for components at the specification level, but still do not provide designers with a standard way of expressing behaviours of components. Industrial specifications using mainstream software engineering practices often suffer from inconsistencies that are due to the difficulties of defining component cooperation in a concurrency scenario.

Current formal approaches to the specification and analysis of components are connected with concurrency arising through composition. This serves the all important purpose of interconnecting services from different components, but does not capture the concurrency at a single interface. For instance, in a reactive system, upon receiving a request on a provided function, the component may have to respond by generating responses concurrently on, say, two of its required functions.

We proposed a language-based representation of component behaviour in Chapter 4. Each component is associated with a specific set of runs. In this way, at each time slot during a period of activation, the corresponding run records the sequence of all services that have occurred on the component. The set of runs that describe the intended behaviour of a component comprises its time-slot language.

The time-slot model explicitly uses time slots, i.e. global clocks, that make the behaviour model highly synchronous. We feel that, for the intended application of the model, it is safe to assume the existence of global clocks, which can unambiguously assign service occurrences to specific time slots. That makes the behaviour model more concrete and explicit.

The discreteness of time slots and downward closure property allowed us to relate our formal description of component behaviour to a more general theory of non-interleaving representation of behaviour, in terms of behavioural presentations and asynchronous transitions systems [Shi88, Shi97]. Using a behavioural presentation for a component allows us to use the temporal relations derived from this model for the orderings between service occurrences on component interfaces. This means that it is appropriate to talk about true concurrency, just as in the event structures [NPW81, Win88], and also simultaneity (by considering the equivalence class generated by the pre-order and the mutual exclusion relations in behaviour presentations). The difference between the two, in software design terms, can be understood as the difference between may and must occur at the same time. The concurrency in the time-slot model is considered at the individual interface level. It makes this language-based model expressive enough to capture concurrent and simultaneous services occurring on the same interface of the component.

We also addressed basic features of the time-slot model for describing sequential, included, alternative and interleaving behaviours. In addition, the service interaction has been explicitly
delivered via the binding operator. Based on the basic event-to-event and function-to-function bindings, the binding operator further supports event-to-function, function-to-event and function-to-function (chained) kinds of partial interactions. The use of additional binding methods boosts the interoperability of components.

Considering time-slot languages as an operational semantic domain, we gave it a protocol-like regular notation – component protocol $\mathcal{P}$, as a syntax language for component behaviour descriptions. By adopting the initial algebra approach, the meaning of any component protocol expression can be uniquely constructed to be a specific time-slot language. In other words, there is no other way to interpret expressions in the semantic domain. Component protocol formulates the component behaviour at the service level rather than the event level. An instant benefit is that synchronous services (functions) can be treated as primitive alphabets rather than a pair of event alphabets. The formal definition of a function alphabet is given in the component signature of the time-slot model.

Next, attention was focussed on the architectural properties of a CBS. We identified conditions, in terms of signatures and languages, which determine the deadlock-freedom of glue, the loyalty of role with glue, the compatibility of role with port and the substitutability of ports. The well-definedness and well-behavedness notions on the component protocol are given to assist the checkings of connectors’ deadlock-freedom of glue and loyalty of role with glue. The language-based nature of our behaviour model allows us to define simply the behaviour refinement relationship by using the generic subset relationship in set theory. The refinement relationship is used to evaluate the compatibility and substitutability properties between components and connectors.

Apart from identifying interoperability between components and connectors, the well-formedness property of each of them promotes reliability allowing engineers to wire them with confidence. The bottom-up approach facilitates the unit-test from atomic component interfaces to composite components. The well-formedness of components is mainly focusing on: the service unmatchability on the static aspect, and the refinement relationship between the user-defined behaviour and the inherent behaviour (the most liberal behaviour) on the dynamic aspect. In this way, the modularity of composable and hierarchical component development can be maintained. The well-formedness of composed components further needs to consider the description consistency in the hierarchy, especially from the concern of behaviour. The well-formedness of connectors mainly concentrates on the deadlock-freedom of their glue and the loyalty of their roles with the glue.
Finally, all the above properties are described in very simple semantics that allow components and connectors to be easily checked. That is, they do not require elaborate and computationally expensive machinery such as model-checking technology.

7.2 Meeting the Goals of the Thesis

At the end of Chapter 2, we raised a discussion on the open issues of modelling CBS in the literature. Our model is proposed to resolve these issues as follows.

Formal Semantics.

As an example of an architectural description language, the Koala component model (as currently used in practice) has no theoretical basis to support rigorous analysis and verification on its time-sensitive behaviours. One of the foremost motivations for this work is to provide a mathematical model to underpin ADLs such as the Koala component model.

We meet our first chief goal of the thesis by introducing a behaviour language, which is solely based on the initial algebra approach to form its operational semantics by the time-slot model and its syntax language by the component protocol. This behaviour language is specially designed for real-time and embedded systems, such as TV platforms.

True Concurrency

Real-time and embedded systems usually require deterministic and time-critical behaviours, processing in a true concurrent environment, such as scheduling algorithm processes. Since the causal ordering of events is strongly depending on time, the ordering of concurrent events is subjective and not distinguished. Therefore, true concurrency models come closer to represent this kind of relative concurrency.

The time-slot model is a true concurrency model that illustrates execution sequences over components by runs, which associate finitely many service occurrences with an infinite set of integer-valued time intervals. Concurrency can be produced in such a way that different services may be associated with the same time slot(s) in a run, and furthermore simultaneity can be identified when different services must be associated with the same time slot(s).

Concurrency Granularity

In CSP and other process algebras, concurrency only arises from the composition of processes. As mentioned above, the time-slot model may produce concurrency without composing processes together. With this concern, our time-slot model is more concurrency-oriented.

The component vector language [MS03] can also describe concurrency in an individual vector, i.e. without process composition. However, concurrency is merely considered in a single
component, with the assumption that services can only perform sequentially in an interface. Concurrency (and simultaneity) may just happen among distinct interfaces of the component. The time-slot model relaxes this assumption by allowing concurrent occurrences of services on the same interface. Concurrency can be naturally described within a single run of the interface. By considering concurrency at the interface level, our time-slot model demonstrates a finer granularity of concurrency than that of the component vector language [MS03] which describes concurrency at the component level. This improvement can facilitate software engineers to develop multi-threaded interfaces for components.

**Notation Readability**

Current behaviour expression languages (such as SOFA's behaviour protocol [PV02]) suffer from a lack of concise notation to denote operation calls (called functions in this thesis), which are mainly split in two events and not in primitive notation. Our behaviour syntax language, the component protocol \( P \), is proposed to tackle this difficulty by adapting service names as its alphabet instead of directly using event names in the behaviour protocol. Each service name denotes an event or a function (operation call). This distinction can be detected by the labelling function \( SEV \) which maps every service name to an actual service. Then, the type of service can be easily recognised in our component signature, where the concept of function has been primitively defined. The component protocol provides a notation based on component services, thus achieving the fourth goal of the thesis aiming to create a more readable expression language to simplify the modelling of operation calls in components.

**7.3 Future Directions**

Many future development of this work are possible. The major interest is that, the formal analysis and reasoning in identifying the complete set of intended behaviours should be supported by automated tools. Ideally, these should be executed behind the scenes, with feedback being provided in the form of generated counterexamples, possibly after executing automatically generated test cases. Feedback provided in the form of counterexamples would help designers explore the consequences of design decisions and identify the complete set of intended behaviours of the system.

Below, we outline a number of the possible directions from our work.
7.3.1 Acceptance Set Semantics

The behaviour refinement relationship \( (t_1 \sqsubseteq t_2 \leftrightarrow \sigma_{t_2}(t_2) \subseteq \sigma_{t_1}(t_1)) \) presented in Definition 5.1.4 is probably enough in a purely language-based model. However, given that we have non-deterministic choice in which, intuitively,

\[ a; (b + c) \neq (a; b) + (a; c) \]

then the semantics should not be entirely language based.

In a natural way to solve this problem, the acceptance set semantics as described by Hennessy [Hen85] should be adopted. Then, the time-slot language \( L \) becomes more complicated.

**Definition 7.3.1** Given a run \( f \in \text{Runs}_\sigma \), we define an acceptance run to be an ordered pair \((f, A)\), where \( A \subseteq P \) denotes the acceptance set.

An acceptance run is a run \( f \) associated with an acceptance set \( A \), that is a set of service names. Next, the partial order of acceptance run is defined as follows.

**Definition 7.3.2** Let \((f, A)\) and \((f', A')\) be acceptance runs, we define \((f, A) \leq (f', A') \Leftrightarrow \exists a \in A : f.a \leq f' \lor (f, A) = (f', A)\). It is easy to show that \( \leq \) is a partial order on the set of all acceptance runs.

Then, the acceptance time-slot language can be defined as below. The idea is that \((f, A) \in L\) if and only if there exists \( a \in A \), such that \((f.a, B) \in L\) for some \( B \subseteq A\), and the language of pairs is closed under the partial order \( \leq \).

**Definition 7.3.3** We define the acceptance time-slot language \( L \subseteq \text{Runs}_\sigma \times \wp(P) \), such that

a) \( L \) is closed under \( \leq \), and
b) \((f, A) \in L \Leftrightarrow \exists a \in A : (f.a, B) \in L\), for some \( B \subseteq A\)

Note that time-slot languages equipped with acceptance sets are important in distinguishing between the expressions \( a; (b + c) \) and \((a; b) + (a; c)\), in the sense that

\[ (a, \{b, c\}) \neq \{(a, \{b\}), (a, \{c\})\} \]

This need to distinguish between meanings of expressions, is what prompted the introduction of refusal or acceptance sets.

Finally, the concentration on acceptance runs can be defined as follows.

**Definition 7.3.4** Let \((f, A)\) and \((f', A')\) be acceptance runs, we define the concatenation on acceptance runs by
Chapter 7 Conclusion

\[ (f \cdot A) \cdot (f', A') = \begin{cases} (f \cdot f', A'), & \text{if } \exists a \in A : f.a \leq f' \\ (f, A), & \text{if } f' = \perp \\ \text{undefined, otherwise} \end{cases} \]

7.3.2 Adaptors

Component adaptation is widely recognised to be one of the crucial problems in CBSE [YS97, Cam99, GS01, Can04], which constituted a starting point of our future work.

As illustrated in preceding examples, a current service interaction is built upon the assumption that all parties in the bindings are mutually matching. However, it is not always the case in practice, because for any given a service, there must exist a particular matching counterpart that has the equivalent functionality, but the opposite prefix to that service. More likely, the matching counterpart is achieved by composing a set of sub-functional services and sometimes may be decomposed from a super-functional service. For this purpose, an adaptor can be produced to bridge these functional incompatibilities. For example, the given well-defined multi-party binding in Figure 7.1 assumes that the event EvtA matches with the events EvtB and EvtC, but the current approach is incapable of specifying that EvtA wants to interact with the combination of EvtB and EvtC. Therefore, by inserting the adaptor AdaptorA in-between, a new event EvtD is introduced as EvtA’s matched partner. It acts as an interceptor to listen to EvtA’s calls and deliberately distribute them to EvtB and EvtC.

![Figure 7.1 Binding with Adaptors](image)

Medvidovic et al [MDE'95] consider adaptor as a special kind of connector. The authors presented a classification framework and taxonomy of software connectors. This taxonomy is based on four service categories (Communication, Coordination, Conversion and Facilitation) and the eight basic connector types (Procedure Call, Event, Data Access, Linkage, Communication Stream, Arbitrator, Adaptor and Distributor). Additional features of each connector type are expressed in terms of the dimensions of the connector type. Introducing these kinds of connectors to our model requires more advanced consideration on the behaviour description techniques and architectural properties of the CBS to be adapted.
7.3.3 Automation

It becomes apparent through the case studies that the process of translating a component protocol into a time-slot language is a tedious task in need of automation. Its highly repetitive nature makes this more appealing and further work should make this forthcoming.

Automation of the process can be seen in two phases. One is to parse component protocol expressions. Automated support for the formal construction boils down to applying the appropriate definitions (Definition 4.3.1 – 4.4.19), and this would require reasonable programming skills to implement them.

More advanced programming skills may be required for the second phase, which involves constructing a CBS model and reasoning various architectural properties. Then, the implementation of the first phase could be used as a back end.

7.4 Afterword

Formal methods are often frowned upon by practitioners in industry. However, apart from providing a powerful tool support, well-grounded formal approaches could be welcomed if they were seen to be useful in locating software design errors due to subtle issues that human inspections tend to miss. This would liberate practitioners to focus on the hard intellectual work of gaining knowledge about the system, obtaining and validating requirements and eventually, with the aid of the formal-based tool, producing high-quality specifications that provide compelling evidence that the behaviour of the system would be predictable at all times.
Appendix A

Proof of Lemma 4.4.7

Let $f, g, h \in \text{Runs}_G$ and $l, m, n \in \mathbb{N}^+$, then 
\[
(f_{\max(l,m)} g)_{\max(l,m)} h = f_{\max(l,m)} (g_{\max(l,m)} h).
\]

(continued from the proof)

Case 5: let $l > n > m$, then by definition 4.4.7,

\[
\text{RHS} := f_{\max(l,m)} (g_{\max(l,m)} h) = f \mid IS(g, m, n-m) \mid h, n, l-n
\]

\[
\text{LHS} := (f_{\max(l,m)} g)_{\max(l,m)} h = (f \mid IS(g, m, l-m) \mid IS(h, n, l-n)
\]

\[
= f \mid IS(g, m, n-m) \mid h, n, l-n
\]

\[
= \text{RHS}
\]

Lemma 4.4.5

Remark 4.4.5 (c)

Case 6: let $l > m = n$, then by definition 4.4.7,

\[
\text{LHS} := (f_{\max(l,m)} g)_{\max(l,m)} h = (f \mid IS(g, m, l-m) \mid IS(h, m, l-m)
\]

\[
\text{RHS} := f_{\max(l,m)} (g_{\max(l,m)} h) = f \mid IS(g, h, m, l-m)
\]

\[
= f \mid IS(g, m, l-m) \mid IS(h, m, l-m)
\]

\[
= \text{LHS}
\]

Lemma 4.4.3

Remark 4.4.5 (c)

Case 7: let $n > l > m$, then by definition 4.4.7,

\[
\text{RHS} := f_{\max(l,m)} (g_{\max(l,m)} h) = IS(f, l, n-l) \mid IS(g, m, n-m) \mid h)
\]

\[
\text{LHS} := (f_{\max(l,m)} g)_{\max(l,m)} h = IS(f \mid IS(g, m, l-m), l, n-l) \mid h
\]

\[
= IS(f, l, n-l) \mid IS(g, m, n-m) \mid h
\]

\[
= \text{RHS}
\]

Lemma 4.4.5

Remark 4.4.5 (c)
### Appendix B

### Example 4.4.9

(continued from Example 4.4.9)

<table>
<thead>
<tr>
<th>ID</th>
<th>Component Signature</th>
<th>Time Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name</td>
<td>Service</td>
</tr>
<tr>
<td>1</td>
<td>s</td>
<td>(?a, !b)</td>
</tr>
<tr>
<td>2</td>
<td>t</td>
<td>(?c, ?d)</td>
</tr>
<tr>
<td>3</td>
<td>u</td>
<td>(?g, !h)</td>
</tr>
<tr>
<td>4</td>
<td>v</td>
<td>!e</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Component Signature</th>
<th>Time Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name</td>
<td>Service</td>
</tr>
<tr>
<td>1</td>
<td>s</td>
<td>(?a, !b)</td>
</tr>
<tr>
<td>2</td>
<td>t</td>
<td>(?c, ?d)</td>
</tr>
<tr>
<td>3</td>
<td>u</td>
<td>(?g, !h)</td>
</tr>
<tr>
<td>4</td>
<td>v</td>
<td>!e</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Component Signature</th>
<th>Time Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name</td>
<td>Service</td>
</tr>
<tr>
<td>1</td>
<td>s</td>
<td>(?a, !b)</td>
</tr>
<tr>
<td>2</td>
<td>t</td>
<td>(?c, ?d)</td>
</tr>
<tr>
<td>3</td>
<td>u</td>
<td>(?g, !h)</td>
</tr>
<tr>
<td>4</td>
<td>v</td>
<td>!e</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Component Signature</th>
<th>Time Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name</td>
<td>Service</td>
</tr>
<tr>
<td>1</td>
<td>s</td>
<td>(?a, !b)</td>
</tr>
<tr>
<td>2</td>
<td>t</td>
<td>(?c, ?d)</td>
</tr>
<tr>
<td>3</td>
<td>u</td>
<td>(?g, !h)</td>
</tr>
<tr>
<td>4</td>
<td>v</td>
<td>!e</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Component Signature</th>
<th>Time Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name</td>
<td>Service</td>
</tr>
<tr>
<td>1</td>
<td>s</td>
<td>(?a, !b)</td>
</tr>
<tr>
<td>2</td>
<td>t</td>
<td>(?c, ?d)</td>
</tr>
<tr>
<td>3</td>
<td>u</td>
<td>(?g, !h)</td>
</tr>
<tr>
<td>4</td>
<td>v</td>
<td>!e</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Component Signature</th>
<th>Time Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name</td>
<td>Service</td>
</tr>
<tr>
<td>1</td>
<td>s</td>
<td>(?a, !b)</td>
</tr>
<tr>
<td>2</td>
<td>t</td>
<td>(?c, ?d)</td>
</tr>
<tr>
<td>3</td>
<td>u</td>
<td>(?g, !h)</td>
</tr>
<tr>
<td>4</td>
<td>v</td>
<td>!e</td>
</tr>
</tbody>
</table>

- 198 -
Bibliography


- 200 -


[Gam94] E. Gamma: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994


[Sel93] B. Selic: *An Efficient Object-Oriented Variation of the Statecharts Formalism for Distributed Real-Time Systems*, IFIP Conference on Hardware Description Languages and Their Applications, Apr. 1993


